Perancangan dan Analisis Algoritma

Tulang punggung teori dari seluruh kurikulum: cara memodelkan masalah komputasi secara presisi, merancang algoritma untuknya, membuktikan algoritma itu benar, dan bernalar formal tentang bagaimana waktu jalannya bertumbuh — sebelum satu baris kode produksi pun ditulis.

Irfan Subakti &
Rully Soelaiman

Profitina
profitina.com

Perancangan dan Analisis Algoritma

Dari Prinsip Dasar hingga Program yang Terbukti Benar

Irfan Subakti & Rully Soelaiman

Profitina

Hak cipta © 2026 Irfan Subakti & Rully Soelaiman. Hak cipta dilindungi undang-undang.

Diterbitkan oleh Profitina · profitina.com

Edisi pertama, September 2026.

Setiap program dalam buku ini sudah dikompilasi dan dijalankan sebelum keluarannya dicetak. Kode sumber lengkapnya ada di folder Code yang menyertai buku ini. Program-program tersebut disediakan untuk keperluan belajar-mengajar; Irfan Subakti & Rully Soelaiman dan penerbit tidak bertanggung jawab atas penggunaannya di luar keperluan itu.

Daftar Isi

Kata Pengantar ..... **lxxii**
Untuk siapa buku ini ditulis lxxii
Isi buku ini lxxii
Cara membacanya lxxiii
Tentang penulis lxxiii
Ucapan terima kasih lxxiii

BAB 1 Fondasi, Kerangka Kerja, dan Persoalan Pertama **1**
1.1 Dari Mana Kita Mulai 1
1.2 Algoritma dan Struktur Data — Dua Pilar 2
1.2.1 Apa itu algoritma 2
1.2.2 Program adalah wujud nyata sebuah algoritma 3
1.2.3 Apa itu struktur data 3
1.2.4 Sasaran yang hendak kita capai 4
1.3 Dua Sasaran Perancangan Algoritma 5
1.3.1 Sasaran pertama: kebenaran 5
1.3.2 Sebuah persoalan yang dinyatakan dengan cermat: pengurutan 6
1.3.3 Insertion sort, dan invarian gelung yang membuktikannya 7
1.3.4 Sasaran kedua: efisiensi 9
1.3.5 Patokan kasar yang menentukan segalanya 10
1.3.6 Kedua sasaran bekerja bersama 11
1.4 Lima Paradigma Perancangan 11
1.4.1 Inkremental, atau *brute force* 11
1.4.2 *Greedy* 12
1.4.3 *Divide and conquer* 12
1.4.4 *Dynamic programming* 13
1.4.5 Matematis dan teori bilangan 14
1.4.6 Lima paradigma dalam satu tabel 16
1.5 Kerangka Lima Fase Pemecahan Persoalan 16
1.5.1 Fase 1 — Memahami 16
1.5.2 Fase 2 — Mengabstraksi 17
1.5.3 Fase 3 — Merancang 17
1.5.4 Fase 4 — Menganalisis 17
1.5.5 Fase 5 — Mengimplementasikan 18
1.6 Persoalan 1 — Columns (E-Olymp 10281) 18
1.6.1 Fase 1 — Memahami 18
1.6.2 Fase 2 — Mengabstraksi 20
1.6.3 Fase 3 — Merancang 22
1.6.4 Fase 4 — Menganalisis 22
1.6.5 Fase 5 — Mengimplementasikan 23
1.7 Persoalan 2 — Journey (E-Olymp 10280) 23
1.7.1 Fase 1 — Memahami 23
1.7.2 Fase 2 — Mengabstraksi 24
1.7.3 Fase 3 — Merancang 26
1.7.4 Fase 4 — Menganalisis 26
1.7.5 Fase 5 — Mengimplementasikan 26
1.8 Persoalan 3 — Unique Numbers (SPOJ DCEPC901) 26
1.8.1 Fase 1 — Memahami 27
1.8.2 Fase 2 — Mengabstraksi 28
1.8.3 Fase 3 — Merancang 30

1.8.4 Fase 4 — Menganalisis ... 30
1.8.5 Fase 5 — Mengimplementasikan ... 30
1.9 Persoalan 4 — Minimum and Maximum Numbers (SPOJ NEWLANG) ... 31
1.9.1 Fase 1 — Memahami ... 31
1.9.2 Fase 2 — Mengabstraksi ... 32
1.9.3 Fase 3 dan 4 — Merancang dan Menganalisis ... 34
1.9.4 Fase 5 — Mengimplementasikan ... 35
1.10 Persoalan 5 — Interesting Sequence (E-Olymp 4894) ... 35
1.10.1 Fase 1 — Memahami ... 36
1.10.2 Fase 2 — Mengabstraksi ... 37
1.10.3 Fase 3 dan 4 — Merancang dan Menganalisis ... 39
1.10.4 Fase 5 — Mengimplementasikan ... 40
1.11 Persoalan 6 — Recursive Function 1 (E-Olymp 10296) ... 40
1.11.1 Fase 1 — Memahami ... 41
1.11.2 Fase 2 — Mengabstraksi ... 41
1.11.3 Fase 3 dan 4 — Merancang dan Menganalisis ... 44
1.11.4 Fase 5 — Mengimplementasikan ... 45
1.12 Keenam Persoalan dalam Sekilas Pandang ... 45
1.12.1 Mengenali paradigma mana yang berlaku ... 46
1.13 Dari Mana Persoalan Ini Berasal: Para *Online Judge* ... 46
1.14 Soal Juri sebagai Penelitian: Dua Makalah dari ITS ... 47
1.14.1 Mengekstrapolasi fungsi periodik ... 48
1.14.2 Membaca angka tulisan tangan ... 50
1.15 Buku Acuan, Peta Jalan, dan Bab-Bab Evaluasi ... 52
1.15.1 Buku acuan ... 52
1.15.2 Peta jalan ... 53
1.15.3 Bab-bab evaluasi ... 54
1.16 Menyiapkan Meja Kerja: VS Code dan Kompilator C++ ... 54
1.16.1 Windows, langkah demi langkah ... 55
1.16.2 macOS, langkah demi langkah ... 55
1.16.3 Linux, langkah demi langkah ... 56
1.16.4 VS Code di setiap sistem ... 56
1.16.5 Program pertama Anda, dikompilasi dan dijalankan ... 56
1.16.6 Memberi masukan pada soal *online judge* ... 57
1.16.7 *Debugging*: menyaksikan program berpikir ... 57
1.17 Profitina dalam Praktik ... 57
1.17.1 Studi kasus yang dikerjakan: Profitina Laundry ... 58
1.18 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya ... 59
1.19 Ringkasan Bab ... 60
1.20 Soal Latihan ... 61
1.21 Penyelesaian Soal Latihan ... 62
Daftar Pustaka ... 64

BAB 2 Memulai: Model RAM, Insertion Sort, dan Pencarian ... 66
2.1 Di Mana Kita Berhenti ... 66
2.2 Sebenarnya Buku Ini tentang Apa ... 67
2.2.1 Yang memang dipelajari di buku ini ... 67
2.2.2 Yang TIDAK dipelajari di buku ini ... 67
2.3 Sekelumit Sejarah Algoritma ... 68
2.3.1 Dari mana kata itu berasal ... 68
2.3.2 Algoritma yang pertama ... 69
2.3.3 Abad kesembilan belas: mesin dan pemrogram pertama ... 69
2.3.4 Abad kedua puluh: apa sebenarnya *komputasi* itu ... 70
2.4 Kosakata: Problem, Instans, Algoritma, Program, Struktur Data ... 72
2.4.1 Problem algoritmik ... 72

2.4.2 Instans sebuah problem ... 72
2.4.3 Algoritma ... 73
2.4.4 Program ... 73
2.4.5 Struktur data ... 74
2.4.6 Kebenaran dan efisiensi, didefinisikan ... 74
2.5 Lima Tujuan Perancangan ... 75
2.6 Model Komputasi RAM ... 76
2.6.1 Aturan main model ini ... 76
2.6.2 Apa yang dihitung satu langkah ... 76
2.6.3 Apa yang BUKAN satu langkah ... 77
2.6.4 Mengapa memilih model ini? ... 77
2.7 Cara Menganalisis Waktu Jalan ... 78
2.7.1 Apa itu ukuran masukan? ... 78
2.7.2 Waktu jalan bukan waktu jam dinding ... 79
2.7.3 Contoh lengkap yang pertama ... 79
2.8 Insertion Sort: Algoritmanya ... 80
2.8.1 Intuisinya: menata kartu di tangan ... 80
2.8.2 Algoritmanya dalam pseudokode ... 81
2.8.3 Menelusuri algoritmanya dengan tangan ... 82
2.8.4 Insertion Sort dalam C++ ... 83
2.9 Membuktikan Insertion Sort Benar: Invarian Gelung ... 85
2.9.1 Apa itu invarian gelung ... 86
2.9.2 Invarian untuk Insertion Sort ... 86
2.9.3 Inisialisasi ... 86
2.9.4 Pemeliharaan ... 87
2.9.5 Terminasi ... 88
2.10 Menghitung Biaya Insertion Sort ... 89
2.10.1 Tabel biaya dan cacah ... 89
2.10.2 Rumus umumnya ... 90
2.11 Kasus Terbaik, Kasus Terburuk, dan Kasus Rata-rata ... 90
2.11.1 Kasus terbaik: larik yang sudah terurut ... 91
2.11.2 Kasus terburuk: larik dengan urutan terbalik ... 92
2.11.3 Kasus rata-rata ... 94
2.11.4 Menguji analisis itu pada mesin ... 95
2.11.5 Mengapa kita memusatkan perhatian pada kasus terburuk ... 98
2.12 Pertumbuhan Fungsi ... 99
2.12.1 Logaritma, dengan kata-kata ... 99
2.12.2 Tabel pembandingnya ... 99
2.12.3 Melihatnya sebagai gambar ... 101
2.12.4 Hierarki laju pertumbuhan ... 102
2.12.5 Ancar-ancar yang praktis ... 103
2.13 Pencarian: Linear Search ... 103
2.13.1 Problemnya ... 103
2.13.2 Algoritmanya ... 104
2.13.3 Invarian gelung untuk linear search ... 104
2.13.4 Analisisnya ... 105
2.13.5 Linear search dalam C++ ... 105
2.13.6 Bisakah kita lebih cepat? ... 106
2.14 Binary Search ... 106
2.14.1 Problemnya ... 106
2.14.2 Gagasan intinya ... 107
2.14.3 Algoritmanya ... 107
2.14.4 Menelusuri binary search dengan tangan ... 108
2.14.5 Mengapa ia $O(\lg n)$ ... 109
2.14.6 Binary search dalam C++, dan jebakan limpah ... 110

2.14.7 Binary search sebagai rekursi 112
2.14.8 Imbal baliknya: prapemrosesan 113
2.15 Menengok Kembali Paradigma Perancangan 113
2.15.1 Pendekatan inkremental 113
2.15.2 Bagi dan kuasai 114
2.15.3 Algoritma ad hoc 114
2.16 Menyambungkan Bab 1 dan Bab 2 115
2.16.1 Letak bab ini di dalam buku 116
2.17 Profitina dalam Praktik 116
2.17.1 Studi kasus yang dapat dijalankan: Profitina Laundry 117
2.18 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 120
2.19 Ringkasan Bab 121
2.20 Soal Latihan 122
2.21 Penyelesaian Soal Latihan 123
Bacaan lanjutan 130
Daftar Pustaka 130

BAB 3 Notasi Asimtotik, Rekurensi, dan Teorema Master 131
3.1 Dari Mana Kita Melanjutkan 131
3.1.1 Kebenaran parsial dan kebenaran total 132
3.1.2 Apa yang masih kurang 133
3.2 Mengapa Kita Memerlukan Bahasa untuk Pertumbuhan 133
3.2.1 Angkanya, supaya Anda percaya 134
3.3 Big-O: Batas Atas Asimtotik 135
3.3.1 Sebuah pembuktian, dikerjakan tuntas: $7n - 3 = O(n)$ 136
3.3.2 Sebuah pembuktian yang ambangnya bekerja sungguhan: $3n2 + 5n + 7 = O(n2)$ 137
3.3.3 Sebuah penyangkalan: $n2 \neq O(n)$ 137
3.3.4 Membiarkan mesin yang memburu saksinya 138
3.4 Big-Ω: Batas Bawah Asimtotik 140
3.5 Big-Θ: Batas Ketat Asimtotik 141
3.6 Little-o, Little-ω, dan Analogi Bilangan 142
3.6.1 Analogi bilangan 143
3.6.2 Penyalahgunaan notasi yang kita semua lakukan 143
3.7 Sifat-Sifat Notasi Tersebut 144
3.7.1 Ketransitifan 144
3.7.2 Kerefleksifan 144
3.7.3 Kesetangkupan dan kesetangkupan transpos 144
3.7.4 Jumlah, hasil kali, dan konstanta 145
3.7.5 Mengapa kita membuang konstanta dan suku berorde lebih rendah — dibuktikan, bukan diklaim 145
3.7.6 Di mana analogi bilangan patah: ketrikotomian gagal 146
3.8 Notasi Baku dan Fungsi-Fungsi yang Sering Muncul 146
3.8.1 Kemonotonan 146
3.8.2 Lantai dan langit-langit 146
3.8.3 Aritmetika modular 147
3.8.4 Polinomial 147
3.8.5 Eksponensial 147
3.8.6 Logaritma — dan notasi yang akan kita pakai di mana-mana 148
3.8.7 Faktorial dan hampiran Stirling 149
3.8.8 Logaritma teriterasi 149
3.8.9 Bilangan Fibonacci 149
3.9 Jumlahan: Aritmetika Gelung 152
3.9.1 Deret aritmetika 152
3.9.2 Hasil yang sama lewat induksi 154
3.9.3 Deret geometri 155

3.9.4 Deret harmonik 156
3.9.5 Jumlahan teleskopis 156
3.9.6 Membatasi sebuah jumlahan dengan integral 157
3.10 Rekurensi: Persamaan yang Menggambarkan Rekursi 160
3.10.1 Apa itu rekurensi 160
3.10.2 Bentuk umum bagi dan kuasai 161
3.10.3 Tiga kerewelan yang boleh kita abaikan 161
3.10.4 Tiga metodenya 161
3.11 Metode 1: Metode Substitusi 162
3.11.1 Sebuah contoh lengkap: $T(n) = 4T(n/2) + n$ adalah $O(n3)$ 162
3.11.2 Percobaan yang gagal: apakah ia $O(n2)$? 163
3.11.3 Perbaikannya: perkuat hipotesisnya 164
3.11.4 Dua teknik substitusi lagi 165
3.11.5 Memeriksa aljabarnya terhadap angka sesungguhnya 165
3.12 Metode 2: Iterasi dan Pohon Rekursi 167
3.12.1 Iterasi murni: $T(n) = n + 3T(\lfloor n/4 \rfloor)$ 168
3.12.2 Menggambar pohonnya: $T(n) = T(n/4) + T(n/2) + n2$ 169
3.12.3 Ketika tiap tingkat berbiaya sama: $T(n) = T(n/3) + T(2n/3) + n$ 170
3.12.4 Mencacah daunnya: dari mana $n\log ba$ berasal 171
3.12.5 Tiga bentuk pohon 172
3.13 Metode 3: Teorema Master 174
3.13.1 Pernyataannya 174
3.13.2 Apa arti tiap kepingnya, dalam kata-kata 175
3.13.3 Peran ε: mengapa harus "secara polinomial" 175
3.13.4 Syarat keteraturan, dalam bahasa sederhana 175
3.13.5 Kartu ringkasannya 176
3.13.6 Di mana teorema itu menolak menjawab: celah-celahnya 177
3.13.7 Strategi lima langkah 178
3.14 Dua Puluh Soal Teorema Master, Dikerjakan Tuntas 178
3.14.1 Pencarian biner: $T(n) = T(n/2) + 1$ 178
3.14.2 $T(n) = 9T(n/3) + n$ 179
3.14.3 Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$ 179
3.14.4 $T(n) = 3T(n/4) + n\lg n$ 179
3.14.5 $T(n) = 4T(n/2) + n3$ 179
3.14.6 Celahnya: $T(n) = 2T(n/2) + n\lg n$ 180
3.14.7 $T(n) = 4T(n/2) + n$ 180
3.14.8 $T(n) = 4T(n/2) + n2$ 180
3.14.9 $T(n) = 8T(n/2) + n2$ 180
3.14.10 Strassen: $T(n) = 7T(n/2) + n2$ 181
3.14.11 $T(n) = 8T(n/2) + n3$ 181
3.14.12 $T(n) = 2T(n/4) + n$ 181
3.14.13 $T(n) = 3T(n/3) + n/2$ 181
3.14.14 $T(n) = 16T(n/4) + n2$ 182
3.14.15 $T(n) = T(n/2) + n$ 182
3.14.16 Celah yang satu lagi: $T(n) = 2T(n/2) + n/\lg n$ 182
3.14.17 $T(n) = 5T(n/2) + n2$ 182
3.14.18 $T(n) = T(2n/3) + 1$ 183
3.14.19 Keteraturan yang benar-benar bekerja: $T(n) = 2T(n/2) + n2/\lg n$ 183
3.14.20 Keteraturan yang gagal: $T(n) = T(n/2) + n(2 - \cos n)$ 183
3.14.21 Memeriksa semuanya dengan mesin 183
3.15 Merge Sort, Diturunkan dari Nol 187
3.15.1 Cetak biru bagi dan kuasai 187
3.15.2 Algoritmanya 188
3.15.3 Penelusuran lengkap atas $A = [1,5,2,4,6,3,2,6]$ 188
3.15.4 Membuktikan MERGE benar dengan loop invariant 188

3.15.5 Rekurensinya, dan tiga cara menyelesaikannya 189
3.15.6 Programnya 190
3.15.7 Berapa harga Merge Sort, dan apa yang dibelinya 193
3.16 Profitina dalam Praktik 193
3.16.1 Studi kasus terkerjakan: Profitina Laundry 194
3.17 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 197
3.18 Ringkasan Bab 198
3.18.1 Menengok ke depan 200
3.19 Soal Latihan 200
3.20 Penyelesaian Soal Latihan 201
Bacaan lanjutan 203
Daftar Pustaka 203

BAB 4 Evaluasi dan Pembahasan I: Fondasi, Analisis, dan Problem Pertama 204
4.1 Posisi Kita Saat Ini 204
4.2 Cara Membaca Sebuah Persoalan Algoritma 205
4.2.1 Langkah 1 — Salin format masukan dan keluaran, apa adanya 205
4.2.2 Langkah 2 — Berilah nama pada yang dicari, dan nama pada datanya 206
4.2.3 Langkah 3 — Ubah kendala menjadi anggaran waktu 206
4.2.4 Langkah 4 — Hitung lima kasus pertama dengan tangan 207
4.2.5 Langkah 5 — Sebutkan bentuknya 207
4.2.6 Langkah 6 — Tuliskan invariannya, rekurensinya, atau rumusnya, lalu buktikan 207
4.2.7 Langkah 7 — Baru sekarang, tulis programnya 208
4.3 Persoalan 1 — LDCSUMAB: Jumlah Dua Pangkat Dua 208
4.3.1 Persoalannya dengan kata-kata kita sendiri 209
4.3.2 Langkah 1 sampai 3: format, lambang, anggaran 209
4.3.3 Langkah 4: kasus-kasus pertama dengan tangan 210
4.3.4 Langkah 5 dan 6: bentuknya, dan rumusnya 211
4.3.5 Bukti kebenaran 213
4.3.6 Kompleksitas 214
4.3.7 Langkah 7: programnya 215
4.3.8 Alternatif: pencarian biner atas nomor blok 216
4.3.9 Memeriksa terhadap gaya kasar 217
4.3.10 Kesalahan yang sering terjadi pada persoalan ini 218
4.4 Persoalan 2 — TOHU: Membantu Tohu 219
4.4.1 Persoalannya dengan kata-kata kita sendiri 219
4.4.2 Langkah 1 sampai 3: format, lambang, anggaran 219
4.4.3 Langkah 4: kasus-kasus pertama dengan tangan 220
4.4.4 Langkah 5 dan 6: memulihkan ak dengan menyelisihkan 221
4.4.5 Menjumlahkan secara teleskopik 222
4.4.6 Bukti kebenaran 223
4.4.7 Kompleksitas, ketelitian, dan limpahan 224
4.4.8 Langkah 7: programnya 224
4.4.9 Alternatif: pecahan eksak, tanpa mengandaikan rumus 226
4.4.10 Kesalahan yang sering terjadi pada persoalan ini 227
4.5 Persoalan Terkerjakan tentang Kebenaran: Invarian dan Penelusuran 227
4.5.1 Persoalan Terkerjakan A — invarian gelung untuk aturan Horner 228
4.5.2 Persoalan Terkerjakan B — menelusuri *insertion sort* 231
4.5.3 Persoalan Terkerjakan C — menelusuri pencarian biner 233
4.6 Persoalan Terkerjakan tentang Pencacahan dan Laju Pertumbuhan 236
4.6.1 Persoalan Terkerjakan D — hitungan langkah yang tepat dari gelung rangkap tiga 236
4.6.2 Persoalan Terkerjakan E — pernyataan asimtotik dengan konstanta yang eksplisit 238
4.6.3 Persoalan Terkerjakan F — mengurutkan tiga belas fungsi menurut laju pertumbuhan 241
4.7 Persoalan Terkerjakan tentang Rekurensi 244
4.7.1 Persoalan Terkerjakan G — metode substitusi, dan hipotesis yang harus diperkuat 244

4.7.2 Persoalan Terkerjakan H — pohon rekursi dengan tabel tingkat 246
4.7.3 Persoalan Terkerjakan I — empat rekurensi dan Teorema Master, termasuk sebuah celah 249
4.8 Persoalan Terkerjakan tentang Bentuk Tertutup dan Rekurensi Tersembunyi 252
4.8.1 Persoalan Terkerjakan J — Kupon Laundry 252
4.8.2 Persoalan Terkerjakan K — menubin lajur $2 \times N$ 254
4.9 Dari Gaya Kasar ke yang Lebih Baik: Perbandingan Berwaktu 258
4.9.1 Persoalan Terkerjakan L — adakah pasangan yang berjumlah x? 258
4.10 Persoalan-Persoalan Bab Ini Dibandingkan 261
4.11 Profitina dalam Praktik 263
4.11.1 Studi kasus terkerjakan: Profitina Laundry 264
4.12 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 267
4.13 Ringkasan Bab 269
4.14 Soal Latihan 269
4.15 Penyelesaian Soal Latihan 270
Daftar Pustaka 272

BAB 5 Pengurutan dalam Praktik: Heapsort dan Quicksort 274
5.1 Dari Mana Kita Melanjutkan 274
5.1.1 Perkakas sejauh ini 274
5.1.2 Celah yang menjengkelkan 275
5.1.3 Mengapa $\Theta(n\lg n)$ adalah sasarannya, dan berapa biaya pengurutan dalam sistem sungguhan 275
5.2 Wawasan dari Selection Sort 276
5.3 Binary Heap: Sebuah Pohon yang Tersembunyi di Dalam Larik 277
5.3.1 Pohon biner hampir lengkap 277
5.3.2 Aritmetika indeksnya, diturunkan 278
5.3.3 Sifat heap 279
5.3.4 Sebuah heap yang konkret 280
5.3.5 Heap itu tidak terurut 281
5.3.6 Tinggi sebuah heap adalah $\lfloor \lg n \rfloor$ 281
5.3.7 Di mana daun-daunnya berada 282
5.3.8 Berapa banyak simpul yang duduk di tiap tinggi 282
5.3.9 Semuanya, di dalam sebuah program 283
5.4 `MAX-HEAPIFY`: Mesin yang Menjaga Heap Tetap Jujur 285
5.4.1 Persoalan yang dipecahkannya 285
5.4.2 Algoritmanya 286
5.4.3 Penelusuran lengkap 286
5.4.4 Mengapa ia benar 289
5.4.5 Waktu jalannya 290
5.4.6 `MAX-HEAPIFY` dalam C++ 291
5.5 `BUILD-MAX-HEAP`: Pembangunan Waktu Linear yang Mengejutkan 293
5.5.1 Algoritmanya 293
5.5.2 Kebenarannya, dengan invarian gelung 294
5.5.3 Penelusuran lengkap 294
5.5.4 Batas yang longgar: $O(n\lg n)$ 298
5.5.5 Batas yang ketat: $O(n)$, diturunkan selangkah demi selangkah 298
5.5.6 Identitas $h = 0\infty h/2h = 2$, dibuktikan 300
5.5.7 Melihat biaya linear itu pada mesin sungguhan 301
5.6 Heapsort 304
5.6.1 Algoritmanya 304
5.6.2 Penelusuran lengkap 305
5.6.3 Kebenarannya, dengan invarian gelung 307
5.6.4 Waktu jalannya 307
5.6.5 Mengapa Heapsort tidak stabil 308

5.6.6 Heapsort dalam C++ 309
5.7 Antrean Prioritas 310
5.7.1 Apa itu antrean prioritas 310
5.7.2 Di mana antrean prioritas dipakai 311
5.7.3 `MAXIMUM`: yang mudah 311
5.7.4 `EXTRACT-MAX` 311
5.7.5 `INCREASE-KEY` 312
5.7.6 `INSERT` 312
5.7.7 Berapa biaya keempat operasinya 313
5.7.8 Antrean prioritas lengkap dalam C++ 314
5.7.9 Memakai pustaka baku saja 316
5.8 Quicksort 318
5.8.1 Gagasannya 318
5.8.2 Bagi dan kuasai, dengan pekerjaannya dipindahkan 318
5.8.3 `PARTITION` versi Lomuto, dan invarian empat wilayah 319
5.8.4 Satu pemartisian Lomuto, ditelusuri dengan tangan 320
5.8.5 `PARTITION` versi Hoare, skema aslinya 321
5.8.6 Pemartisian Hoare, ditelusuri dua kali 322
5.8.7 Prosedur Quicksort itu sendiri 325
5.8.8 Quicksort, ditelusuri tuntas 326
5.9 Menganalisis Quicksort: Kasus Terbaik, Terburuk, dan Rata-Rata 328
5.9.1 Segalanya bergantung pada belahannya 328
5.9.2 Kasus terbaik: belahan yang seimbang sempurna 328
5.9.3 Kasus terburuk: belahan yang setakseimbang mungkin 329
5.9.4 Gagasan belahan yang seimbang, dan belahan 9: 1 332
5.9.5 Kasus rata-rata, dihujahkan dengan dua cara 333
5.9.6 Memeriksa teorinya terhadap sebuah mesin 337
5.10 Quicksort Teracak 339
5.10.1 Perbaikannya 339
5.10.2 Apa yang sesungguhnya dibeli oleh pengacakan 339
5.10.3 Strategi pivot yang lain 340
5.10.4 Rekursi ekor dan tumpukan panggilan 340
5.11 Batas Bawah $\Omega(n\lg n)$ bagi Pengurutan Berbasis Pembandingan 343
5.11.1 Apa yang tergolong pengurutan berbasis pembandingan 344
5.11.2 Pohon keputusan 344
5.11.3 Pembuktiannya 344
5.11.4 Apa yang dilarang dan tidak dilarang oleh batas itu 345
5.12 Memilih Pengurutan: Perbandingan yang Jujur 347
5.12.1 Lanskap lengkapnya 347
5.12.2 Kestabilan, dan mengapa ada yang peduli 348
5.12.3 Perilaku cache, dan mengapa Quicksort menang dalam praktik 350
5.12.4 Kapan memakai yang mana 354
5.13 Profitina dalam Praktik 355
5.13.1 Di mana bab ini muncul pada produk yang sungguhan 355
5.13.2 Studi kasus yang dikerjakan: Profitina Laundry 356
5.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 360
5.15 Ringkasan Bab 361
5.15.1 Menengok ke depan 363
5.16 Soal Latihan 363
5.17 Penyelesaian Soal Latihan 364
Daftar Pustaka 370

BAB 6 Pohon Pencarian Biner dan Pohon AVL 371
6.1 Dari Mana Kita Melanjutkan 371
6.2 Kosakata Pohon 372

6.3 Apa Itu Pohon Pencarian Biner 374
6.4 Sifat Pohon Pencarian Biner Bukan Sifat Heap 375
6.5 Merepresentasikan Pohon Pencarian Biner dalam C++ 376
6.6 Menelusuri Pohon: Tiga Penelusuran 377
6.6.1 Mengapa penelusuran inorder menghasilkan urutan terurut — sebuah bukti 381
6.7 Pencarian 382
6.8 Minimum dan Maksimum 383
6.9 Penerus dan Pendahulu 383
6.10 Penyisipan 387
6.11 Penghapusan: Tiga Kasusnya 389
6.11.1 Kasus 1 — simpulnya adalah simpul daun 389
6.11.2 Kasus 2 — simpulnya punya tepat satu anak 389
6.11.3 Kasus 3 — simpulnya punya dua anak 390
6.12 Pohon Pencarian Biner yang Lengkap dalam C++ 392
6.13 Ketika Pohonnya Rusak: h Bisa Mencapai n 395
6.14 Pohon AVL: Keseimbangan sebagai Invarian 398
6.15 Mengapa Pohon AVL Selalu Pendek: Rekurensi Simpul-Minimum 400
6.15.1 Hubungannya dengan Fibonacci 402
6.15.2 Dari Fibonacci ke $1.44 \lg n$ 403
6.16 Rotasi: Satu-satunya Perkakas 405
6.16.1 Rotasi kanan, lengkap dengan pembaruan penunjuk yang persis 406
6.16.2 Rotasi kiri 408
6.16.3 Apa yang dilakukan rotasi terhadap tinggi 409
6.17 Empat Kasus Rotasi 409
6.17.1 Kasus LL — satu rotasi kanan 410
6.17.2 Kasus LR — rotasi kiri lalu rotasi kanan 411
6.17.3 Kasus RR — satu rotasi kiri 413
6.17.4 Kasus RL — rotasi kanan lalu rotasi kiri 414
6.17.5 Membedakan keempat kasus di dalam kode 414
6.18 Sebuah Contoh Penyisipan, Selangkah demi Selangkah 415
6.18.1 Langkah 1 — sisipkan 14: tidak perlu penyeimbangan ulang 415
6.18.2 Langkah 2 — sisipkan 3: kasus Kiri-Kiri 416
6.18.3 Langkah 3 — sisipkan 45: kasus Kanan-Kanan 417
6.18.4 Langkah 4 — sisipkan 7: kasus Kiri-Kanan 418
6.18.5 Langkah 5 — sisipkan 15: kasus Kanan-Kiri 419
6.19 Pohon AVL yang Lengkap dalam C++ 420
6.19.1 Sebuah implementasi rujukan dalam Java 425
6.20 Penghapusan pada Pohon AVL 427
6.20.1 Mengapa y dan x dipilih dengan cara yang berbeda 428
6.20.2 Mengapa penyisipan paling banyak perlu satu penyeimbangan ulang, sedangkan penghapusan bisa perlu $\lg n$ kali 428
6.20.3 Sebuah contoh penghapusan, selangkah demi selangkah 432
6.21 Memilih: Pohon AVL, Pohon Merah-Hitam, atau Tabel Hash 434
6.21.1 Pohon merah-hitam 434
6.21.2 Tabel hash 435
6.22 Profitina dalam Praktik 439
6.22.1 Studi kasus yang dikerjakan tuntas: Profitina Laundry 440
6.23 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 445
6.24 Ringkasan Bab 448
6.24.1 Menengok ke depan 450
6.25 Soal Latihan 450
6.26 Penyelesaian Soal Latihan 451
Daftar Pustaka 460

BAB 7 Dari Divide and Conquer ke Dynamic Programming **461**
7.1 Dari Mana Kita Melanjutkan 461
7.1.1 Apa yang masih kurang 462
7.2 Divide and Conquer sebagai Disiplin Perancangan 462
7.2.1 Tiga langkahnya 462
7.2.2 Merge Sort, sekali lagi dilihat sebagai bentuk 463
7.2.3 Kapan divide and conquer adalah perkakas yang tepat 464
7.2.4 Kapan divide and conquer memboroskan kerja 464
7.3 Fibonacci: Persoalan Terkecil yang Mematahkan Divide and Conquer 464
7.3.1 Definisinya 464
7.3.2 Rekursi yang lurus-lurus saja 465
7.3.3 Pohon rekursinya 466
7.3.4 Menghitung panggilannya dengan tepat 466
7.3.5 Mengapa biayanya $\Theta(\phi n)$ 467
7.3.6 Berapa banyak kerja yang sesungguhnya terulang 468
7.3.7 Mengukurnya 469
7.4 Empat Perbaikan untuk Fibonacci 471
7.4.1 Perbaikan 1: memoization — ingat apa yang sudah pernah Anda jawab 472
7.4.2 Perbaikan 2: bottom-up — bangun jawaban kecil lebih dulu 473
7.4.3 Perbaikan 3: ruang $\Theta(1)$ 473
7.4.4 Perbaikan 4 (bonus): pangkat matriks, $O(\lg n)$ 474
7.4.5 Keempatnya, diukur 475
7.5 Persoalan Optimasi 479
7.5.1 Apa yang membuat sebuah persoalan menjadi persoalan optimasi 479
7.5.2 Optimal substructure, dinyatakan dengan tepat 480
7.5.3 Argumen potong-dan-tempel 480
7.5.4 Sebuah persoalan *tanpa* optimal substructure 481
7.6 Mengalikan Sederetan Matriks 481
7.6.1 Biaya satu hasil kali matriks 481
7.6.2 Keasosiatifan, dan mengapa pengelompokannya penting 482
7.6.3 Sebuah contoh empat matriks, dihitung harganya 483
7.6.4 Ada berapa banyak parenthesization? 484
7.6.5 Submasalahnya, dan rekurensi yang diturunkan 486
7.6.6 Menghitung submasalahnya 488
7.6.7 Algoritmanya 489
7.6.8 Tabel yang diisi dengan tangan, sel demi sel 490
7.6.9 Tabel c dan rekonstruksi solusinya 492
7.6.10 Waktu $\Theta(n3)$ dan ruang $\Theta(n2)$ 493
7.6.11 Program lengkapnya 494
7.7 Memoization: Mempertahankan Rekursinya 498
7.7.1 Lookup-Chain 498
7.7.2 Diukur: rekursi polos, ber-memoization, bottom-up 499
7.7.3 Imbal-tukarnya, dinyatakan 502
7.8 Empat Langkah Merancang Algoritma Dynamic Programming 502
7.8.1 Langkah 1: cirikan struktur solusi optimalnya 502
7.8.2 Langkah 2: definisikan nilai solusi optimalnya secara rekursif 503
7.8.3 Langkah 3 dan 4: hitung nilainya, lalu susun solusinya 504
7.8.4 Resepnya, dalam satu halaman 505
7.9 Longest Common Subsequence 505
7.9.1 Mengapa orang peduli 505
7.9.2 Subsequence, didefinisikan 505
7.9.3 Brute force, dan mengapa ia gagal 506
7.9.4 Optimal substructure, dibuktikan 507
7.9.5 Rekurensinya 508
7.9.6 Algoritmanya 509

7.9.7 Tabel yang diisi dengan tangan 510
7.9.8 Tabel panahnya 512
7.9.9 Merekonstruksi LCS-nya 512
7.9.10 $\Theta(mn)$, dan varian beruang $\Theta(\min(m,n))$ 513
7.9.11 Program lengkapnya 514
7.10 Submasalah yang Tumpang Tindih versus Submasalah yang Saling Bebas 517
7.10.1 Dua ciri khasnya 518
7.10.2 Diukur 518
7.10.3 Prosedur pengambilan keputusannya 520
7.11 Profitina dalam Praktik 521
7.11.1 Studi kasus yang dikerjakan: Profitina Laundry 522
7.12 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 525
7.13 Ringkasan Bab 527
7.13.1 Menatap ke depan 528
7.14 Soal Latihan 528
7.15 Penyelesaian Soal Latihan 530
Daftar Pustaka 538

BAB 8 Evaluasi dan Pembahasan II: Pengurutan, Pohon Pencarian, dan Pemrograman Dinamis 539
8.1 Di Mana Kita Berdiri 539
8.2 Cara Mendekati Soal Bertingkat 541
8.3 Soal 8.1 — Membangun *Max-Heap*, Langkah demi Langkah 543
8.3.1 Soalnya 543
8.3.2 Yang Anda perlukan dari Bab 5 543
8.3.3 Langkah 1: larik sebagai pohon 544
8.3.4 Langkah 2: pelacakannya, satu panggilan demi satu panggilan 544
8.3.5 Langkah 3: *heapsort* pada hasilnya 546
8.3.6 Langkah 4: mengapa pembangunannya hanya berbiaya $O(n)$ 546
8.3.7 Programnya 547
8.3.8 Periksa diri 550
8.3.9 Belokan keliru yang sering terjadi pada soal ini 551
8.4 Soal 8.2 — Partisi Quicksort, Dilacak; Kasus Terbaik dan Terburuk 551
8.4.1 Soalnya 551
8.4.2 Yang Anda perlukan dari Bab 5 552
8.4.3 Langkah 1: partisinya, satu langkah per baris 552
8.4.4 Langkah 2: seluruh rekursinya 553
8.4.5 Langkah 3: kasus terburuk 554
8.4.6 Langkah 4: kasus terbaik 554
8.4.7 Programnya 555
8.4.8 Periksa diri 559
8.4.9 Belokan keliru yang sering terjadi pada soal ini 560
8.5 Tipe Data yang Dipakai Soal-Soal Pohon 560
8.6 Soal 8.3 — `isBST`: Apakah Pohon Ini Pohon Pencarian Biner? 562
8.6.1 Soalnya 562
8.6.2 Tentang apa soal ini sebenarnya 563
8.6.3 Jawaban yang keliru, dan pohon terkecil yang menyingkapnya 563
8.6.4 Menurunkan jawaban yang benar 564
8.6.5 Programnya 565
8.6.6 Mengapa ini bukan sifat heap 569
8.6.7 Penyelesaian yang menggoda tetapi keliru, dan mengapa ia gagal 569
8.6.8 Cara kedua: penelusuran inorder 570
8.6.9 Belokan keliru yang sering terjadi pada soal ini 571
8.7 Soal 8.4 — `printDescending`: Dari yang Terbesar, Tanpa Daftar 571
8.7.1 Soalnya 571

8.7.2 Tentang apa soal ini sebenarnya 571
8.7.3 Menurunkan jawabannya 572
8.7.4 Programnya 573
8.7.5 Cara kedua: tumpukan eksplisit 575
8.7.6 Belokan keliru yang sering terjadi pada soal ini 575
8.8 Soal 8.5 — `max`: Kunci Terbesar Lewat Satu Lintasan 576
8.8.1 Soalnya 576
8.8.2 Tentang apa soal ini sebenarnya 576
8.8.3 Menurunkan jawabannya 576
8.8.4 Programnya 577
8.8.5 $O(h)$, $O(\lg n)$, dan cara menyatakan batasnya agar ia benar 580
8.8.6 Tanda $\leq$ pada petunjuknya 581
8.8.7 Pohon kosong 581
8.8.8 Belokan keliru yang sering terjadi pada soal ini 581
8.9 Soal 8.6 — `isHeightBalanced`: Syarat AVL dalam Satu Lintasan 582
8.9.1 Soalnya 582
8.9.2 Apa itu syarat AVL 582
8.9.3 Jawaban yang kentara, dan berapa biayanya 583
8.9.4 Jawabannya: satu lintasan, dengan penjaga 584
8.9.5 Programnya 585
8.9.6 Rekurensinya, diselesaikan dengan Teorema Master 589
8.9.7 Belokan keliru yang sering terjadi pada soal ini 590
8.10 Soal 8.7 — `insertHB`: Penyisipan yang Menjaga Pohon Tetap Seimbang 590
8.10.1 Soalnya 590
8.10.2 Kontraknya, dinyatakan dengan persis 591
8.10.3 Menurunkan `insertHB` 591
8.10.4 Sekilas pertama mesinnya bekerja 592
8.11 Soal 8.8 — `rebalanceForLeft`: Kasus LL dan LR 593
8.11.1 Soalnya 593
8.11.2 Satu-satunya perkakas: rotasi 593
8.11.3 Kasus LL — satu rotasi kanan 594
8.11.4 Kasus LR — satu rotasi kiri, lalu satu rotasi kanan 595
8.11.5 Titik paling halus: mengapa $\leq$ dan bukan $<$ 596
8.12 Soal 8.9 — `rebalanceForRight`: Kasus RR dan RL 596
8.12.1 Soalnya 596
8.12.2 Asas pencerminan 596
8.12.3 Kasus RR dan RL, dalam gambar 597
8.12.4 Belokan keliru yang sering terjadi pada Soal 8.8 dan 8.9 598
8.13 Soal 8.10 — `deleteHB`: Penghapusan yang Menjaga Pohon Tetap Seimbang 599
8.13.1 Soalnya 599
8.13.2 Ketiga kasus penghapusan 599
8.13.3 Jebakannya: penyeimbang ulang mana yang dipanggil 600
8.13.4 Mengapa penghapusan mungkin butuh banyak penyeimbangan ulang, dan penyisipan paling banyak satu 601
8.13.5 Pendahulu atau pengganti? 601
8.13.6 Soal 8.7 sampai 8.10 sebagai satu program 603
8.13.7 Belokan keliru yang sering terjadi pada soal ini 608
8.14 Soal 8.11 sampai 8.15 — Pohon Merah-Hitam 608
8.14.1 Soal-soalnya, dan hubungannya dengan soal-soal AVL 608
8.14.2 Kelima sifat merah-hitam 609
8.14.3 Soal 8.11 — mencacah simpul hitam 610
8.14.4 Soal 8.12 — tak ada simpul merah yang beranak merah 610
8.14.5 Soal 8.13 — tinggi-hitam yang seragam 610
8.14.6 Soal 8.14 — pemeriksa yang lengkap 610
8.14.7 Soal 8.15 — rotasi dengan penunjuk induk 611

8.14.8 Kelima soal sebagai satu program yang berjalan 611
8.15 Soal 8.16 — Perkalian Rantai Matriks, Tabelnya Diisi dengan Tangan 618
8.15.1 Soalnya 618
8.15.2 Yang Anda perlukan dari Bab 7 618
8.15.3 Langkah 1: dimensi dan diagonalnya 619
8.15.4 Langkah 2: rantai sepanjang 2 619
8.15.5 Langkah 3: rantai sepanjang 3 619
8.15.6 Langkah 4: rantai sepanjang 4 620
8.15.7 Langkah 5: seluruh rantai 620
8.15.8 Langkah 6: membaca pengurungannya, dan memeriksanya 621
8.15.9 Langkah 7: rekursi polos lawan *memoization* 621
8.15.10 Programnya 622
8.15.11 Periksa diri 626
8.15.12 Belokan keliru yang sering terjadi pada soal ini 626
8.16 Soal 8.17 — *Longest Common Subsequence*, Tabelnya dan Jalan Mundurnya 626
8.16.1 Soalnya 626
8.16.2 Yang Anda perlukan dari Bab 7 626
8.16.3 Langkah 1: tabelnya, satu sel per baris 627
8.16.4 Langkah 2: jalan mundurnya 629
8.16.5 Langkah 3: apakah SORE satu-satunya jawaban? 629
8.16.6 Langkah 4: biayanya 630
8.16.7 Programnya 630
8.16.8 Periksa diri 633
8.16.9 Belokan keliru yang sering terjadi pada soal ini 633
8.17 Dua Puluh Lima Pemeriksaan Cepat 633
8.18 Soal-Soal Lanjutan: Pohon Statistik-Urutan 635
8.18.1 Soal-soalnya 635
8.18.2 Bagaimana tiap penyelesaian diturunkan 636
8.18.3 Penyelesaian lengkap 637
8.18.4 Membaca keluarannya sebagai pemeriksaan 642
8.19 Profitina dalam Praktik 643
8.19.1 Studi kasus terkerjakan: Profitina Laundry 644
8.20 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 650
8.21 Ringkasan Bab 651
8.21.1 Menatap ke depan 652
8.22 Soal Latihan 652
8.23 Penyelesaian Soal Latihan 653
Daftar Pustaka 663

BAB 9 Algoritma Rakus 664
9.1 Dari Mana Kita Melanjutkan 664
9.1.1 Bagaimana jika pilihannya sudah gamblang? 665
9.2 Apa Itu Algoritma Rakus 665
9.2.1 Definisinya 665
9.2.2 Kenyataan yang tidak nyaman 666
9.2.3 Apa yang dikerjakan sisa bab ini 666
9.3 Kegagalan yang Memantik: Memberi Uang Kembalian 667
9.3.1 Persoalannya 667
9.3.2 Aturan rakusnya 667
9.3.3 Contoh sanggahannya 668
9.3.4 Sistem koin tempat kerakusan berhasil 668
9.3.5 Diukur 669
9.4 Persoalan Pemilihan Aktivitas 673
9.4.1 Persoalannya 673
9.4.2 Contoh berjalan 674

9.4.3 Empat calon aturan rakus 675
9.4.4 Substruktur optimal, dibuktikan 676
9.4.5 Program dinamis yang sebentar lagi kita buang 679
9.4.6 Pilihan rakusnya, dan mengapa hanya tersisa satu submasalah 679
9.4.7 Sifat pilihan rakus, dibuktikan dengan argumen penukaran 681
9.4.8 Menyatukan kedua paruhnya 682
9.4.9 Algoritma rakus rekursifnya 683
9.4.10 Penelusuran pada contoh berjalan 684
9.4.11 Algoritma rakus iteratifnya 685
9.4.12 Waktu jalannya 687
9.4.13 Program lengkapnya 688
9.5 Unsur-Unsur Strategi Rakus 692
9.5.1 Prosedur berlangkah enam 692
9.5.2 Bahan pertama: sifat pilihan rakus 693
9.5.3 Bahan kedua: substruktur optimal 694
9.5.4 Cara mengenali kedua bahannya: resep praktis 694
9.6 Rakus lawan Pemrograman Dinamis: Dua Masalah Ransel 695
9.6.1 Kedua persoalannya 695
9.6.2 Aturan rakusnya, dan contoh kasusnya 695
9.6.3 Pecahan: kerakusan berhasil 696
9.6.4 0-1: kerakusan gagal 697
9.6.5 Seberapa parah, dan seberapa sering? 698
9.6.6 Kedua teknik itu, bersebelahan 699
9.6.7 Program lengkapnya 699
9.7 Kode Huffman 704
9.7.1 Persoalannya 704
9.7.2 Contoh berjalan, dengan cacah bitnya dihitung 704
9.7.3 Kode bebas awalan, dan mengapa ia penting 706
9.7.4 Kode sebagai pohon biner 707
9.7.5 Biaya sebuah pohon 708
9.7.6 Algoritmanya 709
9.7.7 Penelusurannya, langkah demi langkah, pada contoh berjalan 710
9.7.8 Kebenaran, bagian satu: lema pilihan rakus 712
9.7.9 Kebenaran, bagian dua: lema substruktur optimal 713
9.7.10 Teoremanya 714
9.7.11 Waktu jalannya 715
9.7.12 Penyandi dan pengurai sandi lengkapnya 715
9.8 Matroid dan Metode Rakus 722
9.8.1 Pertanyaan yang dijawab bagian ini 722
9.8.2 Definisinya, dan apa arti tiap katanya 723
9.8.3 Contoh yang dikerjakan: matroid grafik 724
9.8.4 Dua akibat sifat penukaran 725
9.8.5 Matroid berbobot dan algoritma rakusnya 725
9.8.6 Penerapannya: menjadwalkan tugas berwaktu-satuan dengan tenggat dan denda 727
9.8.7 Dikerjakan dengan tangan pada sebuah contoh 729
9.8.8 Satu algoritma, dua persoalan 731
9.9 Katalog Algoritma Rakus yang Masih Akan Datang 736
9.10 Profitina dalam Praktik 737
9.10.1 Studi kasus yang dikerjakan: Profitina Laundry 738
9.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 745
9.12 Ringkasan Bab 747
9.12.1 Menengok ke depan 748
9.13 Soal Latihan 749
9.14 Penyelesaian Soal Latihan 750
Daftar Pustaka 762

BAB 10 Algoritma Graf Dasar: Penelusuran Melebar dan Penelusuran Mendalam 763
10.1 Dari Mana Kita Melanjutkan 763
10.1.1 Apa yang baru, dan apa yang terbawa dari sebelumnya 764
10.2 Apa Itu Graf 764
10.2.1 Mulailah dari sesuatu yang bisa Anda lihat 764
10.2.2 Menuliskannya 765
10.2.3 Definisi formalnya 765
10.2.4 Dua kelaziman kecil 766
10.3 Untuk Apa Graf Itu 767
10.4 Kosakata Graf 769
10.4.1 Bertetangga, bersisian, tetangga 769
10.4.2 Derajat 770
10.4.3 Lema jabat tangan, dibuktikan 770
10.4.4 Derajat masuk dan derajat keluar 771
10.4.5 Jalan, lintasan, dan lintasan sederhana 772
10.4.6 Siklus dan graf tak bersiklus 774
10.4.7 Graf terhubung 775
10.4.8 Komponen terhubung 775
10.4.9 Komponen terhubung kuat, untuk graf berarah 776
10.4.10 Subgraf dan subgraf merentang 777
10.4.11 Pohon dan hutan 777
10.4.12 Graf lengkap 779
10.4.13 Graf bipartit 779
10.4.14 Graf berbobot 780
10.4.15 Sebanyak apa sisinya bisa jadi? Jarang dan padat 780
10.4.16 Seluruh kosakatanya dalam satu halaman 781
10.5 Tiga Cara Menyimpan Graf 783
10.5.1 Apa yang harus dikerjakan sebuah representasi 783
10.5.2 Daftar sisi 784
10.5.3 Daftar ketetanggaan 785
10.5.4 Matriks ketetanggaan 786
10.5.5 Ruang, berdampingan 787
10.5.6 Keempat pertanyaan, berdampingan 788
10.5.7 Memilih 788
10.5.8 Ketiganya, dalam satu program 789
10.6 Menelusuri Sebuah Graf 792
10.7 Penelusuran Melebar 794
10.7.1 Labirin dan seutas benang 794
10.7.2 Tiga hal yang dihitung BFS 794
10.7.3 Algoritmanya 795
10.7.4 Contoh yang dikerjakan, selangkah demi selangkah 797
10.7.5 Membaca jawabannya 800
10.7.6 Programnya 801
10.7.7 Mengapa ongkosnya $O(V + E)$ 805
10.7.8 Pohon melebar 806
10.7.9 BFS menghitung jarak terpendek: pembuktiannya 806
10.8 Penelusuran Mendalam 810
10.8.1 Labirin, benang, dan sekaleng cat 810
10.8.2 Algoritmanya 811
10.8.3 Kedua pencap waktu 813
10.8.4 Contoh yang dikerjakan, selangkah demi selangkah 814
10.8.5 Hutan mendalam 816
10.8.6 Tanpa rekursi 817
10.8.7 Mengapa ongkosnya $\Theta(V + E)$ 823
10.8.8 Teorema tanda kurung 824

10.8.9 Teorema lintasan putih 828
10.8.10 Menggolongkan sisinya 829
10.9 Enam Hal yang Dapat Dikerjakan Kedua Penelusuran 831
10.9.1 Komponen terhubung 831
10.9.2 Deteksi siklus 833
10.9.3 Menguji kebipartitan 837
10.9.4 Lintasan terpendek pada graf tak berbobot 840
10.9.5 Graf berarah tak bersiklus 841
10.9.6 Pengurutan topologis 842
10.9.7 Jadi: melebar atau mendalam? 848
10.9.8 Diukur 848
10.10 Profitina dalam Praktik 851
10.10.1 Studi kasus yang dikerjakan: Profitina Laundry 853
10.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 859
10.12 Ringkasan Bab 860
10.12.1 Menatap ke depan 861
10.13 Soal Latihan 862
10.14 Penyelesaian Soal Latihan 863
Daftar Pustaka 871

BAB 11 Pohon Merentang Minimum, DAG, dan Pengurutan Topologis 872
11.1 Dari Mana Kita Melanjutkan 872
11.1.1 Yang masih kurang 873
11.2 Graf Berarah Asiklik 874
11.2.1 Siklus, dan apa artinya tidak punya siklus 874
11.2.2 Empat DAG sehari-hari 875
11.2.3 Teorema DAG 875
11.2.4 Membuktikan teorema DAG 876
11.2.5 Mengapa pilihan “simpul pertama” itu penting 877
11.3 Pengurutan Topologis 878
11.3.1 Apa yang kita inginkan 878
11.3.2 Mengapa urutan topologis (hampir) tidak pernah tunggal 878
11.3.3 Contoh berjalan: berpakaian 879
11.3.4 Algoritma 1: pencarian mendalam-dahulu, dan waktu penyelesaiannya 880
11.3.5 Contohnya ditelusuri, satu cap waktu demi satu cap waktu 880
11.3.6 Mengapa ia benar 882
11.3.7 Waktu jalannya $\Theta(V + E)$ 882
11.3.8 Program selengkapnya 883
11.3.9 Algoritma 2: algoritma derajat masuk dari Kahn 886
11.3.10 Kahn ditelusuri pada DAG yang sama 887
11.3.11 Mengapa Kahn benar 887
11.3.12 Waktu jalan Kahn 888
11.3.13 Program selengkapnya, dengan deteksi siklus 888
11.3.14 Yang mana dari keduanya yang dipakai 891
11.4 Pohon Merentang 891
11.4.1 Definisi 892
11.4.2 Pohon merentang punya tepat $V - 1$ sisi 892
11.4.3 Ada berapa banyak pohon merentang? 893
11.5 Persoalan Pohon Merentang Minimum 893
11.5.1 Bobot 893
11.5.2 Sebuah persoalan nyata beserta ongkosnya 894
11.5.3 Sebuah graf contoh 894
11.5.4 Mengapa pencarian paksa sia-sia, dan apa yang kita perlukan sebagai gantinya 895
11.6 Substruktur Optimal MST 895
11.6.1 Klaimnya 895

11.6.2 Buktinya 896
11.6.3 Yang tidak diberikan oleh substruktur optimal sendirian 897
11.7 Sifat Potongan 898
11.7.1 Potongan, sisi yang melintas, dan menghormati sebuah himpunan 898
11.7.2 Teoremanya 898
11.7.3 Buktinya, di dalam sebuah gambar 899
11.7.4 Membacanya sekali lagi, pelan-pelan 900
11.7.5 Akibat yang sungguh-sungguh dipakai algoritmanya 901
11.7.6 Sebuah pemeriksaan yang dikerjakan 901
11.8 Algoritma MST Generik 901
11.8.1 Sisi aman 901
11.8.2 Algoritma generiknya 901
11.8.3 Invarian gelungnya 902
11.8.4 Versi yang lebih khusus 902
11.8.5 Dua cara memilih potongannya 903
11.9 Prim–Jarník 903
11.9.1 Gagasannya: satu awan yang tumbuh 903
11.9.2 Kedua lariknya: key dan π 904
11.9.3 Algoritmanya 904
11.9.4 Antrean prioritas 905
11.9.5 Contohnya ditelusuri, langkah demi langkah 905
11.9.6 Mengapa Prim benar 909
11.9.7 Waktu jalannya 910
11.9.8 Tiga antrean dibandingkan 910
11.9.9 Program selengkapnya 911
11.9.10 Catatan tentang `DecreaseKey` yang malas 915
11.10 Kruskal 915
11.10.1 Gagasannya: sebuah hutan yang melebur 915
11.10.2 Himpunan saling lepas, dari nol 916
11.10.3 Percobaan 1: himpunan sebagai senarai 916
11.10.4 Percobaan 2: himpunan sebagai pohon 917
11.10.5 Akal-akalan 1: gabung menurut peringkat 917
11.10.6 Akal-akalan 2: pemampatan lintasan 918
11.10.7 Mengapa biaya teramortisasinya hampir konstan 919
11.10.8 Program himpunan saling lepasnya 920
11.10.9 Algoritmanya 923
11.10.10 Contohnya ditelusuri, sisi demi sisi 923
11.10.11 Mengapa Kruskal benar 929
11.10.12 Waktu jalannya 930
11.10.13 Program selengkapnya 931
11.11 Prim lawan Kruskal 933
11.11.1 Kedua jawabannya, dibandingkan 933
11.11.2 Jarang dan padat 933
11.11.3 Diukur 934
11.11.4 Tabel keputusannya 938
11.12 Bobot yang Sama, dan Kapan MST-nya Tunggal 939
11.12.1 Seri memecah algoritmanya, bukan jawabannya 939
11.12.2 Bobot yang berbeda-beda berakibat MST yang tunggal 939
11.12.3 Menguji ketunggalannya 940
11.13 Profitina dalam Praktik 940
11.13.1 Pipeline pemindaiannya adalah sebuah DAG, dan penjadwalnya mengurutkannya secara topologis 941
11.13.2 Pohon merentang minimum di dalam deteksi hampir-duplikat 941
11.13.3 Studi kasus yang dikerjakan: Profitina Laundry 941
11.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 947

11.15 Ringkasan Bab 948
11.15.1 Menengok ke depan 949
11.16 Soal Latihan 949
11.17 Penyelesaian Soal Latihan 950
Daftar Pustaka 955

BAB 12 Evaluasi dan Pembahasan III: Algoritma Greedy dan Graf 957
12.1 Posisi Kita Saat Ini 957
12.1.1 Apa yang telah dibangun Bab 9, 10, dan 11 957
12.1.2 Peta pengulangan 958
12.1.3 Susunan bab ini 960
12.2 Sebuah Proyek Perancangan, Dikerjakan dari Awal sampai Akhir 961
12.2.1 Apa itu proyek perancangan, dan kesembilan tahapnya 961
12.2.2 Memilih persoalan yang layak diberi algoritma 961
12.2.3 Tahap 1: persoalannya, dinyatakan 962
12.2.4 Tahap 2: modelnya 963
12.2.5 Tahap 3: algoritmanya dipilih, dan alternatif yang ditolak 964
12.2.6 Tahap 4: pseudokodenya 965
12.2.7 Tahap 5: kompleksitasnya, dihitung lebih dahulu 965
12.2.8 Tahap 6: rancangannya dibuktikan dengan tangan, sebelum ada kode 966
12.2.9 Tahap 7: implementasinya 968
12.2.10 Tahap 8: evaluasinya 975
12.2.11 Tahap 9: kesimpulannya 979
12.2.12 Lima jebakan proyek perancangan, dan satu uji diri 979
12.3 Persoalan Terkerjakan Lanjutan I: Algoritma Greedy 980
12.3.1 Susunan persoalan-persoalan terkerjakan 980
12.3.2 Persoalan Terkerjakan A — menukar uang: satu aturan rakus disanggah, satu lagi dibuktikan 982
12.3.3 Persoalan Terkerjakan B — pemilihan aktivitas, dijalankan dan dibuktikan 986
12.3.4 Persoalan Terkerjakan C — meminimumkan keterlambatan maksimum 990
12.3.5 Persoalan Terkerjakan D — kode Huffman, dibangun langkah demi langkah 992
12.3.6 Persoalan Terkerjakan E — kedua *knapsack* 996
12.3.7 Persoalan Terkerjakan F — matroid yang bekerja: tugas berwaktu satuan dengan tenggat 998
12.3.8 Persoalan B sampai E sebagai satu program 1004
12.4 Persoalan Terkerjakan Lanjutan II: Menelusuri Sebuah Graf 1011
12.4.1 Kedua graf subbab ini 1011
12.4.2 Persoalan Terkerjakan G — pencarian melebar, ditelusuri 1012
12.4.3 Persoalan Terkerjakan H — pencarian mendalam pada digraf, setiap busur digolongkan 1014
12.4.4 Persoalan Terkerjakan I — pencarian mendalam pada graf tak berarah 1019
12.4.5 Persoalan Terkerjakan J — komponen terhubung 1024
12.4.6 Persoalan Terkerjakan K — kebipartitan, dan sertifikatnya 1025
12.4.7 Persoalan Terkerjakan L — pendeteksian siklus, tak berarah dan berarah 1026
12.4.8 Persoalan G, H, J, K, dan L sebagai satu program 1027
12.5 Persoalan Terkerjakan Lanjutan III: DAG dan Pengurutan Topologis 1034
12.5.1 DAG prasyarat subbab ini 1034
12.5.2 Persoalan Terkerjakan M — apakah ia DAG? 1034
12.5.3 Persoalan Terkerjakan N — pengurutan topologis menurut waktu selesai 1035
12.5.4 Persoalan Terkerjakan O — pengurutan topologis dengan algoritma Kahn 1037
12.5.5 Persoalan Terkerjakan P — mencacah urutan topologis 1038
12.5.6 Persoalan Terkerjakan Q — lintasan kritis 1039
12.5.7 Persoalan M sampai Q sebagai satu program 1041
12.6 Persoalan Terkerjakan Lanjutan IV: Pohon Merentang Minimum 1046
12.6.1 Graf berbobot subbab ini 1046
12.6.2 Persoalan Terkerjakan R — Kruskal, dengan penelusuran himpunan saling lepas 1046

12.6.3 Persoalan Terkerjakan S — Prim, dengan larik kunci 1048
12.6.4 Persoalan Terkerjakan T — Prim, dengan tumpukan biner 1049
12.6.5 Persoalan Terkerjakan U — sifat potongan dan sifat siklus 1055
12.6.6 Persoalan Terkerjakan V — penyatuan menurut peringkat dan pemampatan lintasan 1056
12.6.7 Persoalan Terkerjakan W — apakah pohon merentang minimumnya tunggal? 1059
12.6.8 Persoalan R, S, U, V, dan W sebagai satu program 1060
12.7 Latihan Uji Diri Singkat 1067
12.8 Profitina dalam Praktik 1069
12.8.1 Studi kasus terkerjakan: Profitina Laundry 1071
12.9 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 1076
12.10 Ringkasan Bab 1077
12.10.1 Menatap ke depan 1078
12.11 Soal Latihan 1078
12.12 Penyelesaian Soal Latihan 1079
12.12.1 Program-programnya 1082
12.12.2 Bacaan lanjutan 1088
Daftar Pustaka 1089

BAB 13 Lintasan Terpendek Sumber Tunggal 1090
13.1 Dari Mana Kita Melanjutkan 1090
13.1.1 Apa yang terbawa dari Bab 10 dan 11 1090
13.1.2 Apa yang baru 1091
13.2 Persoalan Lintasan Terpendek, Dinyatakan dengan Cermat 1091
13.2.1 Dari “banyaknya sisi” ke “bobot total” 1091
13.2.2 Bobot sebuah lintasan 1092
13.2.3 Bobot lintasan terpendek 1092
13.2.4 Sebuah contoh yang dikerjakan dengan tangan 1093
13.2.5 Keempat ragam persoalannya 1093
13.2.6 Mengapa sumber tunggal yang kita selesaikan 1094
13.2.7 Di mana persoalan ini sungguh-sungguh muncul 1095
13.3 Substruktur Optimal Lintasan Terpendek 1095
13.3.1 Teoremanya 1095
13.3.2 Buktinya, dengan menggunting dan menempel 1096
13.3.3 Membaca buktinya sekali lagi, pelan-pelan 1096
13.3.4 Apa yang kita peroleh dari substruktur optimal 1097
13.4 Ketaksamaan Segitiga 1097
13.4.1 Pernyataannya 1097
13.4.2 Buktinya 1097
13.4.3 Bentuk sisinya, yang itulah yang kita pakai 1098
13.4.4 Sebuah pemeriksaan yang dikerjakan 1099
13.5 Bobot Negatif dan Siklus Negatif 1099
13.5.1 Sisi negatif bukan masalah 1099
13.5.2 Siklus negatif itu mematikan 1100
13.5.3 Lintasan terpendek tidak punya siklus 1101
13.5.4 Algoritma mana bertahan terhadap apa 1101
13.6 Relaksasi: Satu-satunya Primitif 1102
13.6.1 Kedua lariknya: d dan π 1102
13.6.2 Inisialisasi 1102
13.6.3 Langkah relaksasinya 1103
13.6.4 Sebuah relaksasi yang dikerjakan 1103
13.6.5 Sifat batas atas 1104
13.6.6 Sifat tanpa-lintasan 1104
13.6.7 Sifat kekonvergenan 1105
13.6.8 Sifat relaksasi lintasan 1105
13.6.9 Sifat subgraf pendahulu 1106

13.6.10 Kelima sifatnya dalam satu halaman 1107
13.7 Algoritma Dijkstra 1107
13.7.1 Gagasannya 1107
13.7.2 Pseudokodenya 1108
13.7.3 Mengapa bobotnya harus tak negatif 1109
13.7.4 Contohnya, selangkah demi selangkah 1112
13.7.5 Penelusuran yang sama dalam bentuk tabel 1116
13.7.6 Programnya 1116
13.7.7 Apa yang dicetaknya 1119
13.7.8 Kebenarannya 1119
13.7.9 Waktu jalannya 1122
13.7.10 Diukur 1123
13.7.11 Dijkstra, Prim, dan penelusuran melebar 1127
13.8 Algoritma Bellman–Ford 1127
13.8.1 Mengapa kita memerlukannya 1127
13.8.2 Algoritmanya 1127
13.8.3 Mengapa $V - 1$ sapuan sudah cukup 1128
13.8.4 Contohnya, sapuan demi sapuan 1130
13.8.5 Penelusuran yang sama dalam bentuk tabel 1133
13.8.6 Programnya 1133
13.8.7 Apa yang dicetaknya 1137
13.8.8 Pendeteksian siklus negatif, dan mengapa sapuan tambahannya menemukannya 1137
13.8.9 Waktu jalannya 1138
13.9 Lintasan Terpendek pada Graf Berarah Asiklik 1139
13.9.1 Mengapa DAG itu mudah 1139
13.9.2 Algoritmanya 1139
13.9.3 Mengapa tidak diperlukan antrean prioritas 1140
13.9.4 Sebuah contoh yang dikerjakan 1140
13.9.5 Programnya 1142
13.9.6 Waktu jalannya 1145
13.9.7 Lintasan terpanjang dan lintasan kritis 1145
13.10 Menyusun Ulang Lintasannya, Bukan Sekadar Panjangnya 1146
13.10.1 Larik pendahulu itulah jawabannya 1146
13.10.2 Programnya 1146
13.10.3 Tiga jebakan 1149
13.11 Algoritma Mana untuk Graf Mana 1149
13.12 Profitina dalam Praktik 1150
13.12.1 Di mana lintasan terpendek berada di dalam sebuah platform visibilitas AI 1150
13.12.2 Studi kasus yang dikerjakan: Profitina Laundry 1151
13.13 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 1155
13.14 Ringkasan Bab 1157
13.14.1 Menengok ke depan 1158
13.15 Soal Latihan 1158
13.16 Penyelesaian Soal Latihan 1159
Daftar Pustaka 1163

BAB 14 Analisis Amortisasi 1165
14.1 Dari Mana Kita Melanjutkan 1165
14.1.1 Sebuah utang dari Bab 11 1165
14.1.2 Apa yang perlu Anda bawa 1166
14.2 Pertanyaan yang Dijawab Analisis Amortisasi 1166
14.2.1 Sebuah batas yang benar, dan tidak berguna 1166
14.2.2 Faktor n itu, dibuat nyata 1167
14.2.3 Tiga biaya, dan satu-satunya pertidaksamaan yang penting 1168
14.2.4 Analisis amortisasi bukan analisis kasus rata-rata 1169

14.2.5 Ketiga metodenya 1170
14.3 Metode Agregat 1170
14.3.1 Apa itu metodenya 1170
14.3.2 Tumpukan dengan `Multipop` 1171
Batas naifnya, sekali lagi 1172
Pengamatan yang memperbaikinya 1172
Totalnya, dihitung 1173
Mengukurnya 1173
14.3.3 Pencacah biner k-bit 1176
Tabel 8-bit 1177
Seberapa sering bit i berbalik? 1178
Menjumlahkannya 1178
Mengukurnya 1179
14.3.4 Soal 17.1-3, dikerjakan tuntas 1182
Memahami fungsi biayanya 1183
Memecah jumlahnya 1183
Menghitungnya, satu baris demi satu baris 1184
Mengubahnya menjadi sebuah batas 1184
Memeriksanya dengan sebuah program 1185
14.3.5 Apa yang tidak dapat dilakukan metode agregat 1187
14.4 Amortisasi Bukan Kasus Rata-Rata — Dikatakan Lagi 1187
14.5 Metode Akuntansi 1188
14.5.1 Tagihan dan kredit 1188
14.5.2 Invariannya, dan mengapa itulah yang harus Anda buktikan 1189
14.5.3 Tumpukannya, dengan metode akuntansi 1190
14.5.4 Pencacah binernya, dengan metode akuntansi 1191
Mengamati kreditnya 1192
14.5.5 Soal 17.2-2, dikerjakan tuntas 1195
Tagihannya 1195
Kredit sesudah i operasi 1195
Kasus 1: i tepat sebuah pangkat dua, $i = 2k$ dengan $k \geq 0$ 1196
Kasus 2: i terletak tegas di antara dua pangkat, $i = 2k + j$ dengan $k \geq 0$ dan $0 \leq j < 2k$ 1196
Kesimpulannya 1196
Diperiksa dengan sebuah program 1197
14.6 Metode Potensial 1198
14.6.1 Gagasannya 1198
14.6.2 Jumlah teleskopiknya 1199
14.6.3 Satu syaratnya, dan mengapa ia diperlukan 1200
14.6.4 Tumpukannya, dengan metode potensial 1201
14.6.5 Pencacah binernya, dengan metode potensial 1202
Kedua potensialnya, diamati 1203
14.6.6 Soal 17.3-6: antrean dari dua tumpukan 1206
Persoalannya 1206
Rancangannya 1207
Mengapa ia benar 1207
Mengapa batas biaya naifnya buruk 1208
Fungsi potensialnya 1208
Programnya 1209
14.7 Tabel Dinamis 1212
14.7.1 Persoalannya 1212
14.7.2 `TABLE-INSERT` 1213
14.7.3 Metode agregat diterapkan pada `TABLE-INSERT` 1214
14.7.4 Metode akuntansi diterapkan pada `TABLE-INSERT` 1215
14.7.5 Metode potensial diterapkan pada `TABLE-INSERT` 1216
Gambarnya 1218

Programnya 1219
14.7.6 Penghapusan, dan godaan menyeparuhkan pada separuh 1221
Barisan *thrashing*-nya, dituliskan 1221
Perbaikannya 1222
Mengukur kedua kebijakannya 1222
14.7.7 Fungsi potensial untuk kasus gabungannya 1225
14.8 Union–Find: Janji dari Bab 11, Ditepati 1226
14.8.1 Apa yang sedang kita analisis 1226
14.8.2 Akal-akalan 1: gabung menurut peringkat 1227
14.8.3 Akal-akalan 2: pemampatan lintasan 1227
14.8.4 Apa jawabannya 1228
14.8.5 Apa sebenarnya α itu 1228
14.8.6 Mengapa $\alpha(n) \leq 4$ untuk sembarang n yang muat di alam semesta 1229
14.8.7 Mengukurnya 1230
14.9 Memilih Metode 1234
14.10 Profitina dalam Praktik 1235
14.10.1 Di mana amortisasi hidup di dalam sebuah platform visibilitas AI 1235
14.10.2 Studi kasus yang dikerjakan: Profitina Laundry 1236
14.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 1240
14.12 Ringkasan Bab 1241
14.12.1 Menengok ke depan 1242
14.13 Soal Latihan 1243
14.14 Penyelesaian Soal Latihan 1244
Daftar Pustaka 1260

BAB 15 Lintasan Terpendek Semua Pasangan 1262
15.1 Dari Mana Kita Melanjutkan 1262
15.1.1 Apa yang Bab 13 berikan kepada kita 1262
15.1.2 Pertanyaan yang baru 1264
15.2 Persoalan Semua Pasangan, Dengan Persis 1265
15.2.1 Masukannya: sebuah matriks ketetanggaan 1265
15.2.2 Keluarannya: dua matriks, bukan satu 1266
15.2.3 Mencetak sebuah lintasan dari Π 1267
15.2.4 Sebuah pemeriksaan kewarasan yang pertama atas ukurannya 1268
15.3 Algoritma yang Sudah Kentara, dan Harganya 1268
15.3.1 Semua bobot tak negatif: $V \times$ Dijkstra 1269
15.3.2 Ada sisi negatif: $V \times$ Bellman–Ford 1269
15.3.3 Mengapa kita tidak bisa begitu saja tetap memakai Dijkstra 1270
15.3.4 Apa yang sedang kita coba kalahkan 1273
15.4 Lintasan Terpendek dan Perkalian Matriks 1273
15.4.1 Resep pemrograman dinamis, diingat kembali 1273
15.4.2 Langkah 1: struktur sebuah lintasan terpendek 1274
15.4.3 Langkah 2: definisi rekursif $lij(m)$ 1275
15.4.4 Rekurensinya, suku demi suku 1275
15.4.5 Mengapa argumen pertamanya dapat dibuang 1276
15.4.6 Kapan barisannya berhenti berubah? 1277
15.4.7 Langkah 3: menghitung secara bawah-atas — EXTEND-SHORTEST-PATHS 1278
15.4.8 Keserupaan dengan perkalian matriks biasa 1279
15.4.9 Semiring (min, +), dan mengapa keserupaannya persis 1280
15.4.10 SLOW-ALL-PAIRS-SHORTEST-PATHS 1281
15.4.11 Contoh berjalan, setiap matriks antaranya 1282
15.4.12 Satu matriks dihitung sel demi sel 1285
15.4.13 Dari mana matriksnya berasal, dalam satu gambaran 1286
15.5 Memperbaiki Waktu Jalannya: Pemangkatan Berulang 1286
15.5.1 Kita hanya menginginkan matriks yang terakhir 1286

15.5.2 Asosiativitas itulah yang kita jual 1287
15.5.3 Barisan pangkat duanya 1287
15.5.4 Mengapa $\lceil \lg(n-1) \rceil$ hasil kali sudah cukup 1288
15.5.5 FASTER-ALL-PAIRS-SHORTEST-PATHS 1289
15.5.6 Contohnya lagi, dalam dua hasil kali 1289
15.5.7 Apa yang *tidak* diberikan pemangkatan berulang kepada kita 1292
15.6 Algoritma Floyd–Warshall 1292
15.6.1 Cara lain memotong sebuah lintasan 1292
15.6.2 Kedua kasusnya 1293
15.6.3 Mengapa kedua paruhnya dibatasi ke $1, \ldots, k-1$ 1294
15.6.4 Rekurensinya 1294
15.6.5 Algoritmanya 1296
15.6.6 Mengapa k harus menjadi gelung terluar 1296
15.6.7 Matriks pendahulunya 1300
15.6.8 Contoh berjalan, setiap $D(k)$ dan $\Pi(k)$ 1300
15.6.9 Putaran $k = 1$, sel demi sel 1306
15.6.10 Putaran $k = 4$, tempat sebagian besar kerjanya terjadi 1307
15.6.11 Membaca rute dari Π 1307
15.6.12 Waktu jalannya 1308
15.6.13 Mengerjakannya di tempat, dan mengapa itu aman 1308
15.6.14 Mendeteksi siklus berbobot negatif tanpa biaya 1309
15.6.15 Keempat metodenya sepakat 1311
15.7 Penutupan Transitif Sebuah Graf Berarah 1315
15.7.1 Pertanyaannya, dan jawaban yang malas 1315
15.7.2 Jawaban yang lebih baik: tukarlah operatornya 1315
15.7.3 Algoritmanya dan contohnya 1316
15.7.4 Mengapa boolean jauh lebih cepat dalam praktik 1319
15.8 Algoritma Johnson untuk Graf Jarang 1321
15.8.1 Yang diinginkan 1321
15.8.2 Apa yang harus dicapai sebuah pembobotan ulang 1321
15.8.3 Pembobotan ulang yang berhasil 1322
15.8.4 Memilih h: simpul bantunya 1323
15.8.5 Algoritmanya 1324
15.8.6 Sebuah pembobotan ulang kecil yang dikerjakan tuntas 1325
15.8.7 Contoh berjalan bersimpul lima, dibobot ulang 1326
15.9 Algoritma yang Mana, dan Kapan 1330
15.9.1 Keempat ongkosnya berdampingan 1330
15.9.2 Titik silangnya, diukur 1331
15.9.3 Sebuah prosedur pengambilan keputusan 1334
15.10 Profitina dalam Praktik 1334
15.10.1 Di mana jarak semua pasangan muncul di sebuah platform GEO 1334
15.10.2 Studi kasus yang dikerjakan: Profitina Laundry 1335
15.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 1340
15.12 Ringkasan Bab 1341
15.12.1 Menatap ke depan 1342
15.13 Soal Latihan 1342
15.14 Penyelesaian Soal Latihan 1344
Daftar Pustaka 1351

BAB 16 Evaluasi dan Pembahasan IV: Lintasan Terpendek, Analisis Amortisasi, dan Keseluruhan Buku 1352
16.1 Posisi Kita Saat Ini 1352
16.1.1 Apa yang diberikan Bab 13 sampai 15 1352
16.1.2 Seluruh buku pada satu halaman 1353
16.1.3 Bagaimana bab ini disusun 1354

16.2 Cara Menghadapi Persoalan Berbagian dan Bukti ... 1355
16.2.1 Tulislah jawaban dalam urutan yang tetap ... 1355
16.2.2 Ketika sebuah bukti tidak kunjung selesai ... 1355
16.2.3 Lima kebiasaan yang layak dipelihara ... 1356
16.3 Persoalan 16.1 — Lima Belas Pernyataan, Benar atau Salah ... 1356
16.3.1 Persoalannya ... 1356
16.3.2 Apa yang diuji ... 1357
16.3.3 Jawabannya ... 1357
16.3.4 Pernyataan (1) — yang membutuhkan satu kalimat syarat ... 1358
16.3.5 Pernyataan (3), (4), dan (7) — keterurutan dan algoritma yang memakainya ... 1359
16.3.6 Pernyataan (5) — kapan tidak perlu mengurutkan ... 1359
16.3.7 Pernyataan (8) — jebakan dengan dua separuh ... 1360
16.3.8 Pernyataan (9), (10), dan (12) — yang dijamin berhadapan dengan yang lazim ... 1360
16.3.9 Pernyataan (6) — mengapa bobot yang berbeda semua memaksa MST tunggal ... 1361
16.3.10 Pernyataan (11), (14), dan (15) — batas, cacahan, dan definisi ... 1361
16.3.11 Pernyataan (13) — yang dapat Anda sanggah dalam lima detik ... 1362
16.3.12 Kesalahan yang paling sering terjadi pada persoalan ini ... 1364
16.4 Persoalan 16.2 — Subbarisan Menaik Monoton Terpanjang ... 1364
16.4.1 Persoalannya ... 1364
16.4.2 Apa yang diuji ... 1364
16.4.3 Asumsi, dinyatakan dengan benar ... 1365
16.4.4 Menurunkan rekurensinya ... 1365
16.4.5 Pseudokodenya, dibaca baris demi baris ... 1366
16.4.6 Sebuah contoh terkerjakan yang lengkap, dengan tangan ... 1367
16.4.7 Kompleksitas waktunya, dihitung dengan jujur ... 1368
16.4.8 Programnya ... 1368
16.4.9 Cara kedua: $O(n\lg n)$ ... 1369
16.4.10 Kesalahan yang paling sering terjadi pada persoalan ini ... 1371
16.5 Persoalan 16.3 — Pohon Biner Diperluas, Dibuktikan dengan Induksi ... 1372
16.5.1 Persoalannya ... 1372
16.5.2 Apa yang diuji ... 1372
16.5.3 Buktinya ... 1373
16.5.4 Tulisan yang menggoda tetapi ceroboh, dan mengapa ia merugi ... 1373
16.5.5 Dua bukti alternatif ... 1374
16.5.6 Percobaannya ... 1374
16.5.7 Mengapa hasil ini penting ... 1376
16.5.8 Kesalahan yang paling sering terjadi pada persoalan ini ... 1376
16.6 Persoalan 16.4 — Hampiran-2 untuk Persoalan Pedagang Keliling ... 1376
16.6.1 Persoalannya ... 1376
16.6.2 Apa yang diuji, dan kosakata yang Anda butuhkan dari nol ... 1377
16.6.3 Bagian 1 — algoritmanya menghasilkan sebuah tur ... 1378
16.6.4 Bagian 2 — waktu jalannya polinomial ... 1378
16.6.5 Bagian 3 — batas rasionya 2 ... 1379
16.6.6 Apa yang terjadi tanpa ketaksamaan segitiga ... 1380
16.6.7 Sebuah contoh terkerjakan yang lengkap ... 1380
16.6.8 Programnya ... 1382
16.6.9 Seberapa ketat angka 2? ... 1384
16.6.10 Kesalahan yang paling sering terjadi pada persoalan ini ... 1385
16.7 Persoalan 16.5 — Bellman–Ford Berhadapan dengan Dijkstra ... 1385
16.7.1 Persoalannya ... 1385
16.7.2 Apa yang diuji ... 1386
16.7.3 Bagian 1 — perbedaannya, dan penerapannya ... 1386
16.7.4 Bagian 2 — kedua pseudokodenya ... 1387
16.7.5 Bagian 3 — ilustrasinya ... 1389
16.7.6 Programnya ... 1391

16.7.7 Memperlihatkan *mengapa* Dijkstra gagal, bukan sekadar *bahwa* ia gagal 1393
16.7.8 Kesalahan yang paling sering terjadi pada persoalan ini 1395
16.8 Persoalan Terkerjakan tentang Lintasan Terpendek Sumber Tunggal 1396
16.8.1 Persoalan 16.6 — Dijkstra, dengan seluruh tabel sesudah setiap iterasi 1396
16.8.2 Persoalan 16.7 — Bellman–Ford, dan sebuah siklus negatif ditemukan lalu diperlihatkan 1399
16.8.3 Persoalan 16.8 — lintasan terpendek pada graf berarah taksiklik 1404
16.9 Persoalan Terkerjakan tentang Analisis Amortisasi 1407
16.9.1 Persoalan 16.9 — pencacah biner, dengan tiga cara dan dengan induksi 1408
16.9.2 Persoalan 16.10 — tabel dinamis, dengan tiga cara, dan sebuah potensial keliru yang menggoda 1413
16.10 Persoalan Terkerjakan tentang Lintasan Terpendek Semua Pasangan 1417
16.10.1 Persoalan 16.11 — Floyd–Warshall, setiap matriks dihitung sel demi sel 1418
16.10.2 Persoalan 16.12 — pembobotan ulang Johnson, dikerjakan selengkapnya 1422
16.11 Persoalan Terkerjakan Lanjutan 1426
16.11.1 Persoalan 16.13 — lima belas pernyataan lagi 1426
16.11.2 Persoalan 16.14 — lintasan terpanjang pada DAG 1427
16.11.3 Persoalan 16.15 — pohon dengan n simpul mempunyai $n - 1$ sisi 1430
16.11.4 Persoalan 16.16 — pengodean Huffman, dalam tiga bagian 1431
16.11.5 Persoalan 16.17 — Floyd–Warshall berhadapan dengan Johnson 1432
16.11.6 Persoalan 16.18 — lima belas pernyataan sekali lagi 1435
16.11.7 Persoalan 16.19 — *knapsack* 0/1, dan arti "polinomial" 1436
16.11.8 Persoalan 16.20 — hampiran-2 untuk penutup simpul 1439
16.11.9 Persoalan 16.21 — Prim berhadapan dengan Kruskal 1441
16.12 Persoalan-Persoalan Bab Ini Dibandingkan 1442
16.13 Latihan Uji Diri Singkat 1444
16.14 Tabel Pemilihan Algoritma 1447
16.15 Ke Mana Melangkah Selanjutnya 1448
16.16 Profitina dalam Praktik 1450
16.16.1 Studi kasus terkerjakan: Profitina Laundry 1451
16.17 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 1455
16.18 Ringkasan Bab, dan Sepatah Kata Penutup 1456
Sepatah kata penutup 1457
16.19 Soal Latihan 1458
16.20 Penyelesaian Soal Latihan 1459
Daftar Pustaka 1463

Daftar Gambar

Gambar 1.1 Sasaran perancangan struktur data dan algoritma (kebenaran, efisiensi) disandingkan dengan sasaran implementasi (ketangguhan, keluwesan, kegunaan ulang). 4
Gambar 1.2 Persoalan pengurutan: sebuah barisan masuk, sebuah permutasi dari barisan itu keluar, dan kebenaran berarti dua syarat terpisah harus sama-sama terpenuhi. 6
Gambar 1.3 *Columns*, soal E-Olymp nomor 10281, dirumuskan ulang dari jurinya: rumusan soalnya, gambar empat kolom kubus, dan bentuk masukannya. 18
Gambar 1.4 Contoh uji *Columns* dan satu pengecatan sah dengan enam warna. Perhatikan baris ketiga dari bawah: baris itu memuat warna 3 dua kali, dan itu sah karena ada ruang kosong yang memisahkan kedua kubusnya. 20
Gambar 1.5 *Journey*, soal E-Olymp nomor 10280. Cossack Vus harus menyinggahi setiap kota di Potokolandia lalu pulang kembali. 23
Gambar 1.6 Dua contoh uji *Journey*. Dua kota di 1 dan 4 memberi 6; tiga kota di 1,3,1 memberi 4. 24
Gambar 1.7 *Unique Numbers*, soal SPOJ DCEPC901, dengan dua jenis operasinya dan catatan penting bahwa operasinya beruntun saling menyusul. 27
Gambar 1.8 Batasan dan contoh uji *Unique Numbers*: n sampai 108, paling banyak 20 operasi. 28
Gambar 1.9 Perkalian modulo 15. Setiap pasangan baris menunjukkan masukan $0..14$ di atas dan keluaran $(i \times x)$ mod 15 di bawah, untuk $x = 3,5,2,4$ berturut-turut. 29
Gambar 1.10 *Minimum and maximum numbers*, soal SPOJ NEWLANG, dengan batasan dan contoh ujinya. 31
Gambar 1.11 Nilai minimum (atas, berarsir kuning) dan maksimum (bawah, berarsir kelabu) untuk $N = 1..10$ dan $K = 1..5$. Setiap baris adalah satu nilai K; bacalah melintang untuk N yang membesar. 34
Gambar 1.12 *Interesting sequence*, soal E-Olymp nomor 4894. Barisannya dibangun dengan berulang kali menempelkan salinan dirinya sendiri yang sudah digeser. 36
Gambar 1.13 Contoh uji *Interesting sequence*: tiga pertanyaan 17, 239, 385, dengan jawaban 1, 0, 2. 37
Gambar 1.14 Indeks 0 sampai 11 dalam biner, dengan banyaknya bit 1 dan cacah itu modulo 3. Kolom terakhir adalah barisannya sendiri. 38
Gambar 1.15 *Recursive function 1*, soal E-Olymp nomor 10296. Definisinya, masukannya yang berupa satu bilangan bulat, dan contoh uji $f(5) = 5$. 41
Gambar 1.16 Soal SPOJ PERIOD5, *Periodic function, trip 5*: diberi cuplikan sebuah fungsi periodik yang bersih, ekstrapolasikan fungsi itu. Grafik kecilnya memperlihatkan fungsi semacam itu, tersusun dari komponen berperiode 3 dan 5. 48
Gambar 1.17 Sekilas makalah [9]: judul, nama penyusun, wadah terbit (*Journal of Theoretical and Applied Information Technology*, vol. 97, no. 8, 2019), dan sumbangannya. 48
Gambar 1.18 Waktu jalan rata-rata tiga pendekatan ketika N membesar, digambar ulang dari hasil pengukuran pada Gambar 13 di [9]. Metode $N \times N$ yang dioptimalkan tetap mendatar; yang lain menanjak. 49
Gambar 1.19 Peringkat SPOJ untuk PERIOD5 sebagaimana tercantum pada awal 2026. Dari empat ratus dua puluh empat pengiriman hanya dua ratus tiga puluh lima yang diterima, dan entri ITS duduk di peringkat 2 dengan waktu 0,59 detik dalam 4,9 MB. 49
Gambar 1.20 Soal SPOJ HIR, *Hard Image Recognition*: pulihkan bilangan enam angka dari sebuah citra raster hitam-putih, dengan batas 0,1 detik dan batas kode sumber 150 kB. 50
Gambar 1.21 Sepuluh bilangan enam angka yang digambar dengan fon berbeda-beda, sebagian miring, buram, tercoret, berbintik, atau terbalik warnanya — keragaman seperti yang tampak pada gambar contoh soal HIR. 50
Gambar 1.22 Contoh uji HIR. Kedua gambarnya menyandikan 000482 dan 567833; pengiriman yang ditampilkan memperoleh nilai 1 karena jawaban keduanya punya satu angka yang keliru. 51
Gambar 1.23 Sekilas makalah [10]: judul, nama penyusun, wadah terbit (*Journal of Theoretical and Applied Information Technology*, vol. 98, no. 17, 2020), dan sumbangannya. 52

Gambar 1.24 Peringkat SPOJ untuk HIR sebagaimana tercantum pada awal 2026. Dua puluh dua pengguna diterima dari enam ratus enam puluh pengiriman; dua entri teratas, dengan nilai 108, berasal dari kelompok ITS. 52
Gambar 2.1 Muhammad ibn Musa al-Khwarizmi (sekitar 780–850 M), yang nama Latinnya, *Algorismus*, menjadi kata *algoritma*. 69
Gambar 2.2 Charles Babbage (1791–1871), perancang Analytical Engine, mesin hitung serbaguna pertama yang pernah digagas orang. 70
Gambar 2.3 Ada Lovelace (1815–1852), yang menyusun algoritma pertama yang dimaksudkan untuk dijalankan oleh sebuah mesin. 70
Gambar 2.4 Alan Turing (1912–1954), yang memformalkan komputasi lewat mesin Turing pada tahun 1936. 71
Gambar 2.5 Alonzo Church (1903–1995), penemu kalkulus lambda dan penyusun bersama tesis Church–Turing. 71
Gambar 2.6 John von Neumann (1903–1957), yang arsitektur program-tersimpannya mendasari hampir setiap komputer yang dipakai orang hari ini. 71
Gambar 2.7 Algoritma adalah sebuah gagasan; program adalah jadinya gagasan itu pada sebuah mesin tertentu, dalam sebuah bahasa tertentu. 74
Gambar 2.8 Kelima tujuan itu, dalam dua kelompok. Kebenaran dan efisiensi adalah tujuan perancangan dan menjadi pokok bahasan buku ini; kekukuhan, keteradaptasian, dan keterpakaian ulang adalah tujuan implementasi yang termasuk wilayah rekayasa perangkat lunak. 75
Gambar 2.9 Problem pengurutan. Keluaran $b1, \ldots, bn$ harus sekaligus *terurut* ($b1 \leq b2 \leq \cdots \leq bn$) dan merupakan *permutasi* dari masukannya — dua syarat, dan algoritma yang hanya memenuhi salah satunya adalah algoritma yang salah. Waktu jalannya bergantung pada ada berapa elemen, seberapa hampir-terurut elemen itu sudah, dan algoritma mana yang kita pilih. 80
Gambar 2.10 Cara pemain kartu. Tangan kiri memegang deretan terurut 2–4–5–10; tangan kanan baru saja mengambil kartu 7 dan akan menggesernya ke kiri melewati kartu 10 untuk mencapai tempatnya. 81
Gambar 2.11 Untuk suatu ukuran masukan terdapat banyak kemungkinan masukan dan karena itu banyak pula kemungkinan waktu jalan. Kasus terburuk adalah selubung atasnya, kasus terbaik selubung bawahnya, dan kasus rata-rata terletak di antara keduanya. 91
Gambar 2.12 Pertumbuhan $\lg n$, n, n, $n\lg n$, $100n$, $n2$, dan $n3$ dengan kedua sumbu logaritmik. Pada peta semacam itu fungsi nk menjadi garis lurus yang kemiringannya k, sehingga makin curam berarti makin buruk. 101
Gambar 2.13 Laju pertumbuhan yang sama dengan $2n$ ditambahkan. Sumbu tegaknya kini merentang sampai sekitar 10308; setiap kurva polinomial terhimpit ke dasar bagan, tak terbedakan dari sumbunya. 102
Gambar 2.14 Problem pencarian. Pada instans $A = 2,5,4,10,7$ dengan $q = 5$ jawabannya adalah indeks 2; dengan $q = 9$ jawabannya NIL, sebab 9 tidak muncul. 104
Gambar 2.15 Problem pencarian terurut. Satu kata yang ditambahkan — *terurut* — pada spesifikasi masukan itulah yang memungkinkan percepatan sebesar eksponensial. 107
Gambar 2.16 Kelima tujuan itu sekali lagi. Segala isi bab ini mengabdi pada dua yang di sebelah kiri — kebenaran, yang dibuktikan dengan invarian gelung, dan efisiensi, yang diturunkan dengan menghitung. 122
Gambar 3.1 Kebenaran parsial dibandingkan kebenaran total. Pada diagram atas, jaminannya bersyarat — **JIKA** kendali sampai ke kotak keluaran, **MAKA** keluarannya adalah yang dikehendaki. Pada diagram bawah, jaminannya tanpa syarat — kendali **SUNGGUH** sampai ke kotak keluaran, **DAN** keluarannya adalah yang dikehendaki. 132
Gambar 3.2 Big-O sebagai batas atas. Di luar ambang $n0$, waktu jalan $f(n)$ tidak pernah naik melampaui $c \cdot g(n)$. Apa yang terjadi di sebelah kiri $n0$ tidak relevan bagi klaim ini. 136
Gambar 3.3 Big-Ω sebagai batas bawah. Di luar $n0$, waktu jalan $f(n)$ tidak pernah jatuh di bawah $c \cdot g(n)$. 140
Gambar 3.4 Big-Θ sebagai batas ketat. Setelah $n0$, fungsi $f(n)$ terkurung di lorong antara $c1 \cdot g(n)$ dan $c2 \cdot g(n)$, sehingga f dan g tumbuh dengan laju yang sama. 141
Gambar 3.5 Deret aritmetika dan bentuk tertutupnya, $i = 0ni = 1 + 2 + 3 + \cdots + n = n2 + n2$. 152

Gambar 3.6 Pemasangan ala Gauss dibuat geometris. Memasangkan batang pertama dengan yang terakhir, yang kedua dengan yang kedua dari belakang, dan seterusnya, mengemas ulang tangga itu menjadi persegi panjang selebar $n/2$ dan setinggi $n + 1$, yang luasnya $n(n + 1)/2$.....153
Gambar 3.7 Deret aritmetika sebagai luas. Batang-batang setinggi 1,2, … , n duduk mengangkangi garis bergradien 1; luas totalnya $n(n + 1)/2$ adalah segitiga $n2/2$ ditambah n setengah-persegi yang menjorok melewatinya.....154
Gambar 3.8 Langkah induktif ditulis lengkap: andaikan $S(k) = k(k + 1)/2$ untuk $1 \leq k \leq n - 1$, lalu bentangkan $S(n) = S(n - 1) + n$ dan sederhanakan menjadi $n(n + 1)/2$. 155
Gambar 3.9 Deret geometri berhingga, $i = 0nai = 1 + a + a2 + \cdots + an = 1 - an + 11 - a$. 155
Gambar 3.10 Rekurensi Merge Sort sebagai definisi dua kasus: $T(n) = \Theta(1)$ ketika $n = 1$, dan $T(n) = 2T(n/2) + \Theta(n)$ ketika $n > 1$.....160
Gambar 3.11 Tiga metode untuk menyelesaikan rekurensi: substitusi (tebak, lalu sahkan dengan induksi), iterasi atau pohon rekursi (bentangkan rekurensinya dan gambar), dan metode master (sebuah cetakan yang menjawab sekelas rekurensi sekaligus). 162
Gambar 3.12 Metode substitusi yang gagal. Mengandaikan $T(k) \leq ck2$ menghasilkan $T(n) \leq cn2 + n$, dan $cn2 + n \leq cn2$ tidak berlaku untuk pilihan $c > 0$ **mana pun**. 163
Gambar 3.13 Hipotesis yang diperkuat $T(k) \leq c1k2 - c2k$ menutup induksinya: tambahan $-c2n$ menyerap sisa $+n$ kapan pun $c2 \geq 1$.....164
Gambar 3.14 Mengiterasi $T(n) = n + 3T(\lfloor n/4 \rfloor)$ sebanyak tiga kali, mengumpulkan deret geometri $n\sum(3/4)i$, dan menyimpulkan $T(n) = O(n)$.....168
Gambar 3.15 Membangun pohon rekursi untuk $T(n) = T(n/4) + T(n/2) + n2$. Tiap pembentangan mengganti sebuah daun $T(\cdot)$ dengan sebuah simpul yang membawa biaya tak rekursifnya sendiri beserta dua anak baru.....169
Gambar 3.16 Pohon rekursi yang sudah lengkap untuk $T(n) = T(n/4) + T(n/2) + n2$. Total tiap tingkatnya $n2$,$516n2$,$25256n2$, … membentuk deret geometri berasio $5/16 < 1$, yang berjumlah $\Theta(n2)$.....170
Gambar 3.17 Pohon rekursi untuk $T(n) = T(n/3) + T(2n/3) + n$. Tiap tingkat berjumlah tepat n, dan lintasan akar-ke-daun yang terpanjang memiliki $\log 3/2n$ tingkat, sehingga totalnya $O(n\lg n)$. 171
Gambar 4.1 Himpunan persoalan SPOJ (*tutorial*) 15861, *LDCSUMAB — BRVT - Math Team Collection 2013*, dirumuskan ulang dari *judge*-nya: pernyataan, format masukan dan keluaran, serta uji contohnya. 209
Gambar 4.2 Sepuluh suku pertama barisan LDCSUMAB (kiri), dirangkum menjadi peta dari k ke n (tengah), dan pembalikannya $k = n(n + 1)2, n2 + n - 2k = 0, n = -1 + 1 + 8k2$ (kanan).....211
Gambar 4.3 Sebuah penyelesaian LDCSUMAB yang ringkas di samping tabel yang diimplementasikannya: `n = ceil((-1+sqrt(1+8k))/2)`, `r = n(n+1)/2`, `m = n + k - (r+1)`.....215
Gambar 4.4 Sebuah program TOHU alternatif dengan cabang terpisah `if (n==1) printf("%.11lf\n", 2.0/3.0);` dan cabang umum yang menghitung $112 - 12(n + 1)(n + 2) + 23$. 225
Gambar 5.1 Tiga rumus navigasi, pohon yang mereka perikan, dan larik yang memuatnya. Pita-pita warna mengelompokkan simpul menurut kedalamannya dan menandai potongan larik yang bersesuaian di bawahnya; baris berlabel "Level" dihitung dari bawah ke atas, sehingga pita akar bernomor 3 dan baris terbawah bernomor 0.....278
Gambar 5.2 Sepuluh bilangan yang sama dilihat dua kali. Di kiri, pohon biner hampir lengkapnya, dengan indeks larik tiap simpul tercetak di atasnya. Di kanan, larik $A = [16,14,10,8,7,9,3,2,4,1]$, dengan sebuah busur digambar dari tiap simpul ke masing-masing anaknya — ke anak kiri di atas larik, ke anak kanan di bawahnya. Busur-busur itulah yang dihitung oleh aritmetika indeks; tidak satu pun busur semacam itu pernah disimpan.....281
Gambar 5.3 `MAX-HEAPIFY(A, 2)`, langkah (a). Pemanggilannya dimulai. Simpul $i = 2$, yang diarsir, memuat nilai 4. Anak-anaknya adalah $A[4] = 14$ dan $A[5] = 7$. Kedua subpohon di bawahnya sudah merupakan *max-heap* yang sah; hanya simpul 2 yang salah tempat.....287
Gambar 5.4 Langkah (b). Sesudah penukaran pertama. Nilai 14 sudah mengambil tempat 4 di indeks 2, dan si pengganggu 4 kini berada di indeks $i = 4$, yang diarsir. Anak-anaknya adalah $A[8] = 2$ dan $A[9] = 8$ — dan $8 > 4$, jadi penenggelamannya belum selesai.....288

Gambar 5.5 Langkah (c). Perbaikannya rampung. Nilai 4 sudah berhenti di indeks 9, sebuah daun, dan seluruh pohonnya sekali lagi menjadi *max-heap*. Nilai itu menempuh lintasan $2 \rightarrow 4 \rightarrow 9$, yang merupakan satu lintasan akar-ke-daun dan tidak lebih. 289
Gambar 5.6 Masukan bagi `BUILD-MAX-HEAP`: larik $A = [4,1,3,2,16,9,10,14,8,7]$, tanpa urutan tertentu. 294
Gambar 5.7 `BUILD-MAX-HEAP`, langkah (a): $i = 5$**.** Lariknya digambar sebagai pohon hampir lengkap. Simpul 5, yang diarsir, memuat 16. Satu-satunya anaknya adalah $A[10] = 7$. Gelungnya mulai di sini karena simpul 5 adalah simpul terakhir yang masih punya anak sama sekali. 295
Gambar 5.8 Langkah (b): $i = 4$**.** Simpul 4, yang diarsir, memuat 2, dan kedua anaknya $A[8] = 14$ dan $A[9] = 8$ sama-sama lebih besar. Nilai 2 harus tenggelam. 295
Gambar 5.9 Langkah (c): $i = 3$**.** Simpul 3, yang diarsir, memuat 3; anak-anaknya adalah $A[6] = 9$ dan $A[7] = 10$. Anak kanannya yang lebih besar, jadi penukarannya ke kanan. 296
Gambar 5.10 Langkah (d): $i = 2$**.** Simpul 2, yang diarsir, memuat 1, nilai terkecil dalam lariknya. Ia harus tenggelam jauh: dua tingkat, ternyata. 296
Gambar 5.11 Langkah (e): $i = 1$**.** Iterasi terakhir, di akarnya. Simpul 1, yang diarsir, memuat 4. Segala yang di bawahnya sudah merupakan *heap* yang sah, jadi satu pemanggilan `MAX-HEAPIFY` ini merampungkan pekerjaannya. 297
Gambar 5.12 Langkah (f): max-heap yang sudah jadi $A = [16,14,10,8,7,9,3,2,4,1]$. Setiap induk mendominasi kedua anaknya, dan maksimumnya, 16, sudah tiba di akar. 298
Gambar 5.13 Heapsort, langkah (a)–(c). Simpul yang diarsir masih berada di dalam *heap*; yang tidak diarsir sudah ditempatkan pada posisi terurutnya yang final dan tidak lagi menjadi bagiannya. **(a)** *heap* sesudah `BUILD-MAX-HEAP`. **(b)** sesudah menukarkan 16 ke posisi 10 dan memperbaiki akarnya. **(c)** sesudah menukarkan 14 ke posisi 9 dan memperbaikinya. 305
Gambar 5.14 Heapsort, langkah (d)–(f). *Heap*-nya menyusut satu simpul setiap kali dan ekor terurutnya, yang digambar tanpa arsiran dan dijajarkan di sepanjang bagian bawah, tumbuh menjemputnya. **(d)** 10 ditempatkan. **(e)** 9 ditempatkan. **(f)** 8 ditempatkan. 306
Gambar 5.15 Heapsort, langkah (g)–(i). Dengan *heap* yang tinggal empat, tiga, lalu dua simpul, pekerjaan per iterasinya sudah menjadi sepele: tingginya 2, lalu 1, lalu 1. 306
Gambar 5.16 Heapsort, langkah (j) dan (k). (j) sesudah penukaran terakhir, *heap*-nya memuat satu elemen. **(k)** lariknya yang final $A = [1,2,3,4,7,8,9,10,14,16]$, dalam urutan tidak menurun. 307
Gambar 5.17 `PARTITION` versi Hoare, dengan penelusuran lengkap pada [17,12,6,19,23,8,5,10]. Hijau toska menandai elemen yang sudah diketahui $\leq x$, kuning tua yang sudah diketahui $\geq x$. Penunjuk i dan j berangkat dari tepat di luar lariknya lalu merapat; ketika keduanya bersilangan, pemartisiannya selesai dan j dikembalikan. 321
Gambar 5.18 Pohon rekursi kasus terbaik bagi Quicksort. Ukuran submasalah tiap tingkat membelah dua — n, lalu $n/2$ dua kali, lalu $n/4$ empat kali — tetapi pekerjaan *total* pada tiap tingkatnya tetap n, sebab memartisi $2d$ subarray berukuran $n/2d$ berongkos $2d \times n/2d = n$. Ada $\lg n + 1$ tingkat, jadi totalnya $\Theta(n\lg n)$. 329
Gambar 5.19 Rekurensi kasus terburuknya dibuka gulungannya: $T(n) = T(1) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n) = k = 1n\Theta(k) = \Theta k = 1nk = \Theta(n2)$. 330
Gambar 5.20 Pohon rekursi kasus terburuknya. Alih-alih bercabang, pohonnya merosot menjadi satu rantai tunggal sepanjang n — setiap simpul memperanakkan satu anak berukuran satu lebih kecil dan satu anak berukuran 0. Biaya tingkat-tingkatnya adalah $n, n, n-1, n-2, \ldots, 2$, dan menjumlahkannya memberikan $\Theta(n2)$. 331
Gambar 5.21 Pohon rekursi bagi belahan tetap 1: 9. Tulang punggung kirinya, yang ukurannya menyusut sepersepuluh setiap tingkat, menyentuh dasar sesudah $\log 10n$ tingkat. Tulang punggung kanannya, yang ukurannya hanya menyusut dengan faktor 10/9, bertahan sampai kedalaman $\log 10/9n$. Di antara kedua kedalaman itu, beberapa tingkat tidak lengkap dan berongkos kurang dari n; di atas kedalaman yang lebih dangkal, setiap tingkat berongkos tepat n. Totalnya karena itu berada di antara $n\log 10n$ dan $n\log 10/9n$, dan keduanya $\Theta(n\lg n)$. 332
Gambar 5.22 Analisis nasib berselang-seling. Dengan $L(n) = 2U(n/2) + \Theta(n)$ untuk tingkat yang mujur dan $U(n) = L(n-1) + \Theta(n)$ untuk tingkat yang sial, mensubstitusikan yang kedua ke dalam yang pertama memberikan $L(n) = 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) = 2L(n/2-1) + \Theta(n) = \Theta(n\lg n)$. 334
Gambar 5.23 Sebuah belahan sial disusul sebuah belahan mujur. Pemartisian sial atas n elemen menghasilkan potongan berukuran 1 dan potongan berukuran $n-1$; pemartisian mujur atas

potongan itu menghasilkan dua potongan berukuran $(n-1)/2$. Daerah yang diarsir menunjukkan biaya gabungannya, $\Theta(n) + \Theta(n-1) = \Theta(n)$ 335
Gambar 5.24 Sebuah belahan mujur tunggal, sebagai pembanding: satu pemartisian atas n elemen, berongkos $\Theta(n)$, menghasilkan dua potongan berukuran kira-kira $(n-1)/2$. 335
Gambar 6.1 Sebuah pohon pencarian biner. Akarnya adalah 50; segala isi subpohon kiri (30,20,40) berada di bawah 50, dan segala isi subpohon kanan (70,60,80) berada di atasnya. Uji yang sama lolos di setiap simpul lainnya. 374
Gambar 6.2 Sebuah pohon untuk berlatih mencari penerus inorder. Menelusurinya secara inorder memberi 4,8,10,12,14,20,22, dan membaca daftar itu adalah cara termudah untuk memeriksa klaim penerus atau pendahulu mana pun. 382
Gambar 6.3 Sebuah contoh penyisipan. Kunci 50 dijahitkan turun menyusuri pohon — ke kiri di 100, ke kanan di 20, ke kanan di 30 — dan mendarat sebagai anak kanan baru milik 30, ditandai dengan warna kuning tua. Tidak ada hal lain di pohon itu yang bergeser. 388
Gambar 6.4 Kasus 1 penghapusan: simpul yang hendak dihapus, 20, adalah simpul daun. Ia cukup dibuang; sisa pohonnya tidak tersentuh. 389
Gambar 6.5 Kasus 2 penghapusan: simpul 30 hanya punya anak kanan, yaitu 40. Menghapus 30 menaikkan 40 ke tempatnya. 390
Gambar 6.6 Kasus 3 penghapusan: simpul 50 punya dua anak. Penerus inordernya adalah 60, kunci terkecil di subpohon kanannya. Nilai 60 disalin naik ke akar dan simpul 60 yang asli lalu dihapus, yang merupakan penghapusan kasus 1. 390
Gambar 6.7 Sebuah pohon AVL. Kedua subpohon tiap simpul berbeda tinggi paling banyak satu. 399
Gambar 6.8 Sebuah pohon pencarian biner yang **bukan** pohon AVL. Menambahkan simpul 8 dan 3 mendorong dua faktor keseimbangan keluar dari rentang: milik 9 dan 13, keduanya -2. 400
Gambar 6.9 Kedua rotasi. Kelima benda yang sama — simpul x dan y serta subpohon $P1, P2, P3$ — muncul di kedua gambarnya. Rotasi kanan mengubah gambar kiri menjadi gambar kanan; rotasi kiri mengubah gambar kanan kembali menjadi gambar kiri. Keduanya balikan yang persis satu sama lain. 405
Gambar 6.10 Kasus Kiri-Kiri, yang juga muncul pada penghapusan. Satu rotasi kanan terhadap z memulihkan keseimbangannya; y menjadi akar baru subpohonnya, dengan x dan z sebagai anak-anaknya. 410
Gambar 6.11 Kasus Kiri-Kanan, yang juga muncul pada penghapusan. Dua rotasi diperlukan: pertama rotasi kiri terhadap y, yang mengubah persoalannya menjadi kasus LL, lalu rotasi kanan terhadap z. 411
Gambar 6.12 Kasus Kanan-Kanan, yang juga muncul pada penghapusan. Bayangan cermin yang persis dari **Gambar 6.10**: satu rotasi kiri terhadap z. 413
Gambar 6.13 Kasus Kanan-Kiri, yang juga muncul pada penghapusan. Cermin dari **Gambar 6.11**: pertama rotasi kanan terhadap y, yang mengubah zig-zagnya menjadi kasus RR, lalu rotasi kiri terhadap z. 414
Gambar 6.14 Menyisipkan 14 ke dalam sebuah pohon AVL. Setiap faktor keseimbangannya tetap berada di dalam $-1, 0, +1$, jadi tidak ada rotasi yang dilakukan. 415
Gambar 6.15 Menyisipkan 3 menghasilkan faktor keseimbangan -2 di simpul 10. Kasusnya adalah Kiri-Kiri, dan satu rotasi kanan dengan 10 sebagai porosnya memperbaikinya. 416
Gambar 6.16 Menyisipkan 45 membuat simpul 30 berat kanan sebanyak dua. Kasusnya adalah Kanan-Kanan, dan satu rotasi kiri dengan 30 sebagai porosnya memperbaikinya. 417
Gambar 6.17 Menyisipkan 7 menciptakan zig-zag — ke kiri di 10, lalu ke kanan di 5. Dua rotasi diperlukan: kiri dengan 5 sebagai poros, lalu kanan dengan 10 sebagai poros. 418
Gambar 6.18 Menyisipkan 15 membuat simpul 9 berat kanan sebanyak dua sementara simpul 16 condong ke kiri. Kasusnya adalah Kanan-Kiri: rotasi kanan dengan 16 sebagai poros, lalu rotasi kiri dengan 9 sebagai poros. 419
Gambar 6.19 Kelas `AVLTree.java`, bagian pertama. Kelasnya memperluas kelas `Tree` biasa dan menambahkan satu medan — `height` — "for performing height-balanced efficiently". Konstruktornya menetapkan `height = 1 + Math.max(left.getHeight(), right.getHeight())` 426
Gambar 6.20 Kelas `AVLTree.java`, bagian kedua. Pohon kosong adalah sebuah objek dengan `height = 0`; pohon bersimpul tunggal punya `height = 1`. 427

Gambar 6.21 Menghapus dari pohon AVL: simpul 32 hendak dihapus. Tinggi dituliskan di samping simpulnya, dan leluhur tak seimbang yang pertama z, anaknya yang lebih tinggi y, dan anak y yang lebih tinggi x ditandai dengan warna merah. Kotak-kotak kosongnya adalah subpohon nol, dan $T0$ sampai $T3$ menamai keempat subpohon yang akan dipindahkan rotasinya. 432

Gambar 6.22 Pohonnya sesudah rotasi kiri. Simpul 62 adalah akar barunya; 44 turun menjadi anak kirinya dan mengambil subpohon $T1$ (yang berakar di 50) sebagai anak kanannya; 78 tidak berubah sebagai anak kanan 62. 433

Gambar 7.1 Metode *divide and conquer* untuk perancangan algoritma: bagilah persoalan menjadi submasalah yang **saling lepas**, kuasai submasalah itu dengan rekursi, lalu gabungkan solusinya menjadi solusi persoalan asli. 462

Gambar 7.2 Merge Sort sebagai *divide and conquer*. Larik dibelah dua, tiap paruh diurutkan secara rekursif, lalu kedua paruh terurut itu digabungkan. Kalimat yang penting tertulis di bawah gambarnya: **submasalahnya saling bebas, semuanya berbeda.** 463

Gambar 7.3 Bilangan Fibonacci, beserta peringatan yang menyusul persis sesudahnya: *prosedur rekursif yang lurus-lurus saja itu lambat*. Mengapa, dan selambat apa, adalah urusan empat subbab berikutnya. 465

Gambar 7.4 Pohon rekursi `fib(6)` yang lurus-lurus saja. Segitiga kelabu menandai subpohon yang sudah pernah dihitung di suatu tempat di sebelah kirinya dan kini sedang dihitung ulang dari awal. Seluruh paruh kanan pohon itu — seluruh subpohon $F(4)$ — adalah duplikat subpohon yang sudah selesai dikerjakan di sebelah kiri. 466

Gambar 7.5 Argumennya secara garis besar. Kiri: bilangan Fibonacci yang berurutan mendekati rasio emas $\phi \approx 1.61803$, sehingga $Fn \approx 1.6n$. Kanan: banyaknya panggilan $2Fn + 1 - 1$ prosedur naif, pada skala logaritmik, sejajar dengan $1.6n$ — pohonnya punya kira-kira $1.6n$ daun, dan waktu jalannya eksponensial. 467

Gambar 7.6 Perbaikannya, secara ringkas: hitunglah Fn dalam waktu linear dengan *mengingat solusi submasalah yang sudah terselesaikan* — *dynamic programming* — membangun solusinya secara *bottom-up*, menukar ruang dengan waktu. Pasangan yang disorot itulah pengamatan yang akan menjadi §7.4.3: hanya dua nilai yang perlu diingat pada saat mana pun. 471

Gambar 7.7 Persoalan optimasi secara garis besar: ada banyak solusi, satu di antaranya optimal, sebuah solusi adalah *rangkaian pilihan*, dan algoritmanya menghitung nilai optimal serta — bila kita memintanya — solusi optimalnya sendiri. 479

Gambar 7.8 Mengalikan matriks $n \times m$ bernama A dengan matriks $m \times k$ bernama B, di sini dengan $n = 3, m = 2, k = 3$. Baris A yang diarsir dan kolom B yang diarsir bergabung menghasilkan satu entri terarsir $c22$ pada hasilnya, lewat rumus $ci, j = l = 1mai, l \cdot bl, j$ 482

Gambar 7.9 Sebuah contoh empat matriks. Untuk A (30×1), B (1×40), C (40×10), D (10×25), tiga di antara lima pengurungan yang mungkin berbiaya masing-masing 20 700, 41 200, dan 1 400; setiap batang memecah totalnya menjadi tiga perkaliannya. Baris terakhirnya menetapkan notasi yang dipakai mulai dari sini: kita mengelompokkan tanda kurung secara optimal atas $A1 \times A2 \times \cdots \times An$, dengan Ai berupa matriks $di - 1 \times di$ 483

Gambar 7.10 Dua pengamatan kuncinya. Pertama, kurung terluar pasti membelah deretannya pada *suatu* posisi k dengan $i \leq k < j$. Kedua, berapa pun nilai k itu, pengelompokan tanda kurung di kedua sisinya haruslah sendirinya optimal — *optimal substructure*, dan itulah pembenaran bagi rekurensinya. 487

Gambar 7.11 Rekurensi deretan matriks. Kasus basisnya adalah $M(i, i) = 0$; kasus umumnya meminimumkan, atas setiap titik belah k, biaya subderetan kiri ditambah biaya subderetan kanan ditambah $di - 1dkdj$ untuk perkalian terakhirnya. 488

Gambar 7.12 Hitungan submasalahnya: hanya ada $n2 + n = \Theta(n2)$ submasalah (i, j) yang berbeda dengan $1 \leq i \leq j \leq n$. Segitiganya menggambarkan submasalah itu untuk $n = 6$: 21 sel pada dan di atas diagonal. Jurang antara $\Theta(n2)$ pertanyaan dan kunjungan rekursif yang banyaknya eksponensial itulah celah yang dimanfaatkan *dynamic programming*. 489

Gambar 7.13 Sesudah algoritmanya berjalan, $M[1, n]$ memuat nilai optimalnya dan c memuat pembagian optimalnya; sebuah prosedur rekursif sederhana **Print-Optimal-Parens**(c, i, j) menyusuri c dan mencetak pengurungannya. Pohonnya memperlihatkan penyusuran itu pada contoh yang baru saja kita selesaikan dengan tangan, $d = [10,20,3,5,30]$, dengan ongkos tiap perkalian di samping panggilannya. 492

Gambar 7.14 Waktu jalan **Matrix-Chain-Order**: dengan mudah terlihat $O(n3)$, dan sesungguhnya juga $\Omega(n3)$, sehingga $\Theta(n3)$. Grafiknya, pada skala logaritmik, menyandingkan banyaknya calon

pemisahan itu dengan $3n - 1$ panggilan rekursi polos dan $P(n)$ pengurungan pencarian menyeluruh: *dari waktu eksponensial ke polinomial*, seluruh pokok bab ini dalam lima kata. 493

Gambar 7.15 Langkah 1 resepnya: tunjukkan adanya *optimal substructure*. Sebuah solusi adalah satu pilihan dari sejumlah kemungkinan ditambah solusi bagi submasalah yang diciptakan pilihan itu; lalu tunjukkan, dengan potong-dan-tempel, bahwa solusi submasalah itu sendiri haruslah optimal. 503

Gambar 7.16 Langkah 2: tuliskan rekurensinya. Bentuk umumnya adalah $Mopt =$ minpilihan k(jumlah $Mopt$ atas submasalah yang dihasilkan k) + (biaya memilih k) — lalu tunjukkan bahwa banyaknya instans submasalah yang berbeda terbatasi oleh sebuah polinomial. 503

Gambar 7.17 Langkah 3 dan 4: hitung nilai optimalnya secara *bottom-up* supaya hasil bagian yang diperlukan selalu sudah siap (atau pakai *memoization*), carilah peluang mengurangi ruangnya dengan melupakan submasalah yang tidak akan dipakai lagi, dan akhirnya susunlah sebuah solusi optimal dari rangkaian pilihan yang tercatat. 504

Gambar 7.18 Definisinya: Z adalah *subsequence* dari X bila Z dapat dibangkitkan dengan melewatkan sejumlah (boleh nol) karakter X. Untuk $X = $ "ACGGTTA" dan $Y = $ "CGTAT", LCS(X, Y) adalah "CGTA" atau "CGTT". Menyelesaikan persoalannya berarti menemukan *lewatan-lewatannya*. 506

Gambar 7.19 *Optimal substructure* milik LCS. Bekerja dari *ujung* Xm dan Yn: jika karakter terakhirnya sama, karakter itu bergabung ke dalam jawabannya dan kita berekursi pada kedua untai yang masing-masing dipendekkan satu; jika berbeda, kita harus melewatkan satu huruf dari X atau satu huruf dari Y, dan kita memutuskan yang mana dengan membandingkan kedua panjang LCS yang dihasilkan. Pembenarannya adalah argumen potong-dan-tempel. 507

Gambar 7.20 Rekurensi LCS: nol bila salah satu prefiksnya kosong; satu lebih banyak daripada tetangga diagonalnya ketika kedua karakternya cocok; selain itu yang lebih besar di antara tetangga di atas dan tetangga di kirinya. 508

Gambar 7.21 Tabel yang sudah terisi untuk $X = $ "ACGGTTA" dan $Y = $ "CGTAT": tiap sel memuat $c[i, j]$ beserta panahnya $b[i, j]$, dan sel yang diarsir adalah penelusuran balik yang mengeja "CGTT". Garis di sampingnya mengajukan pertanyaan §7.9.10: seberapa banyak ruang yang diperlukan jika kita tidak merekonstruksi subbarisannya sendiri? 510

Gambar 8.1 Sebuah pohon pencarian biner yang menggambarkan definisinya: "Left branch(es) contains the nodes whose values $<$ the value of the node" — cabang-cabang, jamak, bukan anak. 563

Gambar 8.2 Penerus inorder. Merangkai para penggantinya — 8, 10, 12, 14, 20 — menjejakkan urutan terurut yang dihasilkan penelusuran inorder. Membalik rangkaiannya memberikan urutan menurun. 572

Gambar 8.3 Pohon yang **merupakan** pohon AVL. Kedua subpohon tiap simpulnya berselisih tinggi paling banyak satu. 582

Gambar 8.4 Pohon yang **bukan** pohon AVL. Menggantungkan simpul 3 di bawah 5 mendorong dua faktor keseimbangan, milik 9 dan 13, menjadi -2, di luar jangkauan yang sah. 582

Gambar 8.5 Penyisipan ke dalam BST biasa: elemen baru selalu disisipkan sebagai daun. Penurunannya adalah pencariannya, dan pencariannya adalah penempatannya. 591

Gambar 8.6 Menyisipkan 14 ke dalam pohon delapan simpul. Penyisipannya terjadi, faktor keseimbangannya dihitung ulang menaiki lintasannya, dan tak satu pun keluar dari jangkauan yang sah — sehingga `insertHB` melakukan nol rotasi. Inilah kasus yang lazim. 593

Gambar 8.7 Kedua rotasi. Pada kedua gambar, bacaan inordernya adalah $P1$, x, $P2$, y, $P3$, sebagaimana dinyatakan baris di bawah gambarnya. Rotasi kanan mengubah gambar kiri menjadi gambar kanan; rotasi kiri adalah kebalikannya yang persis. 593

Gambar 8.8 Kasus Left-Left: z adalah simpul yang tak seimbang, y anak kirinya, x cucu kirinya. Satu rotasi kanan terhadap z mengangkat y ke puncak. $P1$ sampai $P4$ adalah keempat upapohonnya. 594

Gambar 8.9 Kasus Left-Right. Satu rotasi kanan tunggal terhadap z akan meninggalkan pohonnya sama tak seimbangnya ke arah sebaliknya. Pertama satu rotasi kiri terhadap y meluruskan zig-zagnya menjadi condongan kiri-kiri; lalu satu rotasi kanan terhadap z menuntaskannya. 595

Gambar 8.10 Kasus Right-Right: y adalah anak kanan z dan x adalah anak kanan y. Satu rotasi kiri terhadap z membetulkannya. Ini cermin persis dari **Gambar 8.8**. 597

Gambar 8.11 Kasus Right-Left: zig-zagnya kanan lalu kiri. Satu rotasi kanan terhadap y meluruskannya, lalu satu rotasi kiri terhadap z menuntaskannya. Cermin persis dari **Gambar 8.9**. 598
Gambar 8.12 Kasus penghapusan pertama: simpul yang akan dihapus adalah daun. Cukup singkirkan ia. 599
Gambar 8.13 Kasus kedua: simpulnya punya tepat satu anak. "Copy the child to the node will be deleted, then delete the child" — setara dengan itu, dalam gaya fungsional, kembalikanlah anaknya sebagai pengganti simpulnya. 599
Gambar 8.14 Kasus ketiga: simpulnya punya dua anak. Carilah pengganti inordernya (atau, secara simetris, pendahulunya), salinlah nilai itu ke dalam simpulnya, lalu hapuslah penggantinya dari subpohon kanannya — sebuah soal yang lebih kecil berupa kasus 1 atau kasus 2. 599
Gambar 8.15 Sebelum penghapusan: simpul 32 hendak dihapus, dan leluhur tak seimbang yang pertama $z = 44$, anaknya yang lebih tinggi $y = 62$, serta anak y yang lebih tinggi $x = 78$ sudah ditandai. Tinggi pada gambarnya menghitung simpul, sehingga sebuah daun bernilai 1 — satu lebih besar daripada kesepakatan kita di mana-mana. 602
Gambar 8.16 Setelah satu rotasi kiri terhadap 44. Simpul 62 menjadi akar yang baru, 44 turun ke kirinya sambil memegang subpohon berakar 50, dan 78 tak berubah di kanan. 602
Gambar 9.1 Sebelas aktivitas contoh berjalan, dengan waktu mulai si dan waktu selesai fi, sudah terurut sehingga $f1 \leq f2 \leq \cdots \leq f11$. Tabel yang satu ini menjadi contoh berjalan bagi seluruh §9.4. 674
Gambar 9.2 Perampungan Teorema 9.1 lewat pengontradiksian, sekaligus menggambarkan rekurensi pemrograman dinamis bagi $c[i, j]$ yang mengikutinya. Andaikan ada himpunan $A'kj$ yang lebih baik, menyulihkannya ke dalam Aij akan memberi lebih dari Aij aktivitas yang saling selaras — bertentangan dengan keoptimalan Aij. 678
Gambar 9.3 Penyusutan menjadi satu submasalah: begitu pilihan rakus $a1$ dijatuhkan, tersisalah satu submasalah $S1$, dan solusi optimal bagi persoalan aslinya adalah $a1$ bersama sebuah solusi optimal bagi $S1$. 680
Gambar 9.4 Recursive-Activity-Selector dalam pseudokode. Gelung **while** pada baris 2 mencari aktivitas pertama di Sk yang selesai; pemanggilannya dilakukan dengan $k = 0$, memakai aktivitas fiktif $a0$ dengan $f0 = 0$. 683
Gambar 9.5 Penelusuran atas **Recursive-Activity-Selector** pada sebelas aktivitas **Gambar 9.1**. Tiap baris adalah satu iterasi gelung **while**: batang atasnya adalah aktivitas ak (amber bila diambil), batang-batang toska adalah aktivitas yang sudah terpilih, dan anak panahnya menandai pembandingan sk terhadap f milik pilihan terakhir. Garis-garis putus-putus tebalnya memisahkan pemanggilan-pemanggilan rekursif, masing-masing diberi label argumennya; baris terbawahnya memperlihatkan jawaban $a1, a4, a8, a11$ yang menempati sumber dayanya dari waktu 1 sampai waktu 16. 684
Gambar 9.6 Greedy-Activity-Selector, bentuk iteratif dari **Gambar 9.4**. Peubah k selalu memegang indeks pilihan yang paling mutakhir, sehingga $f[k]$ adalah waktu saat sumber dayanya berikutnya menjadi bebas. 686
Gambar 9.7 Sebuah persoalan penyandian karakter. Sebuah berkas berisi 100 000 karakter hanya memuat a–f, dengan frekuensi sebagaimana ditampilkan (dalam ribuan). Kode panjang tetapnya berbiaya 300 000 bit; kode panjang berubah yang ditampilkan berbiaya 224 000. 705
Gambar 9.8 Pohon-pohon bagi kedua skema penyandian **Gambar 9.7**. Tiap daun membawa sebuah karakter dan frekuensinya; tiap simpul dalam membawa jumlah frekuensi di dalam upapohonnya. **(a)** kode panjang tetap $a = 000, \ldots, f = 101$; **(b)** kode awalan optimal $a = 0, b = 101, \ldots, f = 1100$. 707
Gambar 9.9 Huffman(C) dalam pseudokode. Q adalah antrean prioritas-min yang berkunci frekuensi; gelungnya berjalan $C - 1$ kali, tiap kali menggantikan dua benda yang paling jarang muncul dengan simpul baru yang frekuensinya adalah jumlah keduanya. 709
Gambar 9.10 Penelusuran enam panel atas **Huffman** pada frekuensi **Gambar 9.7**. Tiap panelnya memperlihatkan antreannya terurut menaik menurut frekuensi; pada tiap langkah dua pohon berfrekuensi terendah digabungkan. Daun berupa persegi panjang, simpul dalam berupa lingkaran. **(a)** enam daun awalnya; **(b)–(e)** tahap-tahap antaranya; **(f)** pohon akhirnya. 710
Gambar 9.11 Definisi sebuah matroid: himpunan landasan berhingga S, keluarga turun-temurun $\mathcal{I}$ berisi himpunan bagian yang *bebas*, dan sifat penukarannya. 723

Gambar 9.12 Matroid grafik $MG = (SG, \mathcal{I}G)$ dari sebuah graf tak berarah $G = (V, E)$: himpunan landasannya adalah E, dan sehimpunan sisi bebas tepat ketika ia nirsiklus — yaitu ketika ia membentuk sebuah hutan. **(a)** Sebuah graf G; **(b)** himpunan bebas A berisi empat sisi tanpa siklus; **(c)** himpunan tak bebas B, yang sisi-sisi merahnya menutup sebuah siklus. MG berkaitan erat dengan persoalan pohon merentang minimum, yang dibahas Bab 11 secara terperinci. 724

Gambar 9.13 Sebuah contoh kasus bagi persoalan menjadwalkan tugas berwaktu-satuan dengan tenggat dan denda pada satu pemroses: tujuh tugas $a1, \ldots, a7$ dengan tenggat di dan denda wi....729

Gambar 10.1 Definisi graf, beserta graf contoh yang dipakai sepanjang bab ini: $V = a, b, c, d, e$ dan $E = (a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)$. Lima simpul, tujuh sisi. Perhatikan bahwa gambarnya dan kedua daftar himpunan itu membawa keterangan yang persis sama; gambarnya hanya lebih enak dipandang. 766

Gambar 10.2 Sebuah rangkaian elektronik sebagai graf. Simpulnya adalah titik-titik solder tempat komponen bertemu; sisinya adalah komponennya — resistor, kapasitor — yang menghubungkan titik-titik itu. Pertanyaan "berapa lintasan dengan hambatan terkecil dari `start` ke blok bertanda IC?" adalah pertanyaan lintasan terpendek pada graf ini. 767

Gambar 10.3 Sebuah jaringan penerbangan sebagai graf: simpulnya adalah bandara (HNL, LAX, DFW, STL, FTL, PVD), sisinya adalah rute yang diterbangi di antaranya. "Bisakah saya pergi dari Honolulu ke Providence, dan dengan berapa kali ganti pesawat?" persis merupakan pertanyaan yang dijawab penelusuran melebar di §10.7. 767

Gambar 10.4 Derajat. Setiap simpul graf bab ini diberi label derajatnya alih-alih namanya. Dibaca berlawanan arah jarum jam dari kiri atas, labelnya adalah 3,2,3,3,3; jumlahnya 14, yaitu dua kali banyaknya sisi. 770

Gambar 10.5 Dua lintasan pada graf bab ini, disorot. Di sebelah kiri, $a\ b\ e\ d\ c$, berpanjang 4; di sebelah kanan, $b\ e\ d\ c$, berpanjang 3. Keduanya sederhana: tidak ada simpul yang dipakai dua kali. 773

Gambar 10.6 Lintasan ketiga, $b\ e\ c$, berpanjang 2. Inilah lintasan terpendek dari b ke c pada graf ini — tidak ada sisi (b, c), jadi tidak ada lintasan berpanjang 1. 773

Gambar 10.7 Sebuah siklus pada graf bab ini, disorot: $a\ c\ d\ a$. Berangkatlah dari a, berjalanlah ke c, berjalanlah ke d, berjalanlah kembali ke a — tiga sisi, dan Anda berada di tempat Anda memulai tanpa pernah mengulang satu simpul pun di perjalanan. 774

Gambar 10.8 Terhubung dan tidak terhubung. Di sebelah kiri, setiap simpul dapat dicapai dari setiap simpul lain. Di sebelah kanan simpul-simpul yang sama digambar dengan lebih sedikit sisi, dan dua simpul di ujung kanan terputus dari segitiga di sebelah kiri: tidak ada lintasan yang menghubungkannya, jadi grafnya tak terhubung. 775

Gambar 10.9 Sebuah graf dengan tiga komponen terhubung: sebuah segitiga di sebelah kiri, sebuah bintang di tengah, dan sebuah keping berisi lima simpul di sebelah kanan. Tidak ada sisi yang menyeberang di antara ketiganya, dan di dalam masing-masingnya setiap simpul mencapai setiap simpul lain. 776

Gambar 10.10 Pohon dan hutan. Masing-masing dari keempat graf itu sendiri-sendiri adalah sebuah *pohon*: terhubung, dan tanpa siklus di mana pun di dalamnya. Keempatnya bersama-sama membentuk satu graf yang tak bersiklus tetapi tidak terhubung — sebuah *hutan*, dengan empat komponen. 778

Gambar 10.11 Contoh berjalan untuk penyimpanan graf: graf berarah berisi penerbangan antara tujuh bandara (BOS, JFK, ORD, SFO, LAX, DFW, MIA), dengan tiap busur diberi label nomor penerbangan. Ada sebelas busur. Karena busurnya berarah, penerbangan AA 49 dari DFW ke LAX adalah sisi yang berbeda dari penerbangan DL 335 dari DFW ke ORD, dan tak satu pun di antaranya menyiratkan adanya penerbangan ke arah sebaliknya. 783

Gambar 10.12 Daftar sisi, dibangun pada graf penerbangan **Gambar 10.11**. Wadah bawah memuat ketujuh simpulnya; wadah atas memuat kesebelas sisinya, dan setiap objek sisi menunjuk ke bawah pada kedua ujungnya. Untuk menemukan penerbangan yang meninggalkan DFW, Anda harus memeriksa setiap satu dari sebelas objek sisi itu, sebab tidak ada apa pun di dalam strukturnya yang mengelompokkannya menurut bandara. 784

Gambar 10.13 Daftar ketetanggaan graf bab ini. Lima daftar, satu per simpul. Daftar simpul a memuat b, c, d; daftar simpul b memuat a, e; dan seterusnya. Daftar-daftar itu bersama-sama memuat 14 entri, yaitu $2E$ — lema jabat tangan, dibuat kasatmata. 785

Gambar 10.14 Matriks ketetanggaan graf bab ini, digambar berdampingan dengan grafnya sendiri. Baris c terbaca T, F, F, T, T: simpul c bertetangga dengan a, d, dan e tetapi tidak dengan b, dan

entri diagonalnya F sebab tidak ada gelung-diri. Matriksnya setangkup terhadap diagonalnya sebab grafnya tak berarah, dan ia punya 25 entri untuk graf yang hanya punya 7 sisi. 786

Gambar 10.15 Penelusuran melebar. Blok amber (baris 01–04) menginisialisasi setiap simpul selain sumbernya; blok toska (05–08) menginisialisasi sumbernya dan antreannya; blok abu-abu (09–18) adalah penelusurannya sendiri, dan catatan di sebelah kanan menyatakan apa yang ia capai — *tangani semua anak u sebelum menangani anak mana pun dari anak-anaknya*, dan itulah definisi melebar. 795

Gambar 10.16 Contoh BFS, dua keadaan yang pertama. **Kiri:** sesudah inisialisasi. Hanya s yang kelabu dan hanya s yang punya angka, 0; setiap simpul lain menampilkan ∞. Antreannya memuat s. **Kanan:** sesudah langkah 1. Kedua tetangga s kelabu dengan jarak 1, kedua sisi pohon s–w dan s–r digambar tebal, s hitam, dan antreannya memuat w lalu r — dalam urutan itu, yang menetapkan urutan Adj[s] sebagai $[w, r]$. 799

Gambar 10.17 Langkah 2 dan 3. **Kiri:** w telah dikeluarkan dari antrean dan dihitamkan, menemukan t dan x pada jarak 2; antreannya r, t, x. **Kanan:** r telah dikeluarkan dan dihitamkan, menemukan v pada jarak 2; antreannya t, x, v — tiga simpul yang semuanya berjarak 2, yaitu cincin kedua yang lengkap. 799

Gambar 10.18 Langkah 4 dan 5. **Kiri:** t dikeluarkan, menemukan u pada jarak 3; sisi (t, x) diperiksa lalu ditolak, sebab x sudah kelabu dengan $d[x] = 2$. **Kanan:** x dikeluarkan, menemukan y pada jarak 3; antreannya v, u, y. 799

Gambar 10.19 Langkah 6 dan 7. **Kiri:** v dikeluarkan; satu-satunya tetangganya, r, sudah hitam, jadi tidak ada yang ditemukan dan v sekadar berubah hitam. **Kanan:** u dikeluarkan; kedua tetangganya sudah terlihat. Antreannya tinggal y seorang. 799

Gambar 10.20 Hasilnya. Setiap simpul hitam, antreannya kosong, tiap simpul membawa jaraknya dari s, dan ketujuh sisi tebalnya adalah pohon melebarnya. Ada delapan simpul dan tujuh sisi pohon, persis sebagaimana dituntut hitungan **Gambar 10.10** atas sebuah pohon. 800

Gambar 10.21 Penelusuran mendalam. Prosedur luar DFS(G) memutihkan setiap simpul, menjalankan jamnya, lalu memanggil DFS-Visit pada setiap simpul yang masih putih; tiap panggilan semacam itu mengakarkan satu pohon baru dalam hutan mendalamnya. Prosedur dalam DFS-Visit(u) adalah penelusurannya sendiri: cat u kelabu, cap waktu penemuannya, rekursilah ke setiap tetangga yang putih, lalu cat u hitam dan cap waktu selesainya. 812

Gambar 10.22 Langkah 1 sampai 6 contoh DFS, dalam urutan baca: u berubah kelabu dengan $d = 1$; v ditemukan lewat sisi pohon dengan $d = 2$; y dengan $d = 3$; x dengan $d = 4$; sisi (x, v) diperiksa, didapati menunjuk ke simpul kelabu, dan diberi label **B** untuk sisi balik, digambar sebagai panah putus-putus; dan akhirnya x berubah hitam dengan $f = 5$, ditampilkan sebagai 4/5. 815

Gambar 10.23 Langkah 7 sampai 12. Rekursinya terurai, menghitamkan y pada 3/6 dan v pada 2/7; kembali di daftar ketetanggaan u, sisi (u, x) diperiksa dan diberi label **F** untuk sisi maju; u selesai pada 1/8, melengkapkan pohon pertamanya; gelung luarnya lalu mengakarkan pohon kedua di w dengan $d = 9$; dan sisi (w, y) diberi label **C** untuk sisi silang, sebab y milik pohon pertama. 815

Gambar 10.24 Langkah 13 sampai 16. z ditemukan dari w lewat sisi pohon dengan $d = 10$; gelung-diri (z, z) diperiksa dan diberi label **B**, sebab pada saat itu z kelabu; z selesai pada 10/11; dan w selesai pada 9/12. Setiap simpul kini hitam dan jamnya telah mencapai $12 = 2V$. 816

Gambar 10.25 Contoh DFS yang lebih besar, berpencap waktu. Tiap kotak simpul memuat d/f: s adalah 1/10, z adalah 2/9, y adalah 3/6, x adalah 4/5, w adalah 7/8, t adalah 11/16, v adalah 12/13, u adalah 14/15. Panah kelabu tebal adalah sisi pohon; panah tipisnya diberi label **B** (balik), **F** (maju), dan **C** (silang). Gambar tunggal ini membawa pencap waktunya, hutannya, dan penggolongan sisinya yang lengkap sekaligus. 825

Gambar 10.26 Struktur tanda kurung **Gambar 10.25**. Di atas, hutan mendalamnya digambar ulang dengan sisi pohon menunjuk ke bawah. Di bawah, jamnya dari 1 sampai 16 dengan satu batang mendatar per simpul, merentang $[d, f]$, dan penandakurungannya ditulis di sepanjang bagian bawah: $(s\ (z\ (y\ (x\ x)\ y)\ (w\ w)\ z)\ s)\ (t\ (v\ v)\ (u\ u)\ t)$. Setiap pasang batangnya entah bersarang atau saling lepas; tidak ada yang bertindihan sebagian. 825

Gambar 10.27 Graf **Gambar 10.25** digambar ulang sebagai hutan mendalamnya, dengan sisi pohon tebal dan menunjuk ke bawah. Kini penggolongannya menjadi kasatmata: $x \to z$ dan $u \to t$ melengkung kembali **ke atas** menuju seorang leluhur (**B**); $s \to w$ menunjuk **ke bawah** melewati satu aras antara menuju seorang keturunan (**F**); $w \to x$, $v \to w$, $v \to s$, dan $u \to v$ berjalan **ke samping** di antara cabang atau di antara kedua pohonnya (**C**). 830

Gambar 10.28 Dua graf berarah tak bersiklus. Pada masing-masingnya, ikutilah panahnya dari mana saja dan Anda tidak akan pernah dapat kembali ke tempat Anda memulai: tidak ada siklus berarah. DAG adalah model alami bagi sehimpunan prasyarat — "ini sebelum itu" — dan himpunan prasyarat yang tak bersiklus persis merupakan himpunan yang dapat dipenuhi. 841

Gambar 10.29 DAG berpakaian, diberi pencap waktu oleh sebuah DFS. Sembilan busana; panah dari x ke y berarti x harus dikenakan sebelum y. Capnya adalah shirt 1/8, tie 2/5, jacket 3/4, belt 6/7, watch 9/10, undershorts 11/16, pants 12/15, shoes 13/14, socks 17/18 — delapan belas cap untuk sembilan simpul, sebagaimana dituntut $2V$. Si watch sama sekali tidak punya panah: ia boleh dikenakan kapan saja. 843

Gambar 10.30 Urutan topologisnya: kesembilan simpul yang sama ditata dari kiri ke kanan dalam urutan waktu selesai yang menurun — socks 18, undershorts 16, pants 15, shoes 14, watch 10, shirt 8, belt 7, tie 5, jacket 4 — dan kini **setiap panah menunjuk ke kanan**. Itulah yang disebut pengurutan topologis. 843

Gambar 11.1 Dua graf berarah asiklik. **Kiri:** ikuti anak panahnya dari mana saja dan anak panahnya selalu habis. **Kanan:** tiga sisi yang membentuk segitiga tetapi tetap asiklik, sebab anak panahnya tidak berputar mengelilinginya ke satu arah; membalik sisi dari simpul atas ke simpul kanan bawah akan menutup sebuah siklus berarah. 874

Gambar 11.2 DAG pendahuluan untuk berpakaian, dengan cap waktu pencarian mendalam-dahulu $d[v]/f[v]$ tertulis di samping tiap simpul. Sembilan simpul, sembilan sisi. `watch` sama sekali tidak punya sisi: tidak ada yang mengekang kapan ia dikenakan. 879

Gambar 11.3 Hasilnya: kesembilan simpul **Gambar 11.2** dibentangkan dalam urutan waktu penyelesaian yang menurun. Setiap anak panah kini menunjuk ke kanan, dan itu persis definisi pengurutan topologis. Pasangan d/f dicetak di bawah tiap simpul supaya urutannya dapat diperiksa terhadap **Tabel 11.3**. 881

Gambar 11.4 Sebuah pohon merentang. **Kiri:** graf G. **Kanan:** sebuah pohon merentang G — setiap simpul G hadir, grafnya masih terhubung, dan tidak ada siklus yang tersisa. Hitunglah sisi di sebelah kanan: dua belas, untuk tiga belas simpul. 892

Gambar 11.5 Sebuah graf berbobot: graf tak berarah yang terhubung dengan sebuah bobot pada tiap sisinya. Sebagian sisi yang mahal tak terhindarkan (yang 15 adalah satu-satunya jalan menuju simpulnya); sebagian yang tampak murah justru berlebih. Membedakan keduanya adalah seluruh persoalannya. 895

Gambar 11.6 Sebuah pohon merentang minimum T, dengan salah satu sisinya (u, v) digambar sebagai garis putus-putus. Menghapus sisi itu membelah T menjadi dua bagian. 896

Gambar 11.7 Gambar 11.6 dengan anotasi: kedua bagian yang tersisa sesudah sisi putus-putus (u, v) dihapus dinamai $T1$ (kiri) dan $T2$ (kanan). Klaim Teorema 11.4 adalah bahwa masing-masingnya sendiri merupakan pohon merentang minimum bagi upagraf G yang diinduksi oleh himpunan simpulnya sendiri. 896

Gambar 11.8 Argumen pertukaran yang membuktikan Teorema 11.5. S dan $V - S$ adalah kedua gumpalannya. Sisi merah (u, v) di bawah adalah sisi ringannya, yang diandaikan *tidak* berada di dalam pohonnya. Garis putus-putus adalah lintasan tunggal yang disediakan pohonnya dari u ke v; sisi toska (x, y) adalah sisi pertama lintasan itu yang melintasi potongannya. Menukar (x, y) dengan (u, v) adalah langkah yang diambil buktinya. 899

Gambar 11.9 Langkah 1 penelusuran Prim: pohonnya baru berupa akar a. Ini sekaligus grafnya sendiri — sembilan simpul, empat belas sisi, bobot 1 sampai 14 — dan inilah graf yang dipakai untuk setiap penelusuran di sisa bab ini. 905

Gambar 11.10 Langkah 2: sisi ringan bagi potongan $S = a$ adalah (a, b) berbobot 4, sehingga b bergabung ke pohonnya. 906

Gambar 11.11 Langkah 3: dengan $S = a, b$ sisi-sisi yang melintas adalah $(b, c) = 8$, $(a, h) = 8$, dan $(b, h) = 11$. Dua di antaranya seri sebagai yang teringan; penelusurannya mengambil (b, c) 906

Gambar 11.12 Langkah 4: (c, i) berbobot 2 adalah jalan keluar termurah dari a, b, c, sehingga i bergabung. Ini juga langkah saat kunci h *diperbaiki*, dari 8 menjadi 7. 907

Gambar 11.13 Langkah 5: (c, f) berbobot 4 memasukkan f. Pohonnya kini a, b, c, i, f dan kunci g turun dari 6 menjadi 2. 907

Gambar 11.14 Langkah 6: (f, g) berbobot 2 memasukkan g. Simpul h kini sudah tiga kali kuncinya diturunkan — 8, lalu 7, lalu 1 — dan persis untuk itulah `DecreaseKey` ada. 907

Gambar 11.15 Langkah 7: sisi termurah di seluruh grafnya, (g, h) berbobot 1, baru diambil sekarang — pada langkah ketujuh. Prim mengambil sisi menurut urutan awannya yang tumbuh menjangkau mereka, bukan menurut urutan bobot. 908

Gambar 11.16 Langkah 8: (c, d) berbobot 7 memasukkan d, dan kunci e membaik dari 10 menjadi 9 908

Gambar 11.17 Langkah 9 — gambar terakhirnya. Kesembilan simpulnya hitam dan kedelapan sisi yang tersorot membentuk pohon merentang minimumnya: (a, b), (b, c), (c, i), (c, f), (f, g), (g, h), (c, d), (d, e), dengan bobot total 37. 909

Gambar 11.18 Himpunan saling lepas sebagai senarai. **Atas:** dua himpunan, 1,2,3,4 dan A, B, C; di dalam masing-masingnya, tiap elemen membawa penunjuk balik ke elemen pertama, yang merupakan wakilnya. **Bawah:** hasil penggabungan keduanya — senarai yang lebih pendek sudah ditempelkan dan setiap elemennya diarahkan ulang ke wakil yang baru, yaitu 1. 916

Gambar 11.19 Dua langkah Kruskal yang pertama. **Atas:** (g, h) berbobot 1 — sisi termurah di seluruh grafnya — diterima. **Bawah:** (c, i) berbobot 2 diterima. Tidak seperti Prim, Kruskal memulai dengan sisi termurah secara global, di mana pun ia kebetulan berada. 924

Gambar 11.20 Atas: (f, g) berbobot 2 diterima, meleburkan f dengan g, h. **Bawah:** (a, b) berbobot 4 diterima. Perhatikan bahwa hutannya kini punya empat pohon terpisah yang tumbuh serentak — sesuatu yang tidak pernah dilakukan Prim. 925

Gambar 11.21 Atas: (c, f) berbobot 4 diterima, meleburkan dua pohon menjadi c, f, g, h, i. **Bawah:** penolakan yang pertama — (g, i) berbobot 6 *tidak* tersorot, sebab g dan i sudah berada di satu pohon dan sisi itu akan menutup sebuah siklus. 926

Gambar 11.22 Atas: (c, d) berbobot 7 diterima. **Bawah:** (h, i), yang juga berbobot 7, ditolak. Dua sisi berbobot sama, nasib berlawanan — keputusannya sama sekali tidak berkaitan dengan bobotnya dan sepenuhnya berkaitan dengan partisinya. 926

Gambar 11.23 Atas: (a, h) berbobot 8 diterima. **Bawah:** (b, c) berbobot 8 ditolak. **Inilah langkah saat jawaban Kruskal menyimpang dari jawaban Prim:** Prim mengambil (b, c) dan tidak pernah menimbang (a, h); Kruskal melakukan kebalikannya. Kedua pohonnya berbobot 37 (lihat §11.12). 927

Gambar 11.24 Atas: (d, e) berbobot 9 diterima — sisi kedelapan sekaligus yang terakhir, yang membawa simpul terpencil e masuk ke pohonnya. **Bawah:** (e, f) berbobot 10 ditolak. 928

Gambar 11.25 Dua panel terakhir: (b, h) berbobot 11 dan (d, f) berbobot 14 sama-sama ditolak, dan kedelapan sisi yang tersorot adalah pohon merentang minimum yang sudah jadi, dengan bobot total 37 — bobot yang sama dengan jawaban Prim pada **Gambar 11.17**, lewat himpunan sisi yang berbeda. 928

Gambar 12.1 Graf berbobot dengan sembilan simpul dan empat belas sisi yang dipakai untuk setiap penelusuran pohon merentang minimum di Bab 11. Pohon merentang minimumnya berbobot 37, dan ada dua pohon berbeda dengan bobot itu — dan itulah sebabnya §11.12 ada. 960

Gambar 12.2 Dua graf berarah taksiklik. Di kiri, ikutilah anak panahnya dari mana pun dan Anda akan kehabisan anak panah. Di kanan, tiga anak panah membentuk segitiga yang tetap taksiklik, sebab arahnya tidak berputar searah mengelilinginya; balikkan anak panah bawahnya dan ketiganya akan menutup sebuah siklus berarah, dan spesifikasi alur kerja yang berbentuk seperti itu tidak dapat dijalankan dalam urutan apa pun. 963

Gambar 12.3 Sebuah pohon merentang. **Kiri:** sebuah graf. **Kanan:** sebuah pohon merentangnya — semua simpul hadir, masih terhubung, tanpa siklus; tiga belas simpul dan dua belas sisi, yaitu aturan $V - 1$ pada §11.4.2. Tulang punggung antar-jemput pertanyaan 2 persis benda ini, dipilih seringan mungkin. 964

Gambar 12.4 Graf dengan tiga komponen terhubung: sebuah segitiga, sebuah bintang, dan sepotong bersimpul lima. Tak ada sisi yang melintas di antaranya. Jaringan jalan Rute Laundry punya dua potongan semacam itu, dan yang kedua — Bulak dengan Kenjeran — adalah sebabnya janji dua lompatan tidak boleh diiklankan untuk seluruh kota. 973

Gambar 12.5 Teorema pemilihan aktivitas dirampungkan lewat kontradiksi, bersama rekurensi pemrograman dinamis untuk $c[i, j]$ yang dihasilkan separuh *optimal substructure* argumennya. Penyulihan yang digambar di atasnya — himpunan $A'kj$ yang lebih besar ditempelkan menggantikan Akj — adalah kembaran potong-dan-tempel dari pertukaran pada Persamaan (12.6): di sana satu pekerjaan ditukar dengan yang rakus dan ukurannya tetap; di sini potongan yang lebih besar akan memperbesar keseluruhannya, dan keoptimalan melarang hal itu. 989

Gambar 12.6 Kesebelas aktivitas §9.4.2 beserta waktu mulai dan waktu selesainya, sudah terurut sehingga $f1 \leq f2 \leq \cdots \leq f11$. 989

Gambar 12.7 Penelusuran enam panel atas algoritma Huffman pada data bersimbol enam §9.7. Tiap panel memperlihatkan antreannya terurut menaik menurut frekuensi; pada tiap langkah kedua pohon berfrekuensi terendah digabungkan, daunnya digambar sebagai persegi panjang dan simpul dalamnya sebagai lingkaran. 994

Gambar 12.8 Persoalan pengodean karakter §9.7: berkas berisi 100.000 karakter atas abjad a–f, dengan kode panjang-tetap berongkos 300.000 bit dan kode panjang-berubah berongkos 224.000. Bangun argumennya persis seperti Persoalan Terkerjakan D; hanya abjadnya yang berganti. 996

Gambar 12.9 Contoh §9.8.7 untuk penjadwalan tugas berwaktu satuan dengan tenggat dan penalti: tujuh tugas dengan tenggat di dan penalti wi. Algoritma rakus matroid menjadwalkan lebih awal sebuah himpunan berbobot 230 dan membayar penalti 50. 1003

Gambar 12.10 Pencarian melebar. Blok ambernya menyiapkan setiap simpul selain sumbernya; blok toskanya menyiapkan sumbernya dan antreannya; blok abu-abunya adalah pencariannya. Catatan di sebelah kanan menyatakan invariannya dengan kata-kata — *tangani semua anak u sebelum anak mana pun dari anak-anaknya* — dan kalimat itulah seluruh bedanya dari pencarian mendalam. 1012

Gambar 12.11 Pencarian melebar yang sudah selesai: setiap simpul hitam, antreannya kosong, tiap simpul membawa jaraknya dari s, dan ketujuh sisi tebalnya membentuk pohon melebar atas delapan simpul. 1014

Gambar 12.12 Pencarian mendalam. Prosedur luarnya memutihkan setiap simpul, memulai jamnya dari nol, dan memanggil DFS-Visit pada setiap simpul yang masih putih — tiap panggilan semacam itu mengakari satu pohon hutannya. Prosedur dalamnya mengelabui u, membubuhkan $d[u]$, berekursi ke setiap tetangga putih, lalu menghitamkan u dan membubuhkan $f[u]$. 1015

Gambar 12.13 Sebuah contoh yang lebih besar digambar ulang sebagai hutan mendalamnya, sisi pohonnya tebal dan menurun. Kini penggolongannya menjadi geometris: **B** melengkung kembali ke atas menuju leluhur, **F** menunjuk ke bawah melewati satu tingkat antara menuju keturunan, dan **C** berjalan menyamping di antara cabang atau di antara pohon. 1017

Gambar 12.14 Struktur kurung. Di atas, hutan mendalamnya dengan sisi pohon menunjuk ke bawah; di bawah, jamnya dari 1 sampai 16 dengan satu batang per simpul yang merentang $[d, f]$, dan pengurungannya ditulis di sepanjang dasarnya. Setiap pasangan batang tersarang atau lepas — tak pernah tumpang tindih sebagian. 1018

Gambar 12.15 Pencarian mendalam bersimpul delapan, bercap waktu: s adalah 1/10, z adalah 2/9, y adalah 3/6, x adalah 4/5, w adalah 7/8, t adalah 11/16, v adalah 12/13, u adalah 14/15. Satu gambar membawa cap waktunya, hutannya, dan penggolongan sisinya selengkapnya sekaligus — bentuk yang layak ditiru ketika Anda menuliskan sebuah pencarian sendiri. 1018

Gambar 12.16 DAG urutan untuk berpakaian, dengan cap waktu pencarian mendalam di samping tiap simpul: kemeja 1/8, dasi 2/5, jas 3/4, ikat pinggang 6/7, arloji 9/10, celana dalam 11/16, celana 12/15, sepatu 13/14, kaus kaki 17/18. Sembilan simpul, sembilan busur, delapan belas cap. 1036

Gambar 12.17 Hasilnya: kesembilan busana yang sama ditata dari kiri ke kanan menurut waktu selesai yang menurun — kaus kaki 18, celana dalam 16, celana 15, sepatu 14, arloji 10, kemeja 8, ikat pinggang 7, dasi 5, jas 4 — dan kini setiap anak panah menunjuk ke kanan. Itulah *arti* pengurutan topologis. 1036

Gambar 12.18 Kruskal pada graf bersimpul sembilan Bab 11 di bobot 8: (a, h) diterima, (b, c) ditolak. Prim, pada graf yang sama, mengerjakan sebaliknya — ia mengambil (b, c) dan tak pernah menimbang (a, h). Kedua pohonnya berbobot 37. Bobot yang sama adalah tempat kedua algoritma itu bebas berselisih. 1048

Gambar 12.19 Pohon merentang minimum yang sudah jadi pada graf bersimpul sembilan: (a, b), (b, c), (c, i), (c, f), (f, g), (g, h), (c, d), (d, e), berbobot total 37. Delapan sisi untuk sembilan simpul. Kruskal mencapai himpunan delapan sisi yang berbeda — juga berbobot 37 — sebab dua sisi berbobot 8 seri. 1051

Gambar 12.20 Pertukaran yang membuktikan sifat potongan. S dan $V - S$ adalah kedua gumpalannya; sisi merah (u, v) di bawah adalah sisi ringannya, yang diandaikan tidak berada di pohon; garis bertitik adalah lintasan pohon yang tunggal dari u ke v; dan sisi toska (x, y) adalah

sisi pertama lintasan itu yang melintasi potongannya. Menukar (x, y) keluar demi (u, v) adalah seluruh buktinya. 1055

Gambar 12.21 Himpunan saling lepas sebagai daftar. **Atas:** dua himpunan, tiap unsur membawa penunjuk balik ke wakilnya. **Bawah:** hasil menyatukan keduanya — daftar yang lebih pendek disambungkan dan setiap unsurnya diarahkan ulang. Bandingkan ongkosnya dengan **Tabel 12.35**, yang di dalamnya sebuah penyatuan mengubah persis satu penunjuk. 1058

Gambar 12.22 Definisi graf, dengan contoh bersimpul lima yang dipakai di sepanjang Bab 10: $V = a, b, c, d, e$ dan $E = (a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)$. Gambar dan kedua daftar himpunannya membawa informasi yang identik; gambarnya hanya lebih enak dipandang. Derajat 3,2,3,3,3 berjumlah $14 = 2 \times 7$ — latihan 11 dalam satu gambar. 1069

Gambar 12.23 Sebuah graf berbobot dari Bab 11. Sebagian sisi yang mahal tak terhindarkan — yang berbobot 15 adalah satu-satunya jalan masuk ke simpulnya — dan sebagian yang tampak murah justru mubazir. Graf kemiripan halaman berwatak persis seperti ini, dan algoritma yang sama memisahkan kedua jenis sisinya. 1070

Gambar 13.1 Ketaksamaan segitiga: anak panah berombak di atas adalah lintasan terpendek dari u ke v, pasangan di bawah adalah lintasan terpendek dari u ke x disusul lintasan terpendek dari x ke v. Rute bawah itu adalah *sebuah* lintasan dari u ke v, sehingga ia tidak mungkin lebih murah daripada yang terbaik. 1098

Gambar 13.2 Sebuah graf dengan siklus negatif yang terjangkau. Berkeliling siklus merah $a \to b \to c \to a$ berongkos $2 + (-5) + 1 = -2$, sehingga tiap putaran tambahan membuat "lintasannya" lebih murah dua. Setiap simpul yang terjangkau dari siklus itu — di sini a, b, c, dan d — memiliki $\delta(s, \cdot) = -\infty$. Persoalannya tidak sukar pada graf ini; ia *salah rumus*. 1100

Gambar 13.3 Kedua hasil `RELAX(u,v,w)` dengan $w(u, v) = 2$. **Atas:** $d[u] + w = 5 + 2 = 7$, yang kurang daripada $d[v] = 9$ yang sekarang, sehingga $d[v]$ turun menjadi 7 dan $\pi[v]$ menjadi u. **Bawah:** $d[u] + w = 7$, yang *tidak* kurang daripada $d[v] = 6$ yang sekarang, sehingga sama sekali tidak terjadi apa-apa. Relaksasi yang tidak mengubah apa pun bukanlah langkah yang terbuang; ia adalah penegasan bahwa estimasi yang sekarang sudah mengalahkan rute ini. 1103

Gambar 13.4 Algoritma Dijkstra: baris 1–3 menginisialisasi, baris 4–5 membuat himpunan selesai S yang kosong dan antrean Q, dan baris 6–8 berulang kali menyatakan simpul terdekat yang belum selesai sebagai selesai. Baris 9–11 yang disorot adalah `RELAX`, yang diterapkan pada setiap sisi yang keluar dari simpul itu. 1108

Gambar 13.5 Sebuah graf yang di atasnya Dijkstra diam-diam mengembalikan jawaban yang salah. Tidak ada siklus, sehingga tidak ada siklus negatif, dan jarak sebenarnya adalah $\delta(s, a) = 1$ dan $\delta(s, b) = 3$. Dijkstra memfinalkan a pada 2 — sebelum ia pernah menengok sisi (b, a) — dan tidak pernah meninjaunya kembali. 1110

Gambar 13.6 Contoh Dijkstra, panel (a): grafnya tepat sesudah `INITIALIZE-SINGLE-SOURCE`. $d[s] = 0$ (kuning, hendak dikeluarkan) dan setiap estimasi lainnya adalah ∞. $Q = s = 0$, $u = \infty$, $v = \infty$, $x = \infty$, $y = \infty$. 1113

Gambar 13.7 Panel (b): s biru tua (sudah selesai), kedua sisi yang keluar darinya sudah direlaksasi, dan x — kuning — adalah simpul berikutnya yang akan dikeluarkan sebab $5 < 10$. Perhatikan bahwa u sudah punya estimasi 10; estimasi itu akan diperbaiki sebelum u pernah dikeluarkan. 1113

Gambar 13.8 Panel (c): s dan x biru tua. Estimasi bagi u turun dari 10 menjadi 8 dan sisi pendahulunya berayun dari (s, u) ke (x, u) — anak panah tebalnya kini datang dari arah x. y kuning: pada 7 ia kunci terkecil yang tersisa. 1114

Gambar 13.9 Panel (d): s, x, dan y biru tua. Estimasi v membaik dari 14 menjadi 13 dan anak panah pendahulunya yang tebal berayun dari x ke y. u, kuning pada 8, adalah yang berikutnya. 1114

Gambar 13.10 Panel (e): empat simpul biru tua, v kuning pada 9, dan anak panah pendahulunya kini datang dari u. Setiap sisi di dalam grafnya sudah direlaksasi sekurang-kurangnya sekali kecuali kedua sisi yang meninggalkan v. 1115

Gambar 13.11 Panel (f): setiap simpul biru tua, algoritmanya selesai. Keempat sisi tebal (s, x), (x, u), (u, v), dan (x, y) adalah subgraf pendahulu $G\pi$ — pohon lintasan terpendek yang berakar di s. Membaca sebuah rute darinya tinggal soal menyusuri anak panah tebalnya mundur dari simpul mana pun ke s. 1115

Gambar 13.12 Gambar di balik bukti kebenaran Dijkstra: wilayah hijau-biru muda adalah himpunan selesai S, dan u adalah simpul pertama yang konon dikeluarkan dengan estimasi yang salah. Lintasan terpendek yang sebenarnya meninggalkan S lewat sisi (x, y), sehingga berbentuk $s \rightsquigarrow p1x \to y \rightsquigarrow p2u$ 1120

Gambar 13.13 Bellman–Ford: baris 01–04 menginisialisasi, baris 05–07 menjalankan $V - 1$ sapuan yang masing-masing merelaksasi setiap sisi, dan baris 08–09 adalah pendeteksinya, yang mengembalikan `false` bila masih ada sisi yang dapat direlaksasi. Baris 10 mengembalikan `true` ketika jawabannya layak dipercaya. 1128

Gambar 13.14 Contoh Bellman–Ford, panel (a): inisialisasi. Tiga dari kesepuluh sisinya berbobot negatif — $w(x,t) = -2, w(y,x) = -3, w(t,z) = -4$ — dan itulah persis sebabnya Dijkstra tidak boleh dipakai di sini. Namun tidak ada *siklus* negatif: periksalah $t \to z \to s \to t$, yang berongkos $-4 + 2 + 6 = 4 > 0$, dan $t \to x \to t$, yang berongkos $5 + (-2) = 3 > 0$. 1130

Gambar 13.15 Panel (b): sesudah sapuan 1. Hanya kedua sisi yang meninggalkan s yang berpengaruh, sebab hanya merekalah yang pangkalnya punya estimasi berhingga ketika mereka diperiksa. Anak panah tebal adalah sisi pendahulu yang sekarang. 1131

Gambar 13.16 Panel (c): sesudah sapuan 2. Setiap simpul kini punya estimasi berhingga. $d[x]$ mendarat di 4 lewat y dan sisi negatif (y,x), dan $d[z]$ di 2 lewat t dan sisi negatif (t,z). Dua dari kelima estimasinya masih salah. 1132

Gambar 13.17 Panel (d): sesudah sapuan 3. Estimasi bagi t turun dari 6 menjadi 2, sebab rute $s \to y \to x \to t$ berongkos $7 + (-3) + (-2) = 2$, yang mengalahkan sisi langsung berbobot 6. Anak panah pendahulunya yang tebal berayun dari s ke x. Perbaikan ini tak terlihat sampai sapuan 3 sebab ia memerlukan tiga sisi. 1132

Gambar 13.18 Panel (e), sesudah sapuan 4: $\delta(s,z) = -2$ lewat rute empat sisi $s \to y \to x \to t \to z$, berbobot $7 + (-3) + (-2) + (-4)$. Empat sisi adalah sebanyak-banyaknya sisi lintasan sederhana di dalam graf lima simpul, sehingga keempat sapuan $V - 1 = 4$ diperlukan. 1133

Gambar 13.19 `DAG-Shortest-Paths`: baris 01–03 menginisialisasi, baris 04 mengurutkan simpul secara topologis, dan baris 05–07 merelaksasi sisi keluar tiap simpul tepat sekali, dalam urutan itu. Tidak ada antrean dan tidak ada pengulangan. 1139

Gambar 13.20 Graf contoh Bellman–Ford dengan ketiga sisi mundurnya dibuang: $s \to t$ (6), $s \to y$ (7), $t \to x$ (5), $t \to y$ (8), $t \to z$ (−4), $y \to x$ (−3), $y \to z$ (9). Setiap anak panah kini menunjuk dari kiri ke kanan atau ke bawah, sehingga grafnya sebuah DAG — dan ia masih memuat dua sisi negatif, yang ditangani algoritma DAG tanpa mengeluh. 1141

Gambar 14.1 Operasi $op1, \ldots, op7$ beserta **biaya sebenarnya** ti dan **biaya amortisasi** ai yang kita tetapkan untuknya. Syaratnya adalah $\sum ti \le \sum ai$ atas seluruh barisan, dan sama sekali **bukan** $ti \le ai$ untuk tiap i. 1168

Gambar 14.2 Ketiga teknik yang umum: **metode agregat** membagi biaya total $T(n)$ dengan n; **metode akuntansi** menagih sebagian operasi terlalu mahal supaya membayar di muka untuk yang lain; **metode potensial** menyimpan pembayaran di muka itu sebagai "energi potensial" strukturnya sendiri. 1170

Gambar 14.3 Operasi-operasi tumpukan: `Push` dan `Pop` masing-masing menyentuh satu slot larik dan satu indeks, sehingga keduanya $O(1)$. `Multipop(S,k)` memanggil `Pop` sampai tumpukannya kosong atau k pop sudah dilakukan, sehingga biayanya adalah banyaknya putaran gelungnya. 1171

Gambar 14.4 `Increment(A)`: baris 2–4 berjalan naik dari $A[0]$ sambil mengubah tiap 1 di ekornya menjadi 0, dan baris 6 menyetel 0 pertama yang dicapainya. Bila setiap bitnya 1, pencacahnya diam-diam berputar kembali ke nol, persis seperti register mesin k-bit. 1176

Gambar 14.5 Pencacah 8-bit sesudah tiap satu dari 16 increment; sel yang diarsir pada sebuah baris adalah bit yang dibalik oleh increment berikutnya, sehingga banyaknya adalah biaya sebenarnya increment itu. Total berjalannya sesudah 16 increment adalah 31, di bawah $2 \times 16 = 32$. 1177

Gambar 14.6 Analisis agregat pencacah dalam satu tampilan: buang lantainya, perpanjang jumlahnya sampai tak hingga, keluarkan n, lalu jumlahkan deret geometrinya menjadi 2. 1178

Gambar 14.7 Lima operasi pertama Soal 17.1-3 berbiaya 1,2,1,4,1: pangkat dua 1,2,4 menagih indeksnya sendiri dan operasi lain menagih 1. Totalnya 9, atau 1.8 per operasi. 1182

Gambar 14.8 Penyelesaian lengkap Soal 17.1-3, dari jumlah yang dipecah sampai Total $\le 3n - \lfloor \lg n \rfloor - 2$, lengkap dengan rumus deret geometri dan kesamaan logaritma yang diperlukannya. 1183

Gambar 14.9 Tagihan akuntansi untuk tumpukan: `Push` (biaya sebenarnya 1) ditagih 2, sedangkan `Pop` (biaya 1) dan `Multipop` (biaya $\min(s,k)$) ditagih 0. Ketiga tagihannya konstanta, sehingga n operasi berbiaya $O(n)$. 1190

Gambar 14.10 Penyelesaian Soal 17.2-2 dengan metode akuntansi: tiap operasi ditagih 3, dan kredit sesudah operasi i adalah $3i$ dikurangi biaya sebenarnya sejauh itu, dipecah menjadi dua kasus menurut letak i di antara pangkat-pangkat dua. 1195
Gambar 14.11 Menjumlahkan biaya amortisasi atas $i = 1, \ldots, n$ membuat suku-suku potensialnya saling meniadakan berpasangan, menyisakan yang pertama dan yang terakhir. Perhitungan inilah yang membuat seluruh metodenya bekerja. 1199
Gambar 14.12 Tumpukan dengan metode potensial: dengan Φ = banyaknya elemen di tumpukannya, `Push` berbiaya amortisasi $1 + (s + 1) - s = 2$ dan `Multipop` berbiaya amortisasi $k' - k' = 0$, dengan $k' = \min(k, s)$. 1201
Gambar 14.13 Pencacah dengan metode potensial: dengan bi = banyaknya bit 1 sesudah `Increment` ke-i dan ti = banyaknya bit yang dinolkan increment itu, perubahan potensialnya paling besar $1 - ti$ dan biaya amortisasinya paling besar $(ti + 1) + (1 - ti) = 2$. 1202
Gambar 14.14 `Enqueue` dan `Dequeue` untuk antrean dua-tumpukan: `Enqueue` adalah satu push, dan `Dequeue` mem-pop dari $SOUT$ sesudah lebih dahulu menguras seluruh SIN ke dalamnya, **hanya bila** $SOUT$ **kosong**. Pengujian *underflow*-nya datang sesudah pemindahannya. 1207
Gambar 14.15 Analisis potensial antrean dua-tumpukan. Dengan $\Phi(Di) = 2 \times \text{elements}i$, dengan elements$i$ mencacah item di tumpukan **masukan** saja, `Enqueue` berbiaya amortisasi 3 dan kedua jenis `Dequeue` berbiaya 1. 1208
Gambar 14.16 `TABLE-INSERT(T,x)`: baris 1–3 menangani penyisipan pertama ke dalam tabel tanpa slot, baris 4–9 memperluas (alokasikan dua kali lipat, salin, bebaskan blok lama), dan baris 10–11 menyimpan x lalu mencacahnya. 1213
Gambar 14.17 Analisis agregat `TABLE-INSERT`. Satu satuan untuk tiap penyisipan, ditambah biaya perluasannya $20 + 21 + \cdots + 2\lfloor \lg n \rfloor$, yang kurang dari $2n$; totalnya di bawah $3n$. 1214
Gambar 14.18 Kedua kasus analisis potensial `TABLE-INSERT` menghasilkan tepat 3. Pada kasus tanpa perluasan, biaya sebenarnya 1 dan potensialnya naik 2; pada kasus dengan perluasan, biaya sebenarnya $numi$ dan potensialnya ambruk sebesar $numi - 3$. 1217
Gambar 14.19 Sebuah barisan penyisipan: diagonal tipis adalah num, tangga tebal adalah $size$, dan gigi gergajinya adalah $\Phi = 2\,num - size$, yang jatuh ke 0 pada tiap perluasan lalu mendaki dengan kemiringan 2 sampai menemui $size$ lagi. 1218
Gambar 15.1 Masukan setiap algoritma di bab ini: matriks ketetanggaan $n \times n$, $W = (wij)$, dari Persamaan (15.1). Sisi berbobot negatif diperbolehkan; siklus berbobot negatif tidak. 1265
Gambar 15.2 Contoh berjalan: sebuah graf berarah bersimpul lima dengan sembilan sisi, dua di antaranya negatif ($1 \to 5$ berbobot -4 dan $4 \to 3$ berbobot -5). Ia tidak punya siklus berbobot negatif (§15.6.14 memeriksanya dengan mesin), sehingga setiap $\delta(i, j)$ berhingga. 1266
Gambar 15.3 PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j) memulihkan rute yang sesungguhnya dari matriks pendahulunya, mencetaknya maju sekalipun Π menyimpannya mundur. 1268
Gambar 15.4 Struktur sebuah lintasan terpendek yang bersisi paling banyak m. Bila $i \neq j$, penggallah sisi terakhirnya dan sisanya haruslah lintasan terpendek pula — inilah Teorema 13.1 sedang bekerja. 1274
Gambar 15.5 Definisi $lij(m)$ beserta kasus dasarnya $lij(0)$. Perhatikanlah bahwa superskripnya berada di dalam tanda kurung — ia mencacah sisi, ia bukan pangkat. 1275
Gambar 15.6 Rekurensi Persamaan (15.4), penyederhanaannya yang satu baris, dan fakta, Persamaan (15.6), bahwa barisan $lij(m)$ berhenti berubah pada $m = n - 1$. 1276
Gambar 15.7 EXTEND-SHORTEST-PATHS(L, W), yang mengambil sebuah matriks berisi rute-terbaik-sejauh-ini lalu memperpanjang setiap satunya dengan satu sisi lagi. Tiga gelung bersarang, sehingga waktunya $\Theta(n3)$. 1279
Gambar 15.8 SQUARE-MATRIX-MULTIPLY(A, B), algoritma sekolahan $\Theta(n3)$ untuk $C = A \cdot B$. Taruhlah ia berdampingan dengan **Gambar 15.7** dan keduanya berbeda di persis dua tempat. 1279
Gambar 15.9 SLOW-ALL-PAIRS-SHORTEST-PATHS(W). Ia melakukan $n - 2$ panggilan ke EXTEND-SHORTEST-PATHS, masing-masing berongkos $\Theta(n3)$, sehingga totalnya $\Theta(n4)$. 1281
Gambar 15.10 Graf bersimpul lima itu (Gambar 25.1 pada edisi ketiga buku Cormen dkk.) beserta barisan $L(1), L(2), L(3), L(4)$ yang dihitung SLOW-ALL-PAIRS-SHORTEST-PATHS. Karena $n = 5$, matriks terakhirnya $L(4) = L(n - 1)$ memuat bobot lintasan terpendeknya, dan $L(5) = L(4) \odot W$ sama dengan $L(4)$. 1282
Gambar 15.11 Hanya $\lceil \lg(n - 1) \rceil$ hasil kali yang diperlukan, sebab masing-masing melipatduakan jatah sisinya. Hasil kali terakhirnya $L(2\lceil \lg(n - 1) \rceil)$ sama dengan $L(n - 1)$ menurut Persamaan (15.6). 1287

Gambar 15.12 FASTER-ALL-PAIRS-SHORTEST-PATHS(W). Baris 6 mengkuadratkan matriks yang sedang berjalan alih-alih mengalikannya dengan W, dan itulah seluruh bedanya dari **Gambar 15.9**. 1289
Gambar 15.13 Struktur sebuah lintasan terpendek yang simpul antaranya terletak di $1, \ldots, k$, dipecah menurut apakah k sendiri terpakai. 1293
Gambar 15.14 Lintasan p (Gambar 25.3 pada edisi ketiga buku Cormen dkk.) adalah lintasan terpendek dari i ke j dan k adalah simpul antaranya yang bernomor tertinggi. Kedua paruhnya, $p1$ dan $p2$, lalu punya seluruh simpul antaranya di $1,2, \ldots, k-1$. 1294
Gambar 15.15 Definisi $dij(k)$, kasus dasarnya $dij(0) = wij$, dan rekurensi pada Persamaan (15.7). Matriks terakhirnya $D(n) = (dij(n))$ adalah jawabannya. 1295
Gambar 15.16 FLOYD-WARSHALL(W): lima kata keadaan dan tiga gelung bersarang. Baris 7 adalah rekurensi pada Persamaan (15.7) yang dituliskan lengkap. 1296
Gambar 15.17 Barisan lengkap matriks $D(k)$ dan $\Pi(k)$ yang dihitung Floyd–Warshall untuk graf pada **Gambar 15.2** (Gambar 25.4 pada edisi ketiga buku Cormen dkk.). Baris $k = 0$ adalah masukannya; baris $k = 5$ adalah jawabannya. 1301
Gambar 15.18 Kasus dasar $tij(0)$ dan rekurensi bagi $tij(k)$, yang merupakan rekurensi Floyd–Warshall dengan $\vee$ menggantikan min dan $\wedge$ menggantikan $+$. 1316
Gambar 15.19 TRANSITIVE-CLOSURE(G). Baris 3–7 membangun $T(0)$ dari himpunan sisinya; baris 8–12 adalah gelung rangkap tiga Floyd–Warshall dengan operator boolean. Ia berjalan dalam waktu $\Theta(n3)$. 1317
Gambar 15.20 Sebuah graf berarah dengan sisi $2 \to 3$, $2 \to 4$, $3 \to 2$, $4 \to 1$, $4 \to 3$, beserta matriks $T(k)$ yang dihitung algoritma penutupan transitifnya (Gambar 25.5 pada edisi ketiga buku Cormen dkk.). 1317
Gambar 15.21 Bila sebuah fungsi bobot baru w melestarikan lintasan terpendek dan tak negatif di mana-mana, maka $V \times$ Dijkstra atas w menjawab pertanyaan aslinya. 1322
Gambar 15.22 JOHNSON(G, w). Baris 1 menambahkan simpul bantunya, baris 2 menjalankan Bellman–Ford sekali, baris 6–7 membobot ulang, baris 9–10 menjalankan Dijkstra V kali, dan baris 12 mengubah tiap jawabannya kembali. 1324
Gambar 15.23 Sebuah graf bersimpul empat dengan satu sisi negatif, $0 \to 1$ berbobot -5. Dijkstra tidak diperbolehkan di sini. 1325
Gambar 15.24 Graf yang sama dengan simpul bantu 4 ditambahkan, tersambung ke tiap simpul asli lewat sebuah sisi berbobot 0. Simpul 4 tidak punya sisi masuk, sehingga ia tidak dapat muncul di dalam lintasan terpendek mana pun antara simpul-simpul asli. 1325
Gambar 15.25 Graf bersimpul empat itu sesudah dibobot ulang. Setiap bobotnya kini tak negatif — tiga dari kelimanya persis nol — sehingga algoritma Dijkstra dapat dipakai dari tiap simpul sebagai sumbernya. 1326
Gambar 15.26 Algoritma Johnson pada graf **Gambar 15.2** (Gambar 25.6 pada edisi ketiga buku Cormen dkk.). **(a)** G' dengan bobot aslinya; simpul baru s berwarna biru tua dan tiap simpul v memperlihatkan $h(v) = \delta(s, v)$. **(b)** Sesudah dibobot ulang dengan $w(u, v) = w(u, v) + h(u) - h(v)$; setiap bobotnya kini tak negatif. **(c)–(g)** Dijkstra dijalankan dari tiap simpul bergantian, sumbernya berwarna biru tua dan pohon lintasan terpendeknya digambar tebal berwarna hijau-biru; tiap simpul memperlihatkan $\delta(u, v)$ dan $\delta(u, v)$ yang dipisahkan garis miring, yang berhubungan lewat $duv = \delta(u, v) = \delta(u, v) + h(v) - h(u)$. 1330
Gambar 16.1 Sebuah pohon biner biasa dengan empat simpul (kiri) dan pohon biner diperluas yang diperoleh darinya (kanan). Lingkaran adalah simpul dalam; persegi adalah simpul luar yang menggantikan penunjuk kosong. Empat simpul dalam, lima simpul luar. 1372
Gambar 16.2 Ketaksamaan segitiga: pergi dari u ke v secara langsung tidak pernah lebih buruk daripada pergi dari u ke x lalu dari x ke v. Fakta tunggal inilah yang membuat langkah pemintasan pada Persoalan 16.4 sah. 1377
Gambar 16.3 Algoritma Dijkstra sebagaimana diberikan Bab 13. Baris 1–3 menginisialisasi, baris 7 mengeluarkan simpul belum-selesai terdekat, dan baris 9–11 yang disorot adalah relaksasinya. 1387
Gambar 16.4 Algoritma Bellman–Ford sebagaimana diberikan Bab 13. Baris 1–4 menginisialisasi persis seperti Dijkstra; baris 5–7 adalah $V - 1$ putaran atas semua sisi; baris 8–10 adalah uji siklus negatif yang tidak ada padanannya pada Dijkstra. 1388
Gambar 16.5 Contoh Dijkstra pada saat inisialisasi: $d[s] = 0$, dan $d[u] = d[v] = d[x] = d[y] = \infty$. Bobot sisi $s \to u = 10$, $s \to x = 5$, $u \to v = 1$, $u \to x = 2$, $x \to u = 3$, $x \to v = 9$, $x \to y = 2$, $y \to v = 6$, $y \to s = 7$, $v \to y = 4$. 1389

Gambar 16.6 Contoh Dijkstra ketika berhenti: $d[s] = 0$, $d[x] = 5$, $d[y] = 7$, $d[u] = 8$, $d[v] = 9$. Sisi-sisi tebal adalah pohon lintasan terpendek. Bandingkan bilangannya dengan penelusuran tangan di Subbab 16.7.5. 1390
Gambar 16.7 Contoh Bellman–Ford pada saat inisialisasi. Bobot: $s \to t = 6$, $s \to y = 7$, $t \to x = 5$, $t \to y = 8$, $t \to z = -4$, $x \to t = -2$, $y \to x = -3$, $y \to z = 9$, $z \to s = 2$, $z \to x = 7$. Tiga dari sepuluh bobotnya negatif, jadi Dijkstra tidak dapat dipakai. 1390
Gambar 16.8 Bellman–Ford ketika berhenti: $d[s] = 0$, $d[t] = 2$, $d[x] = 4$, $d[y] = 7$, $d[z] = -2$. Sisi-sisi tebal adalah pohon lintasan terpendek. Lintasan ke t adalah $s \to y \to x \to t$, berbiaya $7 - 3 - 2 = 2$ — tiga sisi, padahal sisi langsung $s \to t$ berbiaya 6. 1391
Gambar 16.9 Pencacah biner 8-bit dari 0 sampai 16. Bit yang diarsir adalah yang terbalik dalam perjalanan ke nilai berikutnya, dan kolom terakhir adalah jumlah berjalan pembalikan — 31 sesudah enam belas penambahan, tepat di bawah $2 \times 16 = 32$. 1409

Daftar Tabel

Tabel 1.1 Struktur data yang dipakai dalam buku ini, dan satu hal yang paling dikuasai masing-masing. 4
Tabel 1.2 Laju pertumbuhan yang umum dijumpai, dan berapa biaya masing-masing pada $n = 1000$. 10
Tabel 1.3 Mengubah sebuah batasan menjadi keputusan rancangan. Angka-angka di kolom tengah dihitung oleh sebuah program C++ pendek; kolom terakhir menerapkan anggaran 108 pada angka-angka itu. 10
Tabel 1.4 Lima paradigma perancangan, satu langkah yang menjadi cirinya, dan di mana masing-masing muncul dalam bab ini. 16
Tabel 1.5 Lima fase itu. Tidak satu pun di sini yang boleh dilewati, dan urutannya tidak bisa ditawar. 16
Tabel 1.6 Tabel memoisasi yang terbentuk sewaktu menghitung $f(12)$, dalam urutan entrinya terisi. 44
Tabel 1.7 Keenam persoalan bab ini, paradigmanya, dan satu kalimat yang menyelesaikan masing-masing. 45
Tabel 1.8 Gejala dan diagnosisnya. Ketika sebuah rumusan persoalan memuat ungkapan di sebelah kiri, teknik di sebelah kanan layak dicoba lebih dulu. 46
Tabel 1.9 Empat *online judge* yang dijumpai dalam buku ini. Ukurannya berubah setiap hari; angka-angkanya adalah yang dilaporkan pada tanggal yang tercantum. 47
Tabel 1.10 Dua keluarga buku di balik buku ini, beserta edisi yang berlaku pada September 2026. 53
Tabel 1.11 Enam belas bab buku ini, beserta bab-bab [11] yang bersesuaian. 53
Tabel 1.12 Keempat bab evaluasi dan cakupan masing-masing. 54
Tabel 1.13 Meja kerja di tiga sistem operasi, dari komputer kosong sampai program yang berjalan. Distribusi Linux lain memakai `sudo dnf install gcc-c++ gdb` (Fedora) atau `sudo pacman -S gcc gdb` (Arch) pada langkah 1. 55
Tabel 2.1 Insertion Sort pada $A = 5,2,4,6,1,3$, satu iterasi gelung luar untuk tiap baris. Batas antara awalan yang terurut dan akhiran yang belum tersentuh bergeser satu tempat ke kanan pada setiap baris. 82
Tabel 2.2 Biaya tiap baris Insertion Sort dan berapa kali baris itu berjalan. Lambang tj adalah berapa kali pengujian baris 4 dinilai untuk nilai j tersebut. 89
Tabel 2.3 Bagaimana lima waktu jalan yang umum itu tumbuh. Nilai terakhirnya begitu besar sehingga sering ditulis begitu saja sebagai ∞; nilai terhitungnya berdigit lebih dari tiga ratus ribu. 99
Tabel 2.4 Ukuran masukan terbesar yang dapat ditangani tiap kelas kompleksitas dalam batas satu sampai dua detik. Bacalah mundur untuk memilih algoritma: pernyataan soal memberi Anda n, dan tabel ini memberi Anda kompleksitas yang harus Anda capai. 103
Tabel 2.5 Binary search untuk $q = 7$ dalam $A = 2,4,5,7,10$. Dua pembandingan sudah cukup, sedangkan linear search memerlukan empat. 109

Tabel 2.6 Keenam problem Bab 1, ditelaah ulang dengan perkakas analisis Bab 2. Setiap klaim kompleksitas yang dibuat secara tak resmi di Bab 1 kini menjadi sesuatu yang dapat Anda turunkan. 115
Tabel 2.7 Topik-topik buku ini dipetakan ke *Introduction to Algorithms*, edisi ke-4 (geometri komputasional hanya ada pada edisi ke-3). Bab ini adalah baris satu dan dua. 116
Tabel 2.8 Cacah pembandingan kasus terburuk untuk binary search pada n elemen. 127
Tabel 3.1 Kelas-kelas pertumbuhan, apa yang terjadi pada masing-masing ketika ukuran masukan digandakan, dan di mana tiap kelas muncul dalam buku ini. 135
Tabel 3.2 Kelima notasi asimtotik dan perbandingan bilangan yang ditiru masing-masingnya. 143
Tabel 3.3 Kesamaan logaritma yang dipakai dalam bab ini. Dua yang terakhir adalah yang membuat Teorema Master bekerja. 148
Tabel 3.4 Fungsi-fungsi yang sering muncul, dalam urutan pertumbuhan menaik. Setiap baris adalah $o(\cdot)$ dari setiap baris di bawahnya. 151
Tabel 3.5 Jumlahan-jumlahan yang dipakai di sepanjang buku ini, dengan bentuk tertutup dan kelas asimtotiknya. 158
Tabel 3.6 Tiga bentuk yang dapat diambil sebuah pohon rekursi, dan apa yang diakibatkan masing-masingnya. Ketiganya berpadanan persis dengan Kasus 3, 2, dan 1 Teorema Master. 172
Tabel 3.7 Tiga kasus Teorema Master untuk $T(n) = a\,T(n/b) + f(n)$, beserta gagasan yang diungkapkan masing-masingnya. 176
Tabel 3.8 Kedua puluh soal terkerjakan §3.14 dalam satu pandangan, lengkap dengan kasus yang menaungi masing-masingnya. 187
Tabel 3.9 Segala isi Bab 3, dalam satu halaman. 199
Tabel 3.10 Algoritma-algoritma yang sudah dijumpai sejauh ini, rekurensinya, dan cara masing-masing diselesaikan. 200
Tabel 4.1 Anggaran kompleksitas. Bacalah batas pada n, bacalah kompleksitas yang terjangkau, lalu tolak setiap gagasan yang melampauinya sebelum Anda menulis satu baris kode pun. 206
Tabel 4.2 Sepuluh bilangan pertama berbentuk $2m + 2n$, dihitung dengan tangan. Kolom n berbunyi satu angka 1, dua angka 2, tiga angka 3, empat angka 4; di dalam tiap blok, m menghitung 0,1,2, ... 210
Tabel 4.3 Lima suku pertama barisan TOHU, dipulihkan dengan tangan dari jumlah-jumlah berbobotnya. Mulai $k = 2$, ak adalah kebalikan hasil kali tiga bilangan bulat berurutan; $a1$ mematahkan pola itu. 220
Tabel 4.4 Aturan Horner pada $2x3 - 6x2 + 2x - 1$ di $x = 3$. Sesudah tiap iterasi, p sama dengan polinomial parsial yang diramalkan invariannya; sesudah iterasi terakhir, nilainya sama dengan $P(3) = 5$. 229
Tabel 4.5 *Insertion sort* pada 8,3,6,3,9,1. Total pergeserannya, 9, sama dengan banyaknya inversi; tj menghitung berapa kali uji `while` pada baris 4 dijalankan dan selalu satu lebih banyak daripada pergeserannya. 232
Tabel 4.6 Pencarian biner pada larik 15 elemen untuk kunci yang ada (44) dan kunci yang tidak ada (20). Jendelanya menyusut $15 \to 7 \to 3 \to 1 \to 0$; empat probe cukup dalam setiap kasus. 234
Tabel 4.7 Menghitung eksekusi terdalam LOOPS untuk $n = 4$ dengan tangan. Subtotalnya adalah bilangan segitiga 10,6,3,1, dan jumlahnya $4 \cdot 5 \cdot 66 = 20$. 237
Tabel 4.8 Ketiga belas fungsi pada Persoalan Terkerjakan F dalam urutan laju pertumbuhan yang menaik. Fungsi pada baris yang sama saling Θ; kolom terakhir membenarkan setiap langkah naik. 242
Tabel 4.9 Pohon rekursi $T(n) = 3T(n/4) + n2$, tingkat demi tingkat. Tiap tingkat berbiaya 316 dari tingkat di atasnya, sehingga akarnya mendominasi. 247
Tabel 4.10 Empat rekurensi melalui Teorema Master. Yang terakhir terletak di celah antara Kasus 2 dan 3 dan dipecahkan dengan membuka rekurensinya. 250
Tabel 4.11 Pelipatduaan cepat menghitung $(F6, F7) = (8,13)$ dalam empat pemanggilan. Tiap baris memeriksa satu kejadian identitas pelipatduaan. 256
Tabel 4.12 Waktu terukur dari **Kode 4.17**. Melipatduakan n mengalikan waktu gaya kasar dengan sekitar 4, sebagaimana diramalkan $\Theta(n2)$, dan waktu urutkan-dan-cari dengan sekitar 2, sebagaimana diramalkan $\Theta(n\lg n)$; keunggulannya sendiri berlipat dua pada setiap baris. 260
Tabel 4.13 Keempat belas persoalan bab ini sekilas: keluarga tempat masing-masing bernaung, alat dari Bab 1 sampai Bab 3 yang menyelesaikannya, dan biaya penyelesaiannya. 262

Tabel 5.1 Empat algoritma pengurutan dari Bab 2 dan 3. Perhatikan bahwa yang cepat justru satu-satunya yang tidak bekerja di tempat. 274
Tabel 5.2 Berkeliling di pohon yang tersembunyi itu dengan aritmetika belaka. Setiap satu di antaranya adalah sebuah instruksi mesin tunggal. 279
Tabel 5.3 Biaya Heapsort, baris demi baris. Suku yang mendominasi adalah $n - 1$ pemanggilan `MAX-HEAPIFY`. 308
Tabel 5.4 Biaya keempat operasi antrean prioritas di bawah tiga implementasi. *Heap* adalah satu-satunya yang tidak punya entri $O(n)$. 313
Tabel 5.5 Merge Sort dan Quicksort adalah bayangan cermin satu sama lain. Masing-masing membelanjakan $\Theta(n)$ per tingkat; keduanya berbeda dalam *di mana* mereka membelanjakannya, dan dalam apakah belahannya terjamin. 319
Tabel 5.6 `PARTITION` versi Lomuto pada [2,8,7,1,3,5,6,4] dengan *pivot* 4. Awalan yang digarisbawahi adalah wilayah 1 — elemen-elemen yang sudah diketahui ≤ 4. 320
Tabel 5.7 Setiap algoritma pengurutan dalam buku ini sejauh ini, dengan batas Θ atas banyaknya pembandingan (kasus terbaik Heapsort mengandaikan kunci yang berbeda-beda). Kolom ruangnya mencacah memori tambahan di luar larik masukannya. 348
Tabel 5.8 Memilih algoritma pengurutan. Baris terakhirnya adalah yang paling sering akan Anda pakai dalam praktik. 354
Tabel 5.9 Tinggi-tinggi untuk Soal 5.2. Tingginya bertambah satu tepat pada tiap pangkat dua, dan itulah yang dikerjakan lantai sebuah logaritma basis dua. 365
Tabel 5.10 Pemartisian Lomuto pada Soal 5.11, iterasi demi iterasi. 367
Tabel 6.1 Sifat pohon pencarian biner dibandingkan sifat tumpukan-maks. Keduanya aturan yang berbeda, keduanya mendukung operasi yang berbeda, dan keduanya disimpan dengan cara yang berbeda. 376
Tabel 6.2 Ketiga penelusuran mendalam-dahulu dan pekerjaan yang dikerjakan masing-masingnya. 381
Tabel 6.3 Ketiga kasus penghapusan pada pohon pencarian biner, dan kerja yang dituntut masing-masingnya di luar pencarian awal. 391
Tabel 6.4 Simpul paling sedikit yang bisa dimuat pohon AVL bertinggi h, dan bilangan Fibonacci yang sama dengannya begitu ditambah satu. 402
Tabel 6.5 Keempat kasus penyeimbangan ulang, dikenali lewat dua faktor keseimbangan, beserta rotasi yang dituntut masing-masingnya. 414
Tabel 6.6 Memilih kamus terurut. k adalah cacah kunci yang sungguh-sungguh dikembalikan kueri rentangnya. 439
Tabel 6.7 Biaya kasus terburuk untuk pohon pencarian biner biasa dan untuk pohon AVL dengan n simpul. 449
Tabel 7.1 Jendela geser `fibIter` sewaktu menghitung $F6$. Sesudah iterasi terakhir `cur` memuat $8 = F6$, dan tidak ada satu larik pun yang pernah dialokasikan. 474
Tabel 7.2 Lima cara menghitung Fn dari bab ini, lengkap dengan biaya masing-masing. Baris pertama adalah yang harus dihindari; baris terakhir adalah yang menang ketika n besar. 479
Tabel 7.3 Tabel M yang sudah selesai untuk $d = 10,20,3,5,30$. Hanya segitiga atasnya yang bermakna, sebab sebuah subderetan haruslah punya $i \leq j$. Jawaban persoalan aslinya ada di pojok kanan atas, $M[1,4] = 1950$. 491
Tabel 7.4 Tabel c yang sudah selesai: $c[i,j]$ mencatat titik belah k yang mencapai minimum pada $M[i,j]$. Ia satu-satunya catatan tentang *bagaimana* optimumnya dicapai, dan §7.6.9 membaca pengurungannya dari sana. 492
Tabel 7.5 *Memoization* melawan *bottom-up*. Keduanya memberi waktu jalan asimtotik yang sama; segala yang berbeda ada di dalam tabel ini. 502
Tabel 7.6 Tabel c untuk $X = $ "ACGGTTA" dan $Y = $ "CGTAT", diisi sel demi sel di §7.9.7. Setiap entri adalah panjang sebuah LCS dari kedua prefiks yang melabeli baris dan kolomnya. Jawabannya, $c[7,5] = 4$, adalah sel yang dicetak tebal. 512
Tabel 7.7 Tabel panah b untuk instans yang sama. ↖ adalah `"copy"` (karakternya cocok), ↑ adalah `"skipx"` (nilainya diambil dari atas), ← adalah `"skipy"` (nilainya diambil dari kiri). Mengikuti panahnya mundur dari pojok kanan bawah akan merekonstruksi sebuah LCS. 512
Tabel 7.8 *Divide and conquer* melawan *dynamic programming*, berdampingan. Baris pertamanyalah yang menentukan mana yang sedang Anda hadapi; segala yang lain mengikutinya. 521
Tabel 7.9 Tabel c untuk Soal 7.10. Rekonstruksilah pengurungan yang ia kodekan. 529

Tabel 7.10 Jendela geser yang menghitung $F8 = 21$. Dua variabel, tujuh iterasi, tanpa larik. 530
Tabel 7.11 Tabel M untuk Soal 7.7. Tabel c-nya adalah $c[1,2] = 1$, $c[1,3] = 1$, $c[1,4] = 3$, $c[1,5] = 3$, $c[1,6] = 3$; $c[2,3] = 2$, $c[2,4] = 3$, $c[2,5] = 3$, $c[2,6] = 3$; $c[3,4] = 3$, $c[3,5] = 3$, $c[3,6] = 3$; $c[4,5] = 4$, $c[4,6] = 5$; $c[5,6] = 5$. 532
Tabel 7.12 Tabel c untuk Soal 7.11. Panjang LCS-nya adalah $c[7,6] = 4$. 533
Tabel 7.13 Tabel c untuk Soal 7.12. Panjang LCS-nya adalah 2, dan tiga LCS berbeda mencapainya. 533
Tabel 7.14 Tabel *edit distance* untuk "kitten" dan "sitting". Jawabannya, $e[6,7] = 3$, adalah sel yang dicetak tebal. 535
Tabel 8.1 Soal-soal bab ini, subbab yang menyelesaikan masing-masing, dan bab yang teorinya dipakai. Nomor soal dan nomor subbab adalah dua barisan yang berbeda; tabel ini menerjemahkan yang satu ke yang lain. 540
Tabel 8.2 Peta pengulangan. Setiap bab sebelum bab ini menyumbangkan sesuatu yang benar-benar diperlukan soal-soal bab ini. 541
Tabel 8.3 `BUILD-MAX-HEAP` pada $A = [5,13,2,25,7,17,20,8,4,11]$, satu baris per panggilan `MAX-HEAPIFY`. Nilai di akar tenggelam tiga aras pada panggilan terakhir. 545
Tabel 8.4 `PARTITION` Lomuto pada $[7,2,9,4,1,8,3,6,5]$ dengan pivot $x = 5$. Delapan perbandingan, empat pertukaran di dalam gelung dan satu di akhir; pivotnya berakhir di indeks $q = 5$. 553
Tabel 8.5 Rekursi yang benar ditelusuri pada pohon jebakan 20105,25,30. Batas $u = 20$ mencapai simpul 25 lewat *dua* aras rekursi; itulah keterangan yang dibuang oleh pemeriksaan lokal. 569
Tabel 8.6 Pemeriksaan simpul terukur berhadapan dengan bentuk tertutupnya. Pemeriksaan keseimbangan yang naif cocok persis dengan $(n + 1)\lg(n + 1) - n$ pada pohon sempurna, cara satu-lintasan cocok dengan n, dan rutin pelabelan tingginya cocok persis dengan $n(n + 1)/2$ pada rantai. 589
Tabel 8.7 Keempat kasus rotasi, sebagaimana diuji di dalam kode. Perhatikan bahwa kasus nolnya tergabung dengan rotasi *tunggal* di kedua sisinya — ≤ 0 di kiri, ≥ 0 di kanan — karena alasan yang diberikan §8.11.5. 598
Tabel 8.8 Kelima soal merah-hitam dalam subbab ini. Kolom terakhir mengatakan apakah sifatnya dapat diperiksa hanya dengan melihat sebuah simpul dan anak-anaknya — pertanyaan yang menentukan Soal 8.3. 609
Tabel 8.9 Soal-soal AVL dan soal-soal merah-hitam berdampingan. Pemeriksaan pohon pencarian dan rotasinya pada hakikatnya sama; soal-soal di tengahnya berbeda sebab invarian keseimbangannya berbeda. 609
Tabel 8.10 Tabel biaya M untuk $d = 4,10,3,12,20,7$. Setiap diagonal diisi sebelum diagonal berikutnya; jawabannya ada di pojok kanan atas, $M[1,5] = 1\,344$. 620
Tabel 8.11 Tabel pemisah c untuk rantai yang sama. Kedua entri tebal adalah entri yang dibaca oleh rekonstruksi di §8.15.8. 620
Tabel 8.12 Tabel c dan b untuk $X = $ SORTED dan $Y = $ STORE. Sel-sel tebal adalah keempat panah diagonal yang dilewati jalan mundur di §8.16.4. 629
Tabel 8.13 Soal 8.18 sampai 8.25 dibandingkan dengan soal-soal yang diperluasnya. Bentuknya sama; hanya invariannya yang berubah. 636
Tabel 8.14 Soal latihan 1: rekursinya berhenti di simpul 45, yang batas atasnya 40 datang dari akarnya dua aras di atas. Simpul 60, 50, dan 70 tak pernah dikunjungi. 654
Tabel 8.15 Soal latihan 6: menyisipkan 1 sampai 7 dalam urutan menaik. Setiap rotasinya adalah kasus Right-Right, sebab masukan menaik selalu condong ke kanan, dan empat rotasi tunggal sudah cukup untuk menjaga tingginya tetap 2 alih-alih 6. 655
Tabel 9.1 Aturan rakus yang sama pada enam sistem koin, diaudit oleh **Kode 9.1** terhadap optimum pemrograman dinamis untuk setiap nilai dari 1 sampai 200. Aturannya tak berubah; hanya struktur masukannya yang berubah. 672
Tabel 9.2 Contoh kasus **Gambar 9.1** dengan durasi tiap aktivitas dihitung. Aktivitas $a10$ adalah raksasanya: dua belas satuan waktu, hampir sepanjang hari. 674
Tabel 9.3 Pengurutan ditambah satu sapuan rakus, diukur oleh **Kode 9.2** pada aktivitas acak. Lajur ketiganya adalah waktu terukur dibagi $n\lg n$; ia bertahan di sekitar 4 di sepanjang rentang n yang 32 kali lipat, dan itulah wajah $\Theta(n\lg n)$ pada mesin yang sungguhan. 687
Tabel 9.4 Enam langkah strategi rakus, dipetakan ke pekerjaan yang dikerjakan di §9.4. Tak ada yang dilewati dan tak ada yang ditambahkan. 693

Tabel 9.5 Contoh kasus ransel berisi tiga barang, dengan kapasitas $W = 50$ kg. Barangnya sudah tersusun menurun menurut nilai per kilogram, jadi barang 1 adalah pilihan pertama algoritma rakusnya. 696
Tabel 9.6 Setiap himpunan bagian contoh kasus di **Tabel 9.5**, lengkap dengan harganya. Optimumnya 2,3 bernilai 220; aturan rakusnya mengembalikan 1,2 bernilai 160. 698
Tabel 9.7 Rakus lawan pemrograman dinamis. Baris yang menentukan adalah yang kedua: sifat pilihan rakus adalah tuntutan tambahan kerakusan, dan justru itulah yang tak dimiliki ransel 0-1. 699
Tabel 9.8 Kedua kode **Gambar 9.7**, dengan biaya masing-masing dihitung karakter demi karakter. Dalam ribuan bit: 300 bagi kode panjang tetap, 224 bagi yang panjang berubah. 705
Tabel 9.9 Lima penggabungan **Huffman** atas frekuensi **Gambar 9.7**, dengan antreannya sesudah masing-masing. Jumlah berjalan label barunya adalah biaya $B(T)$ dalam ribuan bit, dan ia berakhir pada 224 — bilangan yang sama yang diperoleh **Tabel 9.8** dengan menjumlahkan atas daun. 711
Tabel 9.10 Greedy pada tujuh tugas **Gambar 9.13**, satu baris per unsur yang ditinjau dalam urutan denda yang menurun. Bobot awal 230, denda yang dibayar 50. 730
Tabel 9.11 Algoritma rakus dalam buku ini. Lajur ketiganya adalah aturan lokalnya; lajur keempatnya adalah kewajiban pembuktiannya. Perhatikan betapa banyak entri di lajur keempat yang berbunyi "sebuah argumen penukaran". 736
Tabel 9.12 Cacah bentrokan bagi sembilan aktivitas Soal Latihan 9.4, sebelum pemilihan apa pun dilakukan. Lima aktivitas seri pada minimum dua. 751
Tabel 10.1 Derajat masuk dan derajat keluar graf berarah yang dipakai di §10.8. Kedua lajurnya berjumlah $8 = E$, sebagaimana dituntut lemanya. Gelung-diri $z \to z$ menyumbang satu pada derajat keluar z dan satu pada derajat masuknya. 772
Tabel 10.2 Seberapa jauh sesungguhnya jarak antara jarang dan padat. Untuk sejuta simpul, sebuah matriks memuat 1012 entri — satu terabita bahkan pada satu bita per entri — sementara graf jarang dengan simpul yang sama mungkin hanya punya sejuta sisi. Tabel tunggal inilah alasan §10.5 menghabiskan waktu begitu lama pada representasi. 781
Tabel 10.3 Kosakata lengkap §10.4, beserta contoh tiap istilah di dalam graf **Gambar 10.1**. Setiap satu dari kata-kata ini dipakai di suatu tempat dalam sisa bab ini. 782
Tabel 10.4 Biaya ruang ketiga representasi. Untuk graf mungil **Gambar 10.1**, ketiganya berjarak tidak sampai dua kali lipat satu sama lain, dan itulah sebabnya contoh kecil menyesatkan. Pada sejuta simpul, matriksnya sama sekali tidak bisa dipertimbangkan. 787
Tabel 10.5 Biaya setiap operasi dasar pada masing-masing dari ketiga representasi. Tidak ada representasi yang menang di setiap baris, dan itulah sebabnya ketiganya bertahan hidup. Baris terakhirlah yang menentukan isi bab ini: penelusuran atas daftar ketetanggaan bersifat linear; penelusuran yang sama atas matriks bersifat kuadratik. 788
Tabel 10.6 Kapan meraih representasi yang mana, dan alasannya untuk tiap kasus. Jika ragu, pilihlah daftar ketetanggaan; ia jauh lebih sering menjadi jawaban yang benar daripada tidak. 788
Tabel 10.7 BFS lengkap §10.7.4, satu baris per putaran gelung **while**. Setiap simpul dimasukkan ke antrean tepat sekali dan dikeluarkan tepat sekali, dan itulah sebabnya ada delapan langkah untuk delapan simpul. Perhatikan bahwa jarak di dalam antreannya tidak pernah menurun dari kiri ke kanan dan tidak pernah merentang lebih dari satu nilai — sifat yang dibuktikan di Lema 10.3. 800
Tabel 10.8 Keluaran BFS dari s: larik jarak d, larik pendahulu π, dan lintasan terpendek ke tiap simpul yang dibaca dari π dengan menyusuri pendahulunya kembali ke s lalu membaliknya. Panjang tiap lintasan sama dengan nilai d simpulnya, sebagaimana dituntut Teorema 10.4. 800
Tabel 10.9 Pencap waktu dan pendahulu sesudah DFS atas graf §10.8.4. Kedua akarnya adalah u dan w, dua simpul yang tidak punya pendahulu. Kedua belas capnya adalah bilangan bulat 1 sampai 12, masing-masing dipakai sekali. 816
Tabel 10.10 Setiap busur contohnya, digolongkan, beserta uji warna yang membuat keputusannya. Kedelapan busurnya muncul: tiga sisi pohon di pohon pertama, satu di pohon kedua, dua sisi balik, satu maju, satu silang. 816
Tabel 10.11 Keempat golongan sisi: graf mana yang dapat memuatnya, uji warna yang mengenalinya, pola pencap waktu yang dihasilkannya, dan untuk apa masing-masingnya berguna. Pada graf tak berarah hanya dua lajur pertamanya yang mungkin, dan itulah Teorema 10.9. 831

Tabel 10.12 Melebar melawan mendalam, pada sebelas sumbu. Keduanya berongkos sama dan berbeda pada satu baris kode; segala sesuatu lain di dalam tabel ini mengikuti dari satu baris itu. 848

Tabel 10.13 Pengukuran **Kode 10.10**, dengan nisbah pertumbuhannya ditambahkan. Waktu mutlaknya bergantung pada mesinnya dan berbeda beberapa puluh persen dari satu jalan ke jalan berikutnya, jadi *nisbahnya*-lah yang memikul klaimnya: melipatduakan grafnya kira-kira **melipatduakan** waktu daftar ketetanggaannya dan hampir persis **melipatempatkan** waktu matriksnya — tanda tangan $\Theta(V + E)$ dan $\Theta(V2)$ berturut-turut. Kedua lajur terakhirnya, waktu dibagi biaya yang diramalkan, tetap di dalam faktor kecil sepanjang rentang ukuran delapan kali lipat, dan itulah arti "linear" dan "kuadratik" dalam praktik. 850

Tabel 10.14 Keenam lintasan sederhana dari b ke d pada graf **Gambar 10.1**, dalam urutan panjang yang menaik. Yang terpendek berpanjang 2; tidak ada yang berpanjang 1 sebab $(b, d) \notin E$. 864

Tabel 10.15 Penelusuran melebar dari w pada graf §10.7.4. Bandingkan dengan **Tabel 10.7**: delapan langkah yang sama, tetapi pohon yang berbeda dan jarak yang berbeda, sebab sumbernya berpindah. 865

Tabel 10.16 Mengapa pohon melebar §10.7.4 tidak bergantung pada urutan ketetanggaannya: setiap simpul punya tepat satu tetangga yang selangkah lebih dekat ke s, sehingga ia tidak punya pilihan induk. 865

Tabel 10.17 Pencap waktu ketika gelung luar DFS memindai z, y, x, w, v, u. Bandingkan **Tabel 10.9**, yang memakai urutan u, v, w, x, y, z: graf yang sama, algoritma yang sama, hutan yang sama sekali berbeda. 866

Tabel 10.18 Dampak satu jalan baru terhadap cacahan lompatan Profitina Laundry. Satu zona bergabung ke janji dua lompatan dan satu zona meninggalkan ekor empat lompatan; zona yang sama sekali tidak punya jalan tidak terpengaruh, sebab menambahkan sisi di tempat lain tidak dapat menciptakan lintasan ke komponen yang berbeda. 870

Tabel 11.1 Keempat jenis sisi yang dapat dilalui sebuah pencarian mendalam-dahulu pada graf berarah, digolongkan menurut warna ujung seberangnya pada saat sisi itu diperiksa. Seluruh isi §11.2 berputar pada baris kedua. 873

Tabel 11.2 Kesembilan kendala pendahuluan pada **Gambar 11.2**, dibaca dari anak panahnya. Perhatikan apa yang *tidak* ada di sini: tidak ada yang mengaitkan `socks` dengan `shirt`, dan sama sekali tidak ada yang mengaitkan `watch` dengan apa pun. Kebebasan itulah sebabnya jawabannya tidak tunggal. 879

Tabel 11.3 Setiap detak jam mendalam-dahulu pada **Gambar 11.2**. Jamnya maju tepat satu pada tiap penemuan dan tiap penyelesaian, sehingga ia berakhir di $2V = 18$. Bandingkan pasangan d/f-nya dengan angka yang tertulis di samping simpul-simpulnya pada **Gambar 11.2**: keduanya cocok persis. 881

Tabel 11.4 Algoritma Kahn pada **Gambar 11.2**, satu pengeluaran tiap baris. Sembilan simpul keluar dari sembilan langkah, jadi tidak ada siklus. Bandingkan hasilnya dengan **Gambar 11.3**: kedua urutan itu tidak berbagi satu posisi pun, dan keduanya benar. 887

Tabel 11.5 Memilih di antara kedua algoritma pengurutan topologis. Keduanya $\Theta(V + E)$ dan keduanya benar; segala yang ada di tabel ini soal rekayasa, bukan soal kompleksitas. 891

Tabel 11.6 Kedua algoritma bab ini adalah algoritma yang sama dengan dua jawaban berbeda atas baris 3.1 `MoreSpecific-MST`. Selebihnya — bukti kebenarannya, cacah $V - 1$ sisinya, invarian gelungnya — dipakai bersama. 903

Tabel 11.7 Jalan lengkap Prim–Jarník dari **Gambar 11.9** sampai **Gambar 11.17**, dengan antrean prioritasnya dicetak sesudah tiap langkah sebagai `simpul:kunci/π`. Sembilan pengeluaran, delapan sisi, bobot total 37. Barisan bobot yang dikomitmenkan adalah 4, 8, 2, 4, 2, 1, 7, 9 — tidak terurut, sebab Prim tidak mengurutkan. 909

Tabel 11.8 Waktu jalan Prim dengan tiga antrean prioritas. Baris *Fibonacci heap* adalah alasan struktur itu ada: ia membuat `DecreaseKey` menjadi gratis dalam arti teramortisasi, dan `DecreaseKey` adalah operasi yang paling sering dilakukan Prim. 910

Tabel 11.9 Sebuah hutan himpunan saling lepas dengan gabung menurut peringkat dan pemampatan lintasan, dicetak sesudah tiap operasi. Entri yang berubah dicetak tebal. Dua hal yang perlu diperhatikan: peringkat sebuah akar hanya naik ketika dua peringkat yang sama bertemu, dan dalam barisan ini ketujuh penggabungan sungguhan semuanya mempertemukan peringkat yang sama — menjadi 1 pada baris 2–5, menjadi 2 pada baris 6–7, dan menjadi 3 pada baris 8; dan `Union(1,5)` yang "tanpa efek" itu toh tetap mengubah `parent[5]` dari 4

menjadi 0, sebab kedua pencarian yang dilakukannya memampatkan lintasannya dalam perjalanan. Sesudah sapuan pencarian terakhirnya, setiap elemen menunjuk langsung ke akarnya dan peringkatnya — yang memang hanya pernah menjadi batas *atas* bagi tingginya — dibiarkan basi, dan itu tidak berbahaya. 919

Tabel 11.10 Bagaimana fungsi Ackermann balikan tumbuh. Banyaknya atom di alam semesta teramati kira-kira 2266, sehingga untuk setiap persoalan himpunan saling lepas yang akan pernah dijalankan di atas perangkat keras fisik, $\alpha(n) \leq 4$. 920

Tabel 11.11 Jalan lengkap Kruskal dari **Gambar 11.19** sampai **Gambar 11.25**. Empat belas sisi ditimbang, delapan diterima, enam ditolak, bobot total 37. Kolom `Find` menunjukkan wakil yang sungguh-sungguh dikembalikan hutan himpunan saling lepasnya; perhatikan bagaimana c menjadi wakil komponen yang tumbuh itu lalu tetap demikian. 929

Tabel 11.12 Memilih di antara kedua algoritmanya. Dua baris pertamanya meliputi sebagian besar kasus nyata; selebihnya adalah kasus-kasus yang jawabannya tidak gamblang. 938

Tabel 11.13 Uji ketunggalan §11.12.3 diterapkan pada graf contoh sembilan simpul, dihitung dengan memanggil Kruskal dari **Kode 11.5** sekali untuk tiap sisi pohonnya dengan sisi itu dihapus. Tepat satu barisnya berbobot sama, sehingga MST-nya tidak tunggal — dan pohon alternatif yang dihasilkan baris ketujuhnya adalah $(a,b)4, (b,c)8, (c,d)7, (c,f)4, (c,i)2, (d,e)9, (f,g)2, (g,h)1$, yang persis merupakan jawaban Prim dari **Gambar 11.17**. 940

Tabel 12.1 Setiap gagasan Bab 9 sampai 11, subbab yang mengajarkannya, jenis pertanyaan yang dimunculkannya, dan tempat di bab ini persoalan dari jenis itu dikerjakan tuntas. 959

Tabel 12.2 Kesembilan tahap sebuah proyek perancangan, pertanyaan yang dijawab masing-masing tahap, dan bukti yang dihasilkan masing-masingnya. Tahap 1 sampai 6 berlangsung sebelum ada kode apa pun. 961

Tabel 12.3 Enam calon proyek yang dikenai uji pemikul beban. Empat lolos. Dua yang tidak lolos bukanlah program yang buruk; keduanya persoalan *perancangan* yang lemah, sebab keduanya tidak memberi rancangan dan evaluasi apa pun untuk diperdebatkan. 962

Tabel 12.4 Ketujuh belas jalan pada contoh Rute Laundry, dengan jarak dalam ratusan meter. Empat belas zona; enam belas jalan terletak di dalam satu komponen dan yang ketujuh belas, Bulak–Kenjeran, adalah pulau tersendiri. 963

Tabel 12.5 Biaya Rute Laundry yang diramalkan, fase demi fase, ditulis sebelum ada kode yang dikompilasi. Langkah pengurutannya mendominasi, dan §12.2.10 mengukur apakah itulah yang sungguh-sungguh terjadi. 966

Tabel 12.6 Cincin-cincin melebar di sekeliling gerai Sukolilo, dihitung dengan tangan. Enam zona terletak di dalam janji dua lompatan. Dua zona tak terjangkau, dan itu harus dilaporkan programnya sebagai fakta usaha alih-alih sebagai galat. 967

Tabel 12.7 Kruskal pada jalan-jalan Rute Laundry, dengan tangan. Sebelas sisi diterima untuk dua belas zona — persis $V - 1$ — dengan total 69 ratus meter, atau 6,9 km. Lima jalan ditolak sebab kedua ujungnya sudah tersambung. 967

Tabel 12.8 Algoritma Kahn pada kesepuluh tahap alur kerjanya. Sepuluh tahap dikeluarkan dalam sepuluh putaran, jadi spesifikasinya taksiklik dan urutannya sah. Pada putaran 4 *invoice* dikeluarkan sebelum *pretreat*, bukan karena mendesak, tetapi karena antreannya masuk-pertama-keluar-pertama dan invoice diantrekan lebih dahulu. 968

Tabel 12.9 Kedua puluh tiga persoalan terkerjakan Subbab 12.3 sampai 12.6, subbab bab ini tempat masing-masing dikerjakan, dan alat dari Bab 9 sampai 11 yang menyelesaikannya. 981

Tabel 12.10 Kesepuluh pekerjaan pengering pada Persoalan Terkerjakan B, didaftar menurut urutan waktu selesai yang tidak menurun. 986

Tabel 12.11 Kesepuluh pekerjaan pengering beserta durasinya dan banyaknya pekerjaan lain yang berbenturan dengan masing-masingnya. Pekerjaan $a1$, $a2$, $a4$, dan $a10$ seri dengan benturan tersedikit, yaitu dua. 987

Tabel 12.12 Keenam pekerjaan pada Persoalan Terkerjakan C, beserta waktu pemrosesan dan tenggatnya. 990

Tabel 12.13 Tenggat terawal dahulu pada keenam pekerjaan. Hanya $J4$ yang terlambat, sebesar satu satuan, jadi $L\max = 1$. 991

Tabel 12.14 Waktu pemrosesan terpendek dahulu pada keenam pekerjaan yang sama. $L\max = 6$, enam kali lebih buruk daripada tenggat terawal dahulu, walaupun *total* waktu selesainya lebih kecil. 991

Tabel 12.15 Keenam penggabungan algoritma Huffman pada frekuensi lognya. Enam penggabungan untuk tujuh simbol, sebagaimana dituntut $C - 1$. Label simpul terakhirnya, 100, adalah frekuensi totalnya — sebuah pemeriksaan yang berguna. 993

Tabel 12.16 Buku kode Huffman untuk lognya, beserta ongkos tiap simbol. Kedua simbol terjarang, Q dan X, memperoleh kedua kata kode terpanjang dan hanya berbeda pada bit terakhirnya — keduanya bersaudara, dan itulah yang dilakukan penggabungan pertama terhadap keduanya. 995

Tabel 12.17 Keempat barang pada Persoalan Terkerjakan E. 996

Tabel 12.18 Tabel *knapsack* 0-1 $c[i, w]$ untuk Persoalan Terkerjakan E. Lima puluh lima sel, masing-masing satu perbandingan tunggal, jadi seluruh tabelnya berongkos $\Theta(nW) = \Theta(4 \times 10)$. Jawabannya adalah pojok kanan bawahnya, 18. 997

Tabel 12.19 Ketujuh pesanan setrika pada Persoalan Terkerjakan F: masing-masing memakan satu jam, dan pesanan yang terlambat berongkos sebesar pengembalian uangnya. 998

Tabel 12.20 Algoritma rakus matroid pada ketujuh pesanan. Cacah yang tercetak tebal melanggar aturan $Nt \leq t$. Dua pesanan ditolak, dan pengembalian uang yang dibayar adalah jumlah keduanya, $30 + 20 = 50$. 1000

Tabel 12.21 Pencarian melebar atas $G1$ dari A, satu baris per putaran gelung **while**. Sembilan simpul, sembilan langkah: setiap simpul diantrekan sekali dan dikeluarkan sekali. Jarak di dalam antreannya tak pernah menurun dari kiri ke kanan dan tak pernah merentang lebih dari satu nilai — yaitu invarian antrean §10.7.9. 1013

Tabel 12.22 Keluaran pencarian melebar atas $G1$: jarak dan pendahulu. Membaca π mundur dari simpul mana pun lalu membaliknya memberikan sebuah lintasan terpendek, dan panjangnya selalu sama dengan d simpul itu. 1013

Tabel 12.23 Cap waktu dan pendahulu sesudah pencarian mendalam atas $G2$. Jamnya berhenti di $14 = 2V$, dan keempat belas capnya adalah bilangan bulat 1 sampai 14, masing-masing terpakai sekali — itulah pemeriksaan pertama yang harus dijalankan pada jawaban DFS mana pun. 1016

Tabel 12.24 Setiap busur $G2$ digolongkan, beserta uji yang menentukannya. Lima pohon, dua mundur, satu maju, tiga silang — sebelas seluruhnya. Dua sisi mundur berarti grafnya tidak taksiklik. 1016

Tabel 12.25 Pencarian mendalam atas graf tak berarah $G1$ dari A. Jamnya berakhir di $18 = 2V$. Selang-selangnya tersarang sempurna, sebab pohon mendalamnya di sini satu lintasan tunggal yang melewati kesembilan simpul. 1020

Tabel 12.26 Label komponen setiap simpul $G3$. Tiga komponen. Begitu larik ini ada, keterhubungan menjadi satu perbandingan bilangan bulat tunggal, dan itulah sebabnya lariknya — dan bukan pencariannya — yang menjadi jawabannya. 1024

Tabel 12.27 Kedelapan mata ajar digraf prasyarat yang dipakai dalam Persoalan M sampai Q, beserta durasi dan prasyaratnya. Sembilan busur seluruhnya: $A \to D, B \to D, B \to E, C \to G, D \to E, D \to F, E \to H, F \to H, G \to H$. 1034

Tabel 12.28 Cap waktu pencarian mendalam pada DAG prasyarat. Tiga akar — A, B, dan C, ketiga mata ajar tanpa prasyarat — jadi hutannya punya tiga pohon dan $8 - 3 = 5$ sisi pohon. 1035

Tabel 12.29 Algoritma Kahn pada DAG prasyarat, satu baris per penghapusan. Entri yang berubah tercetak tebal. Delapan mata ajar dikeluarkan dalam delapan putaran, jadi grafnya taksiklik dan urutannya sah. 1037

Tabel 12.30 Graf berbobot G pada Persoalan R, S, U, V, dan W: delapan simpul dan tiga belas sisi. Dua sisi, AB dan FG, sama-sama berbobot 3. 1046

Tabel 12.31 Kruskal pada G, dengan hutan himpunan saling lepasnya sesudah setiap keputusan. Tujuh sisi diterima, lima ditolak, satu tak pernah diperiksa. Baris 11 layak dipelajari: sisinya *ditolak*, namun `parent[G]` berubah dari `F` menjadi `B`, sebab kedua panggilan `Find` memampatkan lintasannya dalam perjalanan menuju jawabannya. 1047

Tabel 12.32 Prim–Jarník dari A, satu baris per pengambilan. Delapan pengambilan, tujuh sisi ditetapkan, total 31. Setiap kunci yang turun adalah sebuah `DecreaseKey`: C dari $5/A$ ke $1/B$ pada langkah 2, D dari $6/B$ ke $4/C$ pada langkah 3, E dari $9/C$ ke $2/D$ pada langkah 4, F dari $8/D$ ke $7/E$ pada langkah 5, G dari $10/E$ ke $3/F$ pada langkah 6, dan H dari $12/F$ ke $11/G$ pada langkah 7. 1049

Tabel 12.33 Prim dengan tumpukan biner dan `DecreaseKey` malas pada graf bersimpul sembilan, dari a. Sembilan pengambilan sungguhan menetapkan delapan sisi dengan bobot total 37; empat pengeluaran lainnya mendapati entri basi — salinan lama g, h (dua kali), dan e yang kuncinya sudah turun sejak itu — dan membuangnya. 1050

Tabel 12.34 Setiap sisi bukan-pohon G, siklus yang ditutupnya bersama pohon merentang minimumnya, dan sisi pohon terberat pada siklus itu. Keenamnya tegas terberat, jadi tak satu pun dapat muncul di pohon merentang minimum mana pun — dan itulah bukti lengkap bahwa pohon Persoalan R satu-satunya. 1056

Tabel 12.35 Tujuh penyatuan menurut peringkat. Peringkat sebuah akar naik hanya ketika dua peringkat yang sama bertemu — ketujuh penyatuan di sini kebetulan semuanya pertemuan semacam itu, dan itulah sebabnya strukturnya tetap seimbang sempurna. 1057

Tabel 12.36 Uji ketunggalan pada G: setiap sisi pohon dihapus bergiliran lalu Kruskal dijalankan ulang. Setiap baris tegas lebih berat daripada 31, jadi pohon merentang minimum G **tunggal**. 1059

Tabel 12.37 Pencarian melebar atas $G1$ dari C, berdampingan dengan pencarian dari A. Setiap simpul berubah jaraknya. Jarak terbesar ke simpul mana pun — **eksentrisitas** sumbernya — adalah 4 dari A dan 3 dari C, jadi C lebih di tengah di antara keduanya dan pohonnya satu tingkat lebih dangkal. 1080

Tabel 13.1 Setiap lintasan sederhana dari s ke v di dalam graf contoh Dijkstra, ditimbang satu sisi setiap kalinya; minimumnya, 9, adalah $\delta(s, v)$, yang dicapai oleh s, x, u, v. Mencacah lintasan memang benar tetapi mustahil dipakai secara umum: graf dengan n simpul dapat memiliki lebih dari $(n - 2)!$ lintasan sederhana antara sepasang simpul. 1093

Tabel 13.2 Keempat persoalan lintasan terpendek. Tiga dari empatnya tereduksi ke yang kedua, dan itulah sebabnya yang kedua itulah yang kita selesaikan. 1094

Tabel 13.3 Bentuk sisi ketaksamaan segitiga diperiksa pada setiap sisi graf contoh Dijkstra. Baris-baris yang menunjukkan *kesamaan* — (s, x), (x, u), (u, v), (x, y) — persis keempat sisi pohon lintasan terpendeknya, sebagaimana diterangkan §13.6.9. 1099

Tabel 13.4 Apa yang dikerjakan masing-masing dari ketiga algoritma itu pada tiap rezim bobot. Kata yang perlu diperhatikan adalah *diam-diam*: Dijkstra tidak macet dan tidak mengeluh pada sisi negatif, ia mengembalikan bilangan salah yang tampak masuk akal. §13.7.3 membangun contoh terkecilnya. 1102

Tabel 13.5 Kelima sifat relaksasi. Semuanya dibuktikan sekali, di sini, lalu dipakai oleh ketiga algoritmanya; tidak ada algoritma dalam bab ini yang memerlukan argumen kebenaran yang tidak tereduksi ke tabel ini. 1107

Tabel 13.6 Jalannya Dijkstra selengkapnya pada graf contoh, satu baris per putaran gelung `while`. Kunci di dalam Q hanya pernah turun, sedangkan kunci yang dikeluarkan, 0,5,7,8,9, tidak pernah turun; §13.7.8 membuktikan fakta yang kedua. 1116

Tabel 13.7 Keluaran Dijkstra pada graf contoh: jarak, pendahulu, dan rute yang diperoleh dengan menyusuri π mundur. Baris d-nya cocok dengan **Tabel 13.1**, yang menemukan $\delta(s, v) = 9$ dengan mendaftarkan enam lintasan. 1116

Tabel 13.8 Waktu jalan Dijkstra untuk tiga implementasi antrean prioritas. Total *binary heap* menyederhana menjadi $O(E\lg V)$ bila setiap simpul terjangkau dari s, sehingga $E \geq V - 1$; baris *Fibonacci heap* berasal dari Fredman dan Tarjan [5]. 1122

Tabel 13.9 Penelusuran melebar, Prim, dan Dijkstra dibandingkan. Satu baris yang memisahkan Prim dari Dijkstra adalah kuncinya: Prim mengukur *satu sisi*, Dijkstra mengukur *seluruh perjalanan*. Itu pula sebabnya Prim tahan terhadap bobot negatif sedangkan Dijkstra tidak — Prim tidak pernah menjumlahkan dua bobot, sehingga ia tidak pernah memerlukan ketaksamaan segitiga. 1127

Tabel 13.10 Jalannya Bellman–Ford selengkapnya pada graf contoh, satu baris per sapuan atas kesepuluh sisinya. Pada sapuan 2, $d[x]$ ditetapkan menjadi 11 lalu menjadi 4 *di dalam sapuan yang sama*; hanya nilai pada akhir tiap sapuan yang diperlihatkan. 1133

Tabel 13.11 Keluaran Bellman–Ford pada graf contoh. Tiga dari keempat rute tak sepelenya memakai y dan sisi negatif (y, x), struktur yang mudah terlewat oleh metode rakus yang memfinalkan simpul terlalu dini. 1133

Tabel 13.12 `DAG-Shortest-Paths` pada **Gambar 13.20**, satu baris per simpul dalam urutan topologis. Bandingkan dengan **Tabel 13.10**: Bellman–Ford memerlukan empat sapuan atas sepuluh sisi — empat puluh relaksasi — untuk menyelesaikan graf selengkapnya. Di sini tujuh relaksasi sudah mencukupi. Perhatikan juga bahwa jawabannya berbeda ($\delta(s, t)$ adalah 6 di sini dan 2 di sana) sebab menghapus (x, t) membuang rute yang murah itu; keduanya adalah graf yang berbeda, bukan dua jawaban atas pertanyaan yang sama. 1142

Tabel 13.13 Algoritma sumber tunggal mana yang dipakai. Bacalah dari atas ke bawah: tanyakan lebih dahulu apakah grafnya asiklik, lalu apakah ada bobot yang negatif, lalu seberapa padat ia.

Kedua pertanyaan pertama adalah tentang *kebenaran* dan tidak menerima kompromi; hanya yang ketiga yang tentang kecepatan. 1149
Tabel 14.1 Nilai-nilai $Ak(1)$ yang mendefinisikan α. Lompatan dari $k = 3$ ke $k = 4$ bukan lompatan dalam ukuran, melainkan lompatan dalam *jenis*: 2047 adalah bilangan yang dapat Anda tuliskan, sedangkan $A4(1)$ adalah bilangan yang uraian desimalnya tidak akan muat di alam semesta teramati. 1229
Tabel 14.2 Fungsi Ackermann balikan. Agar $\alpha(n)$ mencapai 5, Anda memerlukan lebih dari $A4(1) >$ 22048 elemen. 1229
Tabel 14.3 Ketiga metodenya dibandingkan. Ketiganya setara kekuatannya untuk contoh-contoh di dalam bab ini; bedanya terletak pada berapa banyak kerja yang dituntut masing-masing dan berapa banyak yang diceritakannya kepada Anda. 1235
Tabel 15.1 Biaya menyelesaikan lintasan terpendek semua pasangan dengan menjalankan algoritma sumber tunggal sekali per simpul. Baris terakhirnyalah yang menyakitkan, dan baris itulah yang diserang sisa bab ini. 1273
Tabel 15.2 EXTEND-SHORTEST-PATHS adalah SQUARE-MATRIX-MULTIPLY dengan min menggantikan $+$, $+$ menggantikan $\times$, dan ∞ menggantikan 0 sebagai nilai awalnya. Selebihnya — struktur gelungnya, pola indeksnya, waktu jalannya — sama persis. 1280
Tabel 15.3 Kedua jadwalnya pada contoh berjalan bersimpul lima. Versi lambatnya menambahkan satu pada jatah sisinya tiap hasil kali; versi cepatnya melipatduakannya. 1292
Tabel 15.4 Setiap algoritma di bab ini, beserta ongkosnya dan kasus yang diperuntukkan baginya. Kedua yang dicetak tebal adalah kedua yang sungguh akan Anda pakai. 1331
Tabel 15.5 Sebuah prosedur pengambilan keputusan bagi persoalan lintasan terpendek semua pasangan. Bekerjalah menurun; "ya" yang pertama adalah jawaban Anda. 1334
Tabel 16.1 Satu gagasan per bab, dan di mana bab ini memakainya. Jika Anda dapat menyebutkan keenam belasnya di luar kepala, seluruh buku ada di tangan Anda. 1354
Tabel 16.2 Kelima belas jawaban Persoalan 16.1, berikut pembenaran satu baris untuk masing-masingnya. Sembilan benar dan enam salah. 1358
Tabel 16.3 Tabel pemrograman dinamis untuk $S = 9,2,5,3,7,11,8,10,13,6$. $L[i]$ adalah panjang subbarisan menaik terpanjang yang berakhir di $S[i]$; $P[i]$ adalah indeks unsur sebelumnya. 1368
Tabel 16.4 Biaya setiap langkah algoritma hampiran pada graf lengkap dengan n simpul. Pohon merentang minimumnya mendominasi. 1378
Tabel 16.5 Jarak Manhattan dalam blok antara enam titik jemput Profitina Laundry. Matriksnya simetris, diagonalnya nol, dan memenuhi ketaksamaan segitiga — tiga syarat yang menjadikannya metrik. 1381
Tabel 16.6 Dijkstra berhadapan dengan Bellman–Ford. Dua baris terakhir menjelaskan pembagian peran di dunia nyata: *router* memakai Bellman–Ford karena ia lokal; aplikasi peta memakai Dijkstra karena ia cepat dan bobot jalan tidak pernah negatif. 1386
Tabel 16.7 Empat putaran Bellman–Ford pada graf **Gambar 16.7**. Perhatikan bahwa $d[t]$ *turun* dari 6 ke 2 pada putaran 2 — taksiran yang oleh Dijkstra sudah dituntaskan — dan bahwa putaran 4 tidak mengubah apa pun, yang lalu dikukuhkan oleh pemeriksaan penutupnya. 1390
Tabel 16.8 Algoritma Dijkstra pada graf Persoalan 16.6, satu baris per `Extract-Min`. Setiap sel adalah $d[v]/\pi[v]$; sel tebal adalah simpul yang dituntaskan pada baris itu. 1397
Tabel 16.9 Bellman–Ford pada graf Persoalan 16.7(a), dengan $d[v]/\pi[v]$ sesudah setiap putaran. Taksiran-taksirannya mapan sesudah putaran 2; putaran 3 dan 4 tidak mengubah apa pun dan pemeriksaan penutupnya mengembalikan TRUE. 1400
Tabel 16.10 Bellman–Ford pada graf Persoalan 16.7(b). Ketiga taksiran pada siklus itu turun 1 pada setiap putaran, dan pemeriksaan penutupnya menemukan bahwa (b, c) masih dapat direlaksasi, sebab $3 + (-2) = 1 < 2$. 1401
Tabel 16.11 Algoritma DAG pada Persoalan 16.8, satu baris per simpul yang diproses dalam urutan topologis. Nilai setiap simpul sudah tuntas pada saat ia diproses, sebab semua sisi masuknya berasal dari simpul yang diproses lebih awal. 1405
Tabel 16.12 Pencacah biner dengan ketiga metode untuk delapan panggilan pertama. Jumlah berjalan $C(i)$ tetap di bawah $2i$ (agregat), tabungannya tidak pernah negatif dan selalu sama dengan banyaknya bit 1 (akuntansi), dan setiap biaya amortisasi tepat 2 (potensial). 1411
Tabel 16.13 Tabel dinamis Persoalan 16.10 dengan ketiga metode untuk sembilan penyisipan pertama. Totalnya tetap di bawah $3i$, tabungannya tetap positif, dan biaya amortisasinya 3 sejak penyisipan kedua — termasuk pada penyisipan mahal $i = 2,3,5,9$. 1415

Tabel 16.14 Matriks-matriks awal Persoalan 16.11: $D(0)$ adalah matriks bobot, dan $\Pi(0)$ mencatat ekor setiap sisi. 1418
Tabel 16.15 Setiap entri yang berubah selama Floyd–Warshall pada Persoalan 16.11, berikut perbandingan yang mengubahnya. Perhatikan bahwa $d32$ berubah dua kali: rute yang ditemukan pada ronde 1 dikalahkan oleh rute yang ditemukan pada ronde 4. 1419
Tabel 16.16 Pembobotan ulang Johnson pada graf Persoalan 16.12. Setiap sisi yang dibobot ulang tak negatif, dan sisi negatif $2 \to 3$ telah menjadi sisi berbobot 0. 1423
Tabel 16.17 Prim dan Kruskal pada graf enam simpul **Tabel 16.5**. Prim menambahkan $F - E$ lalu $E - D$; Kruskal menambahkan $D - E$ lalu $E - F$. *Urutannya* berbeda; *himpunan* sisinya AF, AB, BC, DE, EF dan bobotnya 22 tidak. 1442
Tabel 16.18 Kedua puluh satu persoalan bab ini sekilas: jenis masing-masing, alat yang menyelesaikannya dan di mana buku ini mengajarkannya, dan biaya penyelesaiannya. 1443
Tabel 16.19 Tabel pemilihan algoritma. Bacalah kolom kiri sampai sebuah baris cocok dengan persoalan di depan Anda; tiga kolom lainnya adalah jawabannya, pembenaran yang akan diminta dari Anda, dan anggaran yang harus Anda penuhi. 1447
Tabel 16.20 Tabel-tabel untuk Soal Latihan 2. Maksimumnya $L[8] = 4$, jadi penelusuran baliknya dimulai pada $k = 8$. 1459

Daftar Kode

Kode 1.1 Insertion sort dalam C++, utuh dan dapat dijalankan, mengurutkan barisan dari **Gambar 1.2**. 7
Kode 1.2 Versi penelusuran dari insertion sort yang mencetak invarian gelung sesudah setiap putaran. 8
Kode 1.3 Paradigma inkremental dalam wujudnya yang paling sederhana: satu peubah berjalan, satu kali pindai. 11
Kode 1.4 Merge sort: belah jadi dua, urutkan tiap belahan, gabungkan. Paradigma *divide and conquer* dalam dua puluh baris. 12
Kode 1.5 Rekursi naif dibandingkan rekursi bermemoisasi untuk bilangan Fibonacci. Pencacah pemanggilan membuat selisihnya mustahil dibantah. 13
Kode 1.6 Faktor persekutuan terbesar dihitung dua kali: dengan pencarian menyeluruh, dan dengan algoritma Euclid. 15
Kode 1.7 Pemeriksa *brute force* untuk pewarnaan yang dibangun dalam pembuktian *Columns*. 21
Kode 1.8 Penyelesaian lengkap untuk *Columns*, E-Olymp 10281. 23
Kode 1.9 Penyelesaian lengkap untuk *Journey*, E-Olymp 10280. 26
Kode 1.10 Penyelesaian lengkap untuk *Unique Numbers*, SPOJ DCEPC901. 30
Kode 1.11 Penyelesaian lengkap untuk *Minimum and maximum numbers*, SPOJ NEWLANG. 35
Kode 1.12 Membangun *Interesting sequence* secara *brute force* dan menguji rumus popcount terhadap setiap sukunya. 39
Kode 1.13 Penyelesaian lengkap untuk *Interesting sequence*, E-Olymp 4894. 40
Kode 1.14 Menghitung sub-persoalan berbeda dari f dengan menjelajahi rekursinya memakai sebuah `set`. 43
Kode 1.15 Penyelesaian lengkap untuk *Recursive function 1*, E-Olymp 10296. 45
Kode 1.16 Program pertama: bukti bahwa kompilator, editor, dan terminal sudah bekerja bersama. 56
Kode 1.17 Perencana Profitina Laundry: perjalanan pulang-pergi terpendek bagi mobil antar dan banyaknya label rak yang diperlukan. 59
Kode 2.1 Insertion Sort dalam C++, dengan pengindeksan berbasis-1 agar kodenya persis sejajar dengan pseudokodenya. 83
Kode 2.2 Insertion Sort yang dilengkapi pencatat agar mencetak telusurnya sendiri, satu baris untuk tiap iterasi gelung luar. 84
Kode 2.3 Menghitung $j = 2ntj$ untuk masukan menaik, menurun, dan acak, bersanding dengan nilai kasus terburuk yang diramalkan. 95

Kode 2.4 Mengukur waktu kasus terburuk Insertion Sort pada lima ukuran masukan, masing-masing dua kali lipat ukuran sebelumnya. Kolom `ratio` adalah pengganda biaya yang teramati ketika n dilipatduakan. 97
Kode 2.5 Menghitung perbandingan laju pertumbuhan, termasuk 2106, tanpa pernah menyimpan bilangan sebesar itu. 100
Kode 2.6 Linear search dalam C++, pada data yang sama dengan **Gambar 2.14**. 105
Kode 2.7 Binary search dalam C++, bentuk iteratif, dilengkapi pencatat agar mencetak jendelanya pada tiap langkah. 110
Kode 2.8 Mengapa titik tengah harus dihitung sebagai `left + (right - left) / 2`. 111
Kode 2.9 Binary search sebagai prosedur divide-and-conquer rekursif, yang mencetak jendelanya pada tiap aras rekursi. 112
Kode 2.10 Buku loket Profitina Laundry: penyisipan inkremental untuk menjaganya tetap terurut, binary search untuk menemukan sebuah nota. 117
Kode 2.11 Insertion Sort dengan pengindeksan berbasis-0 yang biasa, tanpa elemen yang dibiarkan kosong. 124
Kode 2.12 Memverifikasi bahwa cacah penggeseran Insertion Sort sama dengan cacah inversi pada masukannya. 125
Kode 2.13 Binary search untuk kemunculan pertama dan terakhir sebuah nilai, ketika ada nilai kembar. 127
Kode 3.1 Sebuah program C++ yang menabelkan fungsi-fungsi pertumbuhan baku. Simpan sebagai `growth.cpp`, kompilasi dengan `g++ -std=c++17 -O2 -o growth growth.cpp`, lalu jalankan `./growth`. 134
Kode 3.2 Memburu secara numerik konstanta c dan $n0$ dalam definisi Big-O. 138
Kode 3.3 Logaritma teriterasi, hampiran Stirling, dan bentuk tertutup Fibonacci, dihitung dan dibandingkan. 150
Kode 3.4 Penjumlahan brutal diperiksa terhadap setiap bentuk tertutup yang diturunkan di §3.9. 158
Kode 3.5 Menghitung dua rekurensi secara eksak, lalu membandingkannya dengan batas yang dibuktikan lewat substitusi. 165
Kode 3.6 Menjumlahkan pohon rekursi tingkat demi tingkat untuk ketiga bentuk pada **Tabel 3.6**, sekaligus memeriksa kesamaan cacah daun. 172
Kode 3.7 Sebuah penyelesai Teorema Master dalam C++, diterapkan pada delapan belas rekurensi. 184
Kode 3.8 Merge Sort dalam C++, dengan penelusuran tercetak dan cacah pembandingan terukur. 190
Kode 3.9 Profitina Laundry: Merge Sort menurut waktu pengambilan yang dijanjikan, sambil mencacah inversi dalam lintasan $\Theta(n\lg n)$ yang sama. 195
Kode 4.1 LDCSUMAB — membalikkan peringkat barisan $2m + 2n$, dalam $O(1)$ per pertanyaan sesudah akar kuadratnya. 215
Kode 4.2 LDCSUMAB dengan pencarian biner atas nomor blok, $\Theta(\lg 50000) \approx 16$ langkah per pertanyaan, tanpa titik-mengambang sama sekali. 217
Kode 4.3 Pemeriksaan silang gaya kasar untuk LDCSUMAB: daftarnya dibangun dan diurutkan secara eksplisit, lalu dibandingkan dengan rumus pada kedelapan ratus dua puluh posisi. 217
Kode 4.4 TOHU — Help Tohu, dalam $O(1)$ per kasus uji, tanpa kasus khusus untuk $n = 1$. 225
Kode 4.5 TOHU dari prinsip pertama: tiap ak dipulihkan secara eksak dari syarat jumlah berbobot, tanpa mengandaikan rumus apa pun, lalu dibandingkan dengan bentuk tertutupnya. 226
Kode 4.6 Aturan Horner terhadap evaluasi naif, dengan penelusuran gelung dan hitungan perkalian. 230
Kode 4.7 *Insertion sort* dengan penelusuran per iterasi, dan hitungan inversi terpisah sebagai pembanding. 232
Kode 4.8 Pencarian biner dengan satu baris keluaran per probe, pada larik Persoalan Terkerjakan C. 235
Kode 4.9 Menghitung eksekusi baris terdalam LOOPS dan membandingkannya dengan $n(n+1)(n+2)6$. 238
Kode 4.10 Pemeriksaan numerik saksi $c1$, $c2$, $n0$ untuk klaim (a) dan (d), serta titik lolos pada klaim (c). 240
Kode 4.11 Logaritma biner ketiga belas fungsi pada empat ukuran, yang memperlihatkan di mana urutan asimtotik sudah dan belum berlaku. 242
Kode 4.12 Kedua rekurensi pada Persoalan Terkerjakan G dievaluasi secara eksak, dengan batas-batas yang telah dibuktikan diperiksa untuk setiap $n \leq 106$. 245

Kode 4.13 Pohon rekursi $T(n) = 3T(n/4) + n2$ pada $n = 256$, tingkat demi tingkat, dan rasio $T(n)/n2$ seiring bertambahnya n. 248
Kode 4.14 Keempat rekurensi pada Persoalan Terkerjakan I dievaluasi secara eksak lalu dibagi dengan batas yang diramalkan. 251
Kode 4.15 LAUNDRT — Kupon Laundry, diselesaikan dalam $O(1)$ secara teleskopik. 253
Kode 4.16 STRIPTILE — penubinan lajur $2 \times N$ dengan pelipatduaan cepat, $\Theta(\log N)$ per pertanyaan. 256
Kode 4.17 Gaya kasar melawan urutkan-dan-cari untuk persoalan jumlah pasangan, diukur waktunya pada masukan yang ukurannya berlipat dua. 259
Kode 4.18 Profitina Laundry — bentangan hari berurutan yang paling menguntungkan, dengan metode minimum awalan, beserta aturan pemutus seri yang diperagakan pada himpunan data kedua 264
Kode 5.1 Setiap fakta struktural tentang *heap* dari Subbab 5.3, dihitung dan diperiksa. 283
Kode 5.2 `MAX-HEAPIFY` dalam C++, yang menarasikan setiap keputusannya sendiri. 291
Kode 5.3 `BUILD-MAX-HEAP`, ditelusuri pada larik dari Subbab 5.5.3 lalu diukur pada larik acak sampai sepuluh juta elemen. 301
Kode 5.4 Heapsort lengkap yang dapat dijalankan, yang menarasikan setiap pengeluarannya. 309
Kode 5.5 Antrean prioritas-maks yang lengkap, dengan keempat operasinya dilatih. 314
Kode 5.6 *Heap* dari pustaka baku C++: `std::priority_queue`, pembanding buatan sendiri, dan algoritma mentah `make_heap` / `sort_heap`. 316
Kode 5.7 Skema pemartisian Lomuto dan Hoare, masing-masing menarasikan langkahnya sendiri. Indeks di sini berbasis 0, jadi kurangilah satu dari setiap indeks pada penelusuran tangan di atas. 323
Kode 5.8 Quicksort dengan pemartisian Lomuto, yang menarasikan rekursinya. Indentasinya menunjukkan kedalaman pemanggilannya. 326
Kode 5.9 Mencacah pembandingan Quicksort pada masukan terurut dan teracak, dengan dan tanpa *pivot* acak, terhadap kedua rumus teoretisnya. 337
Kode 5.10 Mengukur kedalaman rekursi Quicksort naif dan Quicksort rekursi-ekor pada masukan terurut dan teracak 341
Kode 5.11 Batas pohon keputusan yang dihitung untuk n kecil: banyaknya daun yang diperlukan, dan tinggi minimum yang dihasilkannya. 345
Kode 5.12 Sebuah uji kestabilan. Tiap rekaman membawa sebuah tanda yang menunjukkan dari mana ia berangkat, sehingga penataan ulang di antara kunci yang sama menjadi kasatmata. 348
Kode 5.13 Adu langsung keempat pengurutan pada data acak, dengan tiap pengurutan diberi salinan yang segar dan keluarannya diperiksa. 351
Kode 5.14 Profitina Laundry: sebuah *heap* memutuskan urutan mencucinya, dan Quicksort menghasilkan laporan penerimaannya. 356
Kode 6.1 Simpul sebuah pohon pencarian biner, dan pohon **Gambar 6.1** yang dirakit dengan tangan dari tujuh simpul. Kompilasi dengan `g++ -std=c++17 -O2 -o node node.cpp`. 376
Kode 6.2 Ketiga penelusuran dalam C++. Simpan sebagai `traversal.cpp`, kompilasi dengan `g++ -std=c++17 -O2 -o traversal traversal.cpp`, lalu jalankan `./traversal`. 378
Kode 6.3 Pencarian, minimum, maksimum, penerus, dan pendahulu, masing-masing melaporkan lintasan yang ditempuhnya. Kompilasi dengan `g++ -std=c++17 -O2 -o query query.cpp`. 384
Kode 6.4 Pohon pencarian biner yang lengkap. Kompilasi dengan `g++ -std=c++17 -O2 -o bst bst.cpp` lalu jalankan `./bst`. 392
Kode 6.5 Mengukur apa yang dilakukan urutan penyisipan terhadap sebuah pohon pencarian biner. Kompilasi dengan `g++ -std=c++17 -O2 -o degenerate degenerate.cpp`. 395
Kode 6.6 Rekurensi simpul-minimum, kesamaan Fibonacci, dan batas tingginya, semuanya diperiksa secara numerik. Kompilasi dengan `g++ -std=c++17 -O2 -o minnodes minnodes.cpp`. 404
Kode 6.7 Rotasi kanan beserta pembaruan penunjuknya yang persis, diterapkan pada rantai yang condong ke kiri 30 → 20 → 10. Kompilasi dengan `g++ -std=c++17 -O2 -o rotright rotright.cpp`. 406
Kode 6.8 Rotasi kiri, diterapkan pada rantai merosot 10 → 20 → 30. Kompilasi dengan `g++ -std=c++17 -O2 -o rotleft rotleft.cpp`. 408

Kode 6.9 Pohon AVL yang lengkap. Kompilasi dengan `g++ -std=c++17 -O2 -o avl avl.cpp` lalu jalankan `./avl`. 420
Kode 6.10 Satu penyisipan tidak pernah berongkos lebih daripada satu penyeimbangan ulang; satu penghapusan bisa berongkos satu penyeimbangan ulang di setiap aras. Kompilasi dengan `g++ -std=c++17 -O2 -o cascade cascade.cpp` 429
Kode 6.11 Mengukur pohon AVL terhadap BST biasa, pohon merah-hitam, dan tabel hash. Kompilasi dengan `g++ -std=c++17 -O2 -o compare compare.cpp`. 435
Kode 6.12 Papan pengambilan Profitina Laundry: pohon AVL yang berkunci waktu pengambilan yang dijanjikan, mendukung kueri penerus, kueri rentang, dan penghapusan. Kompilasi dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp` 441
Kode 6.13 Cara yang salah dan cara yang benar untuk memverifikasi sifat pohon pencarian biner. Kompilasi dengan `g++ -std=c++17 -O2 -o isbst isbst.cpp`. 446
Kode 6.14 Kunci terkecil ke-k dari sebuah pohon pencarian biner dalam waktu $O(h + k)$. Kompilasi dengan `g++ -std=c++17 -O2 -o kth kth.cpp` 456
Kode 6.15 Pohon statistik terurut: pohon AVL **Kode 6.9** dengan ukuran subpohon di setiap simpul, yang mendukung `select` dan `rank` dalam $O(\lg n)$. Kompilasi dengan `g++ -std=c++17 -O2 -o ostree ostree.cpp`. 457
Kode 7.1 Rekursi Fibonacci naif, lengkap dengan alat ukurnya. Simpan sebagai `fib_naive.cpp`, kompilasi dengan `g++ -std=c++17 -O2 -o fib_naive fib_naive.cpp`, lalu jalankan `./fib_naive` 469
Kode 7.2 Fibonacci ber-*memoization*, *bottom-up*, beruang tetap, dan berpangkat matriks, keempatnya diadu waktunya satu sama lain. Kompilasi dengan `g++ -std=c++17 -O2 -o fib_fast fib_fast.cpp` 475
Kode 7.3 Pencacahan *brute force* atas setiap pengelompokan tanda kurung, dipakai sebagai kebenaran acuan untuk memeriksa hasil program dinamisnya. Kompilasi dengan `g++ -std=c++17 -O2 -o brute brute.cpp`. 485
Kode 7.4 Matrix-Chain-Order dan **Print-Optimal-Parens** dalam C++, lengkap dengan jejak penuh setiap calon belahan. Kompilasi dengan `g++ -std=c++17 -O2 -o mcm mcm.cpp` lalu jalankan `./mcm` 494
Kode 7.5 Rekurensi deretan matriks yang sama diimplementasikan tiga cara — rekursi polos, rekursi ber-*memoization*, tabel *bottom-up* — saling diperiksa satu sama lain dan diukur waktunya. Kompilasi dengan `g++ -std=c++17 -O2 -o mcm_compare mcm_compare.cpp` 499
Kode 7.6 LCS-Length, tabel panahnya, rekonstruksinya, dan varian dua barisnya, lengkap dengan perbandingan waktu dan memori. Kompilasi dengan `g++ -std=c++17 -O2 -o lcs lcs.cpp`. 514
Kode 7.7 Mencacah panggilan melawan submasalah berbeda untuk Merge Sort, Fibonacci naif, dan rekursi deretan matriks yang polos — sebuah uji empiris untuk *overlapping subproblems*. Kompilasi dengan `g++ -std=c++17 -O2 -o overlap overlap.cpp`. 518
Kode 7.8 Profitina Laundry: mencocokkan slip tulisan tangan dengan daftar harga memakai kemiripan LCS, dalam ruang $\Theta(\min(m, n))$. Kompilasi dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp`. 522
Kode 7.9 Program dinamis pengemasan kantong Profitina Laundry dari Soal 7.18, lengkap dengan perbandingan *greedy*. Kompilasi dengan `g++ -std=c++17 -O2 -o bags bags.cpp`. 536
Kode 8.1 Soal 8.1 selengkapnya: `MAX-HEAPIFY` dan `BUILD-MAX-HEAP` dengan pelacakan panggilan demi panggilan, *heapsort* dengan lariknya setelah setiap pengambilan, dan pengukuran batas liniernya. 547
Kode 8.2 Soal 8.2 selengkapnya: partisi Lomuto yang dilacak, rekursi lengkap dengan hitungan perbandingan per panggilan, pembangkit masukan kasus terbaik, dan pengukuran kasus terbaik, terburuk, dan acak. 555
Kode 8.3 Tipe `MyTree` dalam C++, dengan satu-satunya konstruktor `node()` yang menjaga tinggi tersimpannya tetap jujur, dan sebuah pencetak bentuk yang menuliskan seluruh pohon dalam satu baris. 561
Kode 8.4 Soal 8.3 selengkapnya: jawaban selang-terbatas, pemeriksaan lokal yang gagal, cara kedua lewat penelusuran inorder, dan sebuah diagnostik yang menyebut simpul pertama yang bersalah. 565

Kode 8.5 Soal 8.4 selengkapnya: jawaban rekursif empat baris, penelusuran menaik di sampingnya, jawaban iteratif dengan tumpukan eksplisit, dan sebuah penelusuran yang memastikan kedalaman tumpukan $O(h)$. 573
Kode 8.6 Soal 8.5 selengkapnya, dengan pencacah kunjungan simpul yang membuat beda antara cara yang diizinkan dan cara yang dilarang menjadi terukur. 577
Kode 8.7 Soal 8.6 selengkapnya: cara yang boros, cara satu-lintasan dengan penjaga, rutin pelabelan tinggi yang memang kuadrat, dan perbandingan terukur atas sebelas bentuk pohon. 585
Kode 8.8 Soal 8.7 sampai 8.10 dalam satu program: `insertHB`, `rebalanceForLeft`, `rebalanceForRight`, `deleteHB`, ditambah pemeriksaan-diri AVL, pemeriksaan keterurutan, dan uji tekanan terhadap batas tinggi teoretisnya. 603
Kode 8.9 Soal 8.11 sampai 8.15 dalam satu program: pemeriksa-pemeriksa merah-hitam, rotasi imperatif dengan penunjuk induk, serangkaian pemeriksaan, dan peragaan jebakan batas `int` pada §8.6.7. 611
Kode 8.10 Soal 8.16 selengkapnya: `MATRIX-CHAIN-ORDER` dengan pelacakan setiap calon pemisah, `PRINT-OPTIMAL-PARENS`, dan hitungan panggilan rekursi polos dan rekursi bermemo. 622
Kode 8.11 Soal 8.17 selengkapnya: `LCS-LENGTH` dengan pelacakan setiap sel, tabel gabungan nilai dan panah, dan jalan mundur yang menghasilkan satu subbarisan bersama terpanjang. 630
Kode 8.12 Penyelesaian lengkap Soal 8.18 sampai 8.25, dengan perusakan sengaja pada satu medan tersimpan untuk memperlihatkan pemeriksa Soal 8.18 menangkapnya, serta uji tekanan dua ribu elemen. 637
Kode 8.13 Indeks pengambilan Profitina Laundry, dibangun seluruhnya dari kedelapan penyelesaian soal pohon bab ini, dengan audit-diri yang menyebut rekaman yang bercacat dan pembangunan ulang yang selamat dari urutan yang rusak. 645
Kode 8.14 Penyelesaian yang dapat dijalankan untuk soal latihan 4, 5, 7, dan 8, dengan sakelar `STRICT` yang mereproduksi kutu `< 0` soal latihan 8 sesuai permintaan. 657
Kode 9.1 Mengaudit aturan rakus uang kembalian terhadap optimum sejatinya, untuk enam sistem koin dan setiap nilai sampai 200. Kompilasi dengan `g++ -std=c++17 -O2 -o coins coins.cpp` lalu jalankan `./coins`. 669
Kode 9.2 Pemilihan aktivitas secara utuh: pemilih rakus rekursif dan iteratif, pemeriksaan *brute force*, keempat calon aturan berikut contoh sanggahan bagi ketiga yang keliru, dan satu jalannya pengukuran waktu. Kompilasi dengan `g++ -std=c++17 -O2 -o activity activity.cpp` lalu jalankan `./activity`. 688
Kode 9.3 Dua masalah ransel: rakus menurut kerapatan (optimal bagi persoalan pecahan, keliru bagi persoalan 0-1), optimum 0-1 dengan pemrograman dinamis, *brute force* sebagai pemeriksa, dan uji beban atas 120 000 contoh kasus acak. Kompilasi dengan `g++ -std=c++17 -O2 -o knapsack knapsack.cpp` lalu jalankan `./knapsack`. 699
Kode 9.4 Penyandi dan pengurai sandi Huffman yang lengkap: pembangunan yang ditelusuri atas frekuensi **Gambar 9.7**, buku sandinya, $B(T)$ yang dihitung dua cara yang saling bebas, perjalanan bolak-balik terpaket-bit yang sungguhan atas teks biasa, kasus tepi satu lambang, dan satu jalannya pengukuran waktu. Kompilasi dengan `g++ -std=c++17 -O2 -o huffman huffman.cpp` lalu jalankan `./huffman` 715
Kode 9.5 Greedy(M, w) ditulis sekali terhadap orakel kebebasan yang abstrak, lalu diterapkan pada matroid penjadwalan tugas (dengan pemeriksaan *brute force* dan uji beban atas 200 000 contoh kasus) dan pada matroid grafik, tempat ia menjelma menjadi algoritma Kruskal. Kompilasi dengan `g++ -std=c++17 -O2 -o matroid matroid.cpp` lalu jalankan `./matroid`. 731
Kode 9.6 Profitina Laundry: pemilihan aktivitas atas pemesanan pengeringnya (dengan pemeriksaan *brute force*), pemampatan Huffman atas log status hari itu dengan perjalanan bolak-balik yang terverifikasi, dan van pengantarannya sebagai ransel 0-1 tempat kerakusan kehilangan 10% pendapatannya. Kompilasi dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp` lalu jalankan `./laundry` 739
Kode 9.7 Program-program bagi Soal Latihan 9.7, 9.10, dan 9.15: pekerjaan-terpendek-dahulu yang diperiksa terhadap seluruh $n!$ urutan, Huffman atas frekuensi Fibonacci, dan pemilihan aktivitas berbobot tempat aturan rakusnya gagal. Kompilasi dengan `g++ -std=c++17 -O2 -o exercises exercises.cpp` lalu jalankan `./exercises` 758
Kode 10.1 Graf bab ini disimpan dengan tiga cara sekaligus, dengan keempat kueri dasarnya dijawab dari masing-masing representasi. Kompilasilah dengan `g++ -std=c++17 -O2`. 789

Kode 10.2 Penelusuran melebar dalam C++, yang mencetak antreannya pada setiap langkah. Daftar ketetanggaannya dituliskan secara eksplisit alih-alih dibangun dengan `addEdge`, supaya *urutannya* persis sama dengan **Gambar 10.16** sampai **Gambar 10.19** dan penelusuran yang tercetak dapat dibandingkan dengan **Tabel 10.7** baris demi baris. 801
Kode 10.3 Penelusuran mendalam dengan tumpukan eksplisit, yang mencetak tumpukannya pada setiap langkah. Struktur `Frame` membawa posisi lanjutan `next` — tanpanya, kembali ke sebuah simpul akan memulai ulang daftar ketetanggaannya dan penelusurannya akan berputar-putar. 817
Kode 10.4 Penelusuran mendalam, rekursif, pada contoh §10.8.4. Ia mencetak penelusurannya, pencap waktunya, penandakurungan §10.8.8, hutannya, dan penggolongan setiap busurnya. 820
Kode 10.5 Contoh DFS kedua dijalankan selengkapnya, untuk memeriksa pencap waktu dan penandakurungan **Gambar 10.25** dan **Gambar 10.26**. 825
Kode 10.6 Komponen terhubung lewat penelusuran melebar berulang, pada graf yang berbentuk seperti **Gambar 10.9**. Begitu `comp[]` terbangun, "apakah u dan v terhubung?" menjadi satu pembandingan bilangan bulat. 832
Kode 10.7 Deteksi siklus untuk graf tak berarah maupun berarah, diuji pada tujuh graf termasuk contoh-contoh bab ini sendiri. Uji tak berarahnya menerima sembarang tetangga bukan putih selain induknya; uji berarahnya bersikeras pada warna **kelabu**. 834
Kode 10.8 Pengujian kebipartitan lewat pewarnaan dua warna selama sebuah BFS. Ketika berhasil ia mencetak kedua kubunya, dan itulah sertifikatnya; ketika gagal ia menyebut sisi yang tidak dapat diwarnai, dan sisi itu menunjuk pada siklus ganjil. 838
Kode 10.9 Pengurutan topologis DAG berpakaian lewat penelusuran mendalam, disusul lintasan pemeriksaan yang mengecek bahwa setiap busur menunjuk maju dalam urutan yang dihasilkan. 844
Kode 10.10 Mengukur waktu BFS atas daftar ketetanggaan melawan BFS atas matriks ketetanggaan pada graf yang sama. Sebuah pohon merentang dibangun lebih dulu, sehingga grafnya pasti terhubung dan kedua penelusurannya harus mencapai seluruh V simpul — dan programnya memeriksa hal itu sebelum melaporkan waktu apa pun. 848
Kode 10.11 Studi kasus Profitina Laundry: penelusuran melebar pada jaringan penjemputan antarkelurahan, dan pengurutan topologis pada DAG proses pencucian, beserta apa yang terjadi ketika spesifikasi prosesnya dibuat bersiklus. 853
Kode 11.1 Pengurutan topologis memakai waktu penyelesaian mendalam-dahulu, pada DAG berpakaian. Kompilasilah dengan `g++ -std=c++17 -O2 -o toposort toposort.cpp`. 883
Kode 11.2 Pengurutan topologis derajat masuk dari Kahn, berikut uji siklus yang cuma-cuma. Kompilasilah dengan `g++ -std=c++17 -O2 -o kahn kahn.cpp`. 888
Kode 11.3 Prim–Jarník dengan tumpukan biner, menelusuri antreannya pada tiap langkah. Kompilasilah dengan `g++ -std=c++17 -O2 -o prim prim.cpp`. 911
Kode 11.4 Sebuah hutan himpunan saling lepas dengan gabung menurut peringkat dan pemampatan lintasan. Kompilasilah dengan `g++ -std=c++17 -O2 -o dsu dsu.cpp`. 920
Kode 11.5 Algoritma Kruskal dengan hutan himpunan saling lepas yang benar, menelusuri partisinya sebelum tiap keputusan. Kompilasilah dengan `g++ -std=c++17 -O2 -o kruskal kruskal.cpp`. 931
Kode 11.6 Prim dan Kruskal diukur satu terhadap yang lain, termasuk uji kesepakatan teracak atas seribu graf. Kompilasilah dengan `g++ -std=c++17 -O2 -o compare compare.cpp`. 934
Kode 11.7 Studi kasus Profitina Laundry: sebuah pohon merentang minimum atas sepuluh cabang, dan sebuah pengurutan topologis berikut lintasan kritis atas sembilan tahap order. Kompilasilah dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp`. 942
Kode 11.8 Uji berarah yang rusak dan uji tak berarah yang sudah diperbaiki, diukur terhadap tujuh graf yang jawabannya sudah diketahui. Kompilasilah dengan `g++ -std=c++17 -O2 -o undirected undirected.cpp`. 951
Kode 12.1 Rute Laundry: pencarian melebar dan komponen, Kruskal dengan hutan himpunan saling lepas, dan pengurutan topologis Kahn beserta lintasan kritisnya. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o router router.cpp` lalu jalankan `./router`. 968
Kode 12.2 Perkakas evaluasinya: pemeriksaan silang Kruskal terhadap Prim, pemeriksaan pascakondisi pada BFS, dan satu putaran pengukuran waktu pada lima ukuran dengan waktu berlalunya dibagi biaya yang diramalkan. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o routereval routereval.cpp` lalu jalankan `./routereval`. 975

Kode 12.3 Penukaran uang secara rakus dibandingkan dengan optimum pemrograman dinamis, untuk setiap jumlah dari 1 sampai 30 dengan token 1,7,10, disusul dua audit ringkasan. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o change change.cpp`. 983

Kode 12.4 Algoritma rakus matroid untuk tugas berwaktu satuan dengan tenggat dan pengembalian uang, dengan cacah kebebasannya dicetak di setiap langkah dan pemeriksaan gaya kasar atas seluruh 128 upahimpunan. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o tasks tasks.cpp`. 1000

Kode 12.5 Persoalan Terkerjakan B sampai E, dihitung: pemilihan aktivitas dengan empat aturan terhadap gaya kasar; tenggat terawal dahulu terhadap seluruh 720 urutan; Huffman beserta pohonnya, $B(T)$ dengan kedua cara, dan perjalanan pulang-pergi; serta *knapsack*, pecahan dan 0-1. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp` lalu jalankan `./greedy`. 1004

Kode 12.6 Pencarian mendalam atas graf tak berarah, mencetak setiap penemuan, setiap penyelesaian, dan setiap sisi mundur menurut urutan jam, lalu tabel d/f, kedua golongan sisinya, dan struktur kurungnya; dijalankan dari A dan dari E. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o dfsu dfsu.cpp`. 1020

Kode 12.7 Persoalan Terkerjakan G, H, J, K, dan L, dihitung: penelusuran BFS, cap waktu dan penggolongan DFS, komponennya, pewarnaan dua warna beserta sertifikat siklus ganjilnya, dan kedua uji siklus. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o search search.cpp`. 1027

Kode 12.8 Persoalan Terkerjakan M sampai Q: cap waktu dan penggolongan pencarian mendalam, kedua pengurutan topologis, cacah urutan sah beserta uji ketunggalannya, lintasan kritisnya, dan siklus yang disuntikkan. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o dag dag.cpp`. 1041

Kode 12.9 Algoritma Prim dengan `std::priority_queue` dan `DecreaseKey` malas, mencetak tumpukannya sesudah setiap langkah dan mencacah entri basinya; dijalankan dari a dan dari i. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o primheap primheap.cpp`. 1051

Kode 12.10 Persoalan Terkerjakan R, S, U, V, dan W: penelusuran Kruskal beserta hutan himpunan saling lepasnya, penelusuran Prim beserta larik kuncinya, sifat potongan dan sifat siklus, barisan penyatuan menurut peringkat, dan uji ketunggalan yang dijalankan dua kali. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o mst mst.cpp`. 1060

Kode 12.11 Audit situs web Profitina Laundry: kedalaman klik dan halaman yatim lewat pencarian melebar, penggugusan nyaris-kembar lewat pohon merentang minimum dengan sisi beratnya dipotong, dan rencana perbaikan yang diurutkan secara topologis beserta lintasan kritisnya. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o laundry laundry.cpp` 1071

Kode 12.12 Program untuk Soal Latihan 12.1, 12.3, 12.5, 12.7, 12.9, dan 12.11. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o exercises exercises.cpp` lalu jalankan `./exercises`. 1082

Kode 13.1 Dijkstra dan Bellman–Ford pada graf tiga simpul dengan satu sisi negatif dan tanpa siklus. Kompilasilah dengan `g++ -std=c++17 -O2 -o dijfail dijfail.cpp`. 1110

Kode 13.2 Algoritma Dijkstra dengan antrean prioritas *binary heap* pada graf contoh, yang mencetak antreannya pada setiap langkah. Kompilasilah dengan `g++ -std=c++17 -O2 -o dijkstra dijkstra.cpp`. 1116

Kode 13.3 Sebuah pemeriksaan silang sekaligus tolok ukur: Dijkstra *heap*, Dijkstra larik, dan Bellman–Ford harus sepakat pada setiap graf tak negatif, lalu kedua Dijkstra-nya diukur waktunya pada masukan jarang ($m = 8n$) dan padat ($m = n2/4$). 1123

Kode 13.4 Bellman–Ford dengan pendeteksian siklus negatif pada graf contoh, yang mencetak setiap sapuannya. Separuh kedua `main` menurunkan $w(z, s)$ dari 2 menjadi -8, sehingga tercipta siklus $s \to t \to z \to s$ berbobot $6 + (-4) + (-8) = -6$, lalu memperlihatkan pendeteksinya menyala. Kompilasilah dengan `g++ -std=c++17 -O2 -o bellman bellman.cpp` 1134

Kode 13.5 `DAG-Shortest-Paths`: pengurutan topologis Kahn disusul satu sapuan relaksasi. Uji `if ((int)order.size() != n)` adalah pendeteksi siklus yang cuma-cuma — algoritma Kahn tidak mungkin mengeluarkan setiap simpul bila ada siklus — sehingga programnya menolak memberikan jawaban pada graf yang pengandaiannya gagal. Kompilasilah dengan `g++ -std=c++17 -O2 -o dag dag.cpp` 1142

Kode 13.6 Penyusunan ulang lintasan dari π, secara iteratif dan rekursif, dengan dua pengaman: sebuah pengawal siklus di dalam `buildPath`, dan penghitungan ulang bobot tiap lintasan dari

daftar sisinya untuk mencocokkannya dengan d. Grafnya adalah graf contoh Dijkstra ditambah satu simpul q yang tidak ditunjuk apa pun. 1146
Kode 13.7 Studi kasus Profitina Laundry: Dijkstra atas menit berkendara, Bellman–Ford atas biaya bersih dengan satu ruas negatif, dan pendeteksi siklus negatifnya menyala ketika rabatnya dinaikkan terlalu jauh. Kompilasilah dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp` 1151
Kode 13.8 Penyelesaian Soal Latihan 13.13: Bellman–Ford yang diperluas sehingga tiap simpul dilabeli $+\infty$, sebuah jarak berhingga, atau $-\infty$, menurut apakah ia tak terjangkau, terjangkau dengan aman, atau terjangkau dari sebuah siklus negatif. 1161
Kode 14.1 Sebuah tumpukan dengan `Multipop`, diinstrumentasi. Kompilasi dengan `g++ -std=c++17 -O2 -o multipop multipop.cpp` 1173
Kode 14.2 Pencacah biner k-bit dengan pencacah balikan per bit. Kompilasi dengan `g++ -std=c++17 -O2 -o counter counter.cpp` 1179
Kode 14.3 Soal 17.1-3 diuji secara numerik terhadap bentuk tertutupnya dan batasnya. Kompilasi dengan `g++ -std=c++17 -O2 -o ex1713 ex1713.cpp` 1185
Kode 14.4 Metode akuntansi yang diinstrumentasi: saldo kreditnya dicetak sesudah tiap operasi, di samping besaran yang menurut buktinya sama dengannya. Kompilasi dengan `g++ -std=c++17 -O2 -o credit credit.cpp` 1192
Kode 14.5 Soal 17.2-2: saldo kreditnya dilacak dan diuji terhadap $2j + k + 2$. Kompilasi dengan `g++ -std=c++17 -O2 -o ex1722 ex1722.cpp` 1197
Kode 14.6 Metode potensial yang diinstrumentasi pada tumpukannya sekaligus pada pencacahnya. Kompilasi dengan `g++ -std=c++17 -O2 -o potential potential.cpp` 1203
Kode 14.7 Soal 17.3-6: antrean dua-tumpukan, dengan fungsi potensialnya dihitung sesudah tiap operasi. Kompilasi dengan `g++ -std=c++17 -O2 -o queue2 queue2.cpp` 1209
Kode 14.8 `TABLE-INSERT` dengan pelipatan, diinstrumentasi dengan fungsi potensialnya. Kompilasi dengan `g++ -std=c++17 -O2 -o dyntable dyntable.cpp` 1219
Kode 14.9 Kedua kebijakan penyusutan dibandingkan pada barisan *thrashing*-nya. Kompilasi dengan `g++ -std=c++17 -O2 -o dyntable2 dyntable2.cpp` 1222
Kode 14.10 Union–find dengan kedua akal-akalannya, tiap penunjuknya dicacah. Kompilasi dengan `g++ -std=c++17 -O2 -o dsu dsu.cpp` 1230
Kode 14.11 Profitina Laundry: sebuah tabel dinamis, sebuah tumpukan `Multipop`, dan sebuah antrean dua-tumpukan, semuanya diinstrumentasi, sepanjang satu hari kerja. Kompilasi dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp` 1237
Kode 14.12 Sebuah pencacah dengan `Increment` dan `Reset`, keduanya $O(1)$ teramortisasi. Kompilasi dengan `g++ -std=c++17 -O2 -o resetcounter resetcounter.cpp` 1248
Kode 14.13 Tiga kebijakan pertumbuhan dibandingkan: melipatduakan, melipattigakan, dan menambahkan 100 slot tetap. Kompilasi dengan `g++ -std=c++17 -O2 -o growth growth.cpp` 1251
Kode 14.14 Mengukur apa yang sungguh-sungguh dilakukan `std::vector<int>::push_back`. Kompilasi dengan `g++ -std=c++17 -O2 -o vectorgrow vectorgrow.cpp` 1258
Kode 15.1 Dijkstra diaudit terhadap Floyd–Warshall pada dua graf bersisi negatif tanpa siklus negatif. Kompilasilah dengan `g++ -std=c++17 -O2 -o dijkstra_fails dijkstra_fails.cpp` 1270
Kode 15.2 SLOW-ALL-PAIRS-SHORTEST-PATHS selengkapnya, sambil mencetak setiap matriks antaranya. Kompilasilah dengan `g++ -std=c++17 -O2 -o slow_apsp slow_apsp.cpp` 1282
Kode 15.3 FASTER-ALL-PAIRS-SHORTEST-PATHS, versi pemangkatan berulangnya. Kompilasilah dengan `g++ -std=c++17 -O2 -o faster_apsp faster_apsp.cpp` 1289
Kode 15.4 Gelung rangkap tiga yang sama dengan k terluar dan dengan k terdalam, pada dua graf yang dibangun untuk membedakan keduanya. Kompilasilah dengan `g++ -std=c++17 -O2 -o ex_loop_order ex_loop_order.cpp` 1297
Kode 15.5 Floyd–Warshall beserta matriks pendahulunya, mencetak setiap $D(k)$ dan $\Pi(k)$ lalu setiap rutenya. Kompilasilah dengan `g++ -std=c++17 -O2 -o floyd_warshall floyd_warshall.cpp` 1301
Kode 15.6 Pendeteksian siklus berbobot negatif dari diagonalnya, pada tiga graf. Kompilasilah dengan `g++ -std=c++17 -O2 -o negcycle negcycle.cpp` 1309

Kode 15.7 Lima algoritma, satu graf, satu jawaban. Kompilasilah dengan `g++ -std=c++17 -O2 -o crosscheck crosscheck.cpp`. 1311
Kode 15.8 Algoritma penutupan transitif selengkapnya. Kompilasilah dengan `g++ -std=c++17 -O2 -o transitive_closure transitive_closure.cpp` 1317
Kode 15.9 Gelung rangkap tiga booleannya, satu bita per entri dan satu bit per entri, diukur. Kompilasilah dengan `g++ -std=c++17 -O2 -o ex_bitset_tc ex_bitset_tc.cpp`. 1319
Kode 15.10 Algoritma Johnson selengkapnya: Bellman–Ford dari sebuah simpul bantu, pembobotan ulang, V kali jalannya Dijkstra, dan pengubahannya kembali. Kompilasilah dengan `g++ -std=c++17 -O2 -o johnson johnson.cpp`. 1326
Kode 15.11 Floyd–Warshall melawan $V \times$ Dijkstra pada sembilan kerapatan, $V = 600$ di sepanjangnya. Kompilasilah dengan `g++ -std=c++17 -O2 -o crossover crossover.cpp`. 1331
Kode 15.12 Profitina Laundry: waktu berkendara semua pasangan, eksentrisitas, pusat grafnya, diameternya, dan rute di balik perjalanan terburuknya. Kompilasilah dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp`. 1335
Kode 15.13 Gelung rangkap tiga yang sama atas tiga semiring: lintasan terpendek, lintasan terlebar, dan cacah lintasan terpendek. Kompilasilah dengan `g++ -std=c++17 -O2 -o ex_semirings ex_semirings.cpp`. 1347
Kode 16.1 Sanggahan menyeluruh atas pernyataan (13): setiap graf pada empat simpul berlabel dibangun, lalu urutan mendalam dan urutan melebar dari simpul 0 dibandingkan. 1362
Kode 16.2 Algoritma $O(n2)$ Persoalan 16.2 dalam C++, mencetak tabel L dan P. 1368
Kode 16.3 Cara $O(n\lg n)$ untuk Persoalan 16.2, mencetak larik `tail` sesudah setiap unsur. 1370
Kode 16.4 Percobaan di balik Persoalan 16.3: pohon biner acak dengan ukuran dalam tertentu, dengan kedua cacahannya diambil. Perhatikan bahwa `externals` mengembalikan 1 untuk penunjuk kosong — itulah definisi simpul luar, ditulis sebagai kode. 1374
Kode 16.5 Algoritma hampiran Persoalan 16.4 selengkapnya, dengan ketaksamaan segitiga diverifikasi, pohon merentang minimum dibangun dengan Prim, penelusuran penuh dan penelusuran *preorder* dicetak, dan optimumnya ditemukan dengan gaya kasar sebagai pembanding. 1382
Kode 16.6 Kedua algoritma Persoalan 16.5 pada dua graf Bab 13, mencetak vektor taksiran sesudah setiap pengeluaran dan sesudah setiap putaran. 1391
Kode 16.7 Kegagalan Dijkstra pada sisi negatif, dan deteksi siklus negatif oleh Bellman–Ford, pada dua graf terkecil yang memperlihatkan masing-masingnya. 1393
Kode 16.8 Algoritma Dijkstra pada graf Persoalan 16.6, mencetak pasangan $d[v]/\pi[v]$ untuk setiap simpul sesudah setiap pengeluaran, lalu setiap lintasan terpendek yang dibaca balik lewat π. 1397
Kode 16.9 Bellman–Ford putaran demi putaran pada graf-graf Persoalan 16.7, dengan banyaknya perubahan pada setiap putaran, pemeriksaan penutup, dan — ketika pemeriksaan itu gagal — siklus negatif yang dipulihkan dari larik pendahulu. 1401
Kode 16.10 Lintasan terpendek pada DAG Persoalan 16.8: pengurutan topologis secara mendalam, lalu satu relaksasi per sisi dalam urutan topologis, dengan d/π dicetak sesudah setiap simpul. 1406
Kode 16.11 Pencacah biner Persoalan 16.9 dengan total agregat, tabungan akuntansi, dan biaya amortisasi metode potensial dicetak untuk setiap panggilan, disusul jumlah agregat untuk n yang lebih besar. 1411
Kode 16.12 Tabel dinamis berlipat dua Persoalan 16.10, dengan biaya sebenarnya, jumlah berjalan terhadap $3i$, tabungan akuntansi, potensial $2\,num - size$, dan biaya amortisasi setiap penyisipan. 1416
Kode 16.13 Floyd–Warshall pada graf Persoalan 16.11, mencetak $D(k)$ dan $\Pi(k)$ untuk $k = 0, \ldots, 4$ dan menandai setiap entri yang berubah, disusul uji siklus negatif pada diagonalnya. 1420
Kode 16.14 Algoritma Johnson pada graf Persoalan 16.12: Bellman–Ford dari simpul bantu, pembobotan ulang berikut hitungannya, dan Dijkstra dari setiap sumber, dengan jarak d di bawah w' dan jarak D yang dipulihkan berdampingan. 1424
Kode 16.15 Lintasan terpanjang pada DAG berbobot untuk Persoalan 16.14: pengurutan topologis Kahn disusul satu relaksasi pemaksimum per sisi. 1428
Kode 16.16 Floyd–Warshall dan Johnson pada satu graf lima simpul, dengan keenam matriks antara, potensial-potensial, dan sisi-sisi yang dibobot ulang dicetak. 1432
Kode 16.17 Program dinamis *knapsack* 0/1 Persoalan 16.19, mencetak seluruh tabel c dan menyusun kembali himpunan yang terpilih dengan menelusurinya mundur. 1437

Kode 16.18 Hampiran-2 untuk penutup simpul pada Persoalan 16.20, dengan pemasangan yang dibangunnya secara tersirat dan optimum sejati yang ditemukan dengan pencarian menyeluruh. 1439
Kode 16.19 Perencana malam Profitina Laundry: tiga pertanyaan dari satu himpunan data, dijawab oleh algoritma rakus, pohon merentang minimum dengan penelusuran *preorder*, dan Dijkstra — satu baris **Tabel 16.19** masing-masing. 1452
Kode 16.20 Kode Huffman Soal Latihan 8, dengan barisan penggabungannya, kata-kata kodenya, dan perbandingannya terhadap kode berpanjang tetap tiga bit. 1461

Daftar Pseudokode

Pseudokode 2.1 Mencari yang terbesar di antara n bilangan dengan satu kali sapuan. 79
Pseudokode 2.2 Insertion Sort dalam pseudokode yang dipakai di sepanjang buku ini, dengan indeks berbasis-1. 81
Pseudokode 2.3 *Linear search*: periksa elemen dari kiri ke kanan sampai q ditemukan atau lariknya habis. 104
Pseudokode 2.4 *Binary search* pada larik terurut: bandingkan dengan elemen tengah dan buang paruh yang tidak mungkin memuat q. 107
Pseudokode 2.5 Menjumlahkan elemen-elemen sebuah larik. 122
Pseudokode 3.1 Pencarian biner rekursif untuk s pada irisan terurut $A[p..r]$. Satu panggilan rekursif ($a = 1$) pada selang berukuran separuh ($b = 2$), ditambah kerja $\Theta(1)$ per panggilan, memberi $T(n) = T(n/2) + \Theta(1)$. 178
Pseudokode 3.2 Merge Sort pada irisan $A[p..r]$, dengan MERGE diuraikan dalam kata-kata; **Kode 3.8** memberikan kode lengkapnya. 188
Pseudokode 4.1 Menjawab satu pertanyaan LDCSUMAB: terkaan pertama titik-mengambang untuk bloknya, dua gelung pengoreksi, dan posisi di dalam blok. 214
Pseudokode 4.2 Algoritma TOHU: satu evaluasi bentuk tertutup Persamaan (4.7) per pertanyaan. 224
Pseudokode 4.3 Aturan Horner: mengevaluasi $a0 + a1x + \cdots + anxn$ mulai dari koefisien tertinggi ke bawah. 228
Pseudokode 4.4 *Insertion sort*, seperti pada Subbab 2.8.2, dengan indeks yang berjalan dari 1. 231
Pseudokode 4.5 Pencarian biner, seperti pada Subbab 2.14.3, yang mengembalikan sebuah indeks q atau NIL. 233
Pseudokode 4.6 Gelung rangkap tiga yang baris terdalamnya hendak dihitung secara tepat. 236
Pseudokode 4.7 Algoritma Kupon Laundry: satu evaluasi Persamaan (4.14) per pertanyaan. 253
Pseudokode 4.8 Pelipatduaan cepat: pasangan Fibonacci $(Fn, Fn + 1)$ modulo p dalam $\Theta(\log n)$ perkalian. 255
Pseudokode 4.9 Algoritma urutkan-lalu-cari untuk persoalan jumlah pasangan, dalam $\Theta(n\lg n)$. 258
Pseudokode 5.1 `MAX-HEAPIFY`, yang juga disebut `HEAPIFY`: biarkan nilai di simpul i tenggelam sampai ia mendominasi kedua anaknya. 286
Pseudokode 5.2 `BUILD-MAX-HEAP`: perbaiki setiap simpul dalam, dari yang terakhir kembali naik sampai ke akar. 293
Pseudokode 5.3 `HEAPSORT`: bangun sebuah *max-heap*, lalu berulang kali pindahkan akarnya ke ujung *heap* yang menyusut dan perbaiki akarnya. 304
Pseudokode 5.4 `HEAP-MAXIMUM`: kunci terbesar sebuah *max-heap* selalu berada di akarnya. 311
Pseudokode 5.5 `HEAP-EXTRACT-MAX` dalam pseudokode. Simpan akarnya; pindahkan elemen terakhir ke akar; susutkan *heap*-nya satu; panggil `HEAPIFY` pada akarnya untuk memulihkan keteraturan; kembalikan nilai yang tadi disimpan. 311
Pseudokode 5.6 `HEAP-INCREASE-KEY`: naikkan kunci di posisi i, lalu apungkan ia melewati setiap induk yang lebih kecil. 312
Pseudokode 5.7 `HEAP-INSERT` dalam pseudokode. Tumbuhkan *heap*-nya satu slot di ujung, lalu apungkan kunci barunya ke atas, dengan memindahkan setiap induk yang kalah turun ke tempat anaknya, sampai sebuah induk menang atau akarnya tercapai. 313

Pseudokode 5.8 `MAX-HEAP-INSERT`: tambahkan satu slot berisi $-\infty$ di ujung, lalu naikkan ke kunci barunya. 313
Pseudokode 5.9 `PARTITION` versi Lomuto: satu sapuan dari kiri ke kanan yang menumbuhkan wilayah elemen yang tidak lebih besar daripada *pivot*-nya. 319
Pseudokode 5.10 `PARTITION` versi Hoare: dua penunjuk merapat dari kedua ujung dan menukarkan pasangan yang salah tempat sampai keduanya bersilangan. 321
Pseudokode 5.11 `QUICKSORT` dengan pemartisian Lomuto: partisikan, lalu urutkan kedua sisi *pivot*-nya secara rekursif. 325
Pseudokode 5.12 `RANDOMIZED-PARTITION` dan `RANDOMIZED-QUICKSORT`: tukarkan sebuah elemen acak seragam ke slot *pivot*, lalu partisikan seperti biasa. 339
Pseudokode 5.13 `QUICKSORT-TAIL`: berekursi hanya pada potongan yang lebih kecil dan menggelung pada yang lebih besar, sehingga kedalaman tumpukannya tetap $O(\lg n)$. 341
Pseudokode 6.1 Perbaikan Kiri-Kiri: satu rotasi kanan terhadap z, dituliskan sebagai keempat penugasan penunjuknya. 411
Pseudokode 6.2 Perbaikan Kiri-Kanan: rotasi kiri terhadap y mengubah zig-zagnya menjadi kasus Kiri-Kiri, lalu rotasi kanan terhadap z merampungkannya. 412
Pseudokode 6.3 Perbaikan Kanan-Kanan: satu rotasi kiri terhadap z, bayangan cermin **Pseudokode 6.1** 413
Pseudokode 6.4 Perbaikan Kanan-Kiri: rotasi kanan terhadap y, lalu rotasi kiri terhadap z. 414
Pseudokode 7.1 Rekursi yang lurus-lurus saja untuk Fn, disalin langsung dari definisinya. 465
Pseudokode 7.2 Fibonacci dengan *memoization*: rekursi yang sama, yang memeriksa dan mengisi sebuah tabel jawaban. 472
Pseudokode 7.3 Fibonacci secara *bottom-up*: tabelnya diisi dari indeks terkecil ke atas. 473
Pseudokode 7.4 Fibonacci dalam ruang tetap: jendela geser berisi dua nilai menggantikan tabelnya. 473
Pseudokode 7.5 Matrix-Chain-Order dalam pseudokode. Baris 1–2 mengisi diagonalnya dengan nol. Baris 3 adalah yang paling penting: gelung terluarnya berjalan atas **panjang** deretan l, bukan atas i atau j, supaya setiap subderetan terselesaikan sebelum subderetan lebih panjang mana pun yang bersandar padanya. Baris 7–11 mencoba setiap belahan k dan menyimpan baik biaya terbaik di M maupun belahan pemenangnya di c. 489
Pseudokode 7.6 Print-Optimal-Parens: membaca pengurungan optimal dari tabel c. 492
Pseudokode 7.7 Lookup-Chain: rekurensi deretan matriks yang ditulis sebagai rekursi, dengan baris 1 memotong langsung setiap submasalah yang sudah terselesaikan. Seluruh entri M diinisialisasi ke ∞ sebelumnya, dan ∞ sekaligus berperan sebagai penanda "belum terhitung". 498
Pseudokode 7.8 Baris pertama **Lookup-Chain**: kembalikan jawaban tersimpan alih-alih menghitungnya ulang. 498
Pseudokode 7.9 LCS-Length dalam pseudokode. Baris 1–4 mengisi baris nol dan kolom nolnya. Baris 5–15 menyapu tabelnya baris demi baris, mengisi $c[i, j]$ menurut rekurensinya dan mencatat di $b[i, j]$ kasus mana di antara ketiganya yang dipakai: `"copy"`, `"skipx"`, atau `"skipy"`. 509
Pseudokode 8.1 `MAX-HEAPIFY`: nilai di simpul i tenggelam sampai tak ada anak yang lebih besar daripadanya. 543
Pseudokode 8.2 `BUILD-MAX-HEAP`: memperbaiki simpul-simpul dalam dari bawah ke atas. 543
Pseudokode 8.3 `PARTITION` Lomuto: satu sapuan dari kiri ke kanan yang menumbuhkan daerah nilai $\leq x$. 552
Pseudokode 8.4 `QUICKSORT`: partisi, lalu rekursi pada kedua sisi pivot. 552
Pseudokode 8.5 `MATRIX-CHAIN-ORDER`: setiap subrantai diselesaikan sekali, yang terpendek lebih dahulu, dan pemisah pemenangnya dicatat. 618
Pseudokode 8.6 `LCS-LENGTH`: tabel panjang dan tabel panah, diisi baris demi baris, dengan seri diputuskan ke atas. 627
Pseudokode 8.7 `PRINT-LCS`: ikuti panah-panahnya mundur; setiap panah diagonal adalah satu karakter jawaban. 627
Pseudokode 9.1 Greedy(M, w), algoritma rakus matroid berbobot. §9.8.8 memberikannya sebagai program utuh yang dapat dikompilasi, dengan orakel kebebasan yang abstrak. 725
Pseudokode 10.1 Penelusuran melebar dari sumber s, dalam notasi buku ini, dengan nomor baris yang sama seperti **Gambar 10.15**. 796

Pseudokode 10.2 Penelusuran mendalam: prosedur luarnya memulai pohon baru pada setiap simpul yang masih putih, dan DFS-Visit menjalankan penelusurannya sambil mencap waktu penemuan dan waktu selesai. 813
Pseudokode 10.3 Pengurutan topologis dengan penelusuran mendalam: simpul-simpul dalam urutan waktu selesai yang menurun. 842
Pseudokode 10.4 Gelung dalam BFS yang diperluas untuk mencacah lintasan terpendek: $cnt[v]$ menghimpun cacah milik semua tetangga yang satu aras lebih dekat ke sumber. 868
Pseudokode 10.5 Menguji apakah sebuah graf tak berarah dengan $n \geq 1$ simpul adalah pohon: satu pencacahan sisi dan satu penelusuran melebar. 869
Pseudokode 11.1 Pengurutan topologis dengan pencarian mendalam-dahulu, yang mencatat tiap simpul pada saat simpul itu selesai. 880
Pseudokode 11.2 Pengurutan topologis Kahn: berulang kali keluarkan simpul yang tidak punya pendahulu tersisa; kekurangan pada akhirnya membuktikan adanya siklus. 886
Pseudokode 11.3 Algoritma pohon merentang minimum generik: tumbuhkan himpunan sisi A satu sisi aman setiap kalinya. 902
Pseudokode 11.4 Algoritma generik dengan kata "aman" diganti Akibat 11.6: pilih potongan yang menghormati A lalu ambil sisi ringannya. 902
Pseudokode 11.5 Algoritma Prim–Jarník: $key[v]$ adalah bobot sisi termurah dari v ke dalam pohon yang sedang tumbuh, dan $\pi[v]$ adalah ujung lainnya. 904
Pseudokode 11.6 Membuat himpunan bersinggel, dan gabung menurut peringkat: akar berperingkat lebih kecil digantungkan di bawah akar berperingkat lebih besar. 917
Pseudokode 11.7 Pencarian dengan pemampatan lintasan: dalam perjalanan kembali dari akar, setiap simpul pada lintasannya diarahkan ulang ke akar. 918
Pseudokode 11.8 Algoritma Kruskal: pindai sisi-sisinya dari yang teringan sampai yang terberat dan simpan setiap sisi yang menghubungkan dua pohon hutan yang berbeda. 923
Pseudokode 12.1 Rancangan Rute Laundry: pencarian melebar dan komponen untuk wilayah layanan, Kruskal di atas hutan himpunan saling lepas untuk tulang punggungnya, dan algoritma Kahn beserta lintasan kritisnya untuk alur kerjanya. 965
Pseudokode 12.2 Aturan rakus penukaran uang: berulang-ulang serahkan koin terbesar yang tidak melebihi sisa jumlah yang masih terutang. 982
Pseudokode 12.3 Aturan waktu-selesai-terawal untuk pemilihan aktivitas, sebagai satu sapuan atas pekerjaan yang sudah terurut menurut waktu selesai. 987
Pseudokode 12.4 Algoritma Huffman: $n - 1$ penggabungan atas kedua pohon terjarang, tiap penggabungan menciptakan satu simpul dalam. 993
Pseudokode 12.5 Algoritma rakus matroid untuk tugas berwaktu satuan dengan tenggat dan penalti: pertahankan pesanan termahal tetap tepat waktu kapan pun kebebasannya mengizinkan. 999
Pseudokode 12.6 Algoritma Kahn: berulang-ulang keluarkan simpul yang tak lagi punya prasyarat tersisa, lalu buang busur-busur keluarnya. 1037
Pseudokode 12.7 Algoritma Prim dengan tumpukan biner dan `DecreaseKey` malas: kunci yang membaik didorong sebagai entri baru, dan entri yang basi dibuang ketika sampai di puncak. 1049
Pseudokode 13.1 Inisialisasi yang dipakai bersama oleh setiap algoritma sumber tunggal: semua estimasi tak hingga, tanpa pendahulu, dan sumbernya berjarak nol. 1102
Pseudokode 13.2 Merelaksasi sisi (u, v): satu pembandingan, dan dua pemberian nilai bila rute lewat u benar-benar lebih baik. 1103
Pseudokode 13.3 Mencetak lintasan terpendek dari s ke v secara rekursif lewat larik pendahulu. 1146
Pseudokode 16.1 Program dinamis $O(n2)$ untuk subbarisan menaik terpanjang, disusul penelusuran balik yang memulihkan subbarisannya sendiri. 1366
Pseudokode 16.2 Lintasan terpendek sumber tunggal pada graf berarah taksiklik: satu pengurutan topologis, lalu satu relaksasi per sisi. 1404
Pseudokode 16.3 `INCREMENT` pada pencacah biner k-bit: nolkan bit-bit 1 di ujung, lalu jadikan bit 0 yang pertama bernilai 1. 1408
Pseudokode 16.4 `TABLE-INSERT` untuk tabel yang berlipat dua ketika penuh. Baris 3–6 adalah perluasan yang mahal; baris 7 adalah penyisipan dasarnya. 1413
Pseudokode 16.5 Lintasan terpanjang dari s pada DAG berbobot: algoritma lintasan terpendek DAG dengan perbandingan yang dibalik. 1427

Pseudokode 16.6 Floyd–Warshall (kiri) dan Johnson (kanan). Floyd–Warshall tidak pernah mengubah bobotnya; Johnson mengubahnya sekali, di baris 4, sehingga Dijkstra menjadi sah. 1432
Pseudokode 16.7 Program dinamis *knapsack* 0/1: n baris berisi $W + 1$ entri, masing-masing dihitung dalam waktu konstan. 1437
Pseudokode 16.8 Prim (kiri) dan Kruskal (kanan). Baris 5 masing-masing adalah tempat keduanya berbeda: simpul terdekat ke pohon, berhadapan dengan sisi berikutnya yang menyambungkan dua komponen. 1441

Daftar Persamaan

(1.1) 14
(1.2) 21
(1.3) 21
(1.4) 25
(1.5) 29
(1.6) 33
(1.7) 33
(1.8) 37
(1.9) 41
(1.10) 42
(1.11) 42
(2.1) 79
(2.2) 90
(2.3) 92
(2.4) 93
(2.5) 93
(2.6) 94
(2.7) 95
(2.8) 98
(2.9) 102
(2.10) 105
(2.11) 109
(2.12) 113
(2.13) 129
(3.1) 136
(3.2) 140
(3.3) 141
(3.4) 142
(3.5) 148
(3.6) 149
(3.7) 152
(3.8) 155
(3.9) 156
(3.10) 157
(3.11) 161
(3.12) 172
(3.13) 172
(3.14) 175
(3.15) 177

(3.16)....189
(4.1) 212
(4.2) 212
(4.3) 213
(4.4) 219
(4.5) 221
(4.6) 222
(4.7) 222
(4.8) 228
(4.9) 237
(4.10)....239
(4.11)....244
(4.12)....246
(4.13)....247
(4.14)....253
(4.15)....255
(4.16)....255
(5.1) 279
(5.2) 279
(5.3) 279
(5.4) 282
(5.5) 291
(5.6) 299
(5.7) 300
(5.8) 300
(5.9) 320
(5.10)....328
(5.11)....328
(5.12)....329
(5.13)....332
(5.14)....336
(5.15)....336
(5.16)....339
(5.17)....344
(5.18)....345
(5.19)....356
(6.1) 373
(6.2) 375
(6.3) 398
(6.4) 398
(6.5) 401
(6.6) 402
(6.7) 403
(6.8) 403
(6.9) 403
(6.10)....406
(6.11)....434
(7.1) 461
(7.2) 465
(7.3) 468
(7.4) 468

(7.5) 469
(7.6) 474
(7.7) 482
(7.8) 485
(7.9) 485
(7.10) 488
(7.11) 489
(7.12) 494
(7.13) 504
(7.14) 508
(7.15) 522
(7.16) 534
(7.17) 536
(8.1) 546
(8.2) 546
(8.3) 554
(8.4) 554
(8.5) 555
(8.6) 560
(8.7) 560
(8.8) 564
(8.9) 583
(8.10) 584
(8.11) 585
(8.12) 589
(8.13) 589
(8.14) 592
(8.15) 594
(8.16) 595
(8.17) 600
(8.18) 618
(8.19) 621
(8.20) 627
(9.1) 673
(9.2) 676
(9.3) 677
(9.4) 679
(9.5) 680
(9.6) 696
(9.7) 697
(9.8) 708
(9.9) 713
(9.10) 714
(9.11) 726
(9.12) 728
(9.13) 728
(10.1) 770
(10.2) 772
(10.3) 779
(10.4) 805
(10.5) 806

(10.6) 807
(10.7) 808
(10.8) 808
(10.9) 809
(10.10) 817
(10.11) 824
(10.12) 828
(11.1) 894
(11.2) 896
(11.3) 897
(11.4) 900
(11.5) 910
(11.6) 910
(11.7) 917
(11.8) 919
(11.9) 930
(11.10) 930
(12.1) 964
(12.2) 965
(12.3) 966
(12.4) 978
(12.5) 982
(12.6) 988
(12.7) 990
(12.8) 992
(12.9) 995
(12.10) 995
(12.11) 997
(12.12) 999
(12.13) 1013
(12.14) 1025
(12.15) 1036
(12.16) 1038
(12.17) 1048
(12.18) 1051
(12.19) 1058
(13.1) 1092
(13.2) 1092
(13.3) 1096
(13.4) 1096
(13.5) 1097
(13.6) 1098
(13.7) 1106
(13.8) 1120
(13.9) 1121
(13.10) 1122
(13.11) 1129
(13.12) 1137
(13.13) 1138
(14.1) 1168
(14.2) 1169

(14.3) 1178
(14.4) 1183
(14.5) 1183
(14.6) 1184
(14.7) 1184
(14.8) 1189
(14.9) 1196
(14.10) 1199
(14.11) 1200
(14.12) 1200
(14.13) 1215
(14.14) 1216
(14.15) 1226
(14.16) 1228
(14.17) 1228
(14.18) 1253
(15.1) 1265
(15.2) 1267
(15.3) 1275
(15.4) 1276
(15.5) 1277
(15.6) 1278
(15.7) 1295
(15.8) 1295
(15.9) 1300
(15.10) 1300
(15.11) 1316
(15.12) 1316
(15.13) 1322
(15.14) 1324
(16.1) 1352
(16.2) 1366
(16.3) 1373
(16.4) 1377
(16.5) 1380
(16.6) 1407
(16.7) 1408
(16.8) 1409
(16.9) 1413
(16.10) 1418
(16.11) 1422
(16.12) 1423
(16.13) 1437

Kata Pengantar

Ada satu jenis hening tertentu di ruang kelas, yaitu saat seorang mahasiswa baru saja mengerti sesuatu. Bukan hening karena bingung — hening yang itu terasa berat. Ini hening yang datang setengah detik sebelum seseorang berkata, “oh — ya iya.” Setiap halaman buku ini ditulis untuk memperoleh setengah detik itu.

Buku ini membahas perancangan algoritma dan cara membuktikan sesuatu tentangnya. Buku ini tumbuh dari mata kuliah Perancangan dan Analisis Algoritma (EF234405) di Departemen Teknik Informatika, Institut Teknologi Sepuluh Nopember (ITS), Surabaya, dan telah disusun ulang sehingga dapat berdiri sendiri sepenuhnya: Anda tidak memerlukan kelas, dosen, atau dokumen lain untuk belajar darinya. Keenam belas babnya mengikuti irama satu semester yang berlangsung enam belas minggu, dan ilustrasi, contoh, serta penelusuran langkah di dalamnya adalah yang telah diuji, dari tahun ke tahun, pada mahasiswa sungguhan di ruang kelas sungguhan.

Untuk siapa buku ini ditulis

Anda dianggap belum tahu apa-apa. Bukan “belum tahu soal algoritma” — belum tahu apa-apa. Jika Anda belum pernah membuktikan apa pun, belum pernah menulis satu baris C++ pun, belum pernah melihat logaritma dipakai sungguh-sungguh, Andalah pembaca yang dibayangkan saat buku ini ditulis. Setiap lambang diperkenalkan dengan kata-kata sebelum dipakai. Setiap rumus diturunkan, bukan diumumkan. Setiap contoh dikerjakan dengan tangan, satu langkah satu baris, lengkap dengan hitungannya — sebab di hitungan itulah pemahaman benar-benar terjadi, atau diam-diam gagal terjadi.

Prinsip itu punya akibat praktis: buku ini tebal. Buku yang lebih tipis adalah buku yang menganggap Anda sudah paham bagian-bagian yang sulit, dan buku semacam itu tidak berguna bagi orang yang justru belum paham. Ketika harus memilih antara ringkas dan pembaca yang selesai membaca lalu mampu mengimplementasikannya, pembaca yang menang.

Isi buku ini

Bab 1 sampai 3 membangun fondasi: apa itu algoritma, bagaimana membuktikan sebuah algoritma benar, model RAM, notasi asimtotik, rekurensi, dan Teorema Master. Bab 5 sampai 7 membahas pengurutan, pohon pencarian seimbang, bagi dan kuasai, serta pemrograman dinamis. Bab 9 sampai 11 beralih ke algoritma greedy dan ke graf — cara menelusurinya, dan cara merentangnya. Bab 13 sampai 15 membahas lintasan terpendek, analisis amortisasi, dan persoalan semua pasangan.

Bab 4, 8, 12, dan 16 berbeda sifatnya. Masing-masing adalah bab **Evaluasi dan Pembahasan** yang menutup seperempat bagian buku. Di sana tidak ada materi baru; sebaliknya, materi tiga bab sebelumnya dipakai untuk menyelesaikan persoalan utuh secara tuntas: setiap persoalan dinyatakan ulang dengan kata-kata sederhana, setiap langkah penalaran dituliskan, hitungannya ditunjukkan, jalan pintas yang menggoda tetapi keliru disebutkan dan dibongkar, cara kedua diberikan di mana pun cara itu ada, dan disertai program lengkap yang sudah dikompilasi dan

dijalankan. Setiap persoalan ditutup dengan varian "uji diri", agar Anda dapat memeriksa apakah metodenya sudah benar-benar menjadi milik Anda.

Setiap program dalam buku ini utuh, dapat dikompilasi dengan kompilator C++17 standar, dan sudah dijalankan sebelum keluarannya dicetak di sini. Tidak ada satu pun keluaran di halaman-halaman ini yang tidak dihasilkan oleh mesin. Program-program yang sama dikumpulkan, bab demi bab, dalam folder `Code` yang menyertai buku ini, masing-masing dengan README yang menjelaskan cara mengompilasi dan menjalankannya di Windows, macOS, dan Linux.

Cara membacanya

Pelan-pelan, sambil memegang pena. Kerjakan setiap contoh di kertas sebelum membaca penyelesaiannya. Ketika sebuah bukti muncul, tutup baris berikutnya dan cobalah melengkapinya sendiri. Ketika sebuah program muncul, ketikkan dan jalankan, lalu rusakkan dengan sengaja dan lihat seperti apa kerusakannya — di situlah sering kali pelajaran yang sebenarnya berada.

Semua soal latihan di akhir tiap bab dilengkapi penyelesaian lengkap. Gunakan seperti Anda menggunakan kunci jawaban di belakang buku matematika: sesudah usaha Anda sendiri, bukan sebagai penggantinya.

Gambar, tabel, kode, pseudokode, dan persamaan yang bernomor semuanya adalah tautan hidup: pada edisi elektronik, klik "Gambar 5.3" atau "[4]" di bagian mana pun dan Anda langsung dibawa ke sana. Daftar Isi serta daftar gambar, tabel, kode, pseudokode, dan persamaan di bagian depan bekerja dengan cara yang sama.

Tentang penulis

Irfan Subakti menggambarkan dirinya dalam satu baris — *Father | Educator | Entrepreneur | Bridging Tech & Nature: CS Lecturer × Crypto/Digital Business × AI/ML Specialist × Fruit Farmer | Storyteller* — ayah, pendidik, wirausahawan, penjembatan teknologi dan alam: dosen ilmu komputer, pelaku bisnis kripto dan digital, spesialis AI/ML, petani buah, dan pencerita. Setiap bagian dari baris itu ikut masuk ke dalam buku ini. Ia mengajar di Departemen Teknik Informatika ITS sejak 1998. Ia membimbing dan menguji tugas akhir mahasiswa sarjana, mengajar MLOps dalam program IBM SkillsBuild, dan membangun produk: ia mendirikan Profitina, platform visibilitas AI yang rekayasanya muncul di subbab "Profitina dalam Praktik" pada setiap bab. Jauh dari papan ketik, ia merawat kebun buah dan durian keluarga — di sanalah ia belajar bahwa penjelasan terbaik, seperti pohon durian terbaik, memerlukan bertahun-tahun pemangkasan yang sabar. Jiwa penceritanya adalah alasan buku ini memilih kisah ketimbang daftar, setiap kali sebuah kisah bisa mengantarkan maknanya.

Rully Soelaiman telah bertahun-tahun mengajar algoritma di departemen yang sama di ITS; bergenerasi-generasi mahasiswa mengenal beliau sebagai Pak Rully, dosen Competitive Programming terbaik. Sebagian besar materi dalam buku ini — soal, contoh yang dikerjakan, ilustrasi, dan penjelasan yang diasah selama bertahun-tahun di ruang kelas — bermula dari beliau, dan beliau dengan murah hati memberikannya untuk mata kuliah ini dan untuk buku ini.

Ucapan terima kasih

Utang terbesar saya adalah kepada **Rully Soelaiman, S.Kom., M.Kom.** — dosen saya, patron saya, dan di atas segalanya, sahabat baik saya. Bagi orang lain beliau adalah Pak Rully; bagi saya,

Masbro Rully. Begitu banyak bagian buku ini bermula dari beliau: soal-soal yang beliau pilih, contoh-contoh yang beliau kerjakan, ilustrasi yang beliau gambar, penjelasan yang beliau bentuk selama bertahun-tahun mengajar, lalu beliau berikan dengan cuma-cuma agar orang lain dapat belajar. Sudah sepantasnya nama beliau kini berdiri di samping nama saya di sampul buku ini. Banyak hal yang terasa jernih di buku ini jernih karena beliaulah yang menjernihkannya; apa pun yang masih keruh, itu tanggung jawab saya. Terima kasih, Masbro — untuk materinya, untuk standar yang Masbro tetapkan, untuk kesabaran Masbro mengajari saya, dan untuk persahabatan yang bertahan lebih lama daripada semua silabus yang pernah kita ajarkan bersama. Buku ini, dalam arti yang sesungguhnya, juga milik Masbro.

Bentuk buku ini juga berutang pada para mahasiswa yang mempelajari versi-versi awalnya dan mengajukan pertanyaan yang menyingkapkan di mana sebuah penjelasan masih tipis. Standarnya berutang pada Cormen, Leiserson, Rivest, dan Stein, yang *Introduction to Algorithms*-nya tetap menjadi ukuran bagi buku sejenis. Di mana buku ini berbeda dari buku mereka, bedanya adalah buku ini berjalan lebih pelan.

Contoh penerapan di sepanjang buku memakai **Profitina**, platform visibilitas AI yang dibangun di Surabaya, dan **Profitina Laundry**, studi kasus pengajaran yang menyertai buku ini. Yang pertama menunjukkan di mana materi ini mendarat pada produk nyata; yang kedua cukup kecil untuk Anda jalankan sendiri seluruhnya.

Surabaya, September 2026

Irfan Subakti · yifana@gmail.com

Rully Soelaiman · rully130270@gmail.com

Profitina

profitina.com

Bab 1

Fondasi, Kerangka Kerja, dan Persoalan Pertama

Di akhir bab ini Anda akan tahu apa itu algoritma dan bedanya dengan program yang mewujudkannya; Anda akan mampu menyatakan dengan tepat apa artinya sebuah algoritma itu *benar* dan apa artinya ia *efisien*; Anda akan mengenal lima keluarga besar perancangan algoritma dan bisa mengenali keluarga mana yang cocok untuk sebuah persoalan baru; dan Anda akan sudah menyelesaikan enam persoalan lomba yang nyata dari awal sampai tuntas — dari bacaan pertama atas rumusan soal sampai program C++ yang dikompilasi, dijalankan, dan menghasilkan jawaban yang diharapkan juri — memakai metode lima fase yang akan Anda pakai di sepanjang buku ini dan, kalau buku ini berhasil, di sepanjang umur kerja Anda.

1.1 Dari Mana Kita Mulai

Ini adalah buku tentang belajar **berpikir komputasional**. Frasa itu dipakai dengan longgar di banyak tempat, jadi mari kita kunci maknanya. Berpikir komputasional tentang sebuah persoalan berarti empat hal, dan keempatnya akan kita latih di setiap bab:

1. **Memodelkan persoalan secara algoritmis.** Seseorang menyerahkan kepada Anda gambaran sebuah keadaan dalam bahasa manusia biasa — sebuah mobil pengantar barang, setumpuk cucian, barisan angka di layar. Anda harus mengubahnya menjadi objek matematis yang cermat: persisnya apa yang masuk, persisnya apa yang harus keluar, dan persisnya masukan seperti apa saja yang mungkin.
2. **Merancang sebuah algoritma.** Pilih satu siasat — dan kabar baiknya, siasat yang benar-benar berbeda satu sama lain hanya ada segelintir — lalu susun langkah-langkah yang mengubah masukan menjadi keluaran.
3. **Menganalisis kebenaran dan kompleksitas.** *Buktikan* bahwa langkah-langkah Anda selalu memberi jawaban yang betul, untuk setiap masukan yang diizinkan persoalan, bukan sekadar untuk tiga contoh yang kebetulan Anda coba. Sesudah itu hitung berapa banyak waktu dan berapa banyak memori yang dihabiskan langkah-langkah itu saat masukannya membesar.
4. **Mengimplementasikannya dengan rapi.** Terjemahkan rancangan menjadi C++ yang berjalan, dengan struktur data yang tepat.

Perhatikan urutannya. Pahami, lalu rancang, lalu analisis, baru tulis kodenya. **Kode adalah langkah terakhir, bukan langkah pertama.** Banyak mahasiswa, dan cukup banyak pemrogram yang sudah bekerja, langsung melompat ke papan ketik. Mereka membuka editor, mengetik `int main() {`, lalu mulai menebak-nebak. Kadang mereka lolos, untuk persoalan yang mudah. Untuk persoalan yang sulit mereka menghasilkan ratusan baris kode yang salah dengan cara yang tidak bisa mereka temukan, sebab mereka tidak pernah memutuskan apa arti "betul" itu seharusnya.

Pemikiran yang terjadi sebelum kode ditulis justru itulah yang memisahkan seorang insinyur yang baik dari yang hebat. Buku ini sebagian besar berbicara tentang pemikiran itu.

Irama buku ini mengikuti irama satu semester pembelajaran perancangan dan analisis algoritma di perguruan tinggi: enam belas bab, satu bab untuk setiap pekan, dan setiap bab keempat (Bab 4,

8, 12, dan 16) berhenti sejenak untuk mengevaluasi dan membahas tiga bab sebelumnya. Subbab 1.15 membentangkan peta jalan selengkapnya.

Sepatah kata tentang pembaca yang diandaikan buku ini. Buku ini tidak mengandaikan apa pun. Kalau Anda belum pernah menulis satu baris C++ pun, belum pernah melihat lambang $\lfloor x \rfloor$, belum pernah mendengar kata *logaritma*, Andalah pembaca yang kami bayangkan. Setiap lambang diperkenalkan dengan kata-kata sebelum dipakai. Setiap rumus diturunkan, bukan diumumkan. Setiap contoh dikerjakan dengan tangan, satu baris satu langkah, lengkap dengan hitungannya. Ketika sebuah gagasan terasa sulit, ia dinyatakan sekali dalam bahasa matematika lalu dinyatakan lagi dalam bahasa warung di pasar. Anda boleh membaca buku ini hanya dengan pensil, selembar kertas, dan sebuah komputer yang punya kompilator C++, tidak lebih.

1.2 Algoritma dan Struktur Data — Dua Pilar

1.2.1 Apa itu algoritma

> **DEFINISI.** Sebuah **algoritma** adalah prosedur komputasi yang berhingga dan bertahap langkah demi langkah, yang mengubah sebuah *masukan* menjadi sebuah *keluaran*.

Kata itu berasal dari nama cendekiawan Persia abad kesembilan, Muhammad ibn Musa al-Khwarizmi, yang bukunya tentang berhitung dengan angka Hindu-Arab sampai ke Eropa abad pertengahan dengan judul Latin *Algoritmi de numero Indorum*. Gagasannya lebih tua daripada katanya: setiap resep yang bisa diandalkan untuk memperoleh jawaban adalah sebuah algoritma.

Resep memang perbandingan pertama yang paling baik. Bayangkan membuat *nasi goreng*. Anda punya bahan — nasi dingin, sebutir telur, bawang merah, *kecap manis*. Itulah **masukan**. Anda mengikuti langkah-langkah dalam suatu urutan: panaskan wajan, tumis bawang, masukkan nasi, dan seterusnya. Di akhir Anda punya sepiring nasi goreng. Itulah **keluaran**.

Agar sebuah prosedur bisa disebut algoritma dalam arti teknisnya, ia harus punya lima sifat, menurut rumusan klasik yang diberikan Knuth [1]. Masing-masing akan berguna nanti, jadi mari kita bahas satu per satu sambil membayangkan resep tadi.

Sifat 1 — masukan yang terdefinisi jelas. Algoritma harus menyebutkan persis apa yang diterimanya. "Sebuah barisan berisi n bilangan bulat, masing-masing antara 1 dan 1000" adalah masukan yang terdefinisi jelas. "Beberapa bilangan" bukan, sebab itu tidak memberi tahu Anda apakah bilangan negatif boleh muncul, atau apakah daftarnya boleh kosong.

Sifat 2 — keluaran yang terdefinisi jelas. Algoritma harus menyebutkan persis apa yang dihasilkannya, dan apa yang membuat hasil itu *betul*. "Cetak bilangan terkecil" terdefinisi jelas. "Cetak bilangan yang bagus" tidak.

Sifat 3 — kepastian (*definiteness*). Setiap langkah harus ditentukan dengan cermat, tanpa kemenduaan arti. "Tambahkan garam secukupnya" tidak apa-apa di buku masak, sebab juru masak manusia bisa mencicipi. Itu bukan langkah algoritmis, sebab mesin tidak bisa mencicipi. "Tambahkan 3 gram garam" adalah langkah algoritmis.

Sifat 4 — keberhinggaan (*finiteness*). Prosedur itu akhirnya harus berhenti untuk setiap masukan yang sah. Sekumpulan langkah yang berputar selamanya bukan algoritma; itu kutu. Syarat ini tidak sepele. Perhatikan perintah "mulailah dari n; kalau n genap bagi dua, kalau n ganjil ganti dengan $3n + 1$; ulangi sampai Anda mencapai 1." Tidak seorang pun di dunia ini tahu apakah itu berhenti untuk setiap nilai awal — itulah konjektur Collatz yang termasyhur, terbuka sejak 1937. Karena itu kita tidak boleh menyebutnya algoritma untuk apa pun.

Sifat 5 — keterlaksanaan (*effectiveness*). Setiap langkah harus cukup mendasar sehingga benar-benar dapat dikerjakan. "Hitunglah bilangan prima terbesar" tidak terlaksana, sebab bilangan prima terbesar itu tidak ada. "Putuskan apakah sebuah program sembarang akan berhenti" juga tidak terlaksana, menurut hasil termasyhur dari Alan Turing.

> **CONTOH:** Apakah ini sebuah algoritma? > *Masukan:* sebuah daftar berisi n bilangan. > *Langkah 1:* lihat bilangan pertama dan ingat-ingat. > *Langkah 2:* lihat setiap bilangan sisanya satu per satu; kalau ia lebih besar daripada yang Anda ingat, ingatlah ia sebagai gantinya. > *Langkah 3:* saat tidak ada bilangan yang tersisa, umumkan apa yang Anda ingat.
>
> Masukan: terdefinisi. Keluaran: terdefinisi — yaitu bilangan terbesar dalam daftar. Kepastian: "lebih besar daripada" tidak bermakna ganda untuk bilangan. Keberhinggaan: daftarnya berisi n elemen dan kita melihat masing-masing sekali, jadi kita berhenti sesudah n langkah. Keterlaksanaan: membandingkan dua bilangan adalah operasi dasar. Ya — ini sebuah algoritma, dan inilah algoritma pertama yang akan kita tulis dalam C++.

1.2.2 Program adalah wujud nyata sebuah algoritma

Perbedaan ini kecil, tetapi ia sumber kebingungan yang tidak sedikit, jadi mari kita tegas soal ini.

> **DEFINISI.** Sebuah **program** adalah pengungkapan sebuah algoritma dalam suatu bahasa pemrograman tertentu, supaya sebuah mesin tertentu dapat menjalankannya.

Algoritma adalah *gagasan*. Program adalah *perwujudan* gagasan itu. Algoritma yang sama — "ingat yang terbesar sejauh ini" — dapat ditulis dalam C++, dalam Python, dalam Java, di atas kertas, atau dijelaskan lisan kepada seorang teman. Setiap kali ia tetap algoritma yang sama. Hanya programnya yang berubah.

Ini penting karena tiga alasan yang sangat praktis.

Pertama, **Anda bisa salah di dua tempat yang berbeda.** Algoritma Anda bisa salah (gagasannya tidak menyelesaikan persoalan), atau program Anda bisa salah (gagasannya benar tetapi Anda mengetik `<` padahal maksud Anda `<=`). Itu dua penyakit yang berlainan dan obatnya pun berlainan. Kalau Anda sudah membuktikan algoritmanya benar di atas kertas dan programnya masih gagal, berarti Anda punya kutu pengkodean dan Anda harus membaca kodenya. Kalau Anda belum membuktikan apa pun, Anda tidak tahu harus mencari ke mana.

Kedua, Anda bisa membandingkan algoritma tanpa menulis kode sama sekali. Kita akan menghabiskan satu bab penuh untuk belajar mengatakan bahwa satu algoritma sejuta kali lebih cepat daripada yang lain tanpa pernah mengompilasi keduanya.

Ketiga, algoritma adalah bagian yang awet. C++ suatu hari akan digantikan. Pencarian biner (*binary search*) tidak.

1.2.3 Apa itu struktur data

> **DEFINISI.** Sebuah **struktur data** adalah cara menata, menyimpan, dan mengelola data supaya data itu dapat diakses dan diubah secara efisien.

Inilah sebuah perbandingan yang patut terus Anda ingat:

Algoritma seperti resep masakan — langkah-langkahnya. Struktur data seperti peralatan dapur — panci, wajan, ovennya. Resep yang sama dengan peralatan yang lebih baik menghasilkan masakan lebih cepat.

Pilihan struktur data bisa menjadi seluruh perbedaan antara program yang selesai dalam milidetik dan program yang tidak pernah selesai sama sekali. Di Subbab 1.11 Anda akan bertemu

persoalan yang masukannya boleh sebesar 10^{18}. Kalau disimpan dalam larik biasa, jawabannya akan menuntut memori lebih banyak daripada yang ada di seluruh Bumi. Disimpan dalam `std::map`, ia hanya perlu sekitar seribu entri dan berjalan seketika. Algoritmanya sama dalam kedua hal itu. Yang berubah hanya wadahnya.

Inilah struktur-struktur yang akan Anda temui dalam buku ini. Anda belum diharapkan mengenal satu pun di antaranya; masing-masing mendapat pembahasan sendiri ketika ia dibutuhkan.

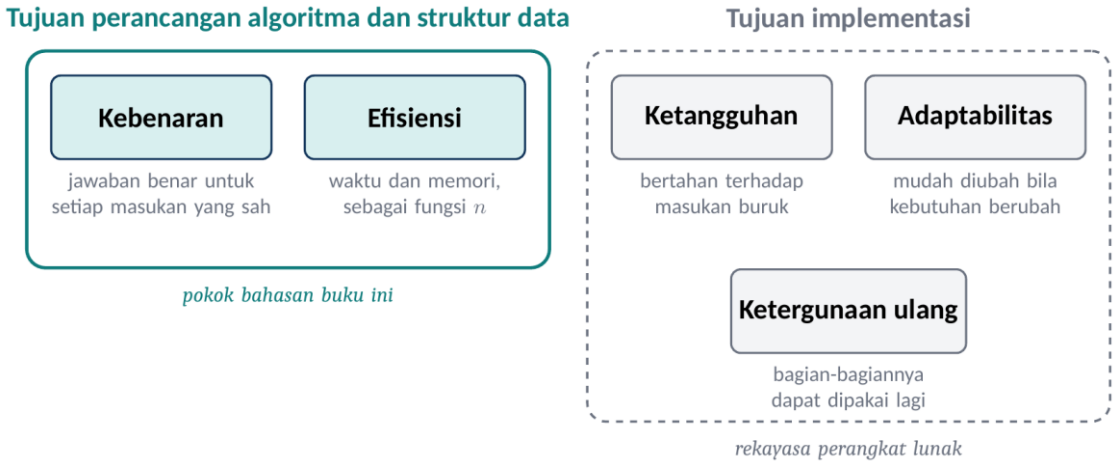
Tabel 1.1 Struktur data yang dipakai dalam buku ini, dan satu hal yang paling dikuasai masing-masing.

Struktur	Nama di C++	Keunggulan utamanya
Larik / larik dinamis	`int a[]`, `vector`	Membaca atau menulis elemen ke-i seketika
Senarai berantai	`list`	Menyisipkan dan menghapus di tengah dengan murah
Tumpukan	`stack`	Masuk terakhir, keluar pertama — *undo*, rekursi, pencocokan kurung
Antrean	`queue`	Masuk pertama, keluar pertama — penjadwalan, penelusuran melebar
Peta hash	`unordered_map`	Mencari berdasarkan kunci, rata-rata dalam waktu konstan
Pohon pencarian seimbang	`map`, `set`	Sama, tetapi kuncinya tetap tersimpan terurut
Heap / antrean prioritas	`priority_queue`	Berulang kali mengambil elemen terkecil atau terbesar
Graf	senarai ketetanggaan	Memodelkan hubungan, jalan, tautan, kebergantungan

Tabel 1.1 adalah peta wilayah, bukan daftar hafalan untuk sekarang. Anda akan menemui setiap barisnya ketika ada persoalan yang memaksa Anda menginginkannya, dan hanya dengan cara itulah orang benar-benar belajar struktur data.

1.2.4 Sasaran yang hendak kita capai

Gambar 1.1 merangkum apa yang hendak kita capai, dan ia membagi sasaran menjadi dua kelompok.



Gambar 1.1 Sasaran perancangan struktur data dan algoritma (kebenaran, efisiensi) disandingkan dengan sasaran implementasi (ketangguhan, keluwesan, kegunaan ulang).

Pasangan di sebelah kiri — **kebenaran** dan **efisiensi** — termasuk ke dalam perancangan algoritma itu sendiri, dan itulah pokok bahasan buku ini. Tiga serangkai di sebelah kanan — **ketangguhan** (program bertahan menghadapi masukan yang buruk), **keluwesan** (program bisa diubah ketika kebutuhan berubah), dan **kegunaan ulang** (sebagiannya bisa dipakai lagi di tempat lain) — termasuk ke dalam rekayasa perangkat lunak. Ketiganya nyata dan penting, tetapi

algoritma yang cepat, dapat dipakai ulang, tangguh, dan *salah* tidak ada gunanya. Jadi kita mulai dari sebelah kiri.

1.3 Dua Sasaran Perancangan Algoritma

Ketika kita merancang sebuah algoritma, ada tepat dua hal yang harus kita jamin. Bukan tiga, bukan satu. Dua.

1.3.1 Sasaran pertama: kebenaran

> **DEFINISI.** Sebuah algoritma disebut **benar** jika, untuk *setiap* masukan yang diizinkan rumusan persoalannya, ia berhenti dan menghasilkan keluaran yang diminta persoalan itu.

Bacalah definisi itu sekali lagi dan perhatikan kata *setiap*. Bukan "contoh-contoh yang ada di rumusan soal". Bukan "kasus-kasus yang terpikir olehku". Bukan "masukan yang kecil". Setiap masukan yang diizinkan batasannya.

Tuntutan ini lebih keras daripada yang disadari kebanyakan pemula, dan inilah sebabnya kita tidak dapat menegakkan kebenaran lewat pengujian. Misalkan sebuah persoalan menyatakan $1 \leq n \leq 1000$ dan $1 \leq a_i \leq 1000$. Maka banyaknya masukan yang mungkin sungguh besar tak terbayangkan — untuk $n = 1000$ saja jumlahnya 1000^{1000}, sebuah bilangan dengan tiga ribu satu angka. Anda boleh menguji satu masukan per nanodetik sampai matahari padam dan Anda belum juga menggores permukaannya. Pengujian dapat menunjukkan adanya kutu; ia tidak akan pernah dapat menunjukkan ketiadaannya. Hanya pembuktian yang bisa.

Ada empat cara baku untuk membuktikan sebuah algoritma benar, dan Anda akan memakai keempatnya dalam buku ini.

Bukti langsung. Tunjukkan, dengan penalaran lurus dari definisi-definisinya, bahwa logika algoritma itu selalu berujung pada jawaban yang diminta. Sebagian besar dari enam persoalan dalam bab ini dibuktikan secara langsung: kita tunjukkan bahwa jawabannya tidak mungkin kurang dari suatu nilai, lalu kita *bangun* sebuah susunan yang mencapai persis nilai itu, dan kedua belahan itu memakukan jawabannya.

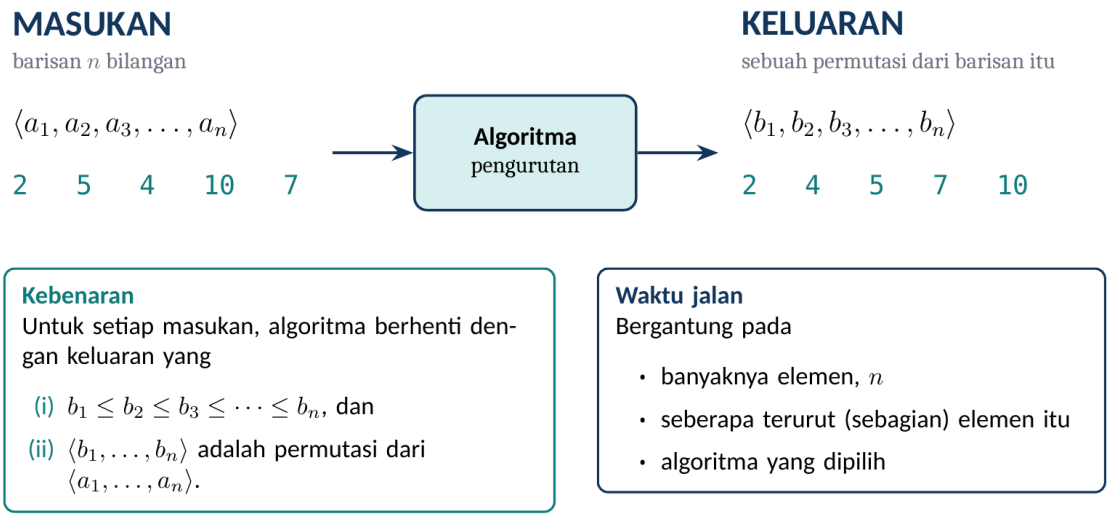
Bukti dengan kontradiksi. Andaikan algoritma itu menghasilkan jawaban yang salah pada suatu masukan, lalu turunkan dari pengandaian itu sesuatu yang mustahil. Karena yang mustahil tidak dapat terjadi, pengandaiannya pasti keliru, jadi algoritmanya benar. Kita akan memakai cara ini untuk persoalan *Journey* di Subbab 1.7: kita andaikan ada perjalanan yang lebih pendek daripada $2 \times$ (rentang), lalu kita tunjukkan bahwa ia pasti gagal menyinggahi salah satu kota.

Invarian gelung (*loop invariant*). Kenali sebuah sifat yang benar sebelum gelung dimulai, tetap benar melintasi setiap putaran gelung, dan — digabung dengan alasan gelung itu berakhir — memberi persis hasil yang kita inginkan. Inilah teknik andalan untuk setiap algoritma yang dibangun di sekitar sebuah gelung, dan ia pantas mendapat subbabnya sendiri. Lihat Subbab 1.3.3.

Induksi. Buktikan pernyataannya untuk kasus terkecil, lalu buktikan bahwa jika ia berlaku untuk kasus berukuran k maka ia pasti berlaku untuk ukuran $k + 1$. Karena ia berlaku di anak tangga terbawah, dan setiap anak tangga mengantar ke anak tangga berikutnya, ia berlaku di mana-mana. Kita memakai induksi di Subbab 1.10 untuk membuktikan sebuah rumus tentang barisan yang panjangnya berlipat dua selamanya.

1.3.2 Sebuah persoalan yang dinyatakan dengan cermat: pengurutan

Sebelum kita dapat membuktikan apa pun tentang algoritma pengurutan, kita harus menyatakan persis apa itu *mengurutkan*. Inilah Fase 1 dari kerangka kerja kita, diterapkan pada persoalan yang paling banyak dipelajari dalam ilmu komputer, dan **Gambar 1.2** menggambarkannya.



Gambar 1.2 Persoalan pengurutan: sebuah barisan masuk, sebuah permutasi dari barisan itu keluar, dan kebenaran berarti dua syarat terpisah harus sama-sama terpenuhi.

> **PERSOALAN PENGURUTAN.** *Masukan:* sebuah barisan berisi n bilangan $a_1, a_2, \ldots, a_n$. *Keluaran:* sebuah barisan $b_1, b_2, \ldots, b_n$ sedemikian rupa sehingga (i) $b_1 \le b_2 \le \cdots \le b_n$, dan (ii) $b_1, b_2, \ldots, b_n$ adalah permutasi dari $a_1, a_2, \ldots, a_n$.

Mari kita bongkar setiap lambangnya, sebab inilah rumusan persoalan formal yang pertama dalam buku ini.

Huruf n melambangkan *berapa banyak bilangan yang ada*. Notasi $a_1, a_2, \ldots, a_n$ berarti: bilangan pertama dinamai a_1 (dibaca "a sub satu"), bilangan kedua dinamai a_2, dan seterusnya sampai yang terakhir, yang dinamai a_n. Angka kecil yang ditulis di bawah dan di kanan itu disebut **subskrip** atau **indeks**; ia sebuah label, bukan perkalian. Lambang $\le$ berarti "kurang dari atau sama dengan". Kata **permutasi** berarti penyusunan ulang: kumpulan bilangan yang sama, boleh jadi dalam urutan yang berbeda, tanpa ada yang ditambahkan dan tanpa ada yang dibuang.

Syarat (i) menyatakan bahwa keluarannya tersusun tidak-menurun. Syarat (ii) menyatakan bahwa keluarannya memuat persis bilangan yang sama dengan masukannya.

Kedua syarat itu sama-sama diperlukan, dan inilah alasannya. Ambil masukan

$$a = [\,2,\ 5,\ 4,\ 10,\ 7\,].$$

Algoritma yang mengeluarkan [2,4,5,7,10] benar: barisannya terurut, dan ia memakai persis bilangan 2,4,5,7,10, yang persis bilangan-bilangan yang tadi diberikan kepada kita.

Algoritma yang mengeluarkan [1,2,3,4,5] memenuhi syarat (i) dengan sangat indah — ia terurut sempurna — dan gagal total pada syarat (ii). Bilangan 1 dan 3 tidak pernah ada di masukan; bilangan 7 dan 10 raib. Algoritma itu **salah**. Pemeriksa yang hanya bertanya "apakah keluarannya terurut?" akan dengan senang hati menerima algoritma ini. Inilah sebabnya rumusan persoalan, bukan pengujian, yang mendefinisikan kebenaran.

Algoritma yang mengeluarkan [2,5,4,10,7] apa adanya memenuhi (ii) dan gagal pada (i). Salah juga.

1.3.3 Insertion sort, dan invarian gelung yang membuktikannya

Berikut sebuah algoritma pengurutan yang sebetulnya sudah Anda kenal, sekalipun Anda belum pernah melihatnya tertulis: begitulah hampir semua orang mengurutkan kartu di tangan. Kartu yang sudah tersusun Anda pegang di tangan kiri. Kartu berikutnya Anda ambil dengan tangan kanan. Anda menggesernya ke kiri melewati setiap kartu yang lebih besar daripadanya, sampai Anda bertemu kartu yang tidak, lalu Anda jatuhkan ke celah itu. Ulangi sampai meja kosong.

Itulah **insertion sort** (pengurutan sisip). **Kode 1.1** adalah keseluruhannya dalam C++.

Kode 1.1 Insertion sort dalam C++, utuh dan dapat dijalankan, mengurutkan barisan dari **Gambar 1.2**.

```
#include <cstdio>

void insertionSort(int a[], int n) {
    for (int j = 1; j < n; j++) {          // a[0..j-1] sudah terurut
        int key = a[j];                    // kartu yang hendak kita letakkan
        int i = j - 1;
        while (i >= 0 && a[i] > key) {     // geser kartu yang lebih besar satu ke kanan
            a[i + 1] = a[i];
            i = i - 1;
        }
        a[i + 1] = key;                    // celah yang ditinggalkan itulah tempat key
    }
}

int main() {
    int a[] = {2, 5, 4, 10, 7};
    int n = 5;
    insertionSort(a, n);
    for (int i = 0; i < n; i++) printf("%d%c", a[i], i + 1 == n ? '\n' : ' ');
    return 0;
}
```

Dikompilasi dengan `g++ -std=c++17 -O2` lalu dijalankan, **Kode 1.1** mencetak:

```
2 4 5 7 10
```

yang persis keluaran yang tampak di sebelah kanan **Gambar 1.2**. Namun jawaban yang betul pada satu masukan tidak membuktikan apa-apa, sebagaimana kita bantah di atas. Mari kita buktikan dengan benar.

> **DEFINISI.** Sebuah **invarian gelung** (*loop invariant*) adalah pernyataan tentang peubah-peubah program yang (a) **benar sebelum gelung berjalan sama sekali** — *inisialisasi*; (b) **tetap benar di akhir setiap putaran, dengan syarat ia benar di awal putaran itu** — *pemeliharaan*; dan (c) **berguna ketika gelung itu akhirnya berhenti** — *terminasi*.

Invarian adalah jembatan antara "saya sudah menjalankan gelungnya beberapa kali dan kelihatannya beres" dan "saya tahu apa yang dikerjakan gelung ini untuk setiap masukan". Gagasan ini berpangkal pada Floyd dan Hoare di tahun 1960-an dan digarap menjadi disiplin pemrograman yang utuh oleh Gries [2], yang bukunya masih merupakan pembahasan paling jernih tentang dari mana invarian itu datang. Inilah invarian untuk insertion sort.

> **INVARIAN GELUNG (insertion sort).** Di awal setiap putaran gelung `for` dengan pencacah j, sub-larik $a[0 \ldots j-1]$ memuat persis elemen-elemen yang semula menempati posisi 0 sampai $j-1$, dan elemen-elemen itu tersusun terurut.

Inisialisasi. Sebelum putaran pertama, $j = 1$. Sub-larik $a[0 \dots 0]$ berisi satu elemen. Satu elemen tunggal pasti terurut — tidak ada pasangan elemen yang bisa salah urutan — dan ia pasti penghuni asli posisi 0, sebab belum ada apa pun yang berpindah. Invarian terpenuhi.

Pemeliharaan. Andaikan invarian terpenuhi di awal sebuah putaran, jadi $a[0 \dots j-1]$ terurut dan memuat elemen yang benar. Di dalam badan gelung kita simpan `key = a[j]`. Gelung `while` kemudian menyalin $a[i]$ ke $a[i+1]$ untuk setiap i yang melangkah ke kiri, selama $a[i] > \mathtt{key}$. Dua hal benar tentang penggeseran ini:

- Tidak ada yang hilang. Nilai yang tertimpa di posisi $i+1$ sudah lebih dulu disalin satu langkah lebih ke kanan (atau, pada geseran yang paling pertama, nilai itu adalah `key` sendiri, yang sudah kita simpan dalam sebuah peubah). Jadi multihimpunan nilai di $a[0 \dots j]$ tidak berubah.
- Gelung berhenti di posisi i pertama yang memenuhi $a[i] \leq \mathtt{key}$, atau di $i = -1$ kalau ia kelewat sampai ujung kiri. Menuliskan `key` ke $a[i+1]$ karena itu menempatkannya sesudah setiap elemen yang $\leq$ ia dan sebelum setiap elemen yang $>$ ia, dan karena elemen di kiri dan di kanannya sudah terurut di antara mereka sendiri, seluruh $a[0 \dots j]$ kini terurut.

Sesudah badan gelung, j bertambah satu, sehingga klaim invarian tentang $a[0 \dots j-1]$ persis sama dengan klaim yang baru saja kita tegakkan tentang $a[0 \dots j]$. Invarian terpelihara.

Terminasi. Gelung berhenti ketika $j = n$. Menyulihkan $j = n$ ke dalam invarian memberi: $a[0 \dots n-1]$ memuat persis elemen-elemen asalnya dan tersusun terurut. Itu persis syarat (ii) dan (i) dari persoalan pengurutan. Algoritmanya benar. ■

Perhatikan bahwa kita sama sekali tidak menyebut bilangan 2,5,4,10,7. Pembuktian itu mencakup setiap masukan dari setiap ukuran — dan untuk itulah pembuktian ada.

Untuk melihat invarian itu dengan mata kepala sendiri, **Kode 1.2** mencetak lariknya sesudah setiap putaran, dengan sebuah garis yang menandai batas antara bagian terurut dan bagian yang belum tersentuh.

Kode 1.2 Versi penelusuran dari insertion sort yang mencetak invarian gelung sesudah setiap putaran.

```
#include <cstdio>

int a[] = {2, 5, 4, 10, 7};
const int n = 5;

void show(const char* label, int j) {
    printf("%s", label);
    for (int i = 0; i < n; i++) {
        if (i == j) printf(" |");          // garis penanda batas terurut/belum
        printf(" %d", a[i]);
    }
    printf("\n");
}

int main() {
    show("start        ", 1);
    for (int j = 1; j < n; j++) {
        int key = a[j];
        int i = j - 1;
        while (i >= 0 && a[i] > key) { a[i + 1] = a[i]; i = i - 1; }
        a[i + 1] = key;
        printf("after j = %d: ", j);
        for (int t = 0; t < n; t++) {
            if (t == j + 1) printf(" |");
            printf(" %d", a[t]);
        }
        printf("\n");
    }
    return 0;
}
```

Keluarannya yang sesungguhnya:

```
start         2 | 5 4 10 7
after j = 1:  2 5 | 4 10 7
after j = 2:  2 4 5 | 10 7
after j = 3:  2 4 5 10 | 7
after j = 4:  2 4 5 7 10
```

Bacalah bagian kiri setiap garis. `2`, lalu `2 5`, lalu `2 4 5`, lalu `2 4 5 10`, lalu seluruh larik. Terurut setiap kali, dan selalu memuat persis elemen yang semula ada di posisi-posisi itu. Itulah invariannya, dibuat kasatmata.

1.3.4 Sasaran kedua: efisiensi

Sebuah algoritma juga harus memakai waktu dan memori sesedikit yang masuk akal. Dua algoritma bisa sama-sama benar sempurna dan tetap dipisahkan oleh jurang yang tidak dapat dijembatani oleh komputer yang lebih cepat mana pun.

> **MENGURUTKAN SATU JUTA BILANGAN.** Bubble sort mengerjakan kira-kira n^2 operasi. Dengan $n = 10^6$ itu berarti $(10^6)^2 = 10^{12}$ operasi — sekitar satu jam pada mesin masa kini. Merge sort mengerjakan kira-kira $n\log_2 n$ operasi. Dengan $n = 10^6$ itu sekitar 2×10^7 operasi — sekitar dua puluh milidetik. Keduanya benar. Yang satu kira-kira lima puluh ribu kali lebih cepat.

Kita mengukur efisiensi dengan **notasi Big-O**, yang akan kita definisikan secara ketat di Bab 3. Untuk sekarang, perlakukan $O(f(n))$ sebagai pernyataan tentang *bagaimana waktu jalan tumbuh ketika masukannya membesar*, dengan mengabaikan faktor konstanta dan suku-suku kecil. Kalau sebuah algoritma $O(n^2)$ dan Anda melipatduakan n, harapkan waktunya melipatempat. Kalau ia $O(n)$ dan Anda melipatduakan n, harapkan waktunya melipatdua. Kalau ia $O(\log n)$ dan Anda melipatduakan n, harapkan waktunya naik satu langkah saja.

Tabel 1.2 mendaftar laju pertumbuhan yang akan Anda temui, lengkap dengan cacah operasi yang tersirat pada $n = 1000$.

Tabel 1.2 Laju pertumbuhan yang umum dijumpai, dan berapa biaya masing-masing pada $n = 1000$.

Notasi	Nama	Operasi saat n=1000	Bagaimana rasanya
$O(1)$	Konstan	1	Seketika
$O(\log n)$	Logaritmik	sekitar 10	Seketika
$O(n)$	Linear	1,000	Cepat
$O(n\log n)$	Linearitmik	sekitar 10,000	Cepat
$O(n^2)$	Kuadratik	1,000,000	Lambat begitu n membesar
$O(n^3)$	Kubik	10^9	Sangat lambat
$O(2^n)$	Eksponensial	sekitar 10^{301}	Mustahil

Dua baris di **Tabel 1.2** pantas mendapat penjelasan. Logaritma $\log_2 n$ adalah *berapa kali Anda dapat membagi dua n sebelum mencapai* 1. Membagi dua 1000 memberi 500,250,125,62,31,15,7,3,1 — sembilan atau sepuluh langkah. Itulah sebabnya $O(\log n)$ ditulis "sekitar 10" dalam tabel. Dan 2^n pada $n = 1000$ adalah 2^{1000}, yang kira-kira 10^{301}: sebuah bilangan dengan tiga ratus dua angka. Alam semesta yang teramati berisi kira-kira 10^{80} atom. Algoritma eksponensial pada $n = 1000$ bukan lambat; ia mustahil dalam arti fisik yang sesungguhnya.

1.3.5 Patokan kasar yang menentukan segalanya

Inilah satu fakta yang paling berguna secara praktis dalam bab ini.

> **WAWASAN KUNCI — ANGGARAN OPERASI.** Sebuah komputer masa kini mengerjakan kira-kira 10^8 sampai 10^9 operasi sederhana per detik. Kalau algoritma Anda, pada masukan terbesar yang diizinkan batasannya, akan mengerjakan jauh lebih banyak daripada kira-kira 10^8 operasi, ia sangat mungkin akan melewati batas waktu.

Yang dimaksud "operasi sederhana" di sini adalah satu penjumlahan, satu perbandingan, satu pengaksesan larik — hal semacam itu, yang menjadi segelintir instruksi mesin. Operasi yang melibatkan `map` atau pembagian beberapa kali lipat lebih mahal; operasi yang melibatkan pembacaan dari cakram ribuan kali lebih mahal. Angka 10^8 itu sengaja dibuat kasar. Ia kasar dan ia cukup.

Mengapa patokan ini begitu penting? Sebab ia mengubah batasan yang tercetak dalam rumusan persoalan menjadi *keputusan rancangan yang diambil sebelum Anda menulis kode apa pun*. Lihat n terbesar yang diizinkan persoalan, putuskan kompleksitas mana yang muat dalam anggaran, dan baru sesudah itu mulailah merancang. **Tabel 1.3** sudah mengerjakan hitungannya untuk Anda.

Tabel 1.3 Mengubah sebuah batasan menjadi keputusan rancangan. Angka-angka di kolom tengah dihitung oleh sebuah program C++ pendek; kolom terakhir menerapkan anggaran 10^8 pada angka-angka itu.

n Terbesar	n	$n\log_2 n$	n^2	Apa yang sanggup Anda pakai
10^3	10^3	$\approx 10^4$	10^6	Bahkan $O(n^3)$ masih aman
10^5	10^5	$\approx 1.7 \times 10^6$	10^{10}	$O(n\log n)$ boleh, $O(n^2)$ tidak
10^6	10^6	$\approx 2.0 \times 10^7$	10^{12}	$O(n\log n)$ boleh, $O(n^2)$ tanpa harapan
10^8	10^8	$\approx 2.7 \times 10^9$	10^{16}	Hanya $O(n)$, dan harus hati-hati
10^{18}	—	—	—	Hanya $O(\log n)$ atau $O(1)$

Lihat baris terakhir **Tabel 1.3**. Ketika sebuah persoalan mengizinkan n sampai 10^{18} — dan Subbab 1.11 memuat persoalan yang persis seperti itu — Anda bahkan tidak dapat *membaca* n buah elemen, apalagi mengolahnya. Setiap algoritma yang menyentuh n benda satu per satu sudah mati sebelum berangkat. Batasan itu sedang memberi tahu Anda, dengan lantang, bahwa penyelesaian yang dimaksudkan tidak pernah melihat masukan satu elemen demi satu elemen. Ia harus memanfaatkan struktur. Pemula melihat 10^{18} lalu panik. Pemecah persoalan yang berpengalaman melihat 10^{18} lalu tersenyum, sebab batasan itu baru saja menyingkirkan sembilan puluh persen ruang pencarian rancangan yang mungkin.

1.3.6 Kedua sasaran bekerja bersama

Kebenaran tanpa efisiensi tidak berguna: program Anda betul tetapi tidak pernah selesai, sehingga juri melaporkan *Time Limit Exceeded* dan nilai Anda nol. Efisiensi tanpa kebenaran sama tidak bergunanya: program Anda selesai seketika lalu mencetak angka yang salah, sehingga juri melaporkan *Wrong Answer* dan nilai Anda nol. Tidak ada nilai sebagian untuk setengah betul dengan cepat.

Anda memerlukan keduanya, sekaligus, dan sisa buku ini berbicara tentang bagaimana memperoleh keduanya.

1.4 Lima Paradigma Perancangan

Tidak ada resep tunggal untuk merancang algoritma. Kalau ada, buku ini cukup setebal satu halaman. Yang ada justru sejumlah kecil *paradigma* — siasat umum, yang masing-masing berlaku untuk satu keluarga persoalan yang luas dan masing-masing punya gejala yang dapat dikenali. Belajar mengenali gejalanya adalah sebagian besar dari keterampilan ini.

Ada lima dalam buku ini.

1.4.1 Inkremental, atau *brute force*

Gagasannya. Bangun penyelesaiannya sepotong demi sepotong. Olah elemen masukan satu per satu sambil memelihara semacam keadaan berjalan, dan ketika masukannya habis, keadaan berjalan itulah jawabannya.

Insertion sort bersifat inkremental: ia memelihara sebuah prefiks yang terurut dan menumbuhkannya satu kartu setiap putaran. Begitu pula algoritma yang paling sederhana dari semuanya, memindai untuk mencari nilai maksimum, yang tampak pada **Kode 1.3**.

Kode 1.3 Paradigma inkremental dalam wujudnya yang paling sederhana: satu peubah berjalan, satu kali pindai.

```
#include <cstdio>

int main() {
    int a[] = {3, 9, 2, 9, 4, 1};
    int n = 6;
    int best = a[0];                        // maksimum dari prefiks satu elemen a[0..0]
    for (int i = 1; i < n; i++)
        if (a[i] > best) best = a[i];       // perluas prefiks sebanyak satu elemen
    printf("maximum = %d\n", best);
    return 0;
}
```

Keluarannya yang sesungguhnya:

```
maximum = 9
```

Invarian gelungnya adalah `best` *memuat nilai terbesar di antara* $a[0 \ldots i-1]$. Sebelum gelung, $i = 1$ dan `best` $= a[0]$, yang memang terbesar dari rentang satu elemen. Setiap putaran membandingkan $a[i]$ dengan sang juara dan menyimpan pemenangnya, jadi invariannya selamat. Ketika gelung berhenti, $i = n$ dan `best` adalah yang terbesar dari $a[0 \ldots n-1]$: seluruh larik. Dua dari enam persoalan bab ini — *Columns* dan *Journey* — diselesaikan oleh gelung yang bentuknya persis seperti ini.

Istilah **brute force** dipakai ketika proses inkremental itu sekadar mencoba setiap kemungkinan. *Brute force* sering kali merupakan pikiran pertama yang tepat, sebab ia biasanya jelas-jelas benar, dan punya sesuatu yang jelas-jelas benar sebagai pembanding sangat berharga ketika Anda sedang mengawakutu sesuatu yang cerdik.

1.4.2 *Greedy*

Gagasannya. Pada setiap langkah, ambil pilihan yang tampak paling baik *saat ini juga*, dan jangan pernah meninjaunya ulang.

Sifat rakus memang menggoda dan sering kali keliru. Kalau Anda berdiri di lereng bukit dalam kabut dan selalu berjalan menanjak, Anda akan sampai ke *sebuah* puncak, tetapi belum tentu yang tertinggi. Algoritma *greedy* (rakus) hanya benar ketika persoalannya memiliki apa yang disebut **sifat pilihan *greedy***: mengambil pilihan terbaik setempat tidak pernah memusnahkan kemungkinan memperoleh penyelesaian terbaik secara menyeluruh. Ketika sebuah persoalan memilikinya, algoritma *greedy* menjadi sangat pendek dan cepat. Ketika tidak, algoritma itu menjadi sangat pendek, cepat, dan salah.

Membuktikan sifat pilihan *greedy* karena itu bukan pilihan bebas, dan Bab 9 dikhususkan untuknya. Dalam bab ini, tiga dari enam persoalan bersifat *greedy* — *Columns*, *Journey*, dan penyusunan angka pada *Minimum and Maximum Numbers* — dan pada setiap kasus kita akan membuktikan bahwa pilihan rakusnya aman alih-alih mengandaikannya begitu saja.

Contoh klasik yang akan Anda temui nanti adalah algoritma Kruskal untuk pohon merentang minimum: berulang kali tambahkan sisi termurah yang tersisa yang tidak membentuk siklus. Ia terbukti optimal, dan buktinya tidak gamblang.

1.4.3 *Divide and conquer*

Gagasannya. Pecah persoalan menjadi sub-persoalan *yang sejenis*, selesaikan sub-persoalan itu secara rekursif, lalu gabungkan jawabannya menjadi jawaban untuk keseluruhannya.

Merge sort adalah ilustrasi bakunya, dan **Kode 1.4** adalah implementasi utuhnya.

Kode 1.4 Merge sort: belah jadi dua, urutkan tiap belahan, gabungkan. Paradigma *divide and conquer* dalam dua puluh baris.

```
#include <cstdio>

int tmp[64];

void mergeSort(int a[], int lo, int hi) {   // mengurutkan a[lo..hi]
    if (lo >= hi) return;                   // deret satu elemen sudah pasti terurut
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, lo, mid);
    mergeSort(a, mid + 1, hi);
    int i = lo, j = mid + 1, k = lo;
    while (i <= mid && j <= hi) tmp[k++] = (a[i] <= a[j]) ? a[i++] : a[j++];
    while (i <= mid) tmp[k++] = a[i++];
    while (j <= hi)  tmp[k++] = a[j++];
    for (int t = lo; t <= hi; t++) a[t] = tmp[t];
}

int main() {
    int a[] = {2, 5, 4, 10, 7, 1, 8, 3};
    int n = 8;
    mergeSort(a, 0, n - 1);
    for (int i = 0; i < n; i++) printf("%d%c", a[i], i + 1 == n ? '\n' : ' ');
    return 0;
}
```

Keluarannya yang sesungguhnya:

```
1 2 3 4 5 7 8 10
```

Ketiga langkah khasnya tampak di dalam kode. *Bagi*: hitung `mid` dan potong rentangnya menjadi dua. *Kuasai*: dua pemanggilan rekursif. *Gabung*: gelung-gelung penggabungan. Bab 3 menunjukkan cara menghitung waktu jalan algoritma semacam itu dari sebuah rekurensi, dan Bab 7 mengembangkan paradigma ini selengkapnya.

1.4.4 *Dynamic programming*

Gagasannya. Seperti *divide and conquer*, pecah persoalan menjadi sub-persoalan — tetapi di sini sub-persoalannya **bertumpang tindih**: sub-persoalan yang sama dibutuhkan berkali-kali. Selesaikan masing-masing sekali saja, simpan jawabannya, dan sesudah itu tinggal melihatnya kembali.

Peragaan yang paling khas adalah barisan Fibonacci, yang didefinisikan oleh $F(0) = 0$, $F(1) = 1$, dan $F(n) = F(n-1) + F(n-2)$ untuk $n \geq 2$. Ditulis sebagai rekursi langsung, menghitung $F(30)$ menghitung ulang $F(28)$ dua kali, $F(27)$ tiga kali, $F(26)$ lima kali, dan seterusnya: banyaknya pemanggilan itu sendiri tumbuh seperti bilangan Fibonacci, yang artinya eksponensial. **Kode 1.5** mengukur kerusakannya lalu memperbaikinya.

Kode 1.5 Rekursi naif dibandingkan rekursi bermemoisasi untuk bilangan Fibonacci. Pencacah pemanggilan membuat selisihnya mustahil dibantah.

```
#include <cstdio>
#include <map>
using namespace std;

long long calls = 0;                        // berapa kali fibNaive dimasuki

long long fibNaive(int n) {
    calls++;
    if (n <= 1) return n;
    return fibNaive(n - 1) + fibNaive(n - 2);
}

map<int, long long> memo;

long long fibMemo(int n) {
    if (n <= 1) return n;
    if (memo.count(n)) return memo[n];      // sudah pernah dihitung: pakai ulang
    return memo[n] = fibMemo(n - 1) + fibMemo(n - 2);
}

int main() {
    calls = 0;
    long long v1 = fibNaive(30);            // selesaikan pemanggilan SEBELUM membaca
`calls`
    long long c1 = calls;
    printf("fibNaive(30) = %lld, calls made = %lld\n", v1, c1);

    long long v2 = fibMemo(30);
    int stored = (int)memo.size();
    printf("fibMemo(30)  = %lld, map entries = %d\n", v2, stored);
    printf("fibMemo(90)  = %lld\n", fibMemo(90));
    return 0;
}
```

Keluarannya yang sesungguhnya:

```
fibNaive(30) = 832040, calls made = 2692537
fibMemo(30)  = 832040, map entries = 29
fibMemo(90)  = 2880067194370816120
```

Dua juta enam ratus sembilan puluh dua ribu lima ratus tiga puluh tujuh pemanggilan untuk menghitung sebuah bilangan yang dicapai memoisasi dengan dua puluh sembilan nilai tersimpan. Dan `fibMemo(90)` menjawab seketika, sementara `fibNaive(90)` akan memakan waktu lebih lama daripada umur alam semesta. Menyimpan jawaban alih-alih menghitungnya ulang disebut **memoisasi**, dan Subbab 1.11 menerapkannya pada persoalan terakhir bab ini.

1.4.5 Matematis dan teori bilangan

Gagasannya. Sebagian persoalan larut seluruhnya begitu Anda melihat potongan matematika yang tepat. Tidak ada gelung atas datanya, tidak ada rekursi — hanya sebuah rumus.

Contoh klasiknya adalah **faktor persekutuan terbesar** (*greatest common divisor*, gcd). Faktor persekutuan terbesar dari dua bilangan bulat positif a dan b, ditulis $\gcd(a, b)$, adalah bilangan terbesar yang membagi habis keduanya. Cara yang gamblang adalah menguji setiap calon dari 1 ke atas, yang memakan kira-kira $\min(a, b)$ langkah. Sebaliknya, metode Euclid, dari sekitar 300 SM, memakai identitas

$$\gcd(a, b) = \gcd(b,\ a \bmod b) \qquad (b > 0), \qquad \gcd(a, 0) = a, \tag{1.1}$$

dengan $a \bmod b$ berarti sisa pembagian a oleh b. Mengapa identitas itu benar? Tulislah $a = qb + r$, dengan q hasil bagi bulatnya dan $r = a \bmod b$. Setiap bilangan d yang membagi habis a dan b

juga membagi habis $a - qb = r$; dan setiap bilangan yang membagi habis b dan r juga membagi habis $qb + r = a$. Jadi pasangan (a, b) dan (b, r) mempunyai faktor persekutuan yang persis sama, dan karena itu faktor persekutuan *terbesar* yang sama pula.

Setiap penerapan Persamaan (1.1) mengganti pasangan itu dengan pasangan yang benar-benar lebih kecil, sebab sisa r selalu kurang dari b. Penyusutannya pun cepat: sesudah dua langkah berturut-turut, bilangan yang lebih besar sudah turun menjadi kurang dari separuh nilainya semula (jika $b \leq a/2$ maka $r < b \leq a/2$; jika $b > a/2$ maka $r = a - b < a/2$). Sebuah bilangan hanya dapat dibagi dua kira-kira sebanyak $\log_2$ dirinya sebelum habis, sehingga yang dibutuhkan tidak pernah lebih dari $O(\text{logmin}(a, b))$ langkah. **Kode 1.6** menyandingkan kedua metode itu.

Kode 1.6 Faktor persekutuan terbesar dihitung dua kali: dengan pencarian menyeluruh, dan dengan algoritma Euclid.

```
#include <cstdio>

long long gcdSlow(long long a, long long b) {   // coba setiap calon pembagi
    long long g = 1;
    for (long long d = 1; d <= a && d <= b; d++)
        if (a % d == 0 && b % d == 0) g = d;
    return g;
}

long long gcdEuclid(long long a, long long b) { // Euclid: gcd(a,b) = gcd(b, a mod b)
    while (b != 0) { long long r = a % b; a = b; b = r; }
    return a;
}

int main() {
    printf("gcdSlow(1071, 462)   = %lld\n", gcdSlow(1071, 462));
    printf("gcdEuclid(1071, 462) = %lld\n", gcdEuclid(1071, 462));
    printf("gcdEuclid(30, 8)     = %lld\n", gcdEuclid(30, 8));
    printf("gcdEuclid(15, 12)    = %lld\n", gcdEuclid(15, 12));
    return 0;
}
```

Keluarannya yang sesungguhnya:

```
gcdSlow(1071, 462)   = 21
gcdEuclid(1071, 462) = 21
gcdEuclid(30, 8)     = 2
gcdEuclid(15, 12)    = 3
```

Mari kita telusuri metode Euclid pada gcd(1071,462), satu baris untuk satu langkah:

$$\text{gcd}(1071,462): \quad 1071 \bmod 462 = 147 \;\Rightarrow\; \text{gcd}(462,147)$$

$$\text{gcd}(462,147): \quad 462 \bmod 147 = 21 \;\Rightarrow\; \text{gcd}(147,21)$$

$$\text{gcd}(147,21): \quad 147 \bmod 21 = 0 \;\Rightarrow\; \text{gcd}(21,0)$$

$$\text{gcd}(21,0) = 21.$$

Tiga langkah, sementara pencarian menyeluruh akan memakan empat ratus enam puluh dua langkah. Subbab 1.8 memuat sebuah persoalan yang *seluruhnya* merupakan penerapan satu kesamaan ini — dan dua baris terakhir keluaran di atas, gcd(30,8) = 2 dan gcd(15,12) = 3, adalah dua bilangan yang akan kita perlukan di sana.

1.4.6 Lima paradigma dalam satu tabel

Tabel 1.4 Lima paradigma perancangan, satu langkah yang menjadi cirinya, dan di mana masing-masing muncul dalam bab ini.

Paradigma	Langkah khasnya	Benar ketika	Muncul di bab ini
Inkremental / *brute force*	Perluas jawaban sebagian satu elemen	Hampir selalu; bisa jadi terlalu lambat	*Columns*, *Journey*
Greedy	Ambil pilihan terbaik setempat, jangan ditinjau ulang	Sifat pilihan *greedy* berlaku	*Columns*, *Journey*, *Min/Max Numbers*
Divide and conquer	Belah, rekursi, gabungkan	Sub-persoalannya saling bebas	Merge sort (**Kode 1.4**)
Dynamic programming	Belah, rekursi, **simpan**	Sub-persoalannya bertumpang tindih	*Recursive Function 1*
Matematis / teori bilangan	Ganti pencarian dengan sebuah rumus	Anda dapat membuktikan rumusnya	*Unique Numbers*, *Interesting Sequence*

Tabel 1.4 pantas Anda tengok kembali setiap kali Anda buntu. Tanyakan, berurutan: bisakah saya cukup memindai? bisakah saya rakus dan membuktikannya? bisakah saya membelahnya? apakah belahannya berulang? adakah rumusnya? Salah satu dari kelimanya hampir selalu berlaku.

1.5 Kerangka Lima Fase Pemecahan Persoalan

Setiap persoalan dalam buku ini — setiap persoalan dalam keempat bab evaluasi (Bab 4, 8, 12, dan 16), setiap persoalan yang kelak diserahkan kepada Anda di tempat kerja — diselesaikan dengan lima fase yang sama, dalam urutan yang sama. Inilah hal terpenting dalam bab ini. Pelajari di sini dan Anda akan memakainya selama dua puluh tahun.

Tabel 1.5 Lima fase itu. Tidak satu pun di sini yang boleh dilewati, dan urutannya tidak bisa ditawar.

Fase	Nama	Apa yang Anda hasilkan
1	Memahami	Pernyataan ulang dengan kata-kata Anda sendiri, masukan/keluaran/batasan yang persis, dan satu contoh kecil yang dikerjakan dengan tangan
2	Mengabstraksi	Inti matematis persoalan, dengan ceritanya dilucuti habis
3	Merancang	Algoritmanya dalam bahasa sehari-hari atau pseudokode, beserta struktur data yang diperlukannya
4	Menganalisis	Bukti kebenaran, kompleksitas waktu, kompleksitas ruang, dan pemeriksaan terhadap batasannya
5	Mengimplementasikan	Kode yang bersih, terjemahan langsung dari Fase 3

Tabel 1.5 menyebutkan kelima fase itu dan, yang lebih berguna, menyebutkan apa yang harus *dihasilkan* masing-masing. Fase yang tidak dapat Anda tunjukkan wujud hasilnya adalah fase yang Anda lewati. Kalau Anda tidak punya contoh yang dikerjakan dengan tangan, Anda belum mengerjakan Fase 1; kalau Anda tidak punya angka kompleksitas, Anda belum mengerjakan Fase 4.

1.5.1 Fase 1 — Memahami

Baca persoalannya. Lalu baca lagi. Lalu, tanpa melihat rumusannya, tuliskan dengan kata-kata Anda sendiri apa yang ditanyakan. Kalau Anda tidak sanggup, berarti Anda belum memahaminya, dan segala sesuatu yang Anda bangun mulai dari sini akan berdiri di atas pasir.

Secara nyata, Fase 1 menghasilkan empat hal:

- **Masukannya.** Persis apa yang datang, dalam urutan yang persis apa, dalam bentuk yang persis apa.
- **Keluarannya.** Persis apa yang harus dicetak, dalam bentuk yang persis apa, termasuk apakah ada baris baru di ujung dan apakah bilangannya dipisahkan spasi atau pindah baris.
- **Batasannya.** Nilai terkecil dan terbesar dari segala sesuatu. Ini bukan hiasan. Sebagaimana ditunjukkan **Tabel 1.3**, batasanlah yang memutuskan rancangan.
- **Sebuah contoh kecil, dikerjakan dengan tangan.** Bukan dengan komputer. Dengan tangan, di atas kertas, dengan pensil. Inilah langkah yang paling sering dilewati dan inilah langkah yang paling sering menemukan salah paham.

> **PERINGATAN:** Masukan contoh dan keluaran contoh di dalam rumusan persoalan adalah bagian dari spesifikasi, bukan tempelan belakangan. Kalau contoh yang Anda kerjakan dengan tangan berbeda dari contoh di rumusan soal, berarti Anda salah membaca persoalannya. Jangan lanjutkan. Cari tahu sebabnya.

1.5.2 Fase 2 — Mengabstraksi

Lucuti ceritanya. Kota-kota, sang Cossack, dan cucian itu hanyalah hiasan; di bawahnya ada sebuah objek matematis. Tanyakan: sebenarnya apa yang diminta dari saya? Sifat apa yang harus dimiliki jawabannya? Besaran apa yang sedang diminimumkan atau dimaksimumkan, dan atas himpunan yang mana?

Di sinilah pemikiran yang sebenarnya terjadi, dan di sinilah sebuah penyelesaian beranjak dari "gelung atas segalanya" menjadi "satu baris hitungan". Di Subbab 1.7 sebuah persoalan yang terdengar seperti persoalan pedagang keliling mengerut menjadi ungkapan $2(x_{\max} - x_{\min})$. Di Subbab 1.10 sebuah barisan yang menggandakan dirinya sendiri mengerut menjadi menghitung bit 1 pada sebuah bilangan. Kedua pengerutan itu tidak terlihat sampai ceritanya dilucuti.

Satu kebiasaan yang berguna dalam Fase 2: cobalah membuktikan sebuah **batas bawah** (jawabannya tidak mungkin lebih kecil daripada L) lalu cobalah **membangun** sesuatu yang mencapai L. Kalau keduanya berhasil, jawabannya persis L dan Anda sudah selesai. Dua dari enam persoalan di bawah diselesaikan persis dengan cara ini.

1.5.3 Fase 3 — Merancang

Sekarang, dan baru sekarang, pilihlah satu paradigma dari **Tabel 1.4** lalu tuliskan algoritmanya. Tulis dalam langkah bernomor, dalam bahasa sehari-hari atau pseudokode. Sebutkan struktur datanya. Jangan menulis C++ dulu.

Rancangan yang ditulis dalam bahasa sehari-hari dapat diperiksa oleh teman, diperdebatkan, dan dibuang dengan murah. Rancangan yang ditulis dalam C++ memperoleh bobot perasaan — Anda sudah menanam modal di dalamnya — dan Anda akan membelanya jauh sesudah semestinya Anda tinggalkan.

1.5.4 Fase 4 — Menganalisis

Dua pertanyaan, keduanya wajib.

Apakah ia benar? Hasilkan sebuah bukti, dengan salah satu dari empat teknik di Subbab 1.3.1. Kalau algoritmanya sebuah gelung, nyatakan invariannya. Kalau ia *greedy*, buktikan pilihan rakusnya aman. Kalau ia rekursif, pakai induksi. "Ia berhasil pada contoh di rumusan soal" bukan jawaban.

Apakah ia cukup cepat? Hitung operasinya sebagai fungsi ukuran masukan, nyatakan dalam Big-O, masukkan masukan terbesar yang diizinkan ke dalam rumus itu, lalu bandingkan hasilnya dengan anggaran 10^8. Kalau tidak muat, kembalilah ke Fase 2 — bukan ke Fase 5. Anda tidak dapat mengoptimalkan diri keluar dari kelas asimtotik yang salah dengan menulis gelung yang lebih ketat.

Nyatakan pula kompleksitas **ruangnya**, dan periksa terhadap batas memori, yang lazimnya 128 atau 256 megabita. Satu `int` memakan 4 bita, jadi larik berisi 10^8 `int` memerlukan 400 megabita dan tidak akan muat.

1.5.5 Fase 5 — Mengimplementasikan

Tulis kodenya. Ia semestinya merupakan terjemahan langsung dan bersih dari Fase 3, dan semestinya pendek.

> **RENUNGKAN.** Kalau kode Anda tampak rumit, besar kemungkinan rancangan Anda belum selesai. Kerumitan dalam implementasi hampir selalu merupakan gejala pertanyaan yang belum terjawab di Fase 2. Kembalilah dan sederhanakan gagasannya; jangan mencoba merapikan kodenya.

Kebanyakan mahasiswa langsung melompat ke Fase 5. Tahanlah diri. Justru empat fase pertama itulah yang membuat Fase 5 menjadi sepele, dan Anda akan memperhatikan, pada enam persoalan berikut ini, bahwa program akhirnya hanya delapan sampai dua puluh baris. Itu bukan karena persoalannya mudah. Itu karena pemikirannya dikerjakan lebih dulu.

1.6 Persoalan 1 — Columns (E-Olymp 10281)

Paradigma: *greedy* / perhitungan langsung.

1.6.1 Fase 1 — Memahami

Gambar 1.3 adalah persoalannya persis seperti yang tampil di juri E-Olymp.

Columns (E-Olymp 10281)

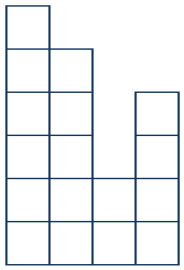
Batas waktu 1 dtk Batas memori 128 MB

Pernyataan

Diberikan n kolom kubus; kolom ke-i tingginya a_i. Tentukan banyak warna terkecil yang cukup untuk mewarnai semua kubus sehingga setiap deretan kubus mendatar dalam satu baris (tanpa celah) dan setiap kolom berisi warna yang berbeda-beda.

Masukan

Baris pertama berisi satu bilangan bulat n ($1 \leq n \leq 1000$), banyaknya kolom. Baris kedua berisi n bilangan bulat $a_1, a_2, \ldots, a_n$ ($1 \leq a_i \leq 1000$), tinggi setiap kolom.



Gambar 1.3 *Columns*, soal E-Olymp nomor 10281, dirumuskan ulang dari jurinya: rumusan soalnya, gambar empat kolom kubus, dan bentuk masukannya.

Dengan kata-kata: Anda diberi n kolom kubus yang berdiri berdampingan di atas tanah. Kolom ke-i setinggi a_i kubus. Anda harus mengecat setiap kubus dengan suatu warna, tunduk pada dua aturan, dan Anda harus memakai warna berbeda sesedikit mungkin.

Masukannya. Baris pertama memuat satu bilangan bulat n, banyaknya kolom, dengan $1 \leq n \leq 1000$. Baris kedua memuat n bilangan bulat $a_1, a_2, \ldots, a_n$, yaitu tingginya, dengan $1 \leq a_i \leq 1000$.

Keluarannya. Satu bilangan: banyaknya warna terkecil yang mencukupi.

Kedua aturannya.

1. Setiap **kolom** harus punya semua kubusnya berlainan warna.
2. Setiap **substring** harus punya semua kubusnya berlainan warna.

Apa itu substring? Inilah satu-satunya kosakata yang diciptakan persoalan ini, dan salah membacanya adalah cara paling lazim untuk kehilangan persoalan ini. Sebuah *substring* adalah deret mendatar berisi kubus-kubus yang duduk bersebelahan pada ketinggian yang sama **tanpa sela di antaranya**. Kalau sebuah kolom terlalu pendek untuk mencapai ketinggian itu, ia meninggalkan lubang, dan lubang itu memecah baris tersebut menjadi dua substring terpisah yang boleh mengulangi warna satu sama lain.

> **CONTOH KERJA TANGAN — tinggi** [6,5,2,4]**.** Mari kita gambar bentuknya baris demi baris, dari atas ke bawah. Tulis `[X]` untuk sebuah kubus dan kosong untuk udara hampa.

Baris 6: `[X]` — hanya kolom 1 yang setinggi ini, jadi satu kubus sendirian Baris 5: `[X][X]` — kolom 1 dan 2; satu substring sepanjang 2 Baris 4: `[X][X] [X]` — kolom 3 tingginya 2, jadi ada **sela**; dua substring, panjangnya 2 dan 1 Baris 3: `[X][X] [X]` — sela yang sama di kolom 3 Baris 2: `[X][X][X][X]` — setiap kolom mencapai ketinggian 2; satu substring sepanjang 4 Baris 1: `[X][X][X][X]` — setiap kolom mencapai ketinggian 1; satu substring sepanjang 4

Gambar tangan itu, yang hanya memakan waktu setengah menit dengan pensil, memuat seluruh penyelesaiannya. Pandanglah gambar itu dan dua kenyataan langsung melompat keluar. Kolom tertinggi punya 6 kubus bertumpuk tegak, dan aturan 1 mengatakan semuanya harus berlainan, jadi kita perlu sekurang-kurangnya 6 warna. Dan baris terbawah adalah substring utuh sepanjang keseluruhan 4 kolom, jadi aturan 2 mengatakan kita perlu sekurang-kurangnya 4 warna. Enam adalah tuntutan yang lebih besar.

Gambar 1.4 menunjukkan contoh uji dari jurinya sendiri: masukan `4` lalu `6 5 2 4`, dengan jawaban `6`, beserta satu pengecatan yang sah.

Columns (E-Olymp 10281)

Batas waktu 1 dtk Batas memori 128 MB

Keluaran

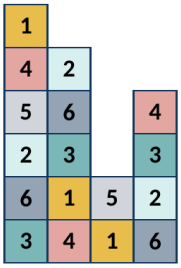
Cetak satu bilangan: banyak warna minimum yang diperlukan.

Contoh

Masukan #1	Jawaban #1
4 6 5 2 4	6

Catatan

Salah satu pewarnaan contoh. Baris ketiga dari bawah memuat warna 3 dua kali; hal itu sah karena ruang kosong di kolom 3 memisahkan kedua kubus tersebut.



Gambar 1.4 Contoh uji *Columns* dan satu pengecatan sah dengan enam warna. Perhatikan baris ketiga dari bawah: baris itu memuat warna 3 dua kali, dan itu sah karena ada ruang kosong yang memisahkan kedua kubusnya.

Perhatikan baik-baik baris ketiga dari bawah pada **Gambar 1.4**. Baris itu terbaca 2, 3, sela, 3. Warna 3 muncul dua kali di baris yang sama — dan ini **sah**, sebab sela yang ditinggalkan kolom ketiga yang pendek memecah baris itu menjadi dua substring yang saling bebas. Inilah jurinya sendiri yang menegaskan, lewat sebuah gambar, apa persisnya arti kata *substring*.

1.6.2 Fase 2 — Mengabstraksi

Lucuti kubus dan catnya. Kita sedang mencari satu bilangan C, yaitu banyaknya warna. Dua kendala mendorong C ke atas, dan kita harus mencari C terkecil yang memenuhi keduanya.

Langkah 1 — kendala tegak. Ambil kolom tertinggi; sebut tingginya $H = \max(a_1, a_2, \dots, a_n)$. Kolom itu memuat H kubus, semuanya dalam satu kolom, dan aturan 1 memaksa semuanya berlainan warna. Anda tidak dapat memberi H benda warna yang berlainan dengan warna yang kurang dari H macam. Karena itu

$$C \geq H = \max_i a_i.$$

Langkah 2 — kendala mendatar. Perhatikan baris 1, baris terbawah. Setiap kolom tingginya sekurang-kurangnya 1 — batasannya menjamin $a_i \geq 1$ — jadi setiap kolom dari n kolom itu punya kubus di baris 1. Tidak ada sela. Seluruh baris terbawah karena itu merupakan satu substring sepanjang n, dan aturan 2 memaksa keseluruhan n kubus itu berlainan warna. Karena itu

$$C \geq n.$$

Perhatikan bahwa kita hanya memerlukan baris *terbawah*. Baris-baris yang lebih atas hanya bisa lebih pendek atau terpecah, jadi mereka tidak pernah menuntut lebih banyak daripada n.

Langkah 3 — gabungkan. Kedua tuntutan harus terpenuhi sekaligus, jadi

$$C \geq \max\left(n,\ \max_i a_i\right). \tag{1.2}$$

Persamaan (1.2) adalah sebuah **batas bawah**: jawabannya tidak mungkin lebih kecil daripada itu. Untuk menuntaskannya, kita harus menunjukkan bahwa batas itu tercapai — bahwa $\max(n, \max_i a_i)$ warna memang cukup.

Langkah 4 — buktikan kecukupannya dengan konstruksi. Ambil $C = \max(n, \max_i a_i)$. Nomori kolomnya $j = 0,1,\ldots,n-1$ dari kiri dan barisnya $r = 0,1,2,\ldots$ dari bawah. Cat kubus di kolom j, baris r dengan warna

$$\text{warna}(j, r) = ((j + r) \bmod C) + 1. \tag{1.3}$$

Dalam Persamaan (1.3), $x \bmod C$ berarti sisa pembagian x oleh C; nilainya selalu salah satu dari $0,1,\ldots,C-1$. Menambahkan 1 mengubahnya menjadi nama warna dari 1 sampai C.

Mengapa setiap kolom aman. Tetapkan sebuah kolom j. Ketika r berjalan menyusuri $0,1,\ldots,a_j - 1$, besaran $j + r$ mengambil a_j nilai berurutan. Bilangan bulat berurutan punya sisa yang berlainan modulo C selama banyaknya paling banyak C — dan $a_j \leq \max_i a_i \leq C$. Jadi semua warna dalam kolom itu berlainan.

Mengapa setiap substring aman. Tetapkan sebuah baris r dan sebuah deret kolom tanpa sela $j, j + 1, \ldots, k$. Ketika j berjalan menyeberangi deret itu, $j + r$ kembali mengambil nilai-nilai berurutan, paling banyak n buah, dan $n \leq C$. Jadi semua warna dalam substring itu berlainan.

Kedua aturan terpenuhi, dengan memakai persis C warna. Batas bawahnya tercapai, jadi jawabannya persis C.

> **WAWASAN KUNCI:** jawaban $= \max(n,\ \max(a_1, a_2, \ldots, a_n))$.

Kalau Anda meragukan konstruksi itu — dan Anda memang semestinya meragukan konstruksi sampai Anda memeriksanya — **Kode 1.7** menerapkannya pada tinggi-tinggi contoh uji lalu menguji *setiap* kolom dan *setiap* deret tanpa sela secara *brute force*.

Kode 1.7 Pemeriksa *brute force* untuk pewarnaan yang dibangun dalam pembuktian *Columns*.

```
#include <cstdio>
#include <algorithm>
using namespace std;

int a[] = {6, 5, 2, 4};                 // tinggi kolom dari contoh uji
const int n = 4;

int color(int j, int r, int C) {       // kolom j (mulai 0), baris r (mulai 0)
    return ((j + r) % C) + 1;           // konstruksi dari pembuktian
}

int main() {
    int C = n;
    for (int j = 0; j < n; j++) C = max(C, a[j]);
    printf("C = %d\n", C);

    bool bad = false;
    for (int j = 0; j < n; j++)         // setiap kolom: semua warna berbeda?
        for (int r = 0; r < a[j]; r++)
            for (int s = r + 1; s < a[j]; s++)
                if (color(j, r, C) == color(j, s, C)) bad = true;

    for (int r = 0; r < C; r++)         // setiap deret maksimal tanpa sela di baris r
        for (int j = 0; j < n; j++)
            for (int k = j + 1; k < n; k++) {
                bool run = true;        // apakah j..k satu deret utuh pada tinggi r?
                for (int t = j; t <= k; t++) if (a[t] <= r) run = false;
                if (run && color(j, r, C) == color(k, r, C)) bad = true;
            }

    printf(bad ? "FAILED: a clash exists\n" : "VERIFIED: no clash anywhere\n");
    return 0;
}
```

Keluarannya yang sesungguhnya:

```
C = 6
VERIFIED: no clash anywhere
```

1.6.3 Fase 3 — Merancang

1. Baca n.
2. Tetapkan $ans \leftarrow n$.
3. Untuk setiap dari n tinggi a_i yang dibaca dari masukan: tetapkan $ans \leftarrow \max(ans, a_i)$.
4. Cetak ans.

Tidak diperlukan larik. Tingginya dilahap satu per satu lalu dibuang; hanya maksimum berjalan yang bertahan. Inilah paradigma inkremental dari Subbab 1.4.1 dalam wujudnya yang paling murni.

1.6.4 Fase 4 — Menganalisis

Kebenaran. Sudah dibuktikan di atas: $\max(n, \max_i a_i)$ sekaligus merupakan batas bawah (Langkah 1–3) dan tercapai (Langkah 4). Invarian gelung untuk langkah 3 adalah *sesudah mengolah i tinggi, $ans = \max(n, a_1, \ldots, a_i)$*, yang terpenuhi di awal dengan $i = 0$ karena $ans = n$, terpelihara oleh `max`, dan pada $i = n$ menyerahkan jawabannya.

Waktu. Satu kali pindai atas n bilangan, kerja konstan per bilangan: $O(n)$. Dengan $n \leq 1000$ itu seribu operasi, melawan anggaran 10^8. Kita memakai seperseratus ribu dari batas waktunya.

Ruang. Tiga bilangan bulat: $O(1)$.

1.6.5 Fase 5 — Mengimplementasikan

Kode 1.8 Penyelesaian lengkap untuk *Columns*, E-Olymp 10281.

```
#include <cstdio>
#include <algorithm>
using namespace std;

int n, height, ans;

int main() {
    scanf("%d", &n);
    ans = n;                        // baris terbawah saja sudah memaksa n warna
    for (int i = 0; i < n; i++) {
        scanf("%d", &height);
        ans = max(height, ans);     // kolom setinggi ini memaksa sebanyak height warna
    }
    printf("%d\n", ans);
    return 0;
}
```

Pada masukan contoh dari jurinya

```
4
6 5 2 4
```

Kode 1.8 sungguh-sungguh mencetak:

```
6
```

sesuai dengan jawaban yang diharapkan pada **Gambar 1.4**. Sebagai pemeriksaan kedua, pada masukan `5` diikuti `1 1 1 1 1` — lima kolom, masing-masing satu kubus — program itu mencetak `5`, dan itu betul: kolom tertinggi hanya menuntut 1 warna, tetapi baris terbawah yang utuh sepanjang lima kolom menuntut 5.

1.7 Persoalan 2 — Journey (E-Olymp 10280)

Paradigma: *greedy* / pengamatan matematis.

1.7.1 Fase 1 — Memahami

Journey (E-Olymp 10280)

Batas waktu 1 dtk Batas memori 128 MB

Pernyataan

Cossack Vus hendak bepergian. Di Potokolandia terdapat n kota pada satu garis lurus, bernomor 1 sampai n; kota i berkoordinat x_i, dan jarak antara kota i dan j adalah $|x_i - x_j|$. Vus harus mengunjungi setiap kota sekurang-kurangnya sekali dan mengakhiri perjalanan di kota tempat ia berangkat; kota awal dan rutenya bebas ia pilih. Tentukan panjang rute terpendek semacam itu.

Masukan

Baris pertama berisi satu bilangan bulat n $(1 \leq n \leq 100)$. Baris kedua berisi n bilangan bulat $x_1, x_2, \ldots, x_n$ $(1 \leq x_i \leq 1000)$.

Gambar 1.5 *Journey*, soal E-Olymp nomor 10280. Cossack Vus harus menyinggahi setiap kota di Potokolandia lalu pulang kembali.

Sebagaimana dinyatakan **Gambar 1.5**: di Potokolandia ada n kota yang terletak pada satu **garis lurus**, bernomor 1 sampai n. Kota i berada di koordinat x_i. Jarak antara kota i dan j adalah

$|x_i - x_j|$ — sepasang garis tegak itu berarti **nilai mutlak**, yaitu selisihnya dengan tanda minusnya dibuang, sehingga jarak tidak pernah negatif.

Cossack Vus ingin menyinggahi setiap kota sekurang-kurangnya sekali lalu berakhir di kota tempat ia memulai. Ia boleh memilih *baik* kota awalnya *maupun* rutenya. Kita harus melaporkan jarak tempuh total yang paling kecil.

Masukannya. Baris pertama: satu bilangan bulat n dengan $1 \leq n \leq 100$. Baris kedua: n bilangan bulat $x_1, \ldots, x_n$ dengan $1 \leq x_i \leq 1000$.

Keluarannya. Satu bilangan bulat, panjang rute minimumnya, sebagaimana ditunjukkan **Gambar 1.6**.

Journey (E-Olymp 10280)

Batas waktu 1 dtk Batas memori 128 MB

Keluaran

Cetak satu bilangan bulat: panjang minimum rute Cossack Vus.

Contoh 1

Masukan #1	Jawaban #1
2 1 4	6

Contoh 2

Masukan #2	Jawaban #2
3 1 3 1	4

Gambar 1.6 Dua contoh uji *Journey*. Dua kota di 1 dan 4 memberi 6; tiga kota di 1,3,1 memberi 4.

CONTOH KERJA TANGAN — contoh uji 1. Dua kota, di koordinat 1 dan 4. Mulailah dari kota 1 (koordinat 1). Berjalanlah ke kanan sampai koordinat 4: itu $|4 - 1| = 3$ satuan. Sekarang singgahi kota 2 — sudah. Berjalanlah balik ke kiri sampai koordinat 1: 3 satuan lagi. Totalnya $3 + 3 = 6$. Jawaban yang diharapkan adalah 6.

CONTOH KERJA TANGAN — contoh uji 2. Tiga kota, di koordinat 1, 3, dan 1. Perhatikan bahwa dua kota berbagi koordinat 1; tidak ada yang melarangnya. Mulailah dari koordinat 1. Kota 3 juga ada di koordinat 1, jadi ia tersinggahi secara cuma-cuma. Berjalanlah ke kanan sampai koordinat 3: 2 satuan. Berjalanlah balik: 2 satuan. Totalnya 4. Jawaban yang diharapkan adalah 4.

1.7.2 Fase 2 — Mengabstraksi

Langkah 1 — kembalikan ke geometri satu dimensi. Lupakan nama kota-kotanya, lupakan Cossack Vus. Kita punya n titik pada sebuah garis bilangan. Kita perlu sebuah lintasan tertutup — lintasan yang berakhir di tempat ia bermula — yang melewati semuanya, dengan panjang total sekecil mungkin.

Ini *terdengar* seperti persoalan pedagang keliling (*traveling salesman problem*), yang termasuk persoalan tersulit dalam ilmu komputer. Pada sebuah garis, ia bukan itu. Geometri menyelamatkan kita.

Langkah 2 — hanya ada dua gerakan. Pada sebuah garis Anda dapat bergerak ke kiri atau bergerak ke kanan. Hanya itu. Jadi rute apa pun adalah sederet gerakan ke kiri dan ke kanan, dan jarak totalnya adalah jumlah panjang gerakan-gerakan itu.

Langkah 3 — cari batas bawahnya. Misalkan $x_{\min}$ koordinat terkecil dan $x_{\max}$ koordinat terbesar. Definisikan **rentang** $R = x_{\max} - x_{\min}$.

Rute itu pada suatu saat pasti berdiri di $x_{\min}$ (ada kota di sana, dan kota itu harus disinggahi) dan pada suatu saat berdiri di $x_{\max}$. Karena rute itu *tertutup* — berakhir di tempat ia bermula — rute itu sesungguhnya sebuah putaran, dan panjang sebuah putaran tidak berubah dari titik mana pun kita mulai membacanya. Maka marilah kita mulai membacanya pada saat ia berdiri di $x_{\min}$. Dibaca maju dari situ, putaran itu terbelah menjadi dua potong:

- **Pergi.** Lintasan itu akhirnya tiba di $x_{\max}$. Perjalanan dari $x_{\min}$ ke $x_{\max}$ berharga sekurang-kurangnya R satuan, sebab pada sebuah garis tidak ada lintasan antara dua titik yang lebih pendek daripada jarak keduanya.
- **Pulang.** Lintasan itu lalu berlanjut sampai ia sudah mengitari seluruh putaran dan berdiri lagi di $x_{\min}$. Perjalanan dari $x_{\max}$ kembali ke $x_{\min}$ sekali lagi berharga sekurang-kurangnya R.

Kedua potong itu tidak bertumpang tindih, jadi setiap rute tertutup yang sah berharga sekurang-kurangnya $R + R = 2R$.

Dalam bentuk **bukti dengan kontradiksi**: andaikan ada rute tertutup yang berharga *kurang* dari $2R$. Potonglah rute itu menjadi potongan "pergi" dan "pulang" seperti tadi. Jumlah panjang keduanya kurang dari $2R$, sehingga sekurang-kurangnya satu potong lebih pendek daripada R. Padahal setiap potong menghubungkan $x_{\min}$ dengan $x_{\max}$, dua titik yang berjarak tepat R — dan tidak ada lintasan di antara keduanya yang lebih pendek daripada R. Itu mustahil, jadi rute semacam itu tidak ada.

Langkah 4 — capai batas itu. Mulailah dari $s = x_{\min}$. Berjalanlah ke kanan terus sampai $x_{\max}$, melewati setiap kota di sepanjang jalan, sebab setiap x_i terletak di antara kedua ujung itu. Itu berharga R. Berjalanlah balik ke kiri sampai $x_{\min}$. Itu berharga R lagi. Totalnya $2R$, dan setiap kota sudah disinggahi. Batas itu tercapai.

Langkah 5 — titik awalnya tidak berpengaruh. Andaikan Anda mulai dari sembarang s dengan $x_{\min} \le s \le x_{\max}$. Berjalan ke kiri ke $x_{\min}$: ongkosnya $s - x_{\min}$. Berjalan ke kanan terus sampai $x_{\max}$: ongkosnya $x_{\max} - x_{\min}$. Berjalan balik ke kiri ke s: ongkosnya $x_{\max} - s$. Kalau dijumlahkan:

$$(s - x_{\min}) + (x_{\max} - x_{\min}) + (x_{\max} - s).$$

Kedua suku yang memuat s adalah $+s$ dan $-s$; keduanya saling meniadakan. Yang tersisa adalah

$$-x_{\min} + x_{\max} - x_{\min} + x_{\max} = 2(x_{\max} - x_{\min}) = 2R. \tag{1.4}$$

Titik awalnya lenyap secara aljabar. Setiap kota awal memberi optimum yang sama, dan itulah sebabnya persoalan ini boleh membiarkan Anda memilihnya.

> **WAWASAN KUNCI:** jawaban $= 2 \times (\max_i x_i - \min_i x_i)$. Banyaknya kota tidak relevan; hanya kedua ujungnya yang berpengaruh.

Periksa Persamaan (1.4) terhadap contoh ujinya. Contoh uji 1: $x_{\max} = 4$, $x_{\min} = 1$, jadi $2(4 - 1) = 6$. Contoh uji 2: $x_{\max} = 3$, $x_{\min} = 1$, jadi $2(3 - 1) = 4$. Keduanya cocok dengan **Gambar 1.6**.

1.7.3 Fase 3 — Merancang

1. Baca n.
2. Berikan nilai awal $d_{\min}$ yang lebih besar daripada koordinat sah mana pun dan $d_{\max}$ yang lebih kecil daripada koordinat sah mana pun.
3. Untuk setiap koordinat yang dibaca: perbarui $d_{\min}$ dan $d_{\max}$.
4. Cetak $2 \times (d_{\max} - d_{\min})$.

Batasannya menyatakan $1 \leq x_i \leq 1000$, jadi memulai dari $d_{\min} = 2000$ dan $d_{\max} = 0$ aman: koordinat pertama yang dibaca akan menggantikan keduanya.

1.7.4 Fase 4 — Menganalisis

Kebenaran. Sudah dibuktikan di atas dengan batas bawah beserta konstruksi yang menyamainya. Invarian gelungnya adalah *sesudah membaca i koordinat, $d_{\min}$ dan $d_{\max}$ adalah yang terkecil dan terbesar di antaranya*.

Waktu. $O(n)$; dengan $n \leq 100$, seratus operasi. **Ruang.** $O(1)$ — koordinatnya tidak pernah disimpan.

1.7.5 Fase 5 — Mengimplementasikan

Kode 1.9 Penyelesaian lengkap untuk *Journey*, E-Olymp 10280.

```
#include <cstdio>
using namespace std;

int main() {
    int n, dist;
    scanf("%d", &n);
    int dmin = 2000, dmax = 0;     // 2000 di atas semua koordinat sah, 0 di bawah semuanya
    for (int i = 0; i < n; i++) {
        scanf("%d", &dist);
        dmin = dmin < dist ? dmin : dist;
        dmax = dmax < dist ? dist : dmax;
    }
    printf("%d\n", 2 * (dmax - dmin));
    return 0;
}
```

Ungkapan `dmin < dist ? dmin : dist` adalah **operator kondisional**: ia menilai ujinya yang ada sebelum `?`, lalu menghasilkan nilai yang ada sebelum `:` kalau ujinya benar, dan nilai yang ada sesudahnya kalau tidak. Jadi baris ini menyimpan mana pun yang lebih kecil di antara `dmin` dan `dist`. Baris berikutnya, dengan kedua cabangnya ditukar, menyimpan mana pun yang lebih besar.

Pada masukan contoh 1 (`2` lalu `1 4`) **Kode 1.9** sungguh-sungguh mencetak `6`; pada masukan contoh 2 (`3` lalu `1 3 1`) ia sungguh-sungguh mencetak `4`. Keduanya cocok dengan **Gambar 1.6**.

1.8 Persoalan 3 — Unique Numbers (SPOJ DCEPC901)

Paradigma: teori bilangan (faktor persekutuan terbesar).

1.8.1 Fase 1 — Memahami

DCEPC901 — Unique Numbers (SPOJ)

Batas waktu 0.204 dtk Batas kode sumber 50 000 B Batas memori 1536 MB

Pernyataan

Seorang guru menuliskan semua bilangan dari 0 sampai $n-1$, lalu melakukan k operasi atasnya, dari dua jenis: `0 x` mengganti setiap bilangan i dengan $(i+x) \bmod n$; `1 x` mengganti setiap bilangan i dengan $(i \times x) \bmod n$. Setelah setiap operasi, laporkan berapa banyak bilangan berbeda yang tersisa.

Catatan: setiap operasi dikenakan pada bilangan hasil operasi sebelumnya, bukan pada bilangan semula.

Masukan

Baris pertama berisi n, baris kedua k, banyaknya operasi. Masing-masing dari k baris berikutnya berisi dua bilangan bulat p dan x.

Keluaran

Keluarkan k baris, masing-masing berisi banyaknya bilangan berbeda yang tersisa.

Gambar 1.7 *Unique Numbers*, soal SPOJ DCEPC901, dengan dua jenis operasinya dan catatan penting bahwa operasinya beruntun saling menyusul.

Gambar 1.7 memasang latarnya. Kita mulai dengan bilangan-bilangan bulat

$$0,1,2,\ldots,n-1,$$

semuanya n buah, semuanya berlainan. Lalu k operasi diterapkan, satu demi satu. Setiap operasi diberikan sebagai dua bilangan bulat p dan x:

- Kalau $p = 0$: **ganti setiap bilangan i dengan $(i + x) \bmod n$.**
- Kalau $p = 1$: **ganti setiap bilangan i dengan $(i \times x) \bmod n$.**

Catatan bercetak tebal pada halaman jurinya penting: *setiap operasi dikerjakan atas bilangan-bilangan yang kita peroleh sesudah operasi sebelumnya, bukan atas bilangan aslinya*. Operasinya beruntun saling menyusul.

Sesudah setiap operasi kita harus melaporkan berapa banyak nilai **berbeda** yang tersisa. Bilangan-bilangannya bisa bertabrakan, dan ketika dua di antaranya menjadi sama, cacahnya turun.

Batasan

$1 \le n \le 10^8$
$1 \le k \le 20$
$0 \le p \le 1$
$0 \le x \le 10^8$

Contoh

Masukan	Jawaban
30 3 1 8 0 3 1 12	15 15 5

Gambar 1.8 Batasan dan contoh uji *Unique Numbers*: n sampai 10^8, paling banyak 20 operasi.

Gambar 1.8 memberikan batasannya: $1 \le n \le 10^8$, $1 \le k \le 20$, $0 \le p \le 1$, $0 \le x \le 10^8$.

Berhentilah sejenak dan bacalah $n \le 10^8$ sambil melihat **Tabel 1.3**. Kalau kita sungguh-sungguh membangun daftar berisi n bilangan, kita akan memerlukan 10^8 bilangan bulat — empat ratus megabita, melampaui batas memori — dan masing-masing dari 20 operasi akan menyentuh semuanya, memberi 2×10^9 operasi, dua puluh kali lipat dari anggaran waktunya. Batasannya sedang memberi tahu kita bahwa penyelesaian yang dimaksudkan sama sekali tidak pernah mewujudkan daftarnya.

Contoh uji pada **Gambar 1.8** adalah $n = 30$ dengan $k = 3$ operasi `1 8`, `0 3`, `1 12`, dan keluaran yang diharapkan adalah 15, 15, 5.

1.8.2 Fase 2 — Mengabstraksi

Langkah 1 — penjumlahan tidak pernah mengubah cacahnya. Tinjau pemetaan $g(i) = (i + x) \bmod n$. Ambil dua nilai berbeda $a \ne b$, keduanya dalam $\{0, \dots, n-1\}$. Mungkinkah keduanya bertabrakan? Kalau $(a + x) \equiv (b + x) \pmod{n}$, kurangkan x dari kedua ruas: $a \equiv b \pmod{n}$. Karena a dan b sama-sama terletak dalam $\{0, \dots, n-1\}$, kongruensi modulo n memaksa $a = b$, bertentangan dengan $a \ne b$. Jadi masukan yang berlainan memberi keluaran yang berlainan: g bersifat **injektif** (satu-satu). Pemetaan satu-satu dari himpunan berhingga ke dirinya sendiri juga bersifat pada, jadi g adalah sebuah **bijeksi** — ia sekadar mengocok bilangan-bilangannya.

Karena itu: **operasi penjumlahan tidak pernah mengubah banyaknya nilai berbeda.** Inilah sebabnya keluaran kedua pada contoh uji mengulang keluaran pertama.

Langkah 2 — perkalian menciutkan semestanya. Sekarang tinjau $f(i) = (i \times x) \bmod n$ untuk $i = 0,1, \dots, n-1$. Nilai apa saja yang bisa keluar?

Misalkan $g = \gcd(n, x)$. Kita mengeklaim bahwa himpunan keluarannya persis kelipatan g di dalam $\{0, \dots, n-1\}$, yaitu

$$\{0,\ g,\ 2g,\ \dots,\ n-g\},$$

yang punya n/g anggota.

Setiap keluaran adalah kelipatan g. Bilangan ix adalah kelipatan x, dan x adalah kelipatan g, jadi ix adalah kelipatan g. Mereduksi modulo n berarti mengurangkan suatu kelipatan n, dan n juga kelipatan g. Kelipatan g dikurangi kelipatan g adalah kelipatan g. Selesai.

Setiap kelipatan g tercapai. Identitas Bézout mengatakan ada bilangan bulat u dan v dengan $ux + vn = g$. Mereduksi modulo n membunuh suku vn, jadi $ux \equiv g \pmod{n}$: mengambil $i = u \bmod n$ menghasilkan keluaran g. Lalu $i = (2u) \bmod n$ menghasilkan $2g$, dan seterusnya. Setiap kelipatan g muncul.

Jadi perkalian dengan x meruntuhkan semestanya dari n nilai menjadi $n/\gcd(n, x)$ nilai, berjarak sama g satu sama lain. Dalam lambang, banyaknya keluaran yang berbeda adalah

$$|\{\,(i \times x) \bmod n \,:\; 0 \le i \le n-1\,\}| \;=\; \frac{n}{\gcd(n,x)}, \tag{1.5}$$

dengan garis tegak yang mengapit sebuah himpunan berarti "banyaknya anggota himpunan itu".

Gambar 1.9 mengilustrasikan keruntuhan ini, dihitung untuk $n = 15$ dan empat pengali yang berbeda.

op	x	$(i \times x) \bmod 15$														
i		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	0	3	6	9	12	0	3	6	9	12	0	3	6	9	12
i		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	5	0	5	10	0	5	10	0	5	10	0	5	10	0	5	10
i		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	0	2	4	6	8	10	12	14	1	3	5	7	9	11	13
i		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	4	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11

Gambar 1.9 Perkalian modulo 15. Setiap pasangan baris menunjukkan masukan 0..14 di atas dan keluaran $(i \times x) \bmod 15$ di bawah, untuk $x = 3,5,2,4$ berturut-turut.

Bacalah **Gambar 1.9** satu blok demi satu blok, sambil mencocokkan setiap blok dengan Persamaan (1.5).

- $x = 3$: keluarannya 0,3,6,9,12,0,3,6,9,12, … — hanya lima nilai berbeda, berulang tiga kali. Dan $\gcd(15,3) = 3$, jadi $15/3 = 5$. Betul.
- $x = 5$: keluarannya 0,5,10,0,5,10, … — tiga nilai berbeda. $\gcd(15,5) = 5$, jadi $15/5 = 3$. Betul.
- $x = 2$: keluarannya 0,2,4,6,8,10,12,14,1,3,5,7,9,11,13 — kelima belas nilainya, sekadar terkocok. $\gcd(15,2) = 1$, jadi $15/1 = 15$. Betul.
- $x = 4$: keluarannya 0,4,8,12,1,5,9,13,2,6,10,14,3,7,11 — lagi-lagi kelima belasnya. $\gcd(15,4) = 1$. Betul.

Empat penegasan yang saling bebas atas rumus itu, digambar dengan tangan sebelum satu baris kode pun ditulis. Inilah Fase 1 dan Fase 2 bekerja bersama persis sebagaimana mestinya.

Langkah 3 — rangkaikan operasinya. Sesudah sebuah perkalian, nilai yang bertahan adalah $0, g, 2g, \ldots$ — sebuah himpunan berukuran $n' = n/g$ yang, sesudah setiap anggotanya dibagi g, berperilaku persis seperti himpunan $\{0,1,\ldots,n'-1\}$. Jadi kita boleh sekadar *mengganti n dengan n'* lalu melanjutkan. Penjumlahan membiarkan n apa adanya. Itulah seluruh algoritmanya: kita tidak pernah melacak bilangan-bilangannya, hanya berapa banyak jumlahnya.

> **WAWASAN KUNCI:** Pelihara n sebagai peubah berjalan. Pada perkalian dengan x, tetapkan $n \leftarrow n/\gcd(n,x)$. Pada penjumlahan, tidak usah apa-apa. Cetak n sesudah setiap operasi.

CONTOH KERJA TANGAN — contoh uji dari jurinya. Mulai dengan $n = 30$.

Operasi 1 adalah `1 8`: sebuah perkalian dengan 8. gcd(30,8): 30 mod 8 = 6, jadi gcd(8,6); 8 mod 6 = 2, jadi gcd(6,2); 6 mod 2 = 0, jadi jawabannya 2. $n \leftarrow 30/2 = 15$. **Cetak 15.**

Operasi 2 adalah `0 3`: sebuah penjumlahan. Penjumlahan adalah bijeksi. n tetap 15. **Cetak 15.**

Operasi 3 adalah `1 12`: sebuah perkalian dengan 12. gcd(15,12): 15 mod 12 = 3, jadi gcd(12,3); 12 mod 3 = 0, jadi jawabannya 3. $n \leftarrow 15/3 = 5$. **Cetak 5.**

Ketiga bilangan yang tercetak adalah 15,15,5 — persis keluaran yang diharapkan pada **Gambar 1.8**.

1.8.3 Fase 3 — Merancang

1. Baca n dan k.
2. Ulangi k kali: baca p dan x; kalau p ganjil (artinya $p = 1$) tetapkan $n \leftarrow n/\gcd(n, x)$; cetak n.

1.8.4 Fase 4 — Menganalisis

Kebenaran. Sudah ditegakkan dalam Langkah 1–3: penjumlahan adalah bijeksi sehingga cacahnya tidak berubah; perkalian dengan x memetakan semestanya pada kelipatan $\gcd(n, x)$, yang banyaknya $n/\gcd(n, x)$; dan himpunan yang bertahan isomorfik terurut dengan sebuah $\{0, \ldots, n' - 1\}$ yang segar, sehingga penalarannya berulang.

Waktu. Satu gcd untuk satu operasi. Algoritma Euclid memakan $O(\log n)$ langkah, jadi seluruh programnya $O(k \log n)$. Dengan $k \leq 20$ dan $n \leq 10^8$ itu paling banyak beberapa ratus operasi — tidak ada apa-apanya.

Ruang. $O(1)$. Empat bilangan bulat. Bandingkan ini dengan empat ratus megabita yang akan dibutuhkan pendekatan naifnya.

1.8.5 Fase 5 — Mengimplementasikan

Kode 1.10 Penyelesaian lengkap untuk *Unique Numbers*, SPOJ DCEPC901.

```
#include <cstdio>
#include <algorithm>
using namespace std;

int main() {
    int n, k, p, x;
    scanf("%d %d", &n, &k);
    while (k--) {
        scanf("%d %d", &p, &x);
        if (p & 1) n /= __gcd(n, x);   // hanya perkalian yang menciutkan semesta
        printf("%d\n", n);
    }
    return 0;
}
```

Tiga hal kecil pada **Kode 1.10** pantas dibicarakan.

`__gcd` adalah fungsi yang disediakan pustaka baku GNU C++ dalam `<algorithm>`; ia mewujudkan algoritma Euclid dari Subbab 1.4.5. Pada C++17 dan sesudahnya Anda boleh sama saja menulis `std::gcd` dari `<numeric>`.

`p & 1` adalah **dan-bit** (*bitwise and*) antara p dan 1, yang bernilai 1 ketika p ganjil dan 0 ketika p genap. Karena p hanya pernah bernilai 0 atau 1, ini cara ringkas untuk menulis `p == 1`.

`while (k--)` membaca nilai `k` yang sekarang sebagai syarat gelung *lalu* menguranginya, sehingga badan gelungnya berjalan persis k kali dan `k` berakhir di -1.

Diberi masukan contoh dari jurinya

```
30
3
1 8
0 3
1 12
```

Kode 1.10 sungguh-sungguh mencetak

```
15
15
5
```

yang persis keluaran yang diharapkan. Perhatikan bahwa program itu membaca `n` dan `k` dengan satu `scanf("%d %d", ...)` walaupun rumusan soalnya meletakkan keduanya pada baris terpisah; `scanf` melompati spasi apa pun, termasuk pindah baris, jadi ini aman.

> **JEBAKAN YANG PERLU DIKETAHUI.** Batasannya mengizinkan $x = 0$. Mengalikan setiap bilangan dengan nol membuat semuanya menjadi nol, jadi mestinya persis satu nilai berbeda yang tersisa. Apakah rumusnya selamat? Menurut kelaziman $\gcd(n, 0) = n$, jadi $n \leftarrow n/n = 1$. Ia memang selamat. Menjalankan **Kode 1.10** pada masukan `30`, `2`, `1 0`, `0 5` sungguh-sungguh mencetak `1` lalu `1`, dan itu betul — dan begitu segalanya sudah runtuh menjadi satu nilai, tidak ada operasi berikutnya yang dapat menaikkan cacahnya lagi.

1.9 Persoalan 4 — Minimum and Maximum Numbers (SPOJ NEWLANG)

Paradigma: konstruksi *greedy*.

1.9.1 Fase 1 — Memahami

NEWLANG — Minimum and maximum numbers (SPOJ)

Batas waktu 0.1 dtk Batas kode sumber 50 000 B Batas memori 1536 MB

Pernyataan

Diberikan N dan K, tentukan bilangan bulat positif terkecil dan terbesar yang masing-masing tepat N digit dan memuat tepat K digit berbeda (0–9). Nol di depan tidak diperbolehkan.

Masukan

Baris pertama berisi T, banyaknya kasus uji (sekitar 200). Masing-masing dari T baris berikutnya berisi `N K`, dengan $1 \le N \le 18$ dan $1 \le K \le 10$. Masukan menjamin jawaban selalu ada.

Keluaran

Untuk setiap kasus uji, keluarkan kedua bilangan itu, dipisahkan spasi, pada baris baru.

Contoh

Masukan	Jawaban

`2`
`1 1`
`3 3`
	`1 9`
	`102 987`

Gambar 1.10 *Minimum and maximum numbers*, soal SPOJ NEWLANG, dengan batasan dan contoh ujinya.

Dari **Gambar 1.10**: diberi N dan K, carilah bilangan bulat positif terkecil dan terbesar yang punya **tepat N angka** dan memakai **tepat K angka berbeda** dari $\{0,1,\ldots,9\}$. Angka nol di depan tidak diperbolehkan.

Masukannya. Baris pertama: T, banyaknya kasus uji (sekitar 200). Masing-masing dari T baris berikutnya memuat N dan K, dengan $1 \le N \le 18$ dan $1 \le K \le 10$. Masukannya dijamin sedemikian rupa sehingga jawabannya selalu ada.

Keluarannya. Untuk setiap kasus uji, kedua bilangan itu dipisahkan sebuah spasi, pada barisnya sendiri.

Contoh ujinya. $T = 2$; kasusnya `1 1` dan `3 3`; keluaran yang diharapkan adalah `1 9` dan `102 987`.

Mari kita pastikan kita memahami "tepat K angka berbeda". Bilangan 102 ditulis dengan tiga karakter 1, 0, 2; himpunan angka yang dipakai adalah $\{0,1,2\}$, yang punya tiga anggota, jadi 102 punya tepat 3 angka berbeda. Bilangan 111 punya satu angka berbeda. Bilangan 121 punya dua angka berbeda, yaitu $\{1,2\}$, walaupun ia punya tiga posisi angka. Keberbedaan itu soal *himpunan* angka yang dipakai, bukan soal banyaknya posisi.

Dua pemeriksaan lagi atas contoh ujinya. Apakah 102 benar-benar bilangan tiga angka terkecil dengan tiga angka berbeda? Calon-calon di bawahnya adalah 100 (angkanya $\{0,1\}$: hanya dua), 101 (angkanya $\{0,1\}$: dua). Jadi ya, 102 yang pertama. Apakah 987 benar-benar yang terbesar? Bilangan tiga angka di atasnya hanyalah 988 sampai 999. Bilangan 988 dan 989 hanya memakai angka $\{8,9\}$, dan setiap bilangan dari 990 sampai 999 diawali dua angka 9, sehingga paling banyak memiliki dua angka berbeda. Tidak satu pun memenuhi syarat, jadi ya, 987 yang terbesar.

Mengapa $N \le 18$? Sebab bilangan 18 angka muat dalam bilangan bulat 64 bit — tetapi perhatikan bahwa kita sama sekali tidak pernah memerlukan hitungan atas bilangan-bilangan ini. Kita mencetaknya angka demi angka sebagai karakter, jadi batas besarnya tidak relevan bagi metode kita. Itu sepotong kecil keberuntungan yang pantas disadari dalam Fase 1.

1.9.2 Fase 2 — Mengabstraksi

Sebuah bilangan N angka dibandingkan dengan bilangan N angka lainnya sebagaimana kata dibandingkan dalam kamus: lihat posisi paling kiri tempat keduanya berbeda, dan mana yang angkanya lebih kecil di situ, itulah bilangan yang lebih kecil. Sebabnya, angka paling kiri membawa bobot 10^{N-1}, yang lebih besar daripada semua yang ada di kanannya digabung jadi satu — bilangan terbesar yang dapat dibentuk dari $N-1$ angka sisanya adalah $\underbrace{99\cdots9}_{N-1} = 10^{N-1} - 1$, yang masih kurang dari satu satuan tempat terdepan.

> **MENGAPA *GREEDY* AMAN DI SINI.** Karena posisi 1 mengalahkan posisi 2 sampai N digabung sekaligus, memilih angka sah terkecil untuk posisi 1 tidak akan pernah menjadi kekeliruan — tidak ada kecerdikan apa pun di belakangnya yang dapat mengimbangi. Penalaran yang sama kemudian berlaku untuk posisi 2 di antara posisi 2.. N, dan seterusnya menyusur bilangan itu ke bawah. Inilah sifat pilihan *greedy*, dan inilah yang mengizinkan segala sesuatu di bawah ini.

Langkah 1 — kasus khusus $K = 1$. Dengan hanya satu angka berbeda yang boleh, bilangannya adalah angka itu diulang N kali. Angkanya tidak boleh 0, sebab angka pertamanya kemudian menjadi nol di depan. Jadi yang terkecil adalah $\underbrace{11\cdots1}_{N}$ dan yang terbesar adalah $\underbrace{99\cdots9}_{N}$. Untuk $N = 1$ itu memberi 1 dan 9, cocok dengan contoh ujinya.

Langkah 2 — nilai minimum ketika $K \ge 2$. Kerjakan dari kiri ke kanan, selalu memilih angka terkecil yang masih sanggup kita pakai.

Posisi 1. Ia tidak boleh 0. Angka bukan-nol terkecil adalah 1. Ambil **1**. Angka yang sudah terpakai: $\{1\}$, yaitu 1 dari K jatah kita.

Posisi 2 dan seterusnya. Angka terkecil dari semuanya adalah 0, dan 0 masih baru, jadi ia bebas dipakai. Sesudah memakainya kita punya $\{0,1\}$, dua angka berbeda, dan kita masih berutang $K - 2$ lagi. Setiap utang memerlukan posisinya sendiri. Kalau panjang seluruhnya N dan posisi 1 sudah terpakai, tersisa $N - 1$ posisi, dan dari situ $K - 2$ posisi terakhir harus memikul angka-angka yang belum terpakai. Tersisalah

$$(N - 1) - (K - 2) = N - K + 1$$

posisi yang boleh diisi dengan 0. Isilah semuanya dengan **0** — mengambil lebih sedikit akan memaksa angka yang lebih besar masuk ke tempat yang lebih berarti, dan itu lebih buruk.

Ke-$K - 2$ posisi terakhir. Posisi-posisi itu harus memikul $K - 2$ angka, masing-masing berbeda dari 0 dan 1 serta berbeda satu sama lain. Supaya bilangannya tetap kecil, kita pakai nilai-nilai terkecil yang tersedia, yaitu $2,3,\ldots,K - 1$, dan kita tempatkan dalam urutan **menaik** supaya yang lebih kecil duduk lebih ke kiri.

Kalau dirangkai:

$$\text{minimum} \;=\; 1\;\underbrace{00\cdots0}_{N-K+1}\;2\,3\cdots(K-1). \tag{1.6}$$

Hitung angkanya: $1 + (N - K + 1) + (K - 2) = N$. Hitung angka berbedanya: $\{1\} \cup \{0\} \cup \{2,\ldots,K - 1\} = \{0,1,\ldots,K - 1\}$, yang anggotanya tepat K.

Langkah 3 — nilai maksimum ketika $K \geq 2$. Bayangan cerminnya. Kerjakan dari kiri ke kanan, selalu memilih angka terbesar yang masih sanggup kita pakai.

Posisi-posisi terdepan. Angka terbesar adalah 9, dan tidak ada kekhawatiran nol di depan. Kita boleh memakai 9 di sebanyak mungkin posisi, tetapi kita harus menyisakan $K - 1$ posisi di ujung untuk angka-angka wajib lainnya. Jadi $N - K + 1$ posisi pertama semuanya mendapat **9**.

Ke-$K - 1$ posisi terakhir. Posisi-posisi itu harus memikul $K - 1$ angka, semuanya berbeda dari 9 dan berbeda satu sama lain, dan sebesar mungkin. Angka-angka itu adalah 8,7,6, … turun sampai $10 - K$, ditempatkan dalam urutan **menurun** supaya yang lebih besar duduk lebih ke kiri.

$$\text{maksimum} \;=\; \underbrace{99\cdots9}_{N-K+1}\;8\,7\cdots(10-K). \tag{1.7}$$

Cacah angkanya: $(N - K + 1) + (K - 1) = N$. Angka berbedanya: $\{9\} \cup \{8,7,\ldots,10 - K\} = \{10 - K,\ldots,9\}$, tepat K buah.

PEMERIKSAAN DENGAN TANGAN — $N = 5$, $K = 3$.

Minimum. Posisi 1 mendapat 1. Nol yang harus ditempatkan: $N - K + 1 = 5 - 3 + 1 = 3$, jadi tiga buah 0. Angka sisanya: $K - 2 = 1$, yaitu 2. Kalau dirangkai: 1 lalu 000 lalu 2 memberi **10002**. Angka yang terpakai: $\{1,0,2\}$ — tiga angka berbeda, sesuai permintaan. Panjangnya lima. Betul.

Maksimum. Sembilan yang harus ditempatkan: $N - K + 1 = 3$, jadi 999. Angka sisanya: $K - 1 = 2$, yaitu 8 lalu 7. Kalau dirangkai: 999 lalu 87 memberi **99987**. Angka yang terpakai: $\{9,8,7\}$ — tiga angka berbeda. Panjangnya lima. Betul.

Gambar 1.11 menabulasikan kedua konstruksi itu, Persamaan (1.6) dan Persamaan (1.7), untuk setiap N dari 1 sampai 10 dan setiap K dari 1 sampai 5.

Minimum

$K \backslash N$	1	2	3	4	5	6	7	8	9	10
1	1	11	111	1111	11111	111111	1111111	11111111	111111111	1111111111
2	–	10	100	1000	10000	100000	1000000	10000000	100000000	1000000000
3	–	–	102	1002	10002	100002	1000002	10000002	100000002	1000000002
4	–	–	–	1023	10023	100023	1000023	10000023	100000023	1000000023
5	–	–	–	–	10234	100234	1000234	10000234	100000234	1000000234

Maksimum

$K \backslash N$	1	2	3	4	5	6	7	8	9	10
1	9	99	999	9999	99999	999999	9999999	99999999	999999999	9999999999
2	–	98	998	9998	99998	999998	9999998	99999998	999999998	9999999998
3	–	–	987	9987	99987	999987	9999987	99999987	999999987	9999999987
4	–	–	–	9876	99876	999876	9999876	99999876	999999876	9999999876
5	–	–	–	–	98765	998765	9998765	99998765	999998765	9999998765

Gambar 1.11 Nilai minimum (atas, berarsir kuning) dan maksimum (bawah, berarsir kelabu) untuk $N = 1..10$ dan $K = 1..5$. Setiap baris adalah satu nilai K; bacalah melintang untuk N yang membesar.

Telusuri baris $K = 5$ pada **Gambar 1.11** di $N = 10$. Nilai minimumnya terbaca 1 0 0 0 0 0 0 2 3 4. Rumus kita: satu angka 1 di depan, lalu $N - K + 1 = 10 - 5 + 1 = 6$ nol, lalu $K - 2 = 3$ angka 2,3,4. Itu adalah 1, enam nol, 234 — persis kesepuluh angka yang sama. Nilai maksimumnya terbaca 9 9 9 9 9 9 8 7 6 5: enam sembilan, lalu $K - 1 = 4$ angka 8,7,6,5. Sama persis juga. Tabel dan rumusnya cocok di mana-mana, dan untuk itulah tabel itu digambar.

> **WAWASAN KUNCI:** Penempatan angka secara *greedy*: letakkan angka-angka ekstrem (1 lalu 0 untuk minimum, 9 untuk maksimum) di posisi yang paling berarti, dan tundalah angka-angka berbeda yang masih wajib ke ujung yang paling tidak berarti.

1.9.3 Fase 3 dan 4 — Merancang dan Menganalisis

1. Baca T. Ulangi T kali:
2. Baca N dan K.
3. Kalau $K = 1$: cetak N salinan `1`, sebuah spasi, N salinan `9`, sebuah pindah baris; lanjut ke kasus berikutnya.
4. Kalau tidak: cetak `1`, lalu $N - K + 1$ salinan `0`, lalu angka 2,3, … , $K - 1$; cetak sebuah spasi; lalu cetak $N - K + 1$ salinan `9`, lalu angka 8,7, … ,$10 - K$; cetak sebuah pindah baris.

Kebenaran. Sudah dibuktikan di atas dengan penalaran pilihan *greedy*, posisi demi posisi, beserta hitungan angka yang menegaskan bahwa kedua konstruksi itu punya tepat N angka dan tepat K angka berbeda.

Waktu. Setiap kasus uji mencetak $2N$ karakter dan mengerjakan kerja konstan per karakter: $O(N)$ per kasus, $O(TN)$ seluruhnya. Dengan $T \approx 200$ dan $N \leq 18$ itu sekitar 7,200 karakter yang dicetak — seketika.

Ruang. $O(1)$. Tidak ada yang disimpan; angkanya dicetak begitu ia dihitung.

1.9.4 Fase 5 — Mengimplementasikan

Kode 1.11 Penyelesaian lengkap untuk *Minimum and maximum numbers*, SPOJ NEWLANG.

```
#include <cstdio>

int main() {
    int t, n, k;
    scanf("%d", &t);
    while (t--) {
        scanf("%d %d", &n, &k);
        if (k == 1) {
            for (int i = 1; i <= n; i++) printf("1");
            printf(" ");
            for (int i = 1; i <= n; i++) printf("9");
            printf("\n");
            continue;
        }
        printf("1");
        for (int i = 1; i <= n - k + 1; i++) printf("0");
        for (int i = 1; i <= k - 2; i++) printf("%d", i + 1);
        printf(" ");
        for (int i = 1; i <= n - k + 1; i++) printf("9");
        for (int i = 1; i <= k - 1; i++) printf("%d", 9 - i);
        printf("\n");
    }
    return 0;
}
```

Dua gelung pantas dilihat sekali lagi. `for (int i = 1; i <= k - 2; i++) printf("%d", i + 1);` mencetak $2,3,\ldots,K-1$ dalam urutan menaik, sebab ketika i berjalan $1,2,\ldots,K-2$ nilai yang tercetak $i+1$ berjalan $2,3,\ldots,K-1$. Dan `for (int i = 1; i <= k - 1; i++) printf("%d", 9 - i);` mencetak $8,7,\ldots,10-K$ dalam urutan menurun, sebab ketika i berjalan $1,\ldots,K-1$ nilai yang tercetak $9-i$ berjalan $8,7,\ldots,10-K$. Ketika $K=2$ gelung yang pertama berjalan nol kali, dan itu persis betul: tidak diperlukan angka tambahan selain 1 dan 0.

Pada masukan contoh dari jurinya

```
2
1 1
3 3
```

Kode 1.11 sungguh-sungguh mencetak

```
1 9
102 987
```

yang cocok persis dengan **Gambar 1.10**. Tiga kasus lain dijalankan sebagai pemeriksaan. Untuk `5 3` ia mencetak `10002 99987`, sesuai dengan pemeriksaan dengan tangan di atas. Untuk `10 5` ia mencetak `1000000234 9999998765`, sesuai dengan baris $K=5$ pada **Gambar 1.11**. Untuk `10 2` ia mencetak `1000000000 9999999998`, sesuai dengan baris $K=2$.

1.10 Persoalan 5 — Interesting Sequence (E-Olymp 4894)

Paradigma: manipulasi bit / pola matematis.

1.10.1 Fase 1 — Memahami

Interesting sequence (E-Olymp 4894)

Batas waktu 1 dtk Batas memori 128 MB

Pernyataan

Barisan 011212201220200112… dibangun sebagai berikut: mula-mula 0; lalu tindakan berikut diulang-ulang: tambahkan barisan yang sudah ada di sebelah kanannya, dengan mengganti 0 menjadi 1, 1 menjadi 2, dan 2 menjadi 0, yaitu

`0 → 01 → 0112 → 01121220 → …`

Tentukan digit ke-m barisan itu.

Masukan

Baris pertama berisi banyaknya uji n $(1 \leq n \leq 1000)$. Masing-masing dari n baris berikutnya berisi m $(1 \leq m \leq 2^{63} - 1)$, posisi digit yang diminta.

Keluaran

Untuk setiap uji, cetak pada baris tersendiri digit pada posisi ke-m barisan itu.

Gambar 1.12 *Interesting sequence*, soal E-Olymp nomor 4894. Barisannya dibangun dengan berulang kali menempelkan salinan dirinya sendiri yang sudah digeser.

Gambar 1.12 mendefinisikan barisannya. Ia bermula dari satu angka tunggal 0. Lalu satu tindakan diulang selamanya: **tempelkan barisan yang sudah Anda punya ke sisi kanannya sendiri, tetapi dengan setiap** 0 **diubah menjadi** 1**, setiap** 1 **menjadi** 2**, dan setiap** 2 **menjadi** 0**.**

Kalau dituliskan, beberapa tahap pertamanya adalah

$$0 \;\rightarrow\; 01 \;\rightarrow\; 0112 \;\rightarrow\; 01121220 \;\rightarrow\; \cdots$$

Mari kita pastikan kita dapat menghasilkan ulang itu dengan tangan, sebab inilah persis "contoh kecil yang dikerjakan dengan pensil" dari Fase 1.

Tahap 0. Barisannya adalah `0`. *Tahap 1.* Ambil `0`. Geser: 0 → 1, memberi `1`. Tempelkan: `0` + `1` = `01`. *Tahap 2.* Ambil `01`. Geser: 0 → 1 dan 1 → 2, memberi `12`. Tempelkan: `01` + `12` = `0112`. *Tahap 3.* Ambil `0112`. Geser: 0 → 1, 1 → 2, 1 → 2, 2 → 0, memberi `1220`. Tempelkan: `0112` + `1220` = `01121220`. *Tahap 4.* Ambil `01121220`. Geser, memberi `12202001`. Tempelkan: `0112122012202001`.

Rumusan soalnya mengutip barisan itu sebagai bermula dengan `011212201220200112...`, dan untai tahap-4 kita `0112122012202001` diikuti dua karakter pertama tahap 5 memberi `011212201220200112`. Cocok.

Tugasnya. Diberi m, laporkan angka ke-m dari barisan itu, dihitung mulai dari 1.

Batasannya. Baris pertama memuat n, banyaknya pertanyaan, dengan $1 \leq n \leq 1000$. Masing-masing dari n baris berikutnya memuat satu m dengan $1 \leq m \leq 2^{63} - 1$.

Batas atas itu, $2^{63} - 1$, kira-kira 9.22×10^{18}. Tengoklah **Tabel 1.3** sekali lagi: tidak ada satu pun yang bahkan *membilang sampai* m yang akan selamat. Kita sedang diberi tahu bahwa ada sebuah rumus.

Interesting sequence (E-Olymp 4894)

Batas waktu 1 dtk Batas memori 128 MB

Contoh

Masukan #1	Jawaban #1
3 17 239 385	1 0 2

Gambar 1.13 Contoh uji *Interesting sequence*: tiga pertanyaan 17, 239, 385, dengan jawaban 1, 0, 2.

1.10.2 Fase 2 — Mengabstraksi

Langkah 1 — beri nama transformasinya. Penggantian $0 \to 1$, $1 \to 2$, $2 \to 0$ tampak seperti permutasi tiga lambang, tetapi lihatlah bilangannya: $0 + 1 = 1$, $1 + 1 = 2$, $2 + 1 = 3$, dan 3 melingkar balik menjadi 0. Transformasinya sekadar **tambah satu, modulo tiga**. Menuliskan $v \mapsto (v + 1) \bmod 3$ menggantikan tabel tiga baris dengan satu potong hitungan, dan hitungan adalah sesuatu yang dapat kita nalar.

Jadi: paruh kedua barisan pada setiap tahap adalah paruh pertamanya dengan setiap nilai dinaikkan 1 modulo 3.

Langkah 2 — rekursi atas penyajian binernya. Barisan itu berlipat dua panjangnya setiap tahap: sesudah k tahap ia punya 2^k angka. Untuk sementara, indekslah angkanya mulai dari 0 alih-alih 1; sebut indeksnya p.

Andaikan p terletak dalam sebuah barisan sepanjang 2^{k+1}. Ada dua kasus.

- Kalau $p < 2^k$, angkanya ada di **paruh pertama**, yang tidak lain adalah barisan sepanjang 2^k. Jadi nilainya sama dengan nilainya satu tahap sebelumnya.
- Kalau $p \geq 2^k$, angkanya ada di **paruh kedua**. Tulis $p = 2^k + q$ dengan $0 \leq q < 2^k$. Paruh kedua adalah paruh pertama yang sudah digeser, jadi nilainya adalah (nilai di q) + 1 mod 3.

Sekarang pikirkan seperti apa kedua kasus itu dalam bentuk **biner**. Penyajian biner sebuah bilangan menuliskannya sebagai jumlah pangkat-pangkat dua; misalnya $13 = 8 + 4 + 1$, jadi dalam biner 13 adalah 1101, artinya satu delapan, satu empat, tidak ada dua, satu satuan. Untuk sebuah bilangan $p < 2^{k+1}$, bertanya "apakah $p \geq 2^k$?" persis sama dengan bertanya "apakah bit berbobot 2^k — bit terdepan — bernilai 1?" Dan membentuk $q = p - 2^k$ persis sama dengan "hapus bit itu".

Jadi rekursinya berbunyi: *kupas bit-bit p satu per satu dari atas; setiap bit yang bernilai* 1 *menambah satu, modulo tiga; setiap bit yang bernilai* 0 *tidak menambah apa-apa.*

Langkah 3 — rumus tertutupnya. Kalau setiap bit 1 menyumbang $+1$ dan setiap bit 0 tidak menyumbang apa pun, dan kita mulai dari nilai 0 pada $p = 0$, maka

$$\text{nilai}(p) \;=\; (\text{banyaknya bit 1 dalam bentuk biner } p) \bmod 3. \tag{1.8}$$

Banyaknya bit 1 pada sebuah bilangan disebut **cacah populasi** bitnya, atau *popcount*.

Bukti dengan induksi untuk Persamaan (1.8), sebab Langkah 2 adalah penalaran sedangkan ini klaimnya. Misalkan S_k melambangkan barisan sesudah k tahap, panjangnya 2^k, dan $P(p)$ melambangkan popcount dari p.

Kasus dasar. S_0 adalah angka tunggal 0 pada indeks $p = 0$. Bentuk biner 0 tidak punya bit 1, jadi $P(0) = 0$, dan $0 \bmod 3 = 0$. Klaimnya terpenuhi.

Langkah induktif. Andaikan $S_k[p] = P(p) \bmod 3$ untuk semua $0 \le p < 2^k$. Ambil sembarang p dengan $0 \le p < 2^{k+1}$. Kalau $p < 2^k$ maka $S_{k+1}[p] = S_k[p] = P(p) \bmod 3$ menurut pengandaian tadi, dan kita selesai. Kalau $p \ge 2^k$, tulis $p = 2^k + q$ dengan $0 \le q < 2^k$. Karena $q < 2^k$, bit berbobot 2^k pada q bernilai 0, jadi menambahkan 2^k sekadar menyalakan bit itu tanpa mengusik yang lain. Karenanya $P(p) = P(q) + 1$. Menurut cara barisan itu dibangun, $S_{k+1}[p] = (S_k[q] + 1) \bmod 3 = (P(q) \bmod 3 + 1) \bmod 3 = (P(q) + 1) \bmod 3 = P(p) \bmod 3$. Klaimnya terpenuhi untuk $k + 1$. Menurut induksi ia terpenuhi untuk setiap k, karenanya untuk setiap indeks. ■

Langkah 4 — sesuaikan pengindeksannya. Persoalannya menomori angkanya mulai dari 1; rumus kita menomorinya mulai dari 0. Jadi angka pada posisi m adalah nilai pada indeks $p = m - 1$.

> **WAWASAN KUNCI:** jawaban = popcount$(m - 1) \bmod 3$.

Gambar 1.14 menabulasikan dua belas indeks pertama, bentuk binernya, popcount-nya, dan popcount modulo tiga.

N	Biner				#bit-1	(#bit-1) mod 3
0	0	0	0	0	0	**0**
1	0	0	0	1	1	**1**
2	0	0	1	0	1	**1**
3	0	0	1	1	2	**2**
4	0	1	0	0	1	**1**
5	0	1	0	1	2	**2**
6	0	1	1	0	2	**2**
7	0	1	1	1	3	**0**
8	1	0	0	0	1	**1**
9	1	0	0	1	2	**2**
10	1	0	1	0	2	**2**
11	1	0	1	1	3	**0**
	MSB			LSB		

Gambar 1.14 Indeks 0 sampai 11 dalam biner, dengan banyaknya bit 1 dan cacah itu modulo 3. Kolom terakhir adalah barisannya sendiri.

Bacalah kolom terakhir **Gambar 1.14** lurus ke bawah: 0,1,1,2,1,2,2,0,1,2,2,0. Sekarang bacalah dua belas angka pertama barisan yang kita bangun dengan tangan di Fase 1, `0112122012202001`: angkanya 0,1,1,2,1,2,2,0,1,2,2,0. Sama persis. Tabel itu *adalah* barisannya.

> **PEMERIKSAAN DENGAN TANGAN — ketiga pertanyaan contoh uji.**
>
> *Pertanyaan* $m = 17$. Maka $p = 16$. Dalam biner $16 = 10000$. Banyaknya bit 1: 1. $1 \bmod 3 = \mathbf{1}$.
>
> *Pertanyaan* $m = 239$. Maka $p = 238$. Ubah ke biner dengan berulang kali mengeluarkan pangkat dua terbesar: $238 - 128 = 110$; $110 - 64 = 46$; $46 - 32 = 14$; $14 - 8 = 6$; $6 - 4 = 2$;

> $2 - 2 = 0$. Jadi $238 = 128 + 64 + 32 + 8 + 4 + 2$, yang dalam biner adalah 11101110. Banyaknya bit 1: 6. $6 \bmod 3 = \mathbf{0}$.
>
> *Pertanyaan* $m = 385$. Maka $p = 384$. $384 - 256 = 128$; $128 - 128 = 0$. Jadi $384 = 256 + 128$, dalam biner 110000000. Banyaknya bit 1: 2. $2 \bmod 3 = \mathbf{2}$.
>
> Ketiga jawabannya adalah 1, 0, 2 — persis yang diharapkan **Gambar 1.13**.

Tiga pemeriksaan sekilas memang menenangkan tetapi belum memutuskan. **Kode 1.12** membangun barisan yang sesungguhnya sampai 128 suku dengan benar-benar mengikuti cara pembangunannya, lalu membandingkan setiap suku dengan rumusnya.

Kode 1.12 Membangun *Interesting sequence* secara *brute force* dan menguji rumus popcount terhadap setiap sukunya.

```
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    vector<int> s(1, 0);                        // barisannya bermula sebagai [0]
    for (int step = 0; step < 7; step++) {      // 7 kali pelipatduaan memberi 128 suku
        vector<int> t = s;
        for (size_t i = 0; i < t.size(); i++) t[i] = (t[i] + 1) % 3;
        s.insert(s.end(), t.begin(), t.end());
    }
    printf("first 32 terms : ");
    for (int i = 0; i < 32; i++) printf("%d", s[i]);
    printf("\n");
    int bad = 0;                                // bandingkan rumus dengan barisan sungguhan
    for (size_t i = 0; i < s.size(); i++)
        if (s[i] != __builtin_popcountll(i) % 3) bad++;
    printf("terms checked  : %d, mismatches : %d\n", (int)s.size(), bad);
    return 0;
}
```

Keluarannya yang sesungguhnya:

```
first 32 terms : 01121220122020011220200120010112
terms checked  : 128, mismatches : 0
```

Kedelapan belas karakter pertamanya, `011212201220200112`, persis awalan yang tercetak pada rumusan persoalan di **Gambar 1.12**, dan tidak satu pun dari 128 suku itu bertentangan dengan rumusnya.

1.10.3 Fase 3 dan 4 — Merancang dan Menganalisis

1. Baca n, banyaknya pertanyaan.
2. Untuk setiap pertanyaan, baca m sebagai bilangan bulat tak bertanda 64 bit, hitung popcount dari $m - 1$, cetak hasilnya modulo 3.

Kebenaran. Sudah dibuktikan dengan induksi pada Langkah 3, ditambah pergeseran indeks pada Langkah 4.

Waktu. $O(1)$ per pertanyaan. Popcount sebuah nilai 64 bit adalah satu instruksi mesin tunggal pada prosesor masa kini, yang dipancarkan GCC untuk `__builtin_popcountll`. Untuk $n \leq 1000$ pertanyaan, kerja totalnya dapat diabaikan.

Ruang. $O(1)$. Kita tidak pernah menyimpan barisannya — dan syukurlah begitu, sebab sesudah 63 kali pelipatduaan ia akan punya 2^{63} suku.

1.10.4 Fase 5 — Mengimplementasikan

Kode 1.13 Penyelesaian lengkap untuk *Interesting sequence*, E-Olymp 4894.

```
#include <iostream>
using namespace std;
typedef unsigned long long LL;

int main() {
    LL n, x;
    cin >> n;
    while (n--) {
        cin >> x;
        LL tmp = __builtin_popcountll(x - 1);   // m berindeks-1 menjadi m-1 berindeks-0
        cout << tmp % 3 << endl;
    }
    return 0;
}
```

Tipe `unsigned long long` memuat bilangan bulat dari 0 sampai $2^{64} - 1$, yang dengan lapang mencakup batas $2^{63} - 1$; sebuah `int` biasa, yang berhenti di sekitar 2.1×10^9, akan meluap dengan mengerikan. `__builtin_popcountll` adalah nama yang dipakai GCC untuk popcount sebuah nilai 64 bit — akhiran `ll` berarti *long long*. Pada C++20 ejaan yang mudah dipindahkan adalah `std::popcount` dari `<bit>`.

Pada masukan contoh dari jurinya

```
3
17
239
385
```

Kode 1.13 sungguh-sungguh mencetak

```
1
0
2
```

sesuai dengan **Gambar 1.13**. Ketika diminta delapan suku pertama ($m = 1$ sampai 8) ia mencetak `0 1 1 2 1 2 2 0`, yang tidak lain barisan tahap-3 `01121220` yang kita bangun dengan tangan.

1.11 Persoalan 6 — Recursive Function 1 (E-Olymp 10296)

Paradigma: *dynamic programming* / memoisasi.

1.11.1 Fase 1 — Memahami

Recursive function 1 (E-Olymp 10296)

Batas waktu 1 dtk Batas memori 128 MB

Pernyataan

Tentukan nilai fungsi

$$f(n) = \begin{cases} 1, & n = 0, \\ f\left(\lfloor \frac{n}{2} \rfloor\right) + f\left(\lfloor \frac{n}{3} \rfloor\right), & n > 0. \end{cases}$$

Masukan

Satu bilangan bulat positif n ($1 \le n \le 10^{18}$).

Keluaran

Cetak nilai $f(n)$.

Contoh

Masukan #1	Jawaban #1
5	5

Gambar 1.15 *Recursive function 1*, soal E-Olymp nomor 10296. Definisinya, masukannya yang berupa satu bilangan bulat, dan contoh uji $f(5) = 5$.

Gambar 1.15 mendefinisikan sebuah fungsi f atas bilangan bulat:

$$f(n) = \begin{cases} 1, & n = 0, \\ f\left(\lfloor \frac{n}{2} \rfloor\right) + f\left(\lfloor \frac{n}{3} \rfloor\right), & n > 0. \end{cases} \tag{1.9}$$

Lambang $\lfloor x \rfloor$ dalam Persamaan (1.9) adalah **floor** (pembulatan ke bawah) dari x: bilangan bulat terbesar yang tidak lebih besar daripada x. Dengan bahasa biasa artinya "bagilah lalu buang bagian pecahannya". Jadi $\lfloor 7/2 \rfloor = 3$, $\lfloor 7/3 \rfloor = 2$, $\lfloor 1/2 \rfloor = 0$. Dalam C++, pembagian bilangan bulat dengan `/` atas nilai tak negatif mengerjakan persis itu, jadi `n / 2` sudah merupakan $\lfloor n/2 \rfloor$.

Masukannya. Satu bilangan bulat n dengan $1 \le n \le 10^{18}$. **Keluarannya.** Nilai $f(n)$. **Batasnya.** Satu detik.

Contoh ujinya adalah $n = 5$ dengan jawaban 5.

> **CONTOH KERJA TANGAN — contoh uji dari jurinya.** $f(5) = f(\lfloor 5/2 \rfloor) + f(\lfloor 5/3 \rfloor) = f(2) + f(1)$. $f(2) = f(1) + f(0)$. $f(1) = f(0) + f(0) = 1 + 1 = 2$. Jadi $f(2) = 2 + 1 = 3$. Jadi $f(5) = 3 + 2 = 5$. Inilah jawaban yang diharapkan.

1.11.2 Fase 2 — Mengabstraksi

Langkah 1 — rekursi naifnya tanpa harapan. Terjemahkan definisinya langsung menjadi kode dan setiap pemanggilan dengan argumen lebih besar daripada 0 melahirkan dua pemanggilan lagi. Berapa banyak pemanggilan yang terjadi seluruhnya? Kita dapat menghitungnya dengan tepat lewat sebuah gambar kecil.

Gambarlah setiap pemanggilan sebagai simpul sebuah pohon. Pemanggilan dengan argumen $u > 0$ mempunyai tepat dua anak, yaitu pemanggilan untuk $\lfloor u/2 \rfloor$ dan $\lfloor u/3 \rfloor$. Pemanggilan dengan argumen 0 tidak mempunyai anak; ia sebuah *daun*, dan ia mengembalikan 1. Setiap nilai dalam perhitungan itu dibangun dengan menjumlahkan angka-angka 1 tersebut, sehingga $f(n)$ persis

sama dengan banyaknya daun. Akhirnya, pohon yang setiap simpul dalamnya mempunyai tepat dua anak selalu memiliki simpul dalam satu lebih sedikit daripada daunnya (setiap simpul dalam menyatukan dua subpohon menjadi satu, dan kita bermula dengan subpohon sebanyak daun lalu berakhir dengan satu). Jadi banyaknya seluruh pemanggilan adalah

$$\text{panggilan}(n) \;=\; f(n) + (f(n) - 1) \;=\; 2f(n) - 1. \tag{1.10}$$

Periksalah Persamaan (1.10) pada contoh ujinya. Karena $f(5) = 5$, persamaan itu meramalkan $2 \cdot 5 - 1 = 9$ pemanggilan. Contoh yang dikerjakan dengan tangan di Subbab 1.11.1 melakukan pemanggilan $f(5), f(2), f(1), f(0), f(0), f(0), f(1), f(0), f(0)$ — sembilan buah.

Untuk masukan terbesar, $f(10^{18}) \approx 2.24 \times 10^{14}$ (Subbab 1.11.3 menghitungnya dengan tepat), sehingga rekursi naifnya melakukan sekitar 4.5×10^{14} pemanggilan. Melawan anggaran 10^8 operasi per detik, itu sekitar 4.5×10^6 detik — kira-kira lima puluh dua hari. Tidak dapat diterima. (Taksiran yang lebih cepat, "dua pemanggilan masing-masing, sedalam kira-kira $\log_2 10^{18} \approx 60$ aras, jadi 2^{60} pemanggilan", melampaui kenyataan, sebab cabang-cabang $\lfloor n/3 \rfloor$ sudah mencapai 0 sesudah hanya sekitar 38 aras. Namun kesimpulannya sama.)

Tetapi perhatikan *mengapa* ia seburuk itu. Lihat lagi penelusuran contoh ujinya: $f(1)$ diperlukan sekali di dalam $f(2)$ dan sekali lagi langsung di dalam $f(5)$. Tidak ada yang baru dihitung pada kesempatan kedua. Inilah persis gejala sub-persoalan bertumpang tindih dari Subbab 1.4.4.

Langkah 2 — hitung ada berapa sub-persoalan yang sungguh-sungguh *berbeda*. Berangkat dari n, setiap nilai yang mungkin ditanyakan kepada kita dicapai lewat serangkaian pembagian dua dan pembagian tiga. Nilai semacam itu berbentuk

$$\left\lfloor \frac{n}{2^a\, 3^b} \right\rfloor \quad \text{untuk bilangan bulat } a, b \geq 0. \tag{1.11}$$

(Pembulatan ke bawah itu sebenarnya boleh dilakukan dalam urutan apa pun di sepanjang jalan, sebab $\lfloor \lfloor n/p \rfloor / q \rfloor = \lfloor n/(pq) \rfloor$ untuk bilangan bulat positif p dan q; jadi Persamaan (1.11) tepat, bukan hampiran.) Seberapa besar a bisa menjadi? Begitu 2^a melampaui n, nilainya 0 dan kita berhenti, jadi $a \leq \log_2 n$. Untuk $n = 10^{18}$ itu paling banyak 60. Serupa itu $b \leq \log_3 n \approx 38$. Jadi ada paling banyak $61 \times 39 = 2{,}379$ pasangan (a, b), dan karenanya paling banyak sekitar 2,400 sub-persoalan yang berbeda — dan banyak di antaranya pun berimpit, sebab pasangan (a, b) yang berbeda bisa memberi floor yang sama.

Dua ribu empat ratus, melawan 4.5×10^{14}. Itulah seluruh wawasannya.

Langkah 3 — memoisasikan. Peliharalah sebuah tabel nilai yang sudah pernah dihitung. Sebelum menghitung $f(u)$, tengoklah tabelnya; kalau ia ada di sana, kembalikan saja. Kalau tidak, hitunglah, simpan, lalu kembalikan.

Tabel yang mana? Bukan larik. Larik yang berindeks n akan memerlukan 10^{18} entri — dengan delapan bita per entri, delapan juta terabita. Kita memerlukan wadah yang hanya menyimpan kunci-kunci yang benar-benar kita temui, yang tersebar di mana saja dalam rentang yang mahabesar. `std::map<long long, long long>` mengerjakan persis itu: ia menampung entri yang kita sisipkan dan tidak lebih, dan mencari sebuah kunci dalam waktu $O(\log(\text{banyak entri}))$.

> **WAWASAN KUNCI:** Walaupun n boleh sebesar 10^{18}, hanya sekitar seribu sub-persoalan berbeda yang pernah muncul. Sebuah peta hash atau peta terurut mengubah perhitungan yang mustahil menjadi perhitungan yang seketika. Algoritmanya tidak berubah; struktur datanyalah yang berubah.

Kode 1.14 memutuskan cacahnya secara empiris alih-alih lewat taksiran: ia menyusuri rekursinya dari masukan sah terbesar dan mencatat setiap nilai berbeda yang pernah dicapainya.

Kode 1.14 Menghitung sub-persoalan berbeda dari f dengan menjelajahi rekursinya memakai sebuah `set`.

```
#include <cstdio>
#include <set>
using namespace std;
typedef long long ll;

set<ll> seen;

void walk(ll u) {
    if (seen.count(u)) return;          // sub-persoalan ini sudah pernah dicapai
    seen.insert(u);
    if (u == 0) return;
    walk(u / 2);
    walk(u / 3);
}

int main() {
    ll n = 1000000000000000000LL;       // 10^18, masukan terbesar yang diizinkan
    walk(n);
    printf("distinct sub-problems for n = %lld : %d\n", n, (int)seen.size());
    seen.clear(); walk(12);
    printf("distinct sub-problems for n = 12 : %d\n", (int)seen.size());
    return 0;
}
```

Keluarannya yang sesungguhnya:

```
distinct sub-problems for n = 1000000000000000000 : 1012
distinct sub-problems for n = 12 : 7
```

Seribu dua belas. Taksiran "sekitar 2,400" tadi adalah batas atas dan kenyataannya bahkan lebih kecil lagi, sebagaimana lazimnya batas atas. Kedua angka itu sama-sama kecil tak berarti.

> **PENELUSURAN DENGAN TANGAN — $f(12)$, dengan isi cache dituliskan pada setiap langkah.**

$f(12) = f(6) + f(4)$ — belum satu pun diketahui, jadi turunlah dulu ke $f(6)$. $f(6) = f(3) + f(2)$ — turun ke $f(3)$. $f(3) = f(1) + f(1)$ — turun ke $f(1)$. $f(1) = f(0) + f(0) = 1 + 1 = 2$. **Simpan** $f(1) = 2$. $f(3) = 2 + 2 = 4$. $f(1)$ yang kedua datang langsung dari cache. **Simpan** $f(3) = 4$. $f(2) = f(1) + f(0) = 2 + 1 = 3$, keduanya dari cache. **Simpan** $f(2) = 3$. $f(6) = 4 + 3 = 7$. **Simpan** $f(6) = 7$. $f(4) = f(2) + f(1) = 3 + 2 = 5$ — **keduanya dari cache, sama sekali tanpa rekursi.** $f(12) = 7 + 5 = 12$.

Hitunglah nilai berbeda yang disentuh penelusuran itu: 12,6,4,3,2,1,0 — tujuh buah, persis seperti yang dilaporkan **Kode 1.14**. Dan jawaban $f(12) = 12$ itu kebetulan, bukan pola; $f(11) = 9$ dan $f(13) = 12$.

Tabel 1.6 Tabel memoisasi yang terbentuk sewaktu menghitung $f(12)$, dalam urutan entrinya terisi.

u	$\lfloor u/2 \rfloor$	$\lfloor u/3 \rfloor$	$f(u)$	Kapan ia masuk cache
0	—	—	1	ditanam sebelum program berjalan
1	0	0	$1+1 = 2$	pertama
2	1	0	$2+1 = 3$	ketiga
3	1	1	$2+2 = 4$	kedua
4	2	1	$3+2 = 5$	kelima
6	3	2	$4+3 = 7$	keempat
12	6	4	$7+5 = 12$	terakhir

Tabel 1.6 adalah penelusuran yang sama, dibentangkan sebagai tabel yang sungguh-sungguh dipegang program itu di dalam memori. Bacalah kolom terakhirnya untuk melihat urutan rekursi mengisinya: nilai yang paling dalam lebih dulu, pertanyaan aslinya paling akhir.

1.11.3 Fase 3 dan 4 — Merancang dan Menganalisis

1. Buat sebuah map kosong dan masukkan satu entri $f(0) = 1$ ke dalamnya.
2. Definisikan `rec(u)`: kalau u adalah salah satu kunci map itu, kembalikan nilai yang tersimpan; kalau tidak, hitung `rec(u/2) + rec(u/3)`, simpan di bawah kunci u, lalu kembalikan.
3. Baca n, cetak `rec(n)`.

Kebenaran. Dengan induksi kuat atas u. Untuk $u = 0$ nilai yang ditanam itu memang definisinya. Untuk $u > 0$, baik $\lfloor u/2 \rfloor$ maupun $\lfloor u/3 \rfloor$ benar-benar lebih kecil daripada u (sebab $u \geq 1$ mengakibatkan $\lfloor u/2 \rfloor \leq u/2 < u$), jadi menurut hipotesis induksi kedua pemanggilan rekursif itu mengembalikan nilai f yang betul pada argumen-argumen tersebut; jumlahnya adalah $f(u)$ menurut definisi. Cache hanya pernah mengembalikan nilai yang dihitung dengan aturan yang sama ini, jadi penyimpanan di cache tidak dapat menyusupkan kekeliruan.

Terminasi. Setiap pemanggilan rekursif punya argumen tak negatif yang benar-benar lebih kecil, jadi rekursinya tidak dapat berlangsung selamanya; ia pasti mencapai 0.

Waktu. Misalkan D banyaknya sub-persoalan berbeda, yang diukur di atas sebagai 1,012 untuk kasus terburuk. Masing-masing dihitung sekali, dan setiap penghitungan mengerjakan sejumlah tetap operasi map yang masing-masing $O(\log D)$. Totalnya $O(D \log D)$, yang untuk $D \approx 10^3$ berarti beberapa puluh ribu langkah dasar. Dinyatakan dalam n, besaran D adalah $O(\log^2 n)$, sehingga waktu jalannya $O(\log^2 n \cdot \log\log n)$.

Ruang. $O(D) = O(\log^2 n)$ entri map — sekitar seribu, atau beberapa puluh kilobita.

Apakah jawabannya muat? f bertumbuh, jadi kita semestinya memeriksa. Menjalankan programnya pada masukan sah terbesar $n = 10^{18}$ memberi $f(n) = 224{,}412{,}958{,}156{,}019$, kira-kira 2.2×10^{14}. Sebuah `long long` memuat nilai sampai kira-kira 9.2×10^{18}, jadi tidak ada *overflow*. Pemeriksaan ini bagian dari Fase 4, dan melewatinya adalah cara yang lazim untuk mengubah algoritma yang benar menjadi *Wrong Answer*.

1.11.4 Fase 5 — Mengimplementasikan

Kode 1.15 Penyelesaian lengkap untuk *Recursive function 1*, E-Olymp 10296.

```
#include <map>
#include <cstdio>
using namespace std;
typedef long long ll;

map<ll, ll> mp;

ll rec(ll u) {
    if (mp.count(u)) return mp[u];      // sudah terpecahkan: serahkan jawaban tersimpan
    ll nval = rec(u / 2) + rec(u / 3);
    mp[u] = nval;
    return nval;
}

int main() {
    mp[0] = 1;                          // kasus dasar, ditanam ke dalam cache
    ll u;
    scanf("%lld", &u);
    printf("%lld\n", rec(u));
    return 0;
}
```

Perhatikan bagaimana kasus dasarnya ditangani pada **Kode 1.15**. Tidak ada `if (u == 0) return 1;` di dalam `rec`. Sebagai gantinya `mp[0] = 1` dituliskan ke dalam cache sebelum pemanggilan pertama, sehingga uji "apakah ia sudah di cache?" yang biasa itulah yang menangkapnya. Ini sepotong kecil penghematan khas pemrograman kompetitif: satu mekanisme mengerjakan dua pekerjaan.

Dijalankan pada masukan contoh `5`, **Kode 1.15** sungguh-sungguh mencetak:

```
5
```

Dijalankan pada `12` ia sungguh-sungguh mencetak `12`, sesuai dengan penelusuran dengan tangan tadi. Dijalankan pada `1000000000000000000` — masukan terbesar yang diizinkan persoalan ini — ia sungguh-sungguh mencetak `224412958156019`, dan menjawab seketika.

1.12 Keenam Persoalan dalam Sekilas Pandang

Tabel 1.7 Keenam persoalan bab ini, paradigmanya, dan satu kalimat yang menyelesaikan masing-masing.

Persoalan	Paradigma	Teknik intinya	Waktu	Jawabannya dalam satu kalimat
Columns	*Greedy*	Maksimum berjalan	$O(n)$	max(cacah kolom, tinggi tertinggi)
Journey	*Greedy*	Rentang lariknya	$O(n)$	$2 \times (\max - \min)$ pada sebuah garis
Unique Numbers	Teori bilangan	Reduksi dengan gcd	$O(k\log n)$	Sebuah perkalian menciutkan semesta sebesar gcd-nya
Min/Max Numbers	*Greedy* konstruktif	Penempatan angka secara rakus	$O(N)$	Angka ekstrem di ujung yang paling berarti
Interesting Sequence	Trik bit	Popcount modulo 3	$O(1)$	Pelipatduaan berarti biner; hitung bit 1-nya
Recursive Function 1	Memoisasi	*Dynamic programming* berbasis map	$O(\log^2 n)$	Sekitar seribu sub-persoalan walaupun $n = 10^{18}$

Lihatlah ke bawah pada kolom "Waktu" **Tabel 1.7**. Tidak satu pun dari keenam penyelesaian itu lebih buruk daripada $O(n)$, dan dua di antaranya sama sekali tidak bergantung pada ukuran masukan. Tidak satu pun dari keenam programnya lebih panjang daripada dua puluh baris. Namun tidak satu pun dari keenamnya *mudah* — masing-masing menuntut wawasan yang sungguh-sungguh di Fase 2. Begitulah wajah bidang ini: kesulitannya tinggal di dalam pemikiran, dan pemikiran, ketika ia berhasil, membuat kodenya menjadi pendek.

1.12.1 Mengenali paradigma mana yang berlaku

Pengenalan pola adalah keterampilan, dan seperti setiap keterampilan ia dibangun dengan melihat banyak contoh. **Tabel 1.8** adalah bekal awalnya: sederet ungkapan yang lazim muncul dalam rumusan persoalan, dan apa yang semestinya dicurigai masing-masing.

Tabel 1.8 Gejala dan diagnosisnya. Ketika sebuah rumusan persoalan memuat ungkapan di sebelah kiri, teknik di sebelah kanan layak dicoba lebih dulu.

Kalau Anda melihat…	Pikirkan…	Contohnya di sini
"Carilah minimum / maksimum dari sesuatu"	Satu kali pindai, agregat berjalan	*Columns*, *Journey*
"Hitung modulo, keterbagian, sisa pembagian"	Gcd dan sifat-sifat teori bilangan	*Unique Numbers*
"Bangunlah X yang optimal"	*Greedy*, posisi paling berarti lebih dulu	*Min/Max Numbers*
"Barisannya berlipat dua setiap langkah"	Penyajian biner, popcount	*Interesting Sequence*
"$f(n)$ bergantung pada $f(n/a)$ dan $f(n/b)$"	Memoisasi dengan map (DP jarang)	*Recursive Function 1*
"n luar biasa besar tetapi keadaan berbedanya sedikit"	DP berbasis map, jangan pernah larik	*Recursive Function 1*
"Singgahi semuanya lalu pulang" pada sebuah garis	Geometri, bukan pedagang keliling	*Journey*
"Warnai / labeli agar tetangganya berbeda"	Batas bawah ditambah konstruksi yang gamblang	*Columns*

Ini semua kiat kasar, bukan hukum. **Tabel 1.8** tidak akan menyelesaikan persoalan untuk Anda; ia akan memberi tahu ke mana perhatian Anda sebaiknya diarahkan lebih dulu, dan itulah sebagian besar isi dari apa yang disebut pengalaman.

1.13 Dari Mana Persoalan Ini Berasal: Para *Online Judge*

Sebuah **online judge** (juri daring) adalah situs web yang menyimpan persoalan beserta data ujinya yang dirahasiakan. Anda mengirimkan sebuah program; jurinya mengompilasi, menjalankannya terhadap setiap uji, membandingkan keluarannya dengan jawaban yang diharapkan karakter demi karakter, lalu mengembalikan sebuah vonis. Vonis yang penting adalah *Accepted*, *Wrong Answer*, *Time Limit Exceeded*, *Memory Limit Exceeded*, *Runtime Error*, dan *Compilation Error*.

Ini bentuk evaluasi yang jujurnya luar biasa. Jurinya tidak peduli seberapa elok gagasan Anda. Ia tidak memberi nilai sebagian untuk kesungguhan usaha. Program Anda menghasilkan jawaban yang betul dalam batas waktu pada uji-uji yang belum pernah Anda lihat, atau tidak — dan uji-uji yang belum pernah Anda lihat itu justru intinya, sebab itulah yang memaksa Anda menalar tentang *semua* masukan, bukan tentang tiga masukan yang ada di rumusan soalnya.

Buku ini terutama mengambil persoalannya dari dua juri, dan **Tabel 1.9** memperkenalkan keduanya bersama dua juri lain yang pantas dikenal ketika Anda melangkah lebih jauh.

Tabel 1.9 Empat *online judge* yang dijumpai dalam buku ini. Ukurannya berubah setiap hari; angka-angkanya adalah yang dilaporkan pada tanggal yang tercantum.

Juri	Alamat	Ukuran, sebagaimana dilaporkan	Sumbangannya bagi buku ini
E-Olymp	`eolymp.com`	9.178 soal dalam arsipnya (September 2026) [3]	*Columns* (10281), *Journey* (10280), *Interesting sequence* (4894), *Recursive function 1* (10296)
SPOJ, Sphere Online Judge	`spoj.com`	sekitar 1,29 juta pengguna terdaftar dan 35,6 juta pengiriman (halaman depan, awal 2026) [4]	*Unique Numbers* (DCEPC901), *Minimum and maximum numbers* (NEWLANG), serta peringkat terbuka penyelesaian terbaik untuk setiap soal
UVa Online Judge	`onlinejudge.org`	lebih dari 4.300 soal (September 2026) [5]	Arsip yang diindeks soal demi soal oleh *Competitive Programming 4* [6]
beecrowd (dahulu URI Online Judge)	`beecrowd.com`	lebih dari satu juta pengembang di lebih dari 100 negara, dan lebih dari 2.500 soal (awal 2026) [7]	Bagian pemula yang ramah dalam bahasa Inggris dan Portugis

E-Olymp [3] adalah juri asal Ukraina dengan arsip soal yang besar dan bertingkat rapi, serta antarmuka yang tersedia dalam bahasa Inggris. Empat dari enam persoalan bab ini tinggal di sana: *Columns* adalah soal 10281, *Journey* adalah 10280, *Interesting sequence* adalah 4894, dan *Recursive function 1* adalah 10296.

SPOJ [4], Sphere Online Judge, adalah salah satu juri tertua dan terbesar; pada awal 2026 halaman depannya melaporkan sekitar 1,29 juta pengguna terdaftar dan lebih dari 35 juta pengiriman. Dua persoalan bab ini yang lain — *Unique Numbers* (DCEPC901) dan *Minimum and maximum numbers* (NEWLANG) — adalah soal SPOJ. SPOJ juga menerbitkan, untuk setiap soal, peringkat penyelesaian tercepat yang diterima, dan itu akan kita manfaatkan di Subbab 1.14.

UVa Online Judge [8], dengan lebih dari 4.300 soal [5], adalah arsip yang diindeks soal demi soal oleh *Competitive Programming 4* [6]; kalau Anda menggarap buku itu, di sinilah Anda mengirimkan jawabannya. **beecrowd** [7], dahulu URI Online Judge, adalah juri terbesar di Amerika Latin, dengan bagian pemula yang cukup luas dalam bahasa Inggris dan Portugis, dan ia tempat yang ramah untuk memulai kalau kumpulan soal E-Olymp dan SPOJ terasa terlalu terjal.

> **CARA MEMAKAI JURI DENGAN BAIK.** Jangan perlakukan juri sebagai kompilator yang lambat membalas. Setiap pengiriman semestinya melewati kelima fase. Kerjakan contoh ujinya dengan tangan *sebelum* Anda mengirim; kalau jawaban tangan Anda berbeda dari contohnya, jangan kirim sama sekali. Ketika Anda mendapat *Wrong Answer*, jangan mengaduk-aduk kode lalu mengirim ulang — kembalilah ke Fase 2 dan tanyakan kelas masukan mana yang belum Anda pikirkan. Ketika Anda mendapat *Time Limit Exceeded*, jangan mengoptimalkan gelungnya sedikit-sedikit — kembalilah ke Fase 4 dan hitung ulang kompleksitasnya terhadap **Tabel 1.3**. Juri menguji penalaran Anda, dan memperlakukannya sebagai alat uji kecepatan mengetik berarti menyia-nyiakannya.

1.14 Soal Juri sebagai Penelitian: Dua Makalah dari ITS

Mudah sekali menganggap soal *online judge* sekadar teka-teki. Ia bukan itu. Departemen Teknik Informatika ITS telah menerbitkan penelitian bereputasi yang landasan percobaannya justru sebuah *online judge*, dan dua contohnya pantas dilihat sekarang, di pekan pertama Anda, sebab keduanya memperlihatkan ke mana bidang ini bermuara.

1.14.1 Mengekstrapolasi fungsi periodik

PERIOD5 — Periodic function, trip 5 (SPOJ)

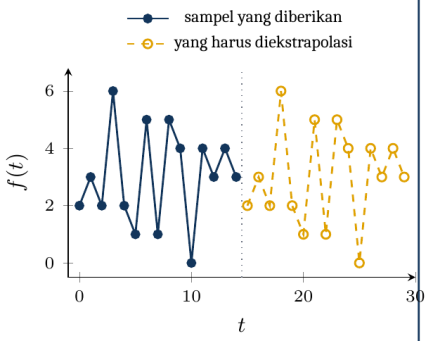
Kategori classical Batas waktu 1–7.5 dtk Batas kode sumber 50 000 B Batas memori 1536 MB

Tugas (ringkasan)

Sebuah *fungsi periodik bersih* adalah jumlah komponen-komponen periodik yang periodenya bilangan bulat. Anda diberi nilai-nilai awal fungsi semacam itu dan harus mengekstrapolasinya: meramalkan nilai-nilai berikutnya.

Motivasi yang dikutip di halaman soal

Halaman soal mengutip NOAA: prakiraan siklus matahari dipakai oleh berbagai lembaga dan industri, karena hambatan atmosfer pada satelit di orbit rendah Bumi berkorelasi dengan siklus itu. Halaman itu menampilkan catatan bintik matahari selama berabad-abad.



Gambar 1.16 Soal SPOJ PERIOD5, *Periodic function, trip 5*: diberi cuplikan sebuah fungsi periodik yang bersih, ekstrapolasikan fungsi itu. Grafik kecilnya memperlihatkan fungsi semacam itu, tersusun dari komponen berperiode 3 dan 5.

Gambar 1.16 menampilkan soal SPOJ PERIOD5. Sebuah fungsi dibangun sebagai jumlah komponen-komponen periodik yang periodenya bilangan bulat; Anda diberi sebagiannya dan harus meramalkan sisanya. Penerapan yang dikutip pada halaman soalnya adalah daur matahari, yang penting karena hambatan satelit di orbit Bumi rendah mengikuti aktivitas bintik matahari.

Soelaiman dkk. [9] menggarapnya dengan sistem persamaan linear, membandingkan perumusan yang belum dioptimalkan pada matriks $N^2 \times N^2$ dengan perumusan yang sudah dioptimalkan pada matriks $N \times N$. **Gambar 1.17** meringkas makalah yang terbit itu.

Journal of Theoretical and Applied Information Technology, vol. 97, no. 8, pp. 2225–2239, 30 April 2019

Efficient Approach for Deterministic Data Extrapolation from a Clean Periodic Function with Periodic Components Representation by a System of Linear Equations

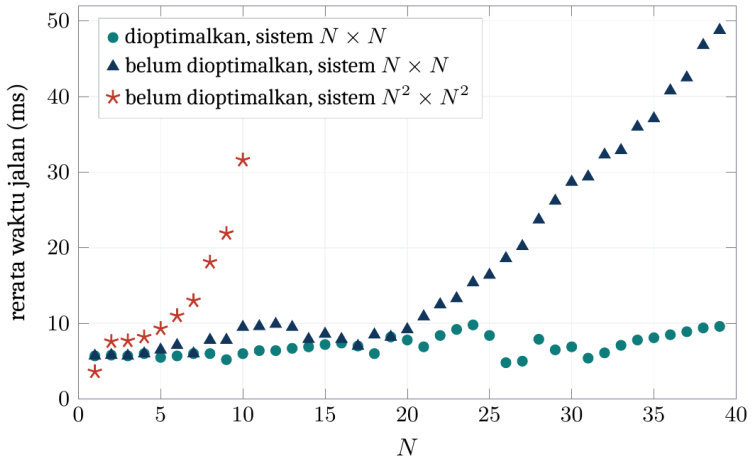
R. Soelaiman, S. Candra, M. M. I. Subakti

Departemen Informatika, Institut Teknologi Sepuluh Nopember (ITS), Surabaya, Indonesia

Ringkasnya

Makalah ini membandingkan cara yang belum dioptimalkan dan yang sudah dioptimalkan untuk mengekstrapolasi fungsi periodik bersih yang komponennya berperiode bilangan bulat. Keduanya mengenali komponen dengan menyelesaikan sistem persamaan linear; formulasi yang dioptimalkan bekerja pada sistem $N \times N$, bukan $N^2 \times N^2$, sehingga lebih cepat, memakai lebih sedikit memori, dan menjangkau rentang periode yang lebih luas. Kebenarannya diuji dengan pengiriman ke Sphere Online Judge, kinerjanya dengan mengukur waktu jalan.

Gambar 1.17 Sekilas makalah [9]: judul, nama penyusun, wadah terbit (*Journal of Theoretical and Applied Information Technology*, vol. 97, no. 8, 2019), dan sumbangannya.



Gambar 1.18 Waktu jalan rata-rata tiga pendekatan ketika N membesar, digambar ulang dari hasil pengukuran pada Gambar 13 di [9]. Metode $N \times N$ yang dioptimalkan tetap mendatar; yang lain menanjak.

Bacalah **Gambar 1.18** dengan mata Subbab 1.3.4. Deret bertanda bintang, yaitu metode $N^2 \times N^2$ yang belum dioptimalkan, sudah keluar dari puncak grafik sebelum $N = 11$. Deret bertanda segitiga, metode $N \times N$ yang belum dioptimalkan, baik-baik saja sampai sekitar $N = 20$ lalu menanjak tajam. Deret bertanda lingkaran, metode yang sudah dioptimalkan, hampir merupakan garis mendatar sampai $N = 39$. Tiga algoritma yang sama-sama benar; satu di antaranya yang dapat dipakai.

PERIOD5: statistik dan penyelesaian terbaik

Pengguna diterima	Pengiriman	Diterima	Jawaban salah	Galat kompilasi	Galat saat jalan	Melewati batas waktu
12	424	235	84	9	57	39

Peringkat	Tanggal	Pengguna	Hasil	Waktu (dtk)	Memori	Bahasa
1	2018-09-01	Steven Candra	accepted	0.59	17M	CPP14
2	2020-01-26	Rully Soelaiman	accepted	0.59	4.9M	CPP
3	2017-05-17	Mateusz Radecki	accepted	0.60	94M	CPP14
4	2017-05-13	Kamil Debowski	accepted	0.72	17M	CPP14
5	2017-05-23	i1e2s	accepted	0.73	17M	CPP14
6	2017-05-24	Balsa	accepted	0.76	88M	CPP14
7	2017-09-23	[Rampage] Blue.Mary	accepted	0.79	3.8M	C++ 4.3.2
8	2017-09-11	[Rampage] Blue.Mary	accepted	0.81	17M	CPP
9	2017-05-04	Domagoj Bradac	accepted	0.82	20M	CPP14

Sebagaimana tercantum pada halaman peringkat SPOJ awal 2026; baris yang disorot adalah entri ITS.

Gambar 1.19 Peringkat SPOJ untuk PERIOD5 sebagaimana tercantum pada awal 2026. Dari empat ratus dua puluh empat pengiriman hanya dua ratus tiga puluh lima yang diterima, dan entri ITS duduk di peringkat 2 dengan waktu 0,59 detik dalam 4,9 MB.

Gambar 1.19 memperlihatkan bagaimana makalah itu mengukur dirinya sendiri: bukan dengan klaim, melainkan dengan peringkat terbuka pada sebuah juri yang dapat diperiksa siapa saja. Perhatikan kolom memorinya — penyelesaian ITS memakai 4,9 MB sementara entri di atasnya memakai 17 MB. Itulah perumusan matriks yang dioptimalkan, menampakkan diri sebagai angka yang dapat diukur.

1.14.2 Membaca angka tulisan tangan

HIR — Hard Image Recognition (SPOJ)

Kategori challenge Batas waktu 0.1 dtk Batas kode sumber 150 000 B Batas memori 1536 MB

Pernyataan

Tulislah program yang mengenali citra bilangan enam digit. Adakalanya citra itu kabur atau terdistorsi, tetapi manusia selalu dapat membaca bilangan yang tertulis. Gambarnya hitam-putih: latar putih dengan citra raster hitam di atasnya; karakter '.' menyatakan putih dan 'X' hitam. (Halaman soal berkisah tentang Byteland, yang nomor paspornya ditulis dengan segala jenis huruf yang ditawarkan pengolah kata.)

Masukan

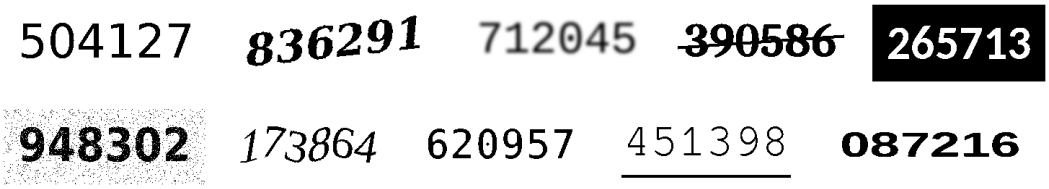
t, banyaknya kasus uji ($t \leq 250$), lalu t uji. Setiap uji diawali dua bilangan bulat H dan W, tinggi dan lebar gambar ($10 \leq H, W \leq 200$), diikuti H baris masing-masing berisi W karakter ('.' atau 'X').

Keluaran

Untuk setiap kasus uji, keluarkan bilangan enam digit yang dikenali pada baris tersendiri. Ini soal *challenge*: skornya adalah banyaknya citra yang terbaca dengan benar.

Gambar 1.20 Soal SPOJ HIR, *Hard Image Recognition*: pulihkan bilangan enam angka dari sebuah citra raster hitam-putih, dengan batas 0,1 detik dan batas kode sumber 150 kB.

Gambar 1.20 menampilkan persoalan yang sangat berbeda. Anda menerima sampai dua ratus lima puluh gambar, masing-masing selebar sampai 200 karakter dan digambar dengan `.` untuk putih dan `X` untuk hitam, lalu Anda harus membaca bilangan enam angka yang tertulis di dalam masing-masing. **Gambar 1.21** memperlihatkan betapa beragam fonnya.



Gambar 1.21 Sepuluh bilangan enam angka yang digambar dengan fon berbeda-beda, sebagian miring, buram, tercoret, berbintik, atau terbalik warnanya — keragaman seperti yang tampak pada gambar contoh soal HIR.

HIR — uji contoh

Masukan

```
2
13 40
.....................................
.....................................
.....................................
...XXX...XXX...XXX.....XX..XXXX..XXXX...
..XX.XX.XX.XX.XX.XX...XXX..XX.XX.X..XX..
..X..XX.XX..X.XX..XX.XXXX..XXXX.....XX..
..X..XX.XX..X.XX..XX.X.XX..XXXX....XX...
..X..XX.XX.XX.XX..X.XXXXXX.X..XX..XX....
..XXXXX.XX.XX.XXXXX....XX..XX.XX.XXXXX..
...XXX...XXX...XXX.....XX...XXX..XXXXX..
.....................................
.....................................
.....................................
12 60
............................................................
............................................................
............XXXX...XXX..XXXXX.XXXX..XXXX..XXXX............
............X.....XX....X..X..X..X....XX....XX............
............X....XX........X..X..X....XX....XX............
............XXXX.XXXXX....XX..XXXX...XX....XX............
...............X.XX..XX...X...X..X....XX....XX............
...............XXXX..XX..XX..XX..XX....XX....XX............
...........X...X..X..XX..X...XX..XXX...XXX...XX............
............XXXX...XXX..XX....XXXX..XXXX..XXXX............
............................................................
............................................................
```

Keluaran

```
000482
567833
```

Skor

Kiriman yang mencetak `000482` dan `567843` mendapat skor 1: jawaban pertama benar, jawaban kedua salah satu digit.

Gambar 1.22 Contoh uji HIR. Kedua gambarnya menyandikan 000482 dan 567833; pengiriman yang ditampilkan memperoleh nilai 1 karena jawaban keduanya punya satu angka yang keliru.

Gambar 1.22 memperlihatkan aturan penilaiannya, yang tidak lazim: ini soal *challenge*, jadi Anda bukan sekadar diterima atau ditolak melainkan diberi nilai, satu angka untuk setiap gambar yang seluruhnya betul. Soelaiman dkk. [10] memakai algoritma genetika untuk merancang arsitektur sebuah jaringan saraf tiruan bagi tugas ini — artinya, sebuah algoritma yang mencari algoritma yang baik — tunduk pada batasan yang kejam berupa batas waktu 0,1 detik dan batas sumber 150 kilobita, yang menutup kemungkinan mengirimkan model besar yang sudah terlatih. **Gambar 1.23** meringkas makalahnya dan **Gambar 1.24** memperlihatkan hasilnya.

Journal of Theoretical and Applied Information Technology, vol. 98, no. 17, pp. 3479–3492, 15 September 2020

Genetic Algorithm Approach for Automated Generation of Neural Networks Architecture for Robust Digit Recognition

R. Soelaiman, Y. H. Pratama, M. M. I. Subakti, Y. Purwananto
Departemen Informatika, Institut Teknologi Sepuluh Nopember (ITS), Surabaya, Indonesia

Ringkasnya

Makalah ini merancang penyelesaian soal SPOJ *Hard Image Recognition* (HIR): membaca bilangan enam digit dari citra raster yang berderau, dengan batas waktu yang sangat singkat dan batas kode sumber yang kecil. Alih-alih memilih arsitektur jaringan saraf tiruan dengan coba-coba, sebuah algoritma genetika mencarinya; bersama set data latih yang dibuat khusus, hasilnya memenuhi batasan dan meraih skor 108 pada SPOJ HIR.

Gambar 1.23 Sekilas makalah [10]: judul, nama penyusun, wadah terbit (*Journal of Theoretical and Applied Information Technology*, vol. 98, no. 17, 2020), dan sumbangannya.

HIR: statistik dan penyelesaian terbaik

Pengguna diterima	Pengiriman	Diterima	Jawaban salah	Galat kompilasi	Galat saat jalan	Melewati batas waktu
22	660	445	0	35	117	63

Peringkat	Tanggal	Pengguna	Hasil	Waktu (dtk)	Memori	Bahasa
1	2020-01-09	Yolanda Hertita	**108**	0.12	5.1M	CPP14
2	2020-09-16	Rully Soelaiman	**108**	0.15	5.0M	CPP
3	2018-12-25	CynthiaDewi	**32**	0.07	17M	CPP14
4	2025-05-22	dasd	**9**	0.05	5.2M	CPP14
5	2010-08-24	.::Manish Kumar::.	**2**	0.00	2.2M	C
6	2011-01-21	Satyarth Kumar Prasad	**2**	0.00	2.6M	C++ 4.3.2
7	2025-05-27	邓紫棋	**2**	0.04	5.2M	CPP14

Sebagaimana tercantum pada halaman peringkat SPOJ awal 2026; baris-baris yang disorot adalah entri ITS.

Gambar 1.24 Peringkat SPOJ untuk HIR sebagaimana tercantum pada awal 2026. Dua puluh dua pengguna diterima dari enam ratus enam puluh pengiriman; dua entri teratas, dengan nilai 108, berasal dari kelompok ITS.

Pelajaran dari **Gambar 1.24** bukanlah bahwa jaringan saraf tiruan itu mengasyikkan. Pelajarannya adalah bahwa batasan dalam rumusan persoalan — 0,1 detik, 150 kB — mendikte seluruh rancangannya, persis sebagaimana dikatakan **Tabel 1.3**. Fase 1 menentukan hasil sebuah makalah penelitian.

1.15 Buku Acuan, Peta Jalan, dan Bab-Bab Evaluasi

1.15.1 Buku acuan

Dua keluarga buku berdiri di balik buku ini, dan **Tabel 1.10** mencantumkannya bersama edisi yang berlaku pada September 2026.

Tabel 1.10 Dua keluarga buku di balik buku ini, beserta edisi yang berlaku pada September 2026.

Buku	Disusun oleh	Edisi	Perannya dalam buku ini
Introduction to Algorithms (CLRS) [11]	Cormen, Leiserson, Rivest, Stein	Edisi ke-4, MIT Press, April 2022 — edisi terkini [12]	Acuan utama: pembuktian dan analisis
Introduction to Algorithms (CLRS)	Cormen, Leiserson, Rivest, Stein	Edisi ke-3, MIT Press, 2009	Masih banyak di perpustakaan perguruan tinggi; nomor babnya berbeda
Competitive Programming 4 (CP4) [6]	Halim, Halim, Effendy	Buku 1 (bab 1–4) dan Buku 2 (bab 5–9), Juli 2020 — edisi terkini [13]	Pendamping praktis: teknik dan soal juri
Competitive Programming 3 (CP3)	Halim, Halim	Edisi ke-3, 2013	Edisi yang pertama kali ditemui banyak mahasiswa

CLRS — Cormen, Leiserson, Rivest, dan Stein — adalah acuan baku untuk perancangan dan analisis algoritma di seluruh dunia. Buku ini mengikuti edisi keempat [11], yang terbit pada April 2022 dan pada September 2026 masih merupakan edisi terbaru [12]; kalau Anda hanya punya akses ke edisi ketiga, penomoran babnya berbeda tetapi hampir seluruh bahannya sama. Edisi keempatlah yang sebaiknya dibeli kalau Anda memang hendak membeli.

CP4 — Halim, Halim, dan Effendy [6], yang terbit dalam dua jilid pada Juli 2020, dengan *Competitive Programming 3* (2013) sebagai pendahulunya — adalah pendamping praktisnya. Tim penyusun CP4 hanya menerbitkan edisi baru bila perubahannya memadai, dan sampai September 2026 edisi kelima belum terbit [13]. Kalau CLRS membuktikan, CP4 melatih: ia mengatalogkan teknik dan menunjuk Anda ke soal-soal juri yang melatih masing-masing teknik itu. Kedua buku itu saling melengkapi, dan buku ini bersandar pada keduanya. Bacalah CLRS untuk memahami mengapa sebuah algoritma bekerja; bacalah CP4 untuk menjadi lebih cekatan mengenali algoritma mana yang harus diraih.

1.15.2 Peta jalan

Tabel 1.11 Enam belas bab buku ini, beserta bab-bab [11] yang bersesuaian.

Bab	Topik	CLRS
1	Fondasi, kerangka kerja, dan persoalan pertama	Bab 1
2	Langkah awal: model RAM, insertion sort, pencarian	Bab 2
3	Kebenaran, analisis asimtotik, Big-O, menyelesaikan rekurensi	Bab 2–4
4	**Evaluasi dan Pembahasan I** (Bab 1–3)	—
5	Pengurutan: heapsort dan quicksort	Bab 6–7
6	Pohon pencarian biner dan pohon AVL	Bab 13
7	*Dynamic programming*	Bab 14
8	**Evaluasi dan Pembahasan II** (Bab 5–7)	—
9	Algoritma *greedy*	Bab 15
10	Algoritma graf dasar	Bab 20
11	Pohon merentang minimum	Bab 21
12	**Evaluasi dan Pembahasan III** (Bab 9–11)	—
13	Lintasan terpendek sumber tunggal	Bab 22
14	Analisis amortisasi	Bab 16
15	Lintasan terpendek semua pasangan	Bab 23
16	**Evaluasi dan Pembahasan IV** (Bab 13–15 dan seluruh buku)	—

Tabel 1.11 pantas ditelaah sejenak, sebab ia memperlihatkan bentuk buku ini secara utuh. Tiga bab pertama membangun perkakasnya — kerangka kerja, model komputasi, analisis asimtotik. Setiap bab keempat berhenti sejenak untuk mengevaluasi dan membahas apa yang datang sebelumnya. Selebihnya adalah satu keluarga algoritma tertentu, dan setiap keluarga adalah salah satu dari lima paradigma **Tabel 1.4** yang dikhususkan untuk sekelas persoalan. Tidak ada satu pun di bab-bab belakangan yang terputus dari apa yang Anda baca hari ini.

1.15.3 Bab-bab evaluasi

Empat bab dalam buku ini tidak membawa topik baru. Sebaliknya, setiap bab itu mengajukan soal-soal tentang tiga bab yang baru saja selesai, lalu mengerjakan setiap soal secara tuntas, sehingga Anda dapat menguji diri lebih dulu dan memeriksa penalaran Anda sesudahnya.

Tabel 1.12 Keempat bab evaluasi dan cakupan masing-masing.

Bab	Judul	Cakupannya
4	Evaluasi dan Pembahasan I	Bab 1–3: kerangka kerja, kebenaran, asimtotik, rekurensi
8	Evaluasi dan Pembahasan II	Bab 5–7: pengurutan, pohon pencarian, *dynamic programming*
12	Evaluasi dan Pembahasan III	Bab 9–11: algoritma *greedy*, graf, pohon merentang
16	Evaluasi dan Pembahasan IV	Bab 13–15, dan seluruh buku

Setiap bab dalam **Tabel 1.12** meminta Anda *merancang dan menganalisis*, bukan mengingat-ingat. Soal yang khas di sana menyodorkan situasi yang belum pernah Anda lihat lalu meminta kelima fase: apa masukan dan keluarannya, apa inti matematisnya, apa algoritmanya, mengapa ia benar, berapa kompleksitasnya. Setiap soal kemudian diselesaikan secara tuntas, setiap langkahnya ditunjukkan.

1.16 Menyiapkan Meja Kerja: VS Code dan Kompilator C++

Setiap program dalam buku ini dimaksudkan untuk Anda ketik, kompilasi, dan jalankan sendiri. Sebelum bab berikutnya meminta Anda melakukannya, mari kita bangun meja kerjanya sekali saja, dengan cermat, di komputer apa pun yang Anda miliki. Anda hanya memerlukan dua hal:

1. **Kompilator** (*compiler*) — program yang mengubah kode C++ yang Anda tulis menjadi program yang dapat dijalankan komputer. Kita memakai **GCC** (perintah `g++`) di Windows dan Linux, serta **Clang** (perintah `clang++`) di macOS. Keduanya memahami C++17 standar, dan hanya itulah yang dipakai buku ini. Rilis terbaru mana pun memadai; pada September 2026 yang terbaru adalah GCC 16.2 (Agustus 2026) [14] dan Clang 22.1 (Juni 2026) [15]. Rilis GCC terkini mengompilasi sebagai C++20 bila tidak diberi tahu lain [14]; itulah sebabnya setiap perintah dalam buku ini menyertakan `-std=c++17` secara tegas.
2. **Editor** — tempat Anda menulis. Kita memakai **Visual Studio Code** (VS Code), editor gratis buatan Microsoft yang berjalan di Windows, macOS, dan Linux, dengan tampilan yang sama di ketiganya.

Kompilatorlah yang bekerja keras; editor membuat pekerjaan itu nyaman. Pasang kompilator lebih dulu, lalu editornya, lalu hubungkan keduanya. **Tabel 1.13** merangkum seluruh prosedur untuk ketiga sistem operasi dalam satu halaman, agar Anda dapat kembali kepadanya kapan saja.

Tabel 1.13 Meja kerja di tiga sistem operasi, dari komputer kosong sampai program yang berjalan. Distribusi Linux lain memakai `sudo dnf install gcc-c++ gdb` (Fedora) atau `sudo pacman -S gcc gdb` (Arch) pada langkah 1.

Langkah	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
1. Pasang kompilator	Pasang MSYS2 dari msys2.org, buka terminal **MSYS2 UCRT64**, jalankan `pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain`	Buka **Terminal**, jalankan `xcode-select --install`	Buka terminal, jalankan `sudo apt update && sudo apt install build-essential gdb`
2. Agar dapat ditemukan	Tambahkan `C:\msys64\ucrt64\bin` ke variabel `Path` pengguna, lalu buka terminal baru	Tidak perlu apa-apa	Tidak perlu apa-apa
3. Periksa	`g++ --version`	`clang++ --version`	`g++ --version`
4. Pasang VS Code	Penginstal dari code.visualstudio.com; centang "Add to PATH"	Unduh, seret *Visual Studio Code* ke *Applications*	`sudo snap install code --classic`, atau berkas `.deb` dari situsnya
5. Tambahkan dukungan C++	Tampilan Extensions (`Ctrl+Shift+X`): pasang **C/C++ Extension Pack** (Microsoft)	Sama, dengan `Cmd+Shift+X`	Sama, dengan `Ctrl+Shift+X`
6. Kompilasi berkas	`g++ -std=c++17 -O2 -Wall -o hello hello.cpp`	`clang++ -std=c++17 -O2 -Wall -o hello hello.cpp`	`g++ -std=c++17 -O2 -Wall -o hello hello.cpp`
7. Jalankan	`.\hello.exe`	`./hello`	`./hello`

1.16.1 Windows, langkah demi langkah

Windows tidak dilengkapi kompilator C++, jadi kita meminjamnya dari **MSYS2**, kumpulan kecil yang terawat baik berisi perkakas yang sama dengan yang dipakai para pemrogram Linux.

1. Buka peramban, kunjungi **msys2.org**, unduh penginstalnya, lalu jalankan. Terima folder usulan `C:\msys64`. Setelah selesai, penginstal menawarkan untuk membuka terminal; biarkan.
2. Dari menu Start buka **MSYS2 UCRT64** (namanya penting: UCRT64 adalah varian modern). Ketik baris berikut lalu tekan Enter: `pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain`. Ketika ditanya anggota mana yang dipasang, tekan Enter untuk menerima semuanya, lalu jawab `Y` untuk melanjutkan. Pengunduhan memakan waktu beberapa menit.
3. Beri tahu Windows di mana kompilator berada. Tekan tombol Windows, ketik *environment variables*, pilih **Edit environment variables for your account**, pilih **Path**, klik **Edit → New**, lalu ketik `C:\msys64\ucrt64\bin`. Klik **OK** di setiap jendela.
4. Buka jendela Command Prompt atau PowerShell yang *baru* (jendela lama tidak melihat Path yang baru) dan ketik `g++ --version`. Jika muncul nomor versi, kompilator sudah terpasang. Jika Windows mengatakan tidak mengenali `g++`, langkah 3 belum tersimpan; ulangi.

1.16.2 macOS, langkah demi langkah

macOS sudah tahu dari mana mendapatkan kompilator; ia hanya perlu izin.

1. Buka **Terminal** (tekan `Cmd+Space`, ketik *Terminal*, tekan Enter).
2. Ketik `xcode-select --install` lalu tekan Enter. Sebuah jendela muncul dan menanyakan apakah *command line developer tools* akan dipasang; klik **Install** dan tunggu.

3. Ketik `clang++ --version`. Baris versi berarti Anda siap. Di Mac, perintah `g++` juga bisa dipakai, tetapi diam-diam ia menjalankan Clang; semua program dalam buku ini dapat dikompilasi dengan keduanya.

1.16.3 Linux, langkah demi langkah

Sebagian besar sistem Linux memasang kompilator dengan satu perintah.

1. Buka terminal (`Ctrl+Alt+T` di Ubuntu).
2. Ketik `sudo apt update && sudo apt install build-essential gdb`, tekan Enter, ketik kata sandi Anda, lalu konfirmasi dengan `Y`.
3. Ketik `g++ --version` untuk memeriksa.

1.16.4 VS Code di setiap sistem

1. Unduh VS Code dari **code.visualstudio.com** dan pasang seperti yang dijelaskan **Tabel 1.13**. Di Windows, centang **Add to PATH** saat pemasangan, agar nanti Anda dapat mengetik `code .` di terminal untuk membuka sebuah folder.
2. Jalankan VS Code, buka tampilan **Extensions** dengan `Ctrl+Shift+X` (`Cmd+Shift+X` di Mac), cari **C/C++ Extension Pack**, lalu pasang yang diterbitkan oleh Microsoft. Ekstensi ini menambahkan pewarnaan sintaksis, garis bawah galat, pelengkapan kode, dan *debugger*.
3. Buat satu folder untuk buku ini, misalnya `DAA` di folder Documents, lalu buka dengan **File → Open Folder**. Simpan satu subfolder per bab — `Chapter01`, `Chapter02`, dan seterusnya — persis seperti susunan folder `Code` yang menyertai buku ini.

1.16.5 Program pertama Anda, dikompilasi dan dijalankan

Buat berkas baru bernama `hello.cpp` di folder Anda, ketikkan **Kode 1.16**, lalu simpan.

Kode 1.16 Program pertama: bukti bahwa kompilator, editor, dan terminal sudah bekerja bersama.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string name;
    cout << "What is your name? ";
    if (!(cin >> name)) name = "friend";      // tidak ada masukan? sapa seorang teman
    cout << "Hello, " << name << "! Your C++17 compiler works." << endl;
    cout << "__cplusplus = " << __cplusplus << endl;
    return 0;
}
```

Sekarang buka terminal di dalam VS Code melalui **Terminal → New Terminal** (atau tekan `Ctrl` bersama tombol *backtick*) dan ketik perintah kompilasi dari baris 6 **Tabel 1.13**:

```
g++ -std=c++17 -O2 -Wall -o hello hello.cpp
```

Setiap bagian baris itu punya tugas. `-std=c++17` memilih standar bahasa C++17 yang dipakai di seluruh buku ini; `-O2` meminta kompilator melakukan optimasi, agar percobaan pengukuran waktu mengukur algoritmanya, bukan terjemahan yang malas; `-Wall` menyalakan peringatan, yang menangkap banyak kekeliruan sebelum menjadi *bug*; `-o hello` memberi nama program hasilnya. Jika kompilator tidak mencetak apa pun, berarti berhasil — diam adalah kabar baik. Jalankan programnya (`.\hello.exe` di Windows, `./hello` di sistem lain) lalu ketik sebuah nama:

```
What is your name? Irfan
Hello, Irfan! Your C++17 compiler works.
__cplusplus = 201703
```

Baris terakhir adalah bukti kecil: `201703` adalah nilai yang diberikan standar C++ kepada makro `__cplusplus` pada C++17, sehingga kini Anda tahu pasti versi bahasa yang dipakai untuk mengompilasi program Anda.

1.16.6 Memberi masukan pada soal *online judge*

Online judge memberikan masukan kepada program melalui *standard input*. Ketika contoh masukan sebuah soal panjang, jangan mengetiknya dengan tangan setiap kali. Simpan di sebuah berkas, misalnya `input.txt`, di samping program, lalu **alihkan** (*redirect*):

```
./columns < input.txt            (macOS, Linux, Command Prompt Windows)
Get-Content input.txt | .\columns.exe      (Windows PowerShell)
```

Program membaca berkas itu persis seolah-olah Anda mengetiknya, dan memang begitulah yang akan dilakukan *judge*. Membandingkan keluaran Anda dengan contoh keluaran pun menjadi urusan melihat dua blok teks pendek.

1.16.7 *Debugging*: menyaksikan program berpikir

Ketika sebuah program memberi jawaban yang salah, cara tercepat untuk mencari tahu sebabnya adalah *menyaksikannya*. Klik di sebelah kiri nomor baris di VS Code: muncul titik merah — sebuah **breakpoint**. Tekan `F5` dan pilih **C++ (GDB/LLDB)** serta kompilator Anda; VS Code membuat berkas konfigurasi kecil yang diperlukannya (`tasks.json` dan `launch.json` di dalam folder `.vscode`) lalu menjalankan program. Program berhenti di titik merah, dan panel **Variables** di sebelah kiri menampilkan nilai setiap variabel pada saat itu. Tekan `F10` untuk menjalankan satu baris, `F11` untuk masuk ke dalam sebuah fungsi, `F5` untuk melanjutkan. Menyaksikan invarian gelung di Bab 2 tetap berlaku, iterasi demi iterasi, di panel Variables adalah salah satu cara terbaik untuk memahaminya.

> **BILA ADA YANG SALAH.** *"g++ is not recognized"* (Windows): entri Path pada langkah 3 belum ada atau terminal dibuka sebelum Anda menambahkannya — buka terminal baru. *"fatal error: iostream: No such file"*: kompilator dipasang dari terminal MSYS2 yang keliru; ulangi langkah 2 di **MSYS2 UCRT64**. *Jendela program berkedip lalu tertutup*: jalankan dari terminal, jangan dengan klik ganda. *Garis bawah galat di mana-mana di VS Code padahal program dapat dikompilasi*: buka Command Palette (`Ctrl+Shift+P`), jalankan **C/C++: Select IntelliSense Configuration**, lalu pilih kompilator yang Anda pasang.

Dengan meja kerja yang sudah berdiri, setiap kode dalam sisa buku ini hanya berjarak satu salin, satu kompilasi, dan satu kali jalan untuk menjadi milik Anda.

1.17 Profitina dalam Praktik

Profitina adalah **platform visibilitas AI (GEO)**, yang beroperasi di `app.profitina.com`. GEO — *Generative Engine Optimization* — adalah disiplin untuk menjadi *layak dikutip oleh mesin penjawab*, sebagaimana SEO adalah disiplin untuk menduduki peringkat dalam daftar tautan. Produk ini memindai situs web sebuah merek, menilai seberapa siap isinya untuk dikutip mesin penjawab AI, menjelaskan setiap sinyal yang diukurnya, lalu membantu membereskan apa yang ditemukannya.

Setiap gagasan dalam bab ini muncul di dalam produk itu dalam pekan pertama pekerjaan rekayasanya.

Batasan menentukan rancangan, sebelum ada kode apa pun. Satu pemindaian menyentuh setiap halaman sebuah situs. Situs merek yang sederhana punya beberapa ratus halaman; sebuah lokapasar punya ratusan ribu. Sebut banyaknya halaman P. Memeriksa apakah isi utama satu halaman dapat diekstraksi bersifat linear terhadap panjang halaman itu, jadi tahap ekstraksi atas seluruh situs adalah $O(\text{total bita})$ — tidak terhindarkan dan itu wajar. Tetapi pemeriksaan naif untuk mencari *halaman yang nyaris kembar*, dengan membandingkan setiap halaman dengan setiap halaman lainnya, adalah $\Theta(P^2)$. Pada $P = 50{,}000$ itu 2.5×10^9 perbandingan, yang menurut **Tabel 1.3** melampaui anggaran satu orde besaran bahkan sebelum kita memperhitungkan biaya membandingkan dua dokumen. Pembetulannya persis langkah Subbab 1.4.5: ganti pencarian dengan hitungan. Cincangkan isi ternormalisasi tiap halaman menjadi sebuah sidik jari lalu kelompokkan berdasarkan sidik jari itu, dan tahapnya menjadi $O(P)$. Jawaban yang sama, kelas asimtotik yang berbeda.

Agregat berjalan biasanya sudah cukup. Dasbor analitik historis menampilkan tren nilai mingguan per merek. Implementasi naifnya menghitung ulang agregat tiap pekan dari seluruh riwayat; implementasi inkremental dari Subbab 1.4.1 memelihara sebuah total berjalan dan memperbaruinya setiap kali pemindaian baru mendarat. Pada tabel yang bertumbuh setiap pekan, yang pertama adalah $\Theta(\text{pekan}^2)$ sepanjang umur akun itu dan yang kedua $\Theta(\text{pekan})$.

Penjadwalan adalah persoalan antrean prioritas. Paket berbayar dipindai ulang dengan irama lebih cepat daripada paket gratis. “Situs mana yang saya pindai berikutnya?” persis sama dengan “berikan saya elemen dengan waktu jatuh tempo terkecil”, dan itulah satu operasi yang menjadi alasan *heap* dibangun. Bab 5 mengembangkan *heap* dengan semestinya; yang penting di sini adalah bahwa pilihan struktur data, bukan pilihan algoritma, itulah yang membuat penjadwalnya murah.

Kebenaran juga punya makna hukum. Platform ini tunduk pada Undang-Undang Pelindungan Data Pribadi Republik Indonesia, UU PDP 27/2022 [16]. Persetujuan harus tegas, data harus dapat diekspor, dan penghapusan akun harus benar-benar menghapus. “Benar untuk setiap masukan yang sah” di Subbab 1.3.1 mencakup pula masukan berupa permintaan seorang pengguna untuk dihapus. Dan kaidah keempat perancangan produk Profitina berlaku bagi algoritma apa pun yang Anda tulis di sini: **aplikasi menyarankan, pengguna memutuskan.** Sebuah nilai adalah hasil pengukuran, tidak pernah sebuah jaminan.

1.17.1 Studi kasus yang dikerjakan: Profitina Laundry

Studi kasus praktik dalam buku ini selalu **Profitina Laundry**, sebuah usaha kecil untuk pengajaran, tidak pernah produk yang sesungguhnya. Inilah persoalan bab 1-nya, dan yang menyenangkan adalah bahwa ia dua di antara enam persoalan kita yang sedang memakai celemek.

Profitina Laundry punya n titik antar di sepanjang satu ruas jalan, Jalan Kertajaya, masing-masing pada penanda meter yang diketahui. Sebuah mobil berangkat dari toko, harus singgah di setiap titik antar, dan harus kembali ke tempatnya bertolak. Itulah *Journey*: rute sependek mungkin adalah $2 \times (\text{terjauh} - \text{terdekat})$, dan tokonya boleh berada di mana saja di sepanjang jalan itu tanpa mengubah jawabannya.

Kembali di toko, cucian bersih ditumpuk pada rak pengering: satu kolom untuk satu titik antar, dan setiap kolom berisi sebanyak keranjang yang dikirim titik itu. Dua keranjang dalam kolom

yang sama harus membawa label yang berbeda, supaya pekerja yang merogoh sebuah tumpukan tidak keliru; dan dua keranjang yang bersebelahan pada rak yang sama harus membawa label yang berbeda, kecuali ada sela yang memisahkannya. Itulah *Columns*: banyaknya label yang diperlukan adalah max(n,tumpukan tertinggi).

Kode 1.17 menyelesaikan keduanya sekaligus.

Kode 1.17 Perencana Profitina Laundry: perjalanan pulang-pergi terpendek bagi mobil antar dan banyaknya label rak yang diperlukan.

```
#include <cstdio>
#include <algorithm>
using namespace std;

int main() {
    int n;
    scanf("%d", &n);                        // banyaknya titik antar di Jalan Kertajaya

    int x, lo = 1000000000, hi = -1000000000;
    for (int i = 0; i < n; i++) {           // penanda meter tiap titik antar
        scanf("%d", &x);
        lo = min(lo, x);
        hi = max(hi, x);
    }

    int h, labels = n;                      // rak terbawah saja sudah perlu n label
    for (int i = 0; i < n; i++) {           // berapa keranjang dikirim balik tiap titik
        scanf("%d", &h);
        labels = max(labels, h);            // tumpukan setinggi itu perlu label sebanyak
itu
    }

    printf("van route            : %d meters\n", 2 * (hi - lo));
    printf("distinct rack labels : %d\n", labels);
    return 0;
}
```

Diberi masukan

```
5
120 340 75 980 410
3 7 2 9 4
```

— lima titik antar pada penanda meter 120,340,75,980,410, yang mengirim balik 3,7,2,9,4 keranjang — **Kode 1.17** sungguh-sungguh mencetak:

```
van route            : 1810 meters
distinct rack labels : 9
```

Periksa keduanya dengan tangan. Titik terdekat ada di 75 m dan terjauh di 980 m, jadi rentangnya 905 m dan perjalanan pulang-perginya $2 \times 905 = 1810$ m. Raknya punya 5 kolom dan tumpukan tertingginya 9 keranjang, jadi $\max(5,9) = 9$ label. Dua wawasan setingkat penelitian, diterapkan pada sebuah usaha binatu, dalam dua puluh baris.

1.18 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Menulis kode sebelum memahami persoalannya. Kesalahan tunggal yang paling mahal dalam bidang ini. Gejalanya: Anda punya dua ratus baris dan tidak tahu baris mana yang salah. Obatnya: tutup editornya, ambil selembar kertas, dan kerjakan Fase 1 dan 2.

Menguji alih-alih membuktikan. "Ia berhasil pada contoh ujinya" nyaris tidak memberi tahu Anda apa-apa, sebagaimana dibantah Subbab 1.3.1. Obatnya: untuk setiap gelung, nyatakan invariannya; untuk setiap pilihan *greedy*, buktikan pilihan itu aman; untuk setiap rekursi, pakai induksi.

Mengabaikan batasannya. Batasan adalah spesifikasi dari anggaran waktu Anda. Membaca $n \leq 10^{18}$ lalu menulis gelung atas n bukanlah kealpaan, melainkan penolakan mengerjakan Fase 4. Obatnya: sebelum merancang, masukkan masukan terbesar yang diizinkan ke dalam **Tabel 1.3** lalu putuskan kompleksitas apa yang boleh Anda pakai.

Mengoptimalkan hal yang keliru. Mengganti `cin` dengan `scanf` tidak akan menyelamatkan algoritma $O(n^2)$ pada $n = 10^6$. Faktor konstanta memindahkan Anda sejauh dua atau lima kali lipat; kelas asimtotik memindahkan Anda sejauh sejuta kali lipat. Obatnya: ketika Anda mendapat *Time Limit Exceeded*, kembalilah ke Fase 2, bukan ke badan gelungnya.

Melupakan kasus tepi yang diizinkan batasannya. Pada *Unique Numbers*, x boleh 0. Pada *Journey*, n boleh 1, dan dua kota boleh berbagi koordinat. Pada *Min/Max Numbers*, K boleh 1, yang merusak rumus umumnya dan menuntut cabangnya sendiri. Obatnya: bacalah baris batasannya, dan untuk setiap besaran tanyakan dengan tegas apa yang terjadi pada nilai sahnya yang terkecil dan yang terbesar.

Luapan bilangan bulat (*integer overflow*). Sebuah `int` di C++ hanya memuat nilai sampai kira-kira 2.1×10^9. *Interesting sequence* mengizinkan m sampai $2^{63} - 1$ dan *Recursive function 1* mengizinkan n sampai 10^{18}; keduanya memerlukan tipe 64 bit. Obatnya: pada Fase 4, hitunglah nilai terbesar yang dapat diambil setiap peubah, lalu pilih tipenya dari situ.

Memilih wadah yang keliru. Larik berindeks n tak tertandingi ketika n kecil dan mustahil ketika n bernilai 10^{18}. Obatnya: tanyakan ada berapa *keadaan berbeda* yang muncul, bukan seberapa besar indeksnya bisa jadi. Di Subbab 1.11 jawabannya 1,012.

Mengacaukan "terurut" dengan "benar". Contoh tandingan [1,2,3,4,5] dari Subbab 1.3.2 bukan lelucon; mahasiswa sungguh-sungguh menulis pemeriksa yang hanya menguji syarat (i). Obatnya: tuliskan kedua syaratnya sebelum menulis algoritmanya.

Salah membaca istilah yang didefinisikan. *Substring* pada *Columns* berarti deret tanpa sela, dan segala sesuatu bergantung padanya. Obatnya: pada Fase 1, nyatakan ulang setiap istilah yang didefinisikan persoalan itu dengan kata-kata Anda sendiri, lalu uji pernyataan ulang Anda terhadap contoh ujinya.

Memperlakukan juri sebagai kompilator. Dua puluh pengiriman dalam sejam berarti dua puluh tebakan. Obatnya: satu pengiriman semestinya mengikuti satu putaran lengkap kelima fase.

1.19 Ringkasan Bab

Sebuah **algoritma** adalah prosedur yang berhingga, pasti, dan terlaksana, yang membawa masukan terdefinisi menjadi keluaran terdefinisi; sebuah **program** adalah algoritma itu yang diungkapkan dalam bahasa yang dapat dijalankan mesin; sebuah **struktur data** adalah cara datanya ditata supaya langkah-langkahnya bisa murah. Algoritma adalah gagasannya, dan itulah bagian yang bertahan lama.

Sebuah algoritma harus sekaligus **benar** — betul untuk setiap masukan yang diizinkan batasannya, yang hanya dapat ditegakkan lewat pembuktian — dan **efisien**. Kebenaran dibuktikan dengan penalaran langsung, dengan kontradiksi, dengan **invarian gelung** (benar saat

masuk, terpelihara oleh setiap putaran, berguna saat keluar), atau dengan **induksi**. Insertion sort dibuktikan benar lewat invariannya di Subbab 1.3.3: di awal putaran j, prefiks $a[0 \ldots j-1]$ memuat penghuni asli posisi-posisi itu, dalam keadaan terurut.

Efisiensi diukur lewat laju pertumbuhan. Kaidah kerjanya adalah bahwa sebuah komputer mengerjakan sekitar 10^8 sampai 10^9 operasi sederhana per detik, sehingga **batasan dalam rumusan persoalan sudah memutuskan rancangannya sebelum satu baris kode pun ditulis**. Batas masukan 10^{18} tidak berarti "tulislah gelung yang cepat"; ia berarti "tidak ada gelung".

Ada lima paradigma perancangan: **inkremental / *brute force*, *greedy*, *divide and conquer*, *dynamic programming***, dan **matematis / teori bilangan**. Mengenali paradigma mana yang diinginkan sebuah persoalan adalah keterampilan intinya, dan **Tabel 1.8** adalah katalog awal gejalanya.

Setiap persoalan diselesaikan dalam lima fase: **Memahami, Mengabstraksi, Merancang, Menganalisis, Mengimplementasikan** — dalam urutan itu, dengan kode di urutan terakhir. Keenam persoalan bab ini masing-masing memakan satu halaman pemikiran dan kurang dari dua puluh baris C++: *Columns* adalah $\max(n, \max a_i)$; *Journey* adalah $2(x_{\max} - x_{\min})$; *Unique Numbers* membagi n yang berjalan dengan $\gcd(n, x)$ pada setiap perkalian; *Minimum and maximum numbers* menempatkan angka ekstrem secara rakus dari ujung yang paling berarti; *Interesting sequence* adalah $\text{popcount}(m-1) \bmod 3$; dan *Recursive function 1* memoisasikan sekitar seribu sub-persoalan dalam sebuah `map` walaupun n mencapai 10^{18}.

Fase abstraksi adalah tempat pekerjaan yang sesungguhnya terjadi. Langkah khasnya adalah membuktikan sebuah **batas bawah** lalu **membangun** sesuatu yang mencapainya; ketika kedua belahan itu berhasil, jawabannya terpaku persis, dan tidak ada lagi yang tersisa untuk diuji.

1.20 Soal Latihan

1.1 Jalankan penyelesaian *Columns* pada $n = 1$ dengan tinggi tunggal 1000. Apa yang dicetaknya, dan kendala yang mana yang sedang bekerja?

1.2 *Columns* dengan tujuh kolom, masing-masing setinggi 1. Ramalkan jawabannya sebelum menjalankan apa pun, lalu jelaskan dalam satu kalimat.

1.3 Dalam pembuktian *Columns* kita mewarnai kubus di kolom j, baris r dengan $((j + r) \bmod C) + 1$. Jelaskan, tanpa memakai kata "jelas", mengapa dua kubus dalam deret mendatar tanpa sela yang sama tidak mungkin menerima warna yang sama.

1.4 *Journey* dengan tiga kota yang semuanya di koordinat 5. Berapa jawabannya, dan apakah rumusnya masih berlaku?

1.5 Ubahlah *Journey* sehingga Cossack Vus **tidak** perlu kembali ke kota tempatnya bertolak. Turunkan jawaban barunya, dan sebutkan kota awal mana yang sebaiknya ia pilih.

1.6 Kerjakan *Unique Numbers* dengan tangan untuk $n = 36$ dan empat operasi `1 10`, `0 7`, `1 6`, `1 4`. Tunjukkan setiap perhitungan gcd-nya.

1.7 Buktikan dengan cermat bahwa operasi "ganti setiap i dengan $(i + x) \bmod n$" tidak pernah mengubah banyaknya nilai berbeda, untuk sembarang $x \geq 0$.

1.8 Batasan *Unique Numbers* mengizinkan $x = 0$ pada sebuah perkalian. Semestinya berapa jawabannya, apa yang diberikan rumusnya, dan mungkinkah cacahnya naik lagi sesudah itu?

1.9 Sebutkan bilangan 4 angka terkecil dan terbesar yang punya tepat 4 angka berbeda, dengan tangan, lalu periksa terhadap **Kode 1.11**.

1.10 Sebutkan bilangan 18 angka terkecil dan terbesar yang punya tepat 1 angka berbeda.

1.11 Carilah suku ke-100 dari *Interesting sequence* dengan tangan. Tunjukkan pengubahan binernya.

1.12 Buktikan bahwa tidak ada suku *Interesting sequence* yang pernah berupa angka selain 0, 1, atau 2.

1.13 Hitunglah $f(10)$ untuk *Recursive function 1* dengan tangan, dengan menyebutkan setiap entri cache dalam urutan ia terbentuk.

1.14 Mengapa *Recursive function 1* tidak dapat memakai larik untuk memoisasinya? Berikan sebuah angka.

1.15 Sebuah persoalan memberi $n \leq 10^5$ dan batas satu detik. Mana di antara $O(n)$, $O(n\log n)$, $O(n^2)$, $O(n^3)$ yang aman? Beri alasan masing-masing dengan hitungan.

1.16 Nyatakan sebuah invarian gelung untuk pemindaian maksimum pada **Kode 1.3** lalu pakailah untuk membuktikan programnya benar.

1.21 Penyelesaian Soal Latihan

Penyelesaian 1.1. Programnya mencetak 1000. Di sini $n = 1$, jadi kendala mendatarnya hanya menuntut 1 warna, sementara satu kolom berisi 1000 kubus menuntut 1000 menurut kendala tegaknya. Jawabannya $\max(1,1000) = 1000$, dan kendala tegaklah yang mengerjakan seluruh pekerjaannya. Dijalankan pada masukan `1` / `1000`, **Kode 1.8** sungguh-sungguh mencetak `1000`.

Penyelesaian 1.2. Jawabannya 7. Kolom tertingginya 1 kubus, jadi kendala tegaknya meminta 1 warna; tetapi setiap kolom mencapai ketinggian 1, jadi baris terbawahnya adalah substring utuh sepanjang ketujuh kolom dan kendala mendatarnya meminta 7. Jadi $\max(7,1) = 7$. Dijalankan pada `7` / `1 1 1 1 1 1 1`, **Kode 1.8** sungguh-sungguh mencetak `7`. Sebagai kasus ketiga, `3` / `2 9 2` mencetak `9`, ketika kolom tengah yang tinggi itu mendominasi.

Penyelesaian 1.3. Ambil sebuah deret tanpa sela di baris r yang menempati kolom $j, j+1, \ldots, k$. Indeks warnanya sebelum penambahan $+1$ adalah $(j+r) \bmod C, (j+1+r) \bmod C, \ldots, (k+r) \bmod C$ — sisa-sisa pembagian dari $k-j+1$ bilangan bulat yang *berurutan*. Bilangan bulat berurutan punya sisa yang berlainan modulo C asalkan banyaknya paling banyak C, sebab sisanya bertambah tepat satu setiap kali dan baru melingkar balik sesudah C langkah. Deret itu punya paling banyak n kolom, dan $C \geq n$ menurut konstruksinya. Jadi tidak ada dua yang berimpit.

Penyelesaian 1.4. Jawabannya 0. Ketiga kotanya berbagi koordinat 5, jadi $x_{\max} = x_{\min} = 5$ dan $2(5-5) = 0$: Cossack Vus sudah berdiri di ketiga kota itu sekaligus dan tidak perlu bergerak. Rumusnya berlaku tanpa perubahan. Dijalankan pada `3` / `5 5 5`, **Kode 1.9** sungguh-sungguh mencetak `0`; pada `1` / `7` ia juga mencetak `0`.

Penyelesaian 1.5. Tanpa kewajiban pulang, rencana terbaiknya adalah bertolak dari satu ujung lalu berjalan ke ujung yang lain, memberi $x_{\max} - x_{\min}$ — persis separuh perjalanan pulang-perginya. Itu batas bawah sebab lintasannya pada suatu saat pasti berada di $x_{\min}$ dan pada saat lain di $x_{\max}$, dan lintasan mana pun di antara kedua titik itu berharga sekurang-kurangnya jarak pisahnya; dan batas itu tercapai oleh sapuan lurus tadi. Ia sebaiknya bertolak dari salah satu ujung, $x_{\min}$ atau $x_{\max}$; bertolak dari titik yang benar-benar di antara keduanya memaksa sebuah

pembalikan arah dan berharga lebih mahal. Sebuah program yang mencetak kedua ragamnya sungguh-sungguh mencetak `round trip = 6, one way = 3` untuk kota-kota di 1,4, dan `round trip = 16, one way = 8` untuk kota-kota di 10,2,7,5.

Penyelesaian 1.6. Mulai dengan $n = 36$. Operasi `1 10` adalah perkalian. gcd(36,10): $36 \bmod 10 = 6$, jadi gcd(10,6); $10 \bmod 6 = 4$, jadi gcd(6,4); $6 \bmod 4 = 2$, jadi gcd(4,2); $4 \bmod 2 = 0$, jawabannya 2. Lalu $n = 36/2 = 18$. **Cetak 18.** Operasi `0 7` adalah penjumlahan: sebuah bijeksi, n tidak berubah. **Cetak 18.** Operasi `1 6`: gcd(18,6): $18 \bmod 6 = 0$, jawabannya 6. Lalu $n = 18/6 = 3$. **Cetak 3.** Operasi `1 4`: gcd(3,4): $3 \bmod 4 = 3$, jadi gcd(4,3); $4 \bmod 3 = 1$, jadi gcd(3,1); $3 \bmod 1 = 0$, jawabannya 1. Lalu $n = 3/1 = 3$. **Cetak 3.** Pada masukan ini **Kode 1.10** sungguh-sungguh mencetak `18`, `18`, `3`, `3`.

Penyelesaian 1.7. Misalkan $g(i) = (i + x) \bmod n$ pada himpunan $S = \{0,1,\dots,n-1\}$. Andaikan $a, b \in S$ dengan $g(a) = g(b)$. Maka $a + x \equiv b + x \pmod{n}$, jadi n membagi $(a + x) - (b + x) = a - b$. Tetapi a dan b sama-sama terletak dalam $\{0,\dots,n-1\}$, jadi $|a - b| \le n - 1 < n$; satu-satunya kelipatan n yang nilai mutlaknya kurang dari n adalah nol, karenanya $a = b$. Jadi g injektif. Pemetaan injektif dari himpunan berhingga ke dirinya sendiri bersifat surjektif, jadi g adalah bijeksi dan $|g(S)| = |S|$: cacah nilai berbedanya tidak berubah. Penalaran ini tidak memakai apa pun tentang x, jadi ia berlaku untuk setiap $x \ge 0$.

Penyelesaian 1.8. Mengalikan setiap bilangan dengan 0 membuat semuanya menjadi 0, jadi semestinya tepat 1 nilai berbeda yang tersisa. Menurut kelaziman baku $\gcd(n, 0) = n$ — setiap bilangan membagi 0, jadi faktor persekutuan terbesar dari n dan 0 adalah n itu sendiri — dan rumusnya memberi $n \leftarrow n/n = 1$. Cocok. Cacahnya tidak akan pernah bisa naik lagi: penjumlahan adalah bijeksi pada himpunan beranggota satu, dan perkalian memetakan satu-satunya nilai 0 menjadi 0. Dijalankan pada `30` / `2` / `1 0` / `0 5`, **Kode 1.10** sungguh-sungguh mencetak `1` lalu `1`.

Penyelesaian 1.9. $N = 4$, $K = 4$. *Minimum:* awali dengan 1; nol yang harus ditempatkan $= N - K + 1 = 4 - 4 + 1 = 1$; angka sisanya $= K - 2 = 2$, yaitu 2 dan 3. Itu memberi 1 0 2 3 = **1023**, angkanya {0,1,2,3}, empat angka berbeda. *Maksimum:* sembilan yang harus ditempatkan $= 1$; angka sisanya $= K - 1 = 3$, yaitu 8,7,6. Itu memberi 9 8 7 6 = **9876**, angkanya {6,7,8,9}, empat angka berbeda. **Kode 1.11** sungguh-sungguh mencetak `1023 9876`.

Penyelesaian 1.10. Dengan $K = 1$ hanya satu angka yang boleh dipakai, diulang delapan belas kali, dan angka itu tidak boleh 0 sebab angka terdepannya lalu menjadi nol. Terkecil: delapan belas buah 1. Terbesar: delapan belas buah 9. **Kode 1.11** pada `18 1` sungguh-sungguh mencetak `111111111111111111 999999999999999999`.

Penyelesaian 1.11. Rumusnya adalah popcount$(m - 1) \bmod 3$, jadi kita perlu $p = 99$. Ubah: $99 - 64 = 35$; $35 - 32 = 3$; $3 - 2 = 1$; $1 - 1 = 0$. Jadi $99 = 64 + 32 + 2 + 1$, yang dalam biner adalah 1100011. Banyaknya bit 1 adalah 4, dan $4 \bmod 3 =$ **1**. **Kode 1.13** pada $m = 100$ sungguh-sungguh mencetak `1`. Dua pemeriksaan lagi: $m = 64$ memberi $p = 63 = 111111_2$, popcount 6, $6 \bmod 3 = 0$, dan programnya mencetak `0`; $m = 1024$ memberi $p = 1023$, sepuluh bit 1, $10 \bmod 3 = 1$, dan programnya mencetak `1`.

Penyelesaian 1.12. Induksi atas tahapnya. Barisan tahap-0 adalah angka tunggal 0, yang terletak dalam {0,1,2}. Kalau setiap angka tahap k terletak dalam {0,1,2}, maka tahap $k + 1$ adalah tahap k diikuti salinannya yang setiap nilai v-nya diganti dengan $(v + 1) \bmod 3$. Operasi $\bmod 3$ mengembalikan nilai dalam {0,1,2} menurut definisi sisa pembagian oleh 3. Jadi setiap angka tahap $k + 1$ pun terletak dalam {0,1,2}, dan klaimnya berlaku untuk semua tahap.

Penyelesaian 1.13. $f(10) = f(5) + f(3)$. Turun ke $f(5) = f(2) + f(1)$; turun ke $f(2) = f(1) + f(0)$; turun ke $f(1) = f(0) + f(0) = 1 + 1 = 2$. **Simpan** $f(1) = 2$. $f(2) = 2 + 1 = 3$. **Simpan** $f(2) = 3$. $f(5) = 3 + 2 = 5$, suku keduanya dari cache. **Simpan** $f(5) = 5$. $f(3) = f(1) + f(1) = 2 + 2 = 4$, keduanya dari cache. **Simpan** $f(3) = 4$. $f(10) = 5 + 4 = \mathbf{9}$. Urutan cache-nya: $f(1), f(2), f(5), f(3), f(10)$, dengan $f(0)$ ditanam sebelum program berjalan. **Kode 1.15** pada masukan `10` sungguh-sungguh mencetak `9`.

Penyelesaian 1.14. Sebuah larik harus dialokasikan sepanjang seluruh rentang indeksnya, dan indeks di sini bisa sebesar 10^{18}. Dengan delapan bita per `long long` itu 8×10^{18} bita — delapan juta terabita — melawan batas memori 128 megabita. Sebuah `map` hanya menyimpan kunci yang benar-benar disinggahi, yang diukur **Kode 1.14** sebagai 1,012 untuk kasus terburuk: sekitar enam belas kilobita muatan. Nisbah antara apa yang akan dibutuhkan larik dan apa yang dibutuhkan map kira-kira 5×10^{14}.

Penyelesaian 1.15. Ambil anggaran kerja 10^8 operasi per detik. $O(n)$: 10^5 operasi. Aman sejauh tiga orde besaran. $O(n\log n)$: $\log_2 10^5 \approx 16.6$, jadi sekitar 1.7×10^6 operasi. Aman. $O(n^2)$: $(10^5)^2 = 10^{10}$ operasi, seratus kali anggarannya. Tidak aman — harapkan sekitar seratus detik. $O(n^3)$: 10^{15} operasi, sepuluh juta kali anggarannya. Tanpa harapan. Jadi $O(n)$ dan $O(n\log n)$ aman; $O(n^2)$ dan $O(n^3)$ tidak.

Penyelesaian 1.16. *Invarian:* di awal putaran dengan pencacah i, peubah `best` memuat maksimum dari $a[0 \ldots i-1]$. *Inisialisasi:* sebelum gelung, $i = 1$ dan `best` $= a[0]$, yang merupakan maksimum dari rentang satu elemen $a[0 \ldots 0]$. *Pemeliharaan:* andaikan invariannya berlaku di awal sebuah putaran. Badan gelung menetapkan `best` menjadi $a[i]$ kalau $a[i] >$ `best`, dan kalau tidak membiarkannya. Bagaimanapun juga, sesudahnya `best` sama dengan $\max(\max(a[0..i-1]), a[i]) = \max(a[0..i])$. Menaikkan i menjadikan ini invarian untuk putaran berikutnya. *Terminasi:* gelungnya berakhir dengan $i = n$, jadi `best` adalah maksimum dari $a[0 \ldots n-1]$, yaitu seluruh lariknya — dan itulah yang dicetak programnya. Pada larik [3,9,2,9,4,1], **Kode 1.3** sungguh-sungguh mencetak `maximum = 9`. ■

Daftar Pustaka

[1] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[2] D. Gries, *The Science of Programming*. New York, NY, USA: Springer-Verlag, 1981.

[3] E-Olymp (Eolymp Basecamp) Online Judge. Accessed: Sep. 24, 2026. [Online]. Available: https://eolymp.com/en/problems

[4] Sphere Online Judge (SPOJ). Accessed: Sep. 19, 2026. [Online]. Available: https://www.spoj.com

[5] "UVa Online Judge," *Wikipedia*. Accessed: Sep. 24, 2026. [Online]. Available: https://en.wikipedia.org/wiki/UVa_Online_Judge

[6] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, 4th ed. Morrisville, NC, USA: Lulu Press, 2020.

[7] beecrowd, community page. Accessed: early 2026. [Online]. Available: https://beecrowd.com

[8] UVa Online Judge. Accessed: Sep. 24, 2026. [Online]. Available: https://onlinejudge.org

[9] R. Soelaiman, S. Candra, and I. Subakti, "Efficient approach for deterministic data extrapolation from a clean periodic function with periodic components representation by a system of linear equations," *Journal of Theoretical and Applied Information Technology*, vol. 97, no. 8, pp. 2225–2239, Apr. 2019.

[10] R. Soelaiman, Y. H. Pratama, I. Subakti, and Y. Purwananto, "Genetic algorithm approach for automated generation of neural networks architecture for robust digit recognition," *Journal of Theoretical and Applied Information Technology*, vol. 98, no. 17, pp. 3479–3492, Sep. 2020.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[12] "Introduction to Algorithms," *Wikipedia*. Accessed: Sep. 24, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Introduction_to_Algorithms

[13] S. Halim, F. Halim, and S. Effendy, *Competitive Programming Book* (official site). Accessed: Sep. 24, 2026. [Online]. Available: https://cpbook.net

[14] "GNU Compiler Collection," *Wikipedia*. Accessed: Sep. 24, 2026. [Online]. Available: https://en.wikipedia.org/wiki/GNU_Compiler_Collection

[15] "Clang," *Wikipedia*. Accessed: Sep. 24, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Clang

[16] Republic of Indonesia, *Undang-Undang Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi*, Jakarta, Indonesia, 2022.

Bab 2
Memulai: Model RAM, Insertion Sort, dan Pencarian

Pada akhir bab ini Anda akan sanggup menyatakan dengan tepat apa itu algoritma dan apa bedanya dengan program; menghitung sendiri *running time* (waktu jalan) sepotong kode, baris demi baris, tanpa perlu menjalankannya; membuktikan bahwa sebuah gelung benar-benar mengerjakan apa yang Anda katakan, dengan *loop invariant* (invarian gelung); menurunkan dari dasar mengapa Insertion Sort memerlukan waktu sebanding dengan n^2 pada kasus terburuk dan hanya n pada kasus terbaik; menjelaskan mengapa algoritma $O(n\lg n)$ selesai dalam sekejap sementara algoritma $O(n^2)$ berjalan seperempat jam; serta menulis, menelusuri, dan menganalisis *linear search* (pencarian linear) maupun *binary search* (pencarian biner) dalam C++, lengkap dengan dua jebakan selisih-satu yang hampir selalu menjerat pemula.

2.1 Di Mana Kita Berhenti

Bab 1 adalah bab *mengerjakan*. Kita mengambil enam problem nyata dari juri daring E-Olymp dan SPOJ — Columns, Journey, Unique Numbers, Minimum and Maximum Numbers, Interesting Sequence, dan Recursive Function 1 — dan keenamnya kita selesaikan. Kita memperkenalkan Kerangka Lima Fase: bacalah soal sampai Anda sanggup menyatakannya ulang dalam satu kalimat; temukan strukturnya; pilih paradigmanya; taksir ongkosnya sebelum menulis kode; baru setelah itu implementasikan dan uji. Kita menulis C++ yang berhasil dikompilasi, berjalan, dan diterima oleh juri.

Yang **tidak** kita lakukan adalah membenarkan satu pun dari semua itu.

Kita berkata "solusi ini $O(n)$, jadi pasti lolos" tanpa pernah sekali pun mendefinisikan apa arti $O(n)$. Kita berkata bahwa metode Euclid hanya memerlukan $O(\log\min(a,b))$ langkah, dan bahwa memoisasi pada sebuah rekursi mengubah yang mustahil menjadi seketika, dengan notasi yang belum kita definisikan. Di Bab 1 semua pernyataan tadi adalah janji. Bab inilah tempat kita mulai melunasinya, dan Bab 3 menuntaskan pelunasannya.

Rencananya sederhana dan berjalan satu arah. Pertama kita sepakati kata-katanya: apa itu *problem*, apa itu *instans* dari sebuah problem, apa itu *algoritma*, dan apa itu *program*. Keempatnya bukan sinonim, dan mencampuradukkannya adalah sumber tunggal terbesar kekacauan berpikir di bidang ini. Lalu kita sepakati sebuah *mesin* — komputer khayalan yang diidealkan, yang disebut model RAM — sebab Anda tidak dapat menghitung langkah sebelum disepakati apa yang disebut satu langkah. Kemudian kita ambil satu algoritma, Insertion Sort, dan kita perlakukan ia habis-habisan: kita uraikan, kita telusuri dengan tangan, kita buktikan kebenarannya, kita hitung operasinya, dan kita turunkan waktu jalannya secara persis. Setelah itu kita lakukan hal yang sama, dengan lebih cepat, untuk dua algoritma pencarian. Di ujung bab ini Anda memiliki sebuah metode yang dapat Anda terapkan pada algoritma apa pun di sisa buku ini.

> ✓ **WAWASAN KUNCI:** Bab 1 memberi Anda *kebiasaan* menyelesaikan problem. Bab 2 memberi Anda *kosakata dan hitungannya* untuk mengatakan mengapa sebuah solusi itu baik.

> Kebiasaan tanpa hitungan adalah tebak-tebakan; hitungan tanpa kebiasaan adalah kemandulan. Anda memerlukan keduanya.

2.2 Sebenarnya Buku Ini tentang Apa

Pada intinya, ilmu komputer adalah tentang satu hal: menyelesaikan problem. Bukan sembarang problem — problem yang muncul di dunia nyata dan dapat dimodelkan secara komputasional. Berikut lima di antaranya, diambil dari lima bidang yang berbeda:

- Mencari rute dari rumah Anda di Keputih ke kantor Anda di Rungkut. Ini adalah problem **lintasan terpendek**, dan kita akan menyelesaikannya dengan benar di Bab 13.
- Menjadwalkan penerbangan pesawat sedemikian rupa sehingga tidak ada dua pesawat yang memperoleh gerbang yang sama pada waktu yang sama. Ini adalah problem **pemenuhan kendala**.
- Menelusuri miliaran halaman web untuk menemukan yang menjawab pertanyaan Anda. Ini adalah **temu kembali informasi**.
- Menyimulasikan aliran udara di atas bilah turbin mesin jet. Ini adalah **komputasi numerik**.
- Memampatkan video agar dapat dialirkan lewat sambungan telepon genggam. Ini adalah **kompresi data**, dan pengodean Huffman yang Anda jumpai di Bab 9 adalah salah satu perkakasnya yang tertua.

Buku ini mengajarkan cara mengambil problem semacam itu dan menyelesaikannya dengan *baik* — dengan algoritma yang sekaligus **benar**, artinya memberikan jawaban yang tepat, dan **efisien**, artinya memberikannya dengan cepat.

2.2.1 Yang memang dipelajari di buku ini

Hanya ada tiga kegiatan, dan segala isi keenam belas bab ini adalah salah satu dari ketiganya.

Merancang algoritma. Menemukan strategi langkah demi langkah yang mengubah masukan menjadi keluaran. Inilah tindakan kreatifnya. Tidak ada prosedur mekanis untuk melakukannya — justru karena itulah kita mempelajari paradigma, yaitu keluarga-keluarga strategi yang pernah berhasil sebelumnya.

Menganalisis algoritma. Membuktikan bahwa strategi itu benar, dan mengukur seberapa cepat ia berjalan. Inilah tindakan matematisnya, dan inilah bagian yang paling sering dilewati mahasiswa dan paling sering disesali kemudian.

Memilih struktur data. Menata data sedemikian rupa sehingga algoritma dapat bekerja dengan murah. Algoritma yang brilian di atas struktur data yang keliru tetap menghasilkan program yang lambat.

2.2.2 Yang TIDAK dipelajari di buku ini

Sama pentingnya untuk mengatakan apa yang *tidak* kita kerjakan, supaya Anda tahu kapan Anda sedang menyimpang.

Bahasa pemrograman. Kita menulis dalam C++ sepanjang buku ini, tetapi kita tidak mempelajari C++. Kita tidak akan membahas *template* demi *template* itu sendiri, atau seluk-beluk *move semantics*. C++ adalah notasi kita, sebagaimana LaTeX adalah notasi seorang matematikawan.

Arsitektur komputer. Kita sengaja mengabstraksikan perangkat kerasnya, dengan memakai model RAM di Subbab 2.6. *Cache*, *pipeline*, dan *branch predictor* itu nyata dan memang penting di industri, tetapi semuanya akan menenggelamkan gagasan yang hendak kita pelajari di sini.

Rekayasa perangkat lunak. Perancangan sistem berskala besar, kerangka kerja pengujian, kendali versi, praktik pendokumentasian — semuanya penting, semuanya diajarkan di tempat lain.

Teori kompleksitas secara mendalam. Kita menyenggol kompleksitas komputasional — kajian tentang apa yang secara hakiki sukar — tetapi fokus kita adalah merancang algoritma yang bekerja, bukan menggolongkan problem yang tidak punya algoritma sama sekali.

Fokus kita adalah **algoritma**: gagasan yang murni dan abstrak tentang cara menyelesaikan sebuah problem, terlepas dari bahasa atau mesin apa yang mengimplementasikannya.

2.3 Sekelumit Sejarah Algoritma

Algoritma jauh lebih tua daripada komputer. Menyadari kenyataan itu adalah jalan tercepat untuk berhenti menganggap algoritma sebagai "sepotong kode" dan mulai menganggapnya sebagai sebuah gagasan.

2.3.1 Dari mana kata itu berasal

Kata *algoritma* berasal dari nama seseorang. Muhammad ibn Musa **al-Khwarizmi** (sekitar 780–850 M) adalah matematikawan, astronom, dan geograf Persia yang bekerja di Baitulhikmah di Baghdad. Ia menulis sebuah kitab tentang penyelesaian persamaan yang judulnya melahirkan kata *aljabar*, dan sebuah kitab lain tentang angka Hindu–Arab yang menerangkan, langkah demi langkah, cara menjumlah, mengurang, mengali, dan membagi dengan sistem posisi desimal yang ketika itu baru tiba dari India. Ketika kitab yang kedua diterjemahkan ke dalam bahasa Latin pada abad kedua belas, namanya dilatinkan menjadi *Algorismus*, dan selama berabad-abad kata "algorisme" berarti prosedur perhitungan desimal yang ia uraikan itu. Lambat laun kata itu bergeser, baik ejaan maupun maknanya, menjadi *algoritma* yang kita kenal sekarang: prosedur apa pun yang persis untuk menghitung sesuatu.

Ada satu pelajaran yang bersembunyi di balik sejarah itu. Sumbangan terbesar al-Khwarizmi bukanlah sebuah teorema. Yang ia sumbangkan adalah sebuah *prosedur*, yang dituliskan begitu jelasnya sehingga orang yang tidak paham mengapa ia bekerja pun tetap dapat menjalankannya dengan benar. Persis itulah algoritma, dan persis itulah yang sedang Anda pelajari untuk ditulis.

Seperti tampak pada **Gambar 2.1**, kita sedang meminjam sebuah nama yang usianya lebih dari seribu tahun.

Gambar 2.1 Muhammad ibn Musa al-Khwarizmi (sekitar 780–850 M), yang nama Latinnya, *Algorismus*, menjadi kata *algoritma*.

2.3.2 Algoritma yang pertama

Jika usia katanya seribu tahun, usia bendanya jauh lebih tua lagi. Algoritma tak-sepele tertua yang masih dipakai sehari-hari adalah **algoritma Euclid** untuk mencari pembagi persekutuan terbesar dua bilangan bulat, yang dituliskan dalam *Elemen* karya Euclid sekitar 300 SM. Anda sudah memakainya di Bab 1, pada problem Unique Numbers. Usianya lebih dari 2.300 tahun dan, untuk keperluannya, ia masih tetap pada dasarnya metode terbaik yang diketahui.

Inilah algoritmanya, dalam satu baris: untuk mencari $\gcd(a, b)$, gantilah pasangan (a, b) dengan pasangan $(b, a \bmod b)$ dan ulangi sampai bilangan kedua menjadi nol; bilangan pertama saat itu adalah jawabannya. Mari kita jalankan pada gcd(48,18), satu langkah satu baris:

1. $a = 48$, $b = 18$. Karena $b \neq 0$, hitung $48 \bmod 18 = 12$, lalu ganti pasangannya menjadi (18,12).
2. $a = 18$, $b = 12$. Karena $b \neq 0$, hitung $18 \bmod 12 = 6$, lalu ganti pasangannya menjadi (12,6).
3. $a = 12$, $b = 6$. Karena $b \neq 0$, hitung $12 \bmod 6 = 0$, lalu ganti pasangannya menjadi (6,0).
4. $a = 6$, $b = 0$. Karena $b = 0$, berhenti dan laporkan 6.

Dan memang $\gcd(48,18) = 6$. Empat baris hitungan, tanpa komputer, tanpa bahasa pemrograman. Itulah algoritma. Euclid tidak pernah melihat mesin, dan prosedurnya berjalan tanpa perubahan pada setiap mesin yang pernah dibuat orang.

2.3.3 Abad kesembilan belas: mesin dan pemrogram pertama

Charles Babbage (1791–1871) merancang, meski tidak pernah rampung membangunnya, **Analytical Engine** — rancangan pertama untuk mesin hitung serbaguna yang dapat diprogram. Mesin itu memiliki "mill" (yang sekarang kita sebut prosesor), "store" (memori), dan hendak diprogram dengan kartu berlubang yang dipinjam dari alat tenun Jacquard. **Gambar 2.2** memperlihatkan wajahnya.

Gambar 2.2 Charles Babbage (1791–1871), perancang Analytical Engine, mesin hitung serbaguna pertama yang pernah digagas orang.

Ada Lovelace (1815–1852), berangkat dari rancangan Babbage, menulis sebuah metode terperinci agar Analytical Engine menghitung bilangan Bernoulli. Metode itu umumnya dianggap algoritma pertama yang pernah ditulis khusus untuk dijalankan oleh sebuah mesin, dan karena itu ia disepakati sebagai pemrogram pertama. Ia bahkan melihat lebih jauh daripada Babbage: ia menulis bahwa suatu saat mesin itu mungkin memanipulasi bukan hanya bilangan, melainkan lambang apa pun yang hubungan antaranggotanya dapat dinyatakan — musik, misalnya. Untuk tahun 1843, itu sebuah pikiran yang luar biasa modern. Ia tampak pada **Gambar 2.3**.

Gambar 2.3 Ada Lovelace (1815–1852), yang menyusun algoritma pertama yang dimaksudkan untuk dijalankan oleh sebuah mesin.

2.3.4 Abad kedua puluh: apa sebenarnya *komputasi* itu

Tiga orang mengubah komputasi dari rekayasa menjadi ilmu.

Alan Turing (1912–1954), dalam makalahnya tahun 1936, mendefinisikan sebuah mesin khayalan — sebuah pita, sebuah kepala yang membaca dan menulis lambang, serta sebuah tabel aturan yang berhingga — dan berargumen bahwa apa pun yang dapat dihitung oleh seorang "penghitung" manusia dengan mengikuti aturan, dapat pula dihitung oleh mesin itu. **Mesin Turing** memberi kita, untuk pertama kalinya, definisi matematis yang persis tentang apa artinya sesuatu itu terhitungkan, dan bersamanya bukti bahwa sejumlah problem yang perumusannya sempurna ternyata *tidak punya* algoritma sama sekali. Lihat **Gambar 2.4**.

Gambar 2.4 Alan Turing (1912–1954), yang memformalkan komputasi lewat mesin Turing pada tahun 1936.

Alonzo Church (1903–1995) tiba di perbatasan yang sama dari arah yang berbeda, lewat **kalkulus lambda**: sebuah notasi yang di dalamnya segala sesuatu adalah fungsi, dan komputasi adalah penyulihan argumen ke dalam tubuh fungsi secara berulang. Sistem Church dan mesin Turing ternyata mendefinisikan kelas fungsi terhitungkan yang persis sama, sebuah kebetulan yang begitu mencolok sehingga sekarang disebut tesis Church–Turing. Setiap bahasa pemrograman fungsional yang hidup sampai hari ini adalah keturunan notasi Church. Ia tampak pada **Gambar 2.5**.

Gambar 2.5 Alonzo Church (1903–1995), penemu kalkulus lambda dan penyusun bersama tesis Church–Turing.

John von Neumann (1903–1957) melakukan sesuatu yang lebih praktis: dalam sebuah laporan tahun 1945, ia memerikan arsitektur yang di dalamnya program disimpan pada memori yang sama dengan data yang diolahnya. Satu keputusan itu — **komputer program-tersimpan** — adalah alasan mengapa laptop Anda dapat menjalankan program yang belum pernah dilihatnya, cukup dengan memuatnya ke dalam RAM. Hampir setiap komputer di dunia masih berupa mesin von Neumann, dan model RAM yang segera kita pakai adalah pengidealan langsung dari rancangannya. Ia tampak pada **Gambar 2.6**.

Gambar 2.6 John von Neumann (1903–1957), yang arsitektur program-tersimpannya mendasari hampir setiap komputer yang dipakai orang hari ini.

✓ **WAWASAN KUNCI:** Algoritma lebih tua daripada komputer. Gagasannya abadi — yang berganti adalah perangkat keras yang menjalankannya. Algoritma Euclid telah hidup lebih panjang daripada negara-kota Yunani, Kekaisaran Romawi, tabung hampa, dan cakram liuk. C++ yang Anda tulis minggu ini akan usang dua puluh tahun lagi; algoritma yang diimplementasikannya mungkin tidak.

2.4 Kosakata: Problem, Instans, Algoritma, Program, Struktur Data

Subbab ini mendefinisikan lima kata. Kelimanya dipakai secara longgar dalam percakapan dan secara ketat dalam buku ini, dan hampir semua kebingungan pemula tentang analisis algoritma dapat dilacak sampai ke pengaburan dua di antaranya. Bacalah subbab ini pelan-pelan.

2.4.1 Problem algoritmik

DEFINISI. Sebuah **problem algoritmik** adalah spesifikasi hubungan yang dikehendaki antara masukan dan keluaran. Ia menyatakan *apa* yang harus dihitung. Ia tidak menyatakan *bagaimana*.

Spesifikasi sebuah problem selalu terdiri atas dua paruh. Paruh pertama menyatakan apa yang tergolong masukan yang sah. Paruh kedua menyatakan, untuk tiap masukan yang sah, apa yang tergolong keluaran yang benar.

Inilah problem yang akan kita tekuni hampir sepanjang bab ini.

CONTOH — problem PENGURUTAN (*sorting*).

Masukan: sebuah barisan berisi n bilangan $\langle a_1, a_2, \dots, a_n \rangle$.

Keluaran: sebuah permutasi (penataan ulang) $\langle a'_1, a'_2, \dots, a'_n \rangle$ dari barisan yang sama itu sedemikian rupa sehingga $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Perhatikan tiga hal pada spesifikasi itu. Ia sama sekali tidak menyebut bahasa pemrograman. Ia sama sekali tidak menyebut metode. Dan ia mencakup *tak berhingga banyaknya* masukan yang berbeda sekaligus: setiap barisan bilangan berhingga yang pernah ada dan yang akan pernah ada.

Lambang n di situ adalah potongan notasi kita yang pertama, dan ia akan mengikuti kita sepanjang buku ini. Ia menyatakan **ukuran masukan** — di sini, ada berapa bilangan. Kurung sudut $\langle \cdots \rangle$ sekadar berarti "sebuah barisan, dalam urutan ini"; tidak seperti himpunan, barisan boleh mengulang nilai dan urutannya berarti. Tanda aksen pada a'_1 dibaca "a-satu-aksen" dan sekadar berarti "elemen pertama keluaran", yang umumnya bukan a_1, elemen pertama masukan.

2.4.2 Instans sebuah problem

DEFINISI. Sebuah **instans** dari suatu problem adalah satu masukan sah tertentu bagi problem itu, beserta segala hal yang diperlukan agar pertanyaannya menjadi konkret.

Problem PENGURUTAN adalah sebuah problem. Barisan $\langle 5,2,4,6,1,3 \rangle$ adalah sebuah *instans* darinya. Keluaran yang benar untuk instans itu adalah $\langle 1,2,3,4,5,6 \rangle$.

Mengapa pembedaan ini cukup penting sampai layak diberi definisi sendiri? Karena *kebenaran adalah pernyataan tentang problem, sedangkan waktu jalan biasanya pernyataan tentang instans.* Sebuah algoritma disebut benar hanya jika ia menghasilkan jawaban yang tepat pada setiap instans, tanpa kecuali. Namun ia bisa saja secepat kilat pada satu instans dan lambat tanpa harapan pada instans lain yang ukurannya persis sama. Kita akan melihat persis hal ini pada Insertion Sort di Subbab 2.11: pada instans $\langle 1,2,3,4,5,6 \rangle$ ia nyaris tidak mengerjakan apa-apa,

sedangkan pada instans ⟨6,5,4,3,2,1⟩ — ukuran sama, bilangan sama — ia mengerjakan kerja sebanyak-banyaknya yang mungkin.

2.4.3 Algoritma

> **DEFINISI.** Sebuah **algoritma** adalah prosedur komputasional yang terdefinisi baik, yang menerima suatu nilai, atau himpunan nilai, sebagai **masukan**, dan menghasilkan suatu nilai, atau himpunan nilai, sebagai **keluaran**. Ia adalah rangkaian langkah komputasional yang mengubah masukan menjadi keluaran.

Algoritma adalah *jawabannya*, sedangkan problem tadi adalah *pertanyaannya*. Dan untuk problem apa pun terdapat tak berhingga banyaknya algoritma yang benar — ada yang cepat, ada yang lambat, ada yang anggun, ada yang buruk rupa. Seluruh tantangan buku ini adalah menemukan, di antara ketakberhinggaan itu, yang efisien.

Agar dapat disebut algoritma sama sekali, sebuah prosedur harus memiliki lima sifat. Inilah kriteria klasik yang berasal dari Knuth, dan Anda semestinya sanggup menguji prosedur apa pun terhadapnya.

1. **Masukan.** Nol atau lebih nilai dipasok dari luar. (Nol pun boleh: "cetak seribu bilangan prima pertama" tidak memerlukan masukan.)
2. **Keluaran.** Sekurang-kurangnya satu nilai dihasilkan. Prosedur yang tidak menghitung apa pun yang dapat diamati tidak sedang menyelesaikan apa pun.
3. **Kepastian.** Setiap langkah dinyatakan secara persis dan tanpa kedwiartian. "Aduk sampai kelihatan pas" tidaklah pasti. "Bandingkan $A[i]$ dengan key, dan jika $A[i]$ lebih besar, salin $A[i]$ ke $A[i+1]$" barulah pasti.
4. **Keberhinggaan.** Prosedurnya berhenti setelah sejumlah berhingga langkah, untuk *setiap* masukan yang sah. Sifat inilah yang memisahkan algoritma dari sekadar prosedur: `while (true) i = i + 1;` itu pasti, terlaksanakan, dan tak berguna.
5. **Keterlaksanaan.** Setiap langkah cukup mendasar sehingga pada prinsipnya seseorang dapat menjalankannya secara persis, dengan pensil dan kertas, dalam waktu berhingga. "Putuskan apakah program sembarang ini berhenti" tidaklah terlaksanakan — Turing membuktikan langkah semacam itu tidak ada.

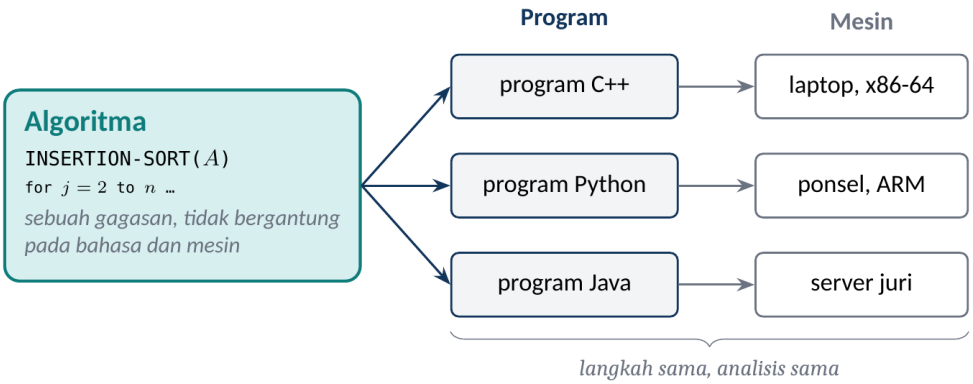
> **CONTOH — menguji sebuah prosedur terhadap kelima sifat itu.** Tinjau prosedur berikut: *"Untuk mengurutkan sebuah senarai, bangkitkan semua kemungkinan penataannya satu per satu; berhentilah ketika Anda membangkitkan satu yang sudah terurut."* Masukan: ada, yaitu senarainya. Keluaran: ada, yaitu senarai terurut. Kepastian: ada, asalkan kita tetapkan urutan pembangkitan permutasinya. Keberhinggaan: ada — penataannya hanya $n!$ buah dan sekurang-kurangnya satu di antaranya terurut, jadi kita pasti berhenti. Keterlaksanaan: ada, seseorang dapat mengerjakannya. Jadi ia *memang* sebuah algoritma. Ia juga algoritma yang buruknya luar biasa: untuk $n = 20$ ia bisa membangkitkan $20! \approx 2.4 \times 10^{18}$ penataan. Menjadi algoritma adalah ambang yang rendah. Menjadi algoritma yang *baik* barulah ambang yang kita pedulikan.

2.4.4 Program

> **DEFINISI.** Sebuah **program** adalah implementasi sebuah algoritma dalam bahasa pemrograman tertentu, yang dinyatakan sedemikian rupa sehingga sebuah mesin tertentu dapat menjalankannya.

Algoritma adalah gagasan abstraknya; program adalah perwujudan konkretnya. Satu algoritma dapat diimplementasikan oleh program yang tak terhitung banyaknya — dalam C++, dalam

Python, oleh seorang mahasiswa dengan rapi dan oleh mahasiswa lain dengan berantakan — dan semuanya tetap algoritma yang sama. Inilah sebabnya kita menganalisis algoritma, bukan program: analisisnya lalu berlaku untuk semua implementasi sekaligus, pada setiap mesin, selamanya. **Gambar 2.7** menggambarkan hubungannya: satu algoritma, banyak program, banyak mesin — dan satu analisis yang mencakup semuanya.



Gambar 2.7 Algoritma adalah sebuah gagasan; program adalah jadinya gagasan itu pada sebuah mesin tertentu, dalam sebuah bahasa tertentu.

2.4.5 Struktur data

> **DEFINISI.** Sebuah **struktur data** adalah cara tertentu menata dan menyimpan data di dalam komputer sehingga data itu dapat diakses dan diubah secara efisien.

Larik, senarai berkait, tumpukan, antrean, tabel hash, *heap*, pohon pencarian biner — semuanya menyimpan jenis informasi yang sama, tetapi masing-masing membuat operasi yang berbeda menjadi murah dan operasi yang berbeda pula menjadi mahal. Larik membuat "berikan saya elemen di posisi 7" seketika, tetapi "sisipkan sesuatu di posisi 7" mahal. Senarai berkait justru sebaliknya.

> **Analogi.** Jika algoritma adalah resepnya — "urutkan bilangan-bilangan ini" — maka struktur data adalah peralatan dapurnya. Resep yang sama akan cepat dengan penggiling listrik dan lambat dengan cobek dan ulekan. Peralatan yang lebih baik tidak mengubah resepnya; ia mengubah berapa lama resep itu dikerjakan. Memilih struktur data *adalah* bagian dari merancang algoritma, dan itulah sebabnya kedua pokok bahasan ini selalu diajarkan bersama.

2.4.6 Kebenaran dan efisiensi, didefinisikan

Dua definisi lagi, lalu lengkaplah semua kata kita.

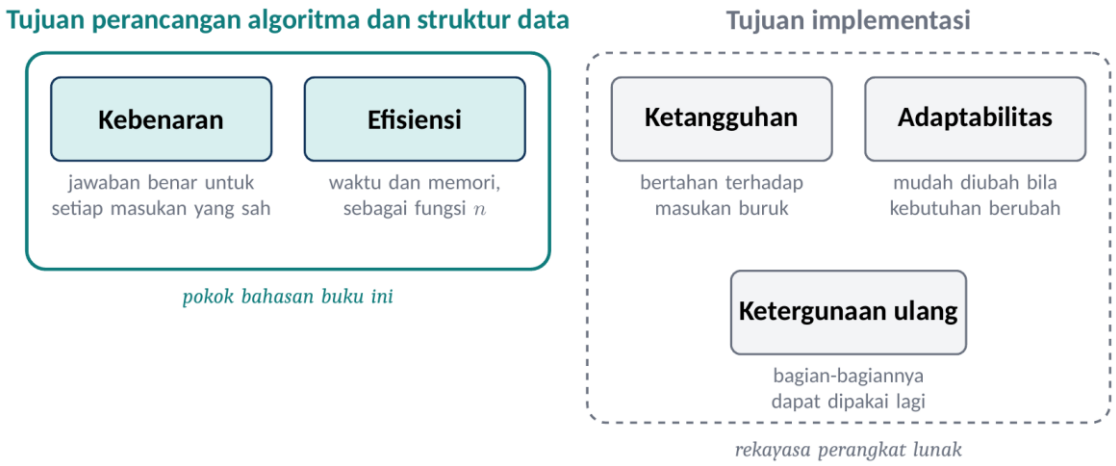
> **DEFINISI.** Sebuah algoritma disebut **benar** untuk suatu problem jika, untuk **setiap** instans sah problem itu, ia berhenti dan menghasilkan keluaran yang memenuhi spesifikasi keluaran problem tersebut. Algoritma yang memberikan jawaban tepat pada setiap instans kecuali satu bukanlah "hampir benar"; ia salah.
>
> **DEFINISI. Efisiensi** sebuah algoritma adalah banyaknya sumber daya yang dihabiskannya sebagai fungsi dari ukuran masukan n — terutama *waktu*, yang diukur sebagai cacah langkah primitif, dan berikutnya *ruang*, yang diukur sebagai cacah sel memori.

Frasa *sebagai fungsi dari ukuran masukan* adalah jantung persoalannya, dan itulah yang dibicarakan sisa bab ini. Kita tidak bertanya "berapa lama ini memakan waktu?"; kita bertanya "bagaimana waktunya **tumbuh** ketika n tumbuh?" Pertanyaan yang kedua punya jawaban yang tidak bergantung pada mesin, sedangkan yang pertama tidak.

> ⚠ **RENUNGKAN:** Di antara tak berhingga banyaknya algoritma benar untuk suatu problem, temukan yang EFISIEN. Itulah seluruh buku ini dalam satu kalimat. Kebenaran memberi Anda jawaban; efisiensi memberi Anda jawaban sebelum tenggat lewat, sebelum baterai habis, sebelum pelanggan pergi.

2.5 Lima Tujuan Perancangan

Ketika kita membangun algoritma dan struktur data, ada lima tujuan yang kita bidik, dan ada gunanya mengetahui sejak awal dua yang mana yang menjadi urusan buku ini dan tiga yang mana yang bukan. **Gambar 2.8** mengelompokkannya: dua yang pertama termasuk *perancangan* struktur data dan algoritmanya, tiga yang terakhir termasuk *implementasinya*.



Gambar 2.8 Kelima tujuan itu, dalam dua kelompok. Kebenaran dan efisiensi adalah tujuan perancangan dan menjadi pokok bahasan buku ini; kekukuhan, keteradaptasian, dan keterpakaian ulang adalah tujuan implementasi yang termasuk wilayah rekayasa perangkat lunak.

Kebenaran (★ utama). Algoritma menghasilkan keluaran yang benar untuk *setiap* masukan yang sah — bukan untuk tiga kasus uji yang kebetulan Anda coba. Pengujian hanya pernah dapat menunjukkan adanya kutu, tidak pernah ketiadaannya, jadi kita membuktikan kebenaran, bukan mengujinya. Perkakasnya adalah invarian gelung, yang kita jumpai di Subbab 2.9, dan induksi matematis, yang sebenarnya perkakas yang sama dengan pakaian berbeda.

Efisiensi (★ utama). Algoritma memakai waktu dan memori sesedikit mungkin. Ini kita ukur dengan analisis asimtotik, yang memberi tahu bagaimana waktu jalan tumbuh ketika ukuran masukan tumbuh. Bab 3 membuat hal ini persis dengan notasi O, Θ, dan Ω; bab ini mengantar Anda sampai ke titik Anda menginginkannya.

Kekukuhan. Program menangani masukan yang keliru atau tak terduga dengan anggun alih-alih macet atau mengembalikan omong kosong. Jika pemakai rutin pengurutan menyodorkan penunjuk kosong, atau panjang larik -1, apa yang terjadi? Itu pertanyaan yang sangat penting — dan itu pertanyaan rekayasa perangkat lunak, sebab ia menyangkut program, bukan algoritma.

Keteradaptasian. Program mudah diubah ketika kebutuhannya berubah, sebagaimana selalu terjadi. Perancangan algoritma yang baik membantu di sini hampir secara cuma-cuma: solusi yang bersih dan tertata, yang dibangun dari bagian-bagian yang dipahami, jauh lebih mudah diubah daripada sebuah kekusutan.

Keterpakaian ulang. Kodenya dapat dipakai lagi pada program lain. Algoritma baku — pengurutan, pencarian, penelusuran graf — adalah komponen pakai-ulang yang paling utama,

dan justru karena itulah semuanya dibakukan dalam pustaka seperti *Standard Template Library* C++. Setiap jam yang Anda habiskan untuk memahami cara kerja `std::sort` adalah satu jam yang tidak perlu Anda habiskan untuk menulisnya sendiri.

> ✓ **WAWASAN KUNCI:** Dalam buku ini kita memusatkan perhatian hampir seluruhnya pada kebenaran dan efisiensi. Ketiga tujuan lain itu penting — merekalah yang memisahkan purwarupa yang bekerja dari produk yang dikirim — tetapi mereka termasuk wilayah rekayasa perangkat lunak. Di sini, jika sebuah algoritma benar dan cepat, ia sudah menunaikan tugasnya.

2.6 Model Komputasi RAM

> ❓ **PERTANYAAN:** Sebelum dapat menganalisis secepat apa sebuah algoritma berjalan, kita harus sepakat dahulu apa arti "satu langkah". Prosesor modern menjalankan sebagian instruksi dalam satu denyut jam dan sebagian lain dalam ratusan denyut. Yang mana yang disebut satu langkah? Bagaimana cara menghitungnya?

Jawabannya: kita berhenti berusaha memerikan mesin yang sesungguhnya, lalu mengambil sebuah mesin ideal yang disepakati bersama. Modelnya disebut **Random Access Machine**, atau *RAM model* (model RAM), dan itulah model yang dipakai untuk menyatakan setiap waktu jalan di dalam buku ini.

2.6.1 Aturan main model ini

Model RAM adalah komputer khayalan dengan sifat-sifat berikut.

1. Ia memiliki memori yang terdiri atas sel-sel bernomor, dan ia boleh membaca atau menulis sel **mana pun** dalam satu langkah waktu, terlepas dari sel yang mana itu. Itulah arti *akses acak*: mencapai sel nomor 1.000.000 berongkos persis sama dengan mencapai sel nomor 3.
2. Instruksi dijalankan satu demi satu, tanpa kesertaan. Prosesornya satu dan ia mengerjakan satu hal pada satu saat.
3. Setiap instruksi **sederhana** memakan tepat **satu** langkah waktu.
4. Datanya berupa bilangan bulat dan bilangan titik-kambang yang ukurannya *wajar* — lebih tepatnya, setiap nilai muat dalam sejumlah konstan sel memori, dan cacah bit dalam satu kata sekurang-kurangnya $\lg n$ sehingga sebuah indeks ke dalam masukan dapat ditampung oleh satu kata. (Lambang $\lg n$ berarti $\log_2 n$, logaritma berbasis 2; kita akan mendefinisikannya dengan cermat di Subbab 2.12.)

2.6.2 Apa yang dihitung satu langkah

Instruksi yang dikenai biaya satu langkah oleh model ini terbagi dalam tiga keluarga.

Aritmetika. Penjumlahan, pengurangan, perkalian, pembagian, sisa bagi (`%`), pembulatan ke bawah, pembulatan ke atas, dan pembandingan dua bilangan. Termasuk pula operasi bitwise: `&`, `|`, `^`, `<<`, `>>`.

Perpindahan data. Memuat sebuah nilai dari memori ke register, menyimpan register kembali ke memori, menyalin satu peubah ke peubah lain. Yang penting, **pengindeksan larik**, `A[i]`, adalah satu langkah: mesin menghitung alamatnya lalu mengambil selnya, dan menurut aturan 1 ongkosnya sama di mana pun sel itu berada.

Kendali. Menguji sebuah syarat dalam `if` atau `while`; melompat ke instruksi lain; memanggil subrutin; kembali dari subrutin.

Jadi dalam model RAM, tiap baris C++ berikut berongkos tepat satu langkah:

```
x = y + z;        // satu penjumlahan, ditambah perpindahannya -- dihitung konstan
if (a > b) ...    // satu pembandingan
t = A[i];         // satu akses larik
i = i - 1;        // satu pengurangan dan satu penyimpanan
```

2.6.3 Apa yang BUKAN satu langkah

Inilah bagian yang dilewati mahasiswa, dan dari sinilah analisis yang keliru berasal.

Sebuah gelung bukan satu langkah. `for (int i = 0; i < n; ++i) x = x + 1;` hanya berisi baris-baris berongkos konstan, tetapi ia menjalankannya n kali. Ongkosnya n langkah, bukan satu. Anda harus selalu mengalikan biaya sebuah baris dengan berapa kali baris itu dijalankan — dan itulah seluruh teknik Subbab 2.7.

Sebuah pemanggilan fungsi bukan satu langkah, kecuali jika fungsinya sendiri berongkos konstan. Memanggil `sort(v)` adalah satu baris pada program Anda dan barangkali $n\lg n$ langkah kerja. Dalam analisis Anda harus membuka kotaknya dan menghitung isinya.

Operasi pada data tak terbatas bukan satu langkah. Mengalikan dua bilangan yang masing-masing berdigit sejuta bukanlah satu operasi tunggal; klausul "ukuran wajar" pada model RAM mengecualikannya. Demikian pula, menyalin sebuah untai sepanjang m berongkos m, bukan 1, sebab itu m perpindahan sel yang terpisah yang mengenakan satu nama saja.

Pemangkatan dengan eksponen sembarang bukan satu langkah. Menghitung 2^k dengan k yang boleh tumbuh bersama n memakan sekitar $\lg k$ perkalian dengan pengkuadratan berulang; mengenakan biaya satu langkah padanya berarti mengizinkan Anda menyelundupkan kerja raksasa melewati sang juru hitung.

> ⚠ **RENUNGKAN:** Bila ragu, bertanyalah: *apakah banyaknya kerja yang dilakukan baris ini bergantung pada n?* Jika jawabannya tidak, ia satu langkah. Jika jawabannya ya, ia bukan satu langkah, dan Anda harus menghitung apa yang ada di dalamnya.

2.6.4 Mengapa memilih model ini?

Model RAM, terus terang saja, adalah sebuah kebohongan. Komputer sungguhan punya *cache* yang membuat sebagian akses memori lima puluh kali lebih lambat daripada yang lain; mereka punya *pipeline* instruksi, prediksi cabang, banyak inti, dan memori maya. Aturan 1 khususnya — bahwa setiap akses memori berongkos sama — justru anggapan yang paling keras dilanggar oleh perangkat keras modern.

Lalu mengapa memakainya?

Karena ia tidak bergantung pada mesin. Jika kita mengukur detik pada jam dinding, algoritma yang sama akan tampak bertambah cepat setiap kali seseorang membeli laptop baru, dan dua mahasiswa yang mengukur kode yang sama akan melaporkan hasil yang berbeda. Model RAM menghitung *operasi*, bukan detik. Cacah operasi adalah sifat algoritma dan masukannya, bukan sifat toko tempat Anda membeli komputer.

Karena ia berdaya ramal. Inilah pembenaran yang sesungguhnya, dan sifatnya empiris, bukan teoretis: algoritma yang menurut model RAM lebih cepat umumnya *memang* lebih cepat pada perangkat keras nyata, begitu n membesar. Model ini benar dalam mengurutkan peringkat walaupun keliru dalam konstantanya. Anda akan melihat hal itu diperagakan dengan pengukur waktu di Subbab 2.11.4.

Karena ia cukup sederhana untuk dipakai berhitung. Model yang memerikan hierarki *cache* secara setia akan begitu rumitnya sehingga tak seorang pun dapat menganalisis apa pun dengannya. Model semacam itu memang ada — model memori-luar, model *cache-oblivious* — dan dipakai ketika lalu lintas memori benar-benar mendominasi. Untuk semua isi buku ini, RAM adalah tingkat kerincian yang tepat.

> ✓ **WAWASAN KUNCI:** Sebuah model bukanlah pemerian kenyataan; ia adalah keputusan tentang apa yang boleh diabaikan. Model RAM mengabaikan segala hal tentang perangkat keras kecuali bahwa memorinya seragam dan kerjanya berurutan. Ia menyimpan persis apa yang dibutuhkan untuk membandingkan algoritma secara adil, dan membuang persis apa yang kalau tidak dibuang akan membuat perbandingan itu bergantung pada anggaran Anda.

2.7 Cara Menganalisis Waktu Jalan

Sekarang kita punya mesinnya. Inilah metode menghitung di atasnya.

> **DEFINISI. Waktu jalan** (*running time*) sebuah algoritma pada suatu masukan tertentu adalah cacah seluruh langkah primitif yang dijalankan dalam model RAM. Kita menuliskannya $T(n)$, dibaca "T dari n", dengan n adalah ukuran masukan. Ia adalah **cacah operasi**, tidak pernah jumlah detik.

Prosedurnya terdiri atas tiga bagian, dan tidak pernah berubah:

1. **Berikan biaya pada tiap baris.** Dalam model RAM setiap baris kode lurus berongkos suatu konstanta. Konstanta itu kita beri nama: c_1 untuk baris pertama, c_2 untuk baris kedua, dan seterusnya. Kita tidak berusaha mencari nilai angkanya — nilai itu bergantung pada kompilator dan prosesor, dan, seperti akan kita lihat, nilainya tidak akan berarti.
2. **Hitung berapa kali tiap baris dijalankan.** Inilah bagian yang menuntut pemikiran, sebab ia bergantung pada masukan.
3. **Jumlahkan biaya × cacah untuk semua baris.** Hasilnya adalah $T(n)$.

Hanya itu saja. Seluruh keterampilannya terletak pada langkah 2.

2.7.1 Apa itu ukuran masukan?

Lambang n harus didefinisikan untuk tiap problem, sebab "ukuran" berarti hal yang berlainan di lingkungan yang berlainan:

- **Pengurutan dan pencarian:** n adalah cacah elemen dalam larik. Inilah yang berlaku sepanjang bab ini.
- **Operasi matriks:** n adalah dimensi matriksnya, sehingga masukannya sebenarnya memuat n^2 bilangan. Karena itu algoritma yang berjalan dalam waktu n^3 sebenarnya hanya sekitar $n^{1.5}$ kali ukuran masukannya — layak diingat sebelum Anda menyebutnya lambat.
- **Algoritma graf:** diperlukan dua parameter, n (atau $|V|$) untuk cacah simpul dan m (atau $|E|$) untuk cacah sisi. Mulai Bab 10 seterusnya, semua waktu jalan kita menyebut keduanya.
- **Problem teori bilangan:** n adalah cacah **bit** dalam bilangan masukan, bukan bilangan itu sendiri. Menguji apakah sebuah bilangan 2.048 bit itu prima adalah problem berukuran 2.048, bukan berukuran 2^{2048}. Salah di sini, maka setiap klaim kompleksitas dalam kriptografi akan tampak menggelikan bagi Anda.

> ⚠ **RENUNGKAN:** Selalu nyatakan apa arti n sebelum Anda memakainya. "Ini berjalan dalam $O(n)$" tidak bermakna apa-apa sampai pembaca tahu apakah n menghitung elemen, simpul, atau bit.

2.7.2 Waktu jalan bukan waktu jam dinding

Ini perlu diulang, sebab inilah kesalahpahaman yang paling umum di bidang ini. Waktu jalan, dalam buku ini, adalah cacah operasi primitif yang dijalankan. Ia bukan jumlah detik yang berlalu. Keduanya memang berkaitan — pada satu mesin tertentu, detik kira-kira sebanding dengan operasi — tetapi hanya cacah operasi yang merupakan sifat algoritmanya.

Jika Anda mengukur sebuah program dan memperoleh 0,8 detik, yang Anda pelajari adalah sesuatu tentang laptop Anda, bendera kompilasi Anda, suasana hati sistem operasi Anda, dan, di suatu tempat di dalamnya, algoritma Anda. Jika Anda menghitung bahwa ia menjalankan sekitar $n^2/2$ operasi, yang Anda pelajari adalah sesuatu yang lima puluh tahun lagi masih akan benar.

2.7.3 Contoh lengkap yang pertama

Mari kita jalankan seluruh prosedurnya sekali pada sesuatu yang mungil, supaya metodenya jelas sebelum datanya menjadi rumit. **Pseudokode 2.1** mencari nilai terbesar dalam sebuah larik berisi n bilangan:

Pseudokode 2.1 Mencari yang terbesar di antara n bilangan dengan satu kali sapuan.

```
FIND-MAX(A, n)
1  max = A[1]
2  for i = 2 to n
3      if A[i] > max
4          max = A[i]
5  return max
```

Sekarang ketiga langkahnya.

Langkah 1, biaya. Baris 1 adalah satu akses larik dan satu penyimpanan: biaya c_1. Mesin gelung pada baris 2 — uji $i \leq n$, naikkan i — berongkos c_2 untuk tiap pengujian. Baris 3 adalah satu akses larik dan satu pembandingan: c_3. Baris 4 adalah satu akses dan satu penyimpanan: c_4. Baris 5 adalah satu pengembalian: c_5.

Langkah 2, cacah. Baris 1 berjalan sekali. Pengujian pada baris 2 berjalan n kali seluruhnya: sekali untuk tiap $i = 2,3,\ldots,n$, yaitu $n-1$ pengujian yang berhasil, ditambah satu pengujian terakhir dengan $i = n+1$ yang gagal dan mengakhiri gelung. Baris 3 berjalan sekali untuk tiap pengujian yang berhasil, jadi $n-1$ kali. Baris 4 berjalan hanya ketika pengujian pada baris 3 berhasil; sebut cacahnya k, dengan $0 \leq k \leq n-1$. Baris 5 berjalan sekali.

Langkah 3, penjumlahan.

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4 k + c_5. \tag{2.1}$$

Karena k berada di suatu tempat antara 0 dan $n-1$, kita dapat membatasi Persamaan (2.1) dari kedua sisi:

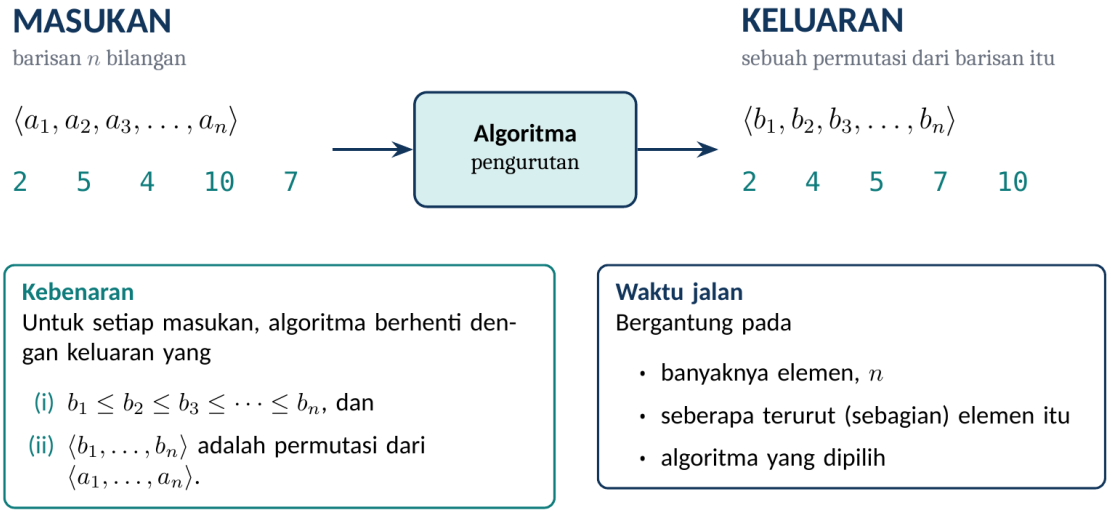
$$c_1 + c_2 n + c_3(n-1) + c_5 \;\leq\; T(n) \;\leq\; c_1 + c_2 n + (c_3 + c_4)(n-1) + c_5.$$

Kedua ujungnya berbentuk $an + b$ untuk konstanta a dan b. Jadi apa pun masukannya, $T(n)$ adalah fungsi linear dari n. Di Bab 3 kita akan mengatakan $T(n) = \Theta(n)$. Perhatikan apa yang terjadi pada k: ia mengubah konstantanya, tetapi tidak mengubah bentuknya. Itulah pola yang akan Anda jumpai berulang-ulang.

2.8 Insertion Sort: Algoritmanya

Insertion Sort — *insertion sort* (pengurutan sisip) — adalah algoritma sungguhan pertama yang kita analisis secara tuntas, dan kita menganalisisnya tuntas karena ia spesimen pengajaran yang sempurna: ia cukup pendek untuk ditampung di kepala, ia sungguh berguna dalam praktik untuk larik kecil atau larik yang hampir terurut, bukti kebenarannya adalah contoh buku teks untuk invarian gelung, dan kasus terbaik serta kasus terburuknya berbeda secara mencolok, yang memaksa kita berhadapan dengan analisis tiga kasus di Subbab 2.11.

Problem yang diselesaikannya adalah problem PENGURUTAN di Subbab 2.4.1. **Gambar 2.9** menyatakan problem itu secara formal, dengan masukan di kiri, keluaran di kanan, dan — di bagian bawah — dua pertanyaan yang justru menjadi alasan keberadaan seluruh bab ini: apakah ia benar, dan berapa lama ia memakan waktu?



Gambar 2.9 Problem pengurutan. Keluaran $\langle b_1, \ldots, b_n \rangle$ harus sekaligus *terurut* ($b_1 \leq b_2 \leq \cdots \leq b_n$) dan merupakan *permutasi* dari masukannya — dua syarat, dan algoritma yang hanya memenuhi salah satunya adalah algoritma yang salah. Waktu jalannya bergantung pada ada berapa elemen, seberapa hampir-terurut elemen itu sudah, dan algoritma mana yang kita pilih.

Bacalah kotak kiri-bawah gambar itu dengan cermat, sebab di dalamnya ada jebakan. Kebenaran menuntut **dua** syarat, bukan satu. Prosedur yang mengeluarkan ⟨1,2,3,4,5⟩ untuk setiap masukan menghasilkan barisan yang terurut sempurna dan sama sekali salah, sebab keluarannya bukan permutasi dari masukannya. Keterurutan saja bukanlah pengurutan.

2.8.1 Intuisinya: menata kartu di tangan

Bayangkan Anda sedang bermain kartu. Beberapa kartu ada di tangan kiri Anda, dan Anda menjaganya selalu terurut. Sisa kartunya tertelungkup di atas meja. Dengan tangan kanan Anda mengambil kartu teratas dari meja, dan Anda harus menaruhnya ke tangan kiri sedemikian rupa sehingga tangan itu tetap terurut. Bagaimana caranya? Anda menggeser kartu baru itu ke kiri melewati setiap kartu yang lebih besar daripadanya, lalu menjatuhkannya ke celah yang terbuka. Persis itulah **Gambar 2.10**.

Gambar 2.10 Cara pemain kartu. Tangan kiri memegang deretan terurut 2–4–5–10; tangan kanan baru saja mengambil kartu 7 dan akan menggesernya ke kiri melewati kartu 10 untuk mencapai tempatnya.

Dua ciri gambar itu menjadi dua ciri algoritmanya.

Pertama, **bagian yang terurut bertambah satu pada tiap langkah**. Sebelum Anda mengambil kartu 7, empat kartu sudah tertata; sesudah Anda menaruhnya, lima kartu tertata. Inilah metode *inkremental* di Subbab 2.15.1: perluas solusi yang benar untuk $j - 1$ benda menjadi solusi yang benar untuk j benda.

Kedua, **Anda menemukan tempatnya dengan menyapu dari kanan ke kiri, bukan dengan mencari**. Anda membandingkan kartu baru itu dengan kartu paling kanan di tangan Anda, lalu dengan kartu berikutnya, dan seterusnya. Anda berhenti begitu bertemu kartu yang tidak lebih besar daripada kartu Anda, sebab semua yang lebih ke kiri pasti lebih kecil lagi — tangan itu sudah terurut, jadi satu pembandingan yang gagal menjamin semua pembandingan sisanya juga akan gagal. Berhenti lebih awal itulah sebabnya kasus terbaik Insertion Sort jauh lebih baik daripada kasus terburuknya.

2.8.2 Algoritmanya dalam pseudokode

Pseudokode 2.2 adalah prosedurnya. Lariknya bernama A, panjangnya `A.length`, dan indeksnya berjalan dari 1 sampai `A.length` — kita menghitung mulai dari 1, sebagaimana yang dilakukan CLRS [1], sebab dengan begitu matematikanya lebih bersih. Subbab 2.8.4 menjelaskan cara memperoleh indeks berbasis-1 dalam C++, yang menghitung mulai dari 0.

Pseudokode 2.2 Insertion Sort dalam pseudokode yang dipakai di sepanjang buku ini, dengan indeks berbasis-1.

```
INSERTION-SORT(A)
1  for j = 2 to A.length          // mulai dari elemen ke-2
2      key = A[j]                 // ambil kartunya
3      i = j - 1                  // mulai membandingkan dari sebelah kiri
4      while i > 0 and A[i] > key // geser elemen yang lebih besar ke kanan
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key             // sisipkan kartunya
```

Mari kita telusuri ketujuh barisnya dengan kata-kata, sebab setiap subbab berikutnya merujuk kembali kepadanya.

Baris 1 menjalankan gelung luar. Peubah j menandai kartu yang hendak kita sisipkan. Kita mulai dari $j = 2$, bukan $j = 1$, sebab larik satu elemen $A[1..1]$ sudah terurut dan tidak ada yang perlu dikerjakan.

Baris 2 menyimpan $A[j]$ ke dalam sebuah peubah bernama `key`. Penyimpanan ini sangat penting: penggeseran pada baris 5 sebentar lagi akan menimpa $A[j]$, dan seandainya kita tidak menyimpan salinannya, kita akan kehilangan nilai yang justru hendak kita tempatkan. Pemula yang menghilangkan baris 2 lalu menulis `A[i+1] = A[j]` pada baris 7 akan menghasilkan larik yang penuh nilai kembar.

Baris 3 menetapkan i agar menunjuk elemen paling kanan dari bagian yang sudah terurut, yaitu $A[j-1]$.

Baris 4 adalah pengujian gelung dalam. Ia memuat dua syarat yang dirangkai dengan `and`, dan keduanya berarti. Syarat $i > 0$ mencegah kita melangkah keluar dari ujung kiri larik. Syarat $A[i] > key$ menghentikan kita begitu bertemu elemen yang tidak lebih besar daripada kuncinya — itulah jalan keluar dini dari Subbab 2.8.1. Urutan kedua syarat itu tidak boleh sembarangan: dalam C++, `&&` dinilai dari kiri ke kanan dan berhenti pada nilai salah yang pertama, jadi menulis `i > 0 && A[i] > key` menjamin kita tidak pernah membaca $A[0]$. Tukar keduanya, maka program itu membaca di luar batas.

Baris 5 menggeser satu elemen ke kanan, membuka sebuah celah. *Baris 6* melangkah ke kiri. Bersama-sama keduanya membentuk satu iterasi penggeseran.

Baris 7 menjatuhkan kunci ke dalam celah itu. Mengapa $A[i+1]$ dan bukan $A[i]$? Sebab gelungnya keluar *setelah* i diturunkan melewati elemen terakhir yang ia pindahkan, sehingga posisi i memuat sesuatu yang memang harus tetap berada di sebelah kiri kuncinya, dan posisi $i+1$ adalah lubangnya. Inilah kesalahan selisih-satu yang paling sering terjadi pada algoritma ini, dan Subbab 2.9 akan membuktikan bahwa $i+1$-lah yang benar.

2.8.3 Menelusuri algoritmanya dengan tangan

Mari kita urutkan $A = \langle 5,2,4,6,1,3 \rangle$ sampai tuntas dengan tangan. **Tabel 2.1** menelusurinya. Bacalah sambil menaruh lariknya di hadapan Anda, dan periksa tiap barisnya.

Tabel 2.1 Insertion Sort pada $A = \langle 5,2,4,6,1,3 \rangle$, satu iterasi gelung luar untuk tiap baris. Batas antara awalan yang terurut dan akhiran yang belum tersentuh bergeser satu tempat ke kanan pada setiap baris.

Langkah	Key	Tindakan	Hasil
$j = 2$	2	$2 < 5$, jadi geser 5 ke kanan. Sisipkan 2 di posisi 1.	[2,5,4,6,1,3]
$j = 3$	4	$4 < 5$, geser 5 ke kanan. $4 > 2$, berhenti. Sisipkan 4 di posisi 2.	[2,4,5,6,1,3]
$j = 4$	6	$6 > 5$, tak ada pemindahan. Sudah pada tempatnya.	[2,4,5,6,1,3]
$j = 5$	1	$1 < 6$, geser 6. $1 < 5$, geser 5. $1 < 4$, geser 4. $1 < 2$, geser 2. Sisipkan 1 di posisi 1.	[1,2,4,5,6,3]
$j = 6$	3	$3 < 6$, geser 6. $3 < 5$, geser 5. $3 < 4$, geser 4. $3 > 2$, berhenti. Sisipkan 3 di posisi 3.	[1,2,3,4,5,6]

Sekarang mari kita bentangkan baris kelima, $j = 5$, menjadi operasi satuan, satu operasi satu baris, sebab di baris itulah semua kerjanya terjadi dan sebuah ringkasan justru menyembunyikannya.

Pada awal iterasi ini lariknya adalah [2,4,5,6,1,3] dan empat masukan pertamanya sudah terurut.

1. Baris 2: $key \leftarrow A[5] = 1$. Lariknya tidak berubah; kita sekadar memegang salinan angka 1.
2. Baris 3: $i \leftarrow 4$.
3. Baris 4: uji. Apakah $i > 0$? $4 > 0$, ya. Apakah $A[4] > key$? $A[4] = 6$ dan $6 > 1$, ya. Masuk ke dalam gelung.

4. Baris 5: $A[5] \leftarrow A[4]$, sehingga lariknya menjadi [2,4,5,6,6,3]. Angka 1 tampak hilang, tetapi kita sudah menyimpannya di `key`.
5. Baris 6: $i \leftarrow 3$.
6. Baris 4: uji. $3 > 0$, ya. $A[3] = 5 > 1$, ya. Masuk lagi.
7. Baris 5: $A[4] \leftarrow A[3]$, menghasilkan [2,4,5,5,6,3].
8. Baris 6: $i \leftarrow 2$.
9. Baris 4: uji. $2 > 0$, ya. $A[2] = 4 > 1$, ya. Masuk lagi.
10. Baris 5: $A[3] \leftarrow A[2]$, menghasilkan [2,4,4,5,6,3].
11. Baris 6: $i \leftarrow 1$.
12. Baris 4: uji. $1 > 0$, ya. $A[1] = 2 > 1$, ya. Masuk lagi.
13. Baris 5: $A[2] \leftarrow A[1]$, menghasilkan [2,2,4,5,6,3].
14. Baris 6: $i \leftarrow 0$.
15. Baris 4: uji. Apakah $i > 0$? $0 > 0$ bernilai **salah**. Gelungnya berakhir. Perhatikan bahwa kita berhenti pada syarat yang *pertama*; seandainya kita menguji $A[0]$, kita akan membaca memori yang bukan milik kita.
16. Baris 7: $A[i + 1] = A[1] \leftarrow key = 1$, menghasilkan [1,2,4,5,6,3].

Enam belas baris untuk memindahkan satu kartu. Pengujian gelung `while` pada baris 4 dinilai lima kali: empat kali berhasil dan sekali gagal. Ingat angka itu — lima — sebab di Subbab 2.10 ia akan disebut t_5, dan seluruh analisis waktu jalan dibangun dari angka-angka semacam itu.

2.8.4 Insertion Sort dalam C++

Sekarang programnya. Ada satu keputusan yang perlu dijelaskan lebih dahulu. Pseudokode kita mengindeks mulai dari 1; C++ mengindeks mulai dari 0. Ada tiga jalan keluar: menulis ulang algoritmanya untuk indeks berbasis-0, menerjemahkan dalam kepala di setiap baris, atau mendeklarasikan lariknya satu elemen lebih panjang dan tidak pernah memakai elemen 0. Sepanjang bab ini kita mengambil jalan ketiga, sebab dengan begitu C++-nya dapat disandingkan baris demi baris dengan pseudokodenya, dan pembaca yang memeriksa program terhadap buktinya tidak perlu memegang sebuah terjemahan di kepalanya. Dalam kode produksi Anda tentu akan memakai indeks berbasis-0; Soal 2.3 meminta Anda melakukan perubahan itu.

Kode 2.1 Insertion Sort dalam C++, dengan pengindeksan berbasis-1 agar kodenya persis sejajar dengan pseudokodenya.

```
// Insertion Sort, ditulis agar sejajar baris demi baris dengan pseudokodenya.
// Lariknya berbasis-1: A[0] sengaja dibiarkan tak terpakai supaya indeks
// C++-nya persis sama dengan indeks yang dipakai pseudokode di buku ini.
#include <iostream>
#include <vector>
using namespace std;

void insertionSort(vector<int>& A, int n) {
    for (int j = 2; j <= n; ++j) {
        int key = A[j];          // kartu yang kita ambil
        int i = j - 1;           // mulai membandingkan dengan kartu di kirinya
        while (i > 0 && A[i] > key) {
            A[i + 1] = A[i];     // geser kartu yang lebih besar satu tempat ke kanan
            i = i - 1;
        }
        A[i + 1] = key;          // jatuhkan kartunya ke lubang yang tadi dibuka
    }
}

void print(const vector<int>& A, int n) {
    for (int i = 1; i <= n; ++i) cout << A[i] << (i < n ? ' ' : '\n');
}

int main() {
    vector<int> A = {0, 5, 2, 4, 6, 1, 3};   // A[0] tak terpakai
    int n = 6;
    cout << "before: "; print(A, n);
    insertionSort(A, n);
    cout << "after:  "; print(A, n);
    return 0;
}
```

Dikompilasi dengan `g++ -std=c++17 -O2` lalu dijalankan, **Kode 2.1** mencetak:

```
before: 5 2 4 6 1 3
after:  1 2 3 4 5 6
```

Itu jawaban yang sama dengan yang kita peroleh dengan tangan di **Tabel 2.1**, dan itulah paling sedikit yang pantas kita harapkan.

Namun program yang hanya mencetak jawaban tidak mengajarkan apa pun tentang *bagaimana* jawaban itu muncul. **Kode 2.2** menambahkan satu baris keluaran untuk tiap iterasi gelung luar, sehingga mesinnya menirukan telusur tangan pada **Tabel 2.1**.

Kode 2.2 Insertion Sort yang dilengkapi pencatat agar mencetak telusurnya sendiri, satu baris untuk tiap iterasi gelung luar.

```
// Insertion Sort dengan telusur tercetak: satu baris untuk tiap iterasi gelung
// luar, menirukan dengan mesin persis telusur tangan yang dikerjakan di teks.
#include <iostream>
#include <vector>
using namespace std;

void show(const vector<int>& A, int n) {
    cout << '[';
    for (int i = 1; i <= n; ++i) cout << A[i] << (i < n ? "," : "");
    cout << ']';
}

int main() {
    vector<int> A = {0, 5, 2, 4, 6, 1, 3};
    int n = 6;
    cout << "start        "; show(A, n); cout << '\n';
    for (int j = 2; j <= n; ++j) {
        int key = A[j];
        int i = j - 1;
        int shifts = 0;
        while (i > 0 && A[i] > key) { A[i + 1] = A[i]; --i; ++shifts; }
        A[i + 1] = key;
        cout << "j=" << j << " key=" << key
             << " shifts=" << shifts
             << " insert at " << (i + 1) << "  ";
        show(A, n);
        cout << '\n';
    }
    return 0;
}
```

Keluaran sesungguhnya adalah:

```
start        [5,2,4,6,1,3]
j=2 key=2 shifts=1 insert at 1  [2,5,4,6,1,3]
j=3 key=4 shifts=1 insert at 2  [2,4,5,6,1,3]
j=4 key=6 shifts=0 insert at 4  [2,4,5,6,1,3]
j=5 key=1 shifts=4 insert at 1  [1,2,4,5,6,3]
j=6 key=3 shifts=3 insert at 3  [1,2,3,4,5,6]
```

Bandingkan keluaran itu baris demi baris dengan **Tabel 2.1**. Kolom `Hasil` pada tabel dan larik berkurung siku pada program cocok pada kelima barisnya; cacah `shifts` cocok dengan banyaknya kata "geser" pada kolom `Tindakan` di tabel itu — satu, satu, nol, empat, tiga. Telusur tangan dan mesin sepakat. Kesepakatan itu bukan sekadar formalitas: setiap kali Anda menemukan sebuah algoritma, telusurilah dengan tangan *dan* pasangi pencatat, lalu buatlah keduanya cocok. Di tempat keduanya berselisih, salah satunya mengandung kutu, dan biasanya berharga untuk mengetahui yang mana.

2.9 Membuktikan Insertion Sort Benar: Invarian Gelung

> ❓ **PERTANYAAN:** Kita sudah menyaksikan Insertion Sort bekerja pada satu larik berisi enam bilangan. Bagaimana kita *membuktikan* bahwa ia menghasilkan keluaran terurut untuk **semua** masukan — untuk setiap larik, dengan panjang berapa pun, berisi kombinasi bilangan apa pun?

Anda tidak dapat menjawabnya dengan pengujian. Masukannya tak berhingga banyaknya; yang dapat Anda coba berhingga banyaknya. Perkakas yang menutup jurang itu adalah **invarian gelung**.

2.9.1 Apa itu invarian gelung

> **DEFINISI.** Sebuah **invarian gelung** (*loop invariant*) adalah pernyataan tentang keadaan komputasi yang bernilai benar tepat sebelum setiap iterasi gelung dimulai, dan benar kembali tepat setelah gelung itu selesai.

Kata "invarian" berarti *tak berubah*: kebenaran pernyataan itu tidak berubah selama gelungnya berjalan, meskipun data yang dibicarakannya berubah terus-menerus. Invarian yang baik adalah sebuah kalimat yang menjawab pertanyaan *"apa yang sejauh ini sudah dicapai gelung ini?"*

Untuk memakai invarian sebagai bukti, Anda harus menegakkan tiga hal.

1. **Inisialisasi.** Invariannya benar sebelum iterasi yang pertama sekali dimulai.
2. **Pemeliharaan.** *Jika* invariannya benar sebelum suatu iterasi, *maka* ia masih benar sebelum iterasi berikutnya. Dengan kata lain, satu putaran gelung melestarikannya.
3. **Terminasi.** Ketika gelungnya akhirnya berhenti, invarian itu — digabung dengan alasan mengapa gelungnya berhenti — memberi kita sifat yang sebenarnya ingin kita buktikan.

Dua langkah pertama persis sama dengan induksi matematis. Inisialisasi adalah kasus dasarnya; pemeliharaan adalah langkah induktifnya. Jika invariannya berlaku di awal dan tiap iterasi melestarikannya, maka menurut induksi ia berlaku di awal iterasi 1, iterasi 2, iterasi 3, dan seterusnya untuk berapa pun banyaknya iterasi. Terminasi adalah langkah tambahan yang tidak diberikan oleh induksi semata: di situlah Anda menguangkan invarian menjadi pernyataan tentang jawabannya.

> **Mengapa langkah ketiga tidak boleh dilewati.** Mudah sekali menyatakan invarian yang benar tetapi tak berguna. "Pada awal tiap iterasi, j adalah bilangan bulat" sungguh invarian, dan ia tidak memberi tahu apa pun tentang pengurutan. Invarian yang berguna adalah yang kebenarannya *pada saat gelung berhenti* justru merupakan hal yang hendak Anda buktikan. Merancang invariannya adalah bagian yang sukar dan kreatif; memeriksanya adalah pekerjaan mekanis.

2.9.2 Invarian untuk Insertion Sort

Inilah invariannya.

> **INVARIAN GELUNG (Insertion Sort).** Pada awal tiap iterasi gelung `for` di baris 1, sublarik $A[1 \ldots j-1]$ terdiri atas elemen-elemen yang semula berada di $A[1 \ldots j-1]$, tetapi dalam keadaan terurut.

Bacalah dua kali. Ia memuat dua klausa, dan keduanya diperlukan, persis karena alasan yang sudah diperingatkan **Gambar 2.9**: awalannya harus **terurut**, dan ia harus memuat **elemen yang sama** dengan yang semula ada di sana. Buang klausa kedua, maka invariannya akan dipenuhi juga oleh algoritma yang menimpa awalan itu dengan nol.

Mari kita periksa invariannya terhadap telusur kita. Pada awal iterasi $j = 5$ di **Tabel 2.1**, lariknya adalah [2,4,5,6,1,3]. Sublarik $A[1..4] = [2,4,5,6]$ terurut: ✓. Dan $\{2,4,5,6\}$ persis himpunan nilai yang semula menempati keempat posisi itu pada larik awal $[5,2,4,6,\ldots]$, yaitu 5,2,4,6: ✓. Kedua klausanya berlaku.

2.9.3 Inisialisasi

Kita harus menunjukkan bahwa invariannya benar sebelum iterasi pertama, yang dimulai dengan $j = 2$.

Dengan menyulihkan $j = 2$ ke dalam invariannya, klaimnya adalah tentang sublarik $A[1 \dots 2 - 1] = A[1 \dots 1]$. Sublarik itu memuat tepat satu elemen, yaitu $A[1]$.

Apakah ia terurut? Barisan berisi satu elemen terurut secara hampa: keterurutan menuntut $a_1 \leq a_2 \leq \cdots$, dan dengan hanya satu elemen tidak ada pasangan elemen bersebelahan untuk dibandingkan, sehingga tidak ada syarat yang dapat gagal. Apakah ia penataan ulang dari elemen yang semula ada di $A[1..1]$? Ya — belum ada apa pun yang dijalankan, jadi $A[1]$ masih memuat persis nilai aslinya.

Kedua klausanya berlaku sebelum iterasi pertama. ✓

> **Catatan tentang kebenaran hampa.** Pemula kerap merasa tidak nyaman bahwa "senarai satu elemen itu terurut" terasa seperti mengakali. Ia bukan akal-akalan. Definisi *terurut* adalah "setiap pasangan bersebelahan berada dalam urutan tak-menurun". Pada senarai satu elemen tidak ada satu pun pasangan bersebelahan, dan pernyataan tentang semua anggota koleksi kosong otomatis benar — tidak tersedia contoh penyangkalnya. Ini alasan yang sama dengan mengapa "setiap unikorn di ruangan ini berwarna ungu" itu benar.

2.9.4 Pemeliharaan

Kita harus menunjukkan bahwa jika invariannya berlaku pada awal sebuah iterasi, ia berlaku pula pada awal iterasi berikutnya. Jadi andaikan, untuk suatu nilai j tertentu dengan $2 \leq j \leq n$, bahwa $A[1 \dots j - 1]$ adalah penataan ulang terurut dari isi aslinya. Kita harus menunjukkan bahwa setelah tubuh gelungnya berjalan, $A[1 \dots j]$ adalah penataan ulang terurut dari isi *asli miliknya* — sebab pada awal iterasi berikutnya indeks gelungnya $j + 1$ dan invariannya lalu berbicara tentang $A[1 \dots (j + 1) - 1] = A[1 \dots j]$.

Tubuh gelungnya berjalan begini. Baris 2 menyalin $A[j]$ ke `key`, jadi nilainya aman. Baris 3 menetapkan $i = j - 1$. Lalu gelung `while` dalam berjalan.

Untuk menalar tentang gelung dalam, kita perlu invarian untuk *gelung itu* juga. Gelung bersarang, maka buktinya pun bersarang.

> **INVARIAN GELUNG DALAM.** Pada awal tiap iterasi gelung `while` di baris 4, semua hal berikut benar: (a) posisi $A[1 \dots i]$ memuat nilainya dari awal iterasi luar, tanpa berubah; (b) posisi $A[i + 2 \dots j]$ memuat nilai yang semula berada di $A[i + 1 \dots j - 1]$, masing-masing telah berpindah satu tempat ke kanan; (c) posisi $A[i + 1]$ adalah sebuah *lubang* — apa pun isinya adalah kembaran basi yang bebas kita timpa; dan (d) setiap nilai di $A[i + 2 \dots j]$ benar-benar lebih besar daripada `key`.

Inisialisasi gelung dalam. Saat masuk, $i = j - 1$. Klausa (a) menyatakan $A[1..j - 1]$ tak tersentuh — benar, kita belum mengubah apa pun. Klausa (b) berbicara tentang $A[j + 1 \dots j]$, sebuah jangkauan kosong, jadi ia benar secara hampa. Klausa (c) menyatakan $A[j]$ adalah lubang, dan itu benar sebab kita sudah menyalin nilainya ke `key` dan tidak lagi memerlukannya di sana. Klausa (d) berbicara tentang jangkauan kosong yang sama dan bersifat hampa. ✓

Pemeliharaan gelung dalam. Andaikan keempat klausa berlaku dan pengujian gelungnya berhasil, sehingga $i > 0$ dan $A[i] > key$. Baris 5 menyalin $A[i]$ ke lubang di $A[i + 1]$; nilai $A[i]$ sekarang muncul di posisi $i + 1$, satu tempat di kanan tempatnya semula, yang memperluas klausa (b) sebanyak satu elemen, dan karena $A[i] > key$ elemen baru itu memenuhi klausa (d) juga. Posisi i kini menjadi kembaran basi, artinya lubang yang baru. Baris 6 menurunkan i, sehingga klausa (c) kembali menyebut lubangnya dengan benar dan klausa (a) menyusut persis sebanyak satu elemen yang baru saja kita pindahkan. Keempat klausanya berlaku kembali. ✓

Terminasi gelung dalam. Gelungnya berhenti karena salah satu dari dua alasan.

Alasan 1: $i = 0$. Maka klausa (d) menyatakan setiap elemen $A[2 \dots j]$ melampaui `key`, dan klausa (c) menyatakan $A[1]$ adalah lubangnya. Baris 7 menulis `key` ke $A[1]$. Hasilnya, $A[1 \dots j]$ adalah `key` yang diikuti $A[2..j]$, yang menurut klausa (b) dan invarian luar adalah $A[1..j-1]$ terurut yang lama, dalam urutan aslinya, yang telah digeser ke kanan. Barisan terurut yang semua anggotanya melampaui `key`, yang didahului oleh `key`, adalah barisan terurut. ✓

Alasan 2: $A[i] \leq key$ *untuk suatu* $i > 0$. Maka klausa (a) menyatakan $A[1 \dots i]$ adalah i elemen pertama dari awalan terurut yang lama, sehingga setiap satunya $\leq A[i] \leq key$. Klausa (d) menyatakan setiap elemen $A[i+2 \dots j]$ bernilai $> key$. Baris 7 menulis `key` ke lubang di $A[i+1]$. Membaca dari kiri ke kanan, kita sekarang memperoleh: elemen-elemen yang semuanya $\leq key$, lalu `key`, lalu elemen-elemen yang semuanya $> key$, dengan masing-masing dari kedua blok terluar itu terurut di dalam dirinya sendiri sebab keduanya adalah potongan bersambung dari awalan yang tadinya sudah terurut. Jadi seluruh $A[1 \dots j]$ terurut. ✓

Pada kedua kasus, baris 7 menulis ke $A[i+1]$, dan pada kedua kasus $A[i+1]$ persis adalah lubangnya. Itu menuntaskan pertanyaan selisih-satu yang diajukan di Subbab 2.8.2: kuncinya diletakkan di $i+1$, tidak pernah di i, sebab baris 6 selalu memindahkan i satu tempat melewati elemen terakhir yang digesernya.

Terakhir, tidak ada nilai yang diciptakan atau dimusnahkan: setiap operasi tulis entah menyalin sebuah elemen ke sel tetangganya atau menuliskan kembali `key` yang disimpan, sehingga $A[1 \dots j]$ adalah permutasi dari isi aslinya. Kedua klausa invarian luar kini berlaku untuk $j+1$. Pemeliharaan terbukti. ✓

2.9.5 Terminasi

Gelung luar menaikkan j sebanyak 1 setiap kali dan berhenti ketika j melampaui n, yaitu ketika $j = n+1$. (Ia pasti berhenti: j mulai dari 2, naik ketat sebanyak 1 setiap iterasi, dan pengujiannya terhadap batas tetap n, jadi setelah paling banyak $n-1$ iterasi pengujiannya gagal. Insertion Sort memenuhi sifat keberhinggaan di Subbab 2.4.3.)

Sulihkan $j = n+1$ ke dalam invariannya. Ia berbunyi: $A[1 \dots (n+1)-1] = A[1 \dots n]$ terdiri atas elemen-elemen yang semula berada di $A[1 \dots n]$, dalam keadaan terurut.

Tetapi $A[1 \dots n]$ adalah seluruh lariknya. Jadi seluruh larik memuat elemen aslinya dalam keadaan terurut. Itu persis spesifikasi keluaran problem PENGURUTAN — kedua klausanya. Insertion Sort benar. ■

> ✓ **WAWASAN KUNCI:** Invarian gelung adalah teknik pembuktian baku untuk algoritma iteratif, dan Anda akan memakainya di setiap bab dari sini sampai akhir buku. Anggaplah invarian sebagai jawaban atas pertanyaan: *"Apa yang selalu benar tentang keadaan komputasi pada titik ini?"* Tuliskan kalimat itu sebelum Anda menulis gelungnya, dan gelung itu biasanya akan benar sejak percobaan pertama.

> ⚠ **RENUNGKAN:** Terminasi adalah langkah yang di sini memerlukan kerja paling sedikit, dan ia adalah langkah yang paling menentukan. Perhatikan bahwa kita harus menggabungkan *dua* fakta: invariannya, dan alasan gelungnya berhenti ($j = n+1$). Invarian saja tidak pernah membuktikan apa pun tentang jawabannya; yang membuktikan adalah invarian *yang dinilai pada syarat keluarnya*.

2.10 Menghitung Biaya Insertion Sort

Kebenaran sudah tuntas. Sekarang: berapa lama ia memakan waktu?

Kita terapkan metode Subbab 2.7 persis seperti adanya. Berikan biaya konstan pada tiap baris, hitung berapa kali tiap baris berjalan, lalu jumlahkan hasil kalinya.

2.10.1 Tabel biaya dan cacah

Ada satu besaran yang tidak dapat kita pastikan di muka. Gelung `while` dalam di baris 4 berjalan dengan cacah yang berbeda untuk tiap j, bergantung pada datanya. Maka kita beri ia nama.

> **DEFINISI.** Misalkan t_j menyatakan berapa kali **pengujian** gelung `while` di baris 4 dijalankan untuk nilai j tersebut.

Bacalah definisi itu secara persis, sebab segala yang mengikutinya bergantung padanya. Ia menghitung *pengujian*, bukan pengujian yang *berhasil*. Untuk sembarang j, pengujiannya dinilai sejumlah kali, dan semuanya kecuali yang terakhir berhasil lalu mengirim kita masuk ke tubuh gelung. Jadi tubuh gelungnya — baris 5 dan 6 — dijalankan $t_j - 1$ kali. Satu selisih-satu itulah alasan mengapa tabel di bawah ini memuat t_j di satu baris dan $t_j - 1$ di dua baris berikutnya.

Kembalilah ke telusur terbentang di Subbab 2.8.3. Untuk $j = 5$ kita menilai pengujiannya lima kali: empat berhasil dan satu gagal. Jadi $t_5 = 5$, dan baris 5 serta 6 masing-masing berjalan $t_5 - 1 = 4$ kali — yang cocok dengan `shifts=4` pada keluaran programnya.

Setelah t_j terdefinisi, inilah tabel lengkapnya, yang sekarang akan kita pakai untuk menurunkan segalanya.

Tabel 2.2 Biaya tiap baris Insertion Sort dan berapa kali baris itu berjalan. Lambang t_j adalah berapa kali pengujian baris 4 dinilai untuk nilai j tersebut.

Baris	Biaya	Cacah eksekusi
1: `for j = 2 to A.length`	c_1	n
2: `key = A[j]`	c_2	$n - 1$
3: `i = j - 1`	c_3	$n - 1$
4: `while i > 0 and A[i] > key`	c_4	$\sum_{j=2}^{n} t_j$
5: `A[i+1] = A[i]`	c_5	$\sum_{j=2}^{n} (t_j - 1)$
6: `i = i - 1`	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7: `A[i+1] = key`	c_7	$n - 1$

Tiap cacah pantas memperoleh satu kalimat pembenaran, sebab cacah yang tak dapat Anda benarkan adalah cacah yang Anda terka.

Baris 1 berjalan n kali. Kepala gelungnya dinilai sekali untuk tiap nilai $j = 2,3, \ldots, n$ yang masuk ke tubuh gelung — itu $n - 1$ penilaian — ditambah satu penilaian terakhir dengan $j = n + 1$ yang gagal lalu keluar. Totalnya n. "Satu pengujian tambahan untuk mengetahui bahwa gelungnya sudah selesai" ini adalah kenyataan umum tentang gelung, dan dari sanalah datang selisih n terhadap $n - 1$ antarbaris.

Baris 2, 3, dan 7 berjalan $n-1$ kali. Ketiganya berada di dalam tubuh gelung, yang dimasuki sekali untuk tiap j dari 2 sampai n. Itu $n-1$ nilai.

Baris 4 berjalan $\sum_{j=2}^{n} t_j$ kali. Untuk tiap j, pengujiannya dinilai t_j kali menurut definisi; jumlahkan untuk semua j.

Baris 5 dan 6 berjalan $\sum_{j=2}^{n}(t_j-1)$ kali. Untuk tiap j, tubuh gelungnya dimasuki sekali untuk tiap pengujian yang *berhasil*, dan itu satu lebih sedikit daripada cacah pengujiannya.

Lambang $\sum$ dibaca "jumlah dari". Ungkapan $\sum_{j=2}^{n} t_j$ berarti: ambil t_2, lalu t_3, lalu t_4, dan seterusnya sampai t_n, lalu jumlahkan semuanya. Jika $n=6$ dan nilai-nilai t-nya adalah 2,2,1,5,4, maka $\sum_{j=2}^{6} t_j = 2+2+1+5+4 = 14$.

2.10.2 Rumus umumnya

Menjumlahkan biaya × cacah ke bawah menurut **Tabel 2.2** memberikan waktu jalan untuk **sembarang** masukan:

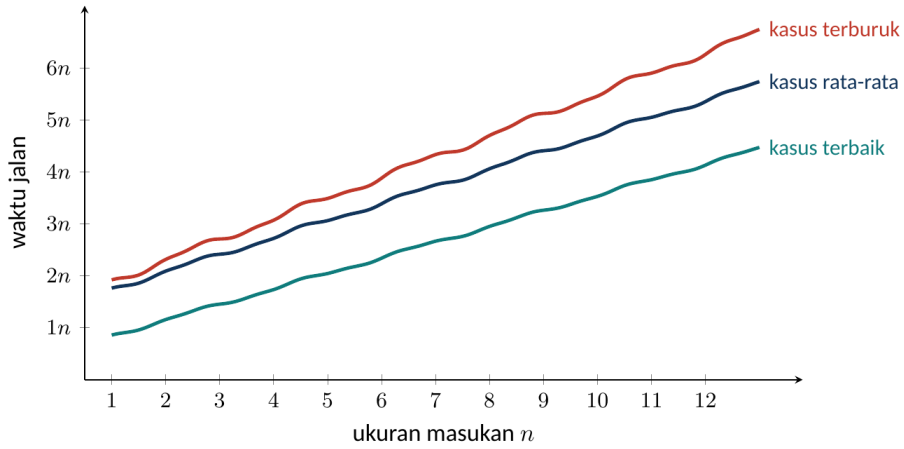
$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j-1) + c_6 \sum_{j=2}^{n}(t_j-1) + c_7(n-1). \tag{2.2}$$

Perhatikan apa saja yang terkandung dalam Persamaan (2.2). Konstanta $c_1, \ldots, c_7$ bergantung pada kompilator dan prosesor; kita tidak akan pernah memerlukan nilainya. Besaran n adalah ukuran masukan. Dan bilangan $t_2, t_3, \ldots, t_n$ bergantung pada **larik tertentu mana yang diberikan kepada kita**. Dua larik dengan panjang sama dapat menghasilkan himpunan nilai t_j yang sama sekali berbeda, dan karena itu waktu jalan yang sama sekali berbeda pula.

> ✓ **WAWASAN KUNCI:** $T(n)$ tidak ditentukan oleh n semata. Ungkapan di atas berlubang, dan lubangnya berbentuk $\sum t_j$, dan hanya masukan yang sesungguhnya yang dapat mengisinya. Inilah sebabnya subbab berikutnya harus terbelah menjadi tiga kasus: kita harus mengatakan lebih dahulu masukan yang *mana* yang sedang kita bicarakan sebelum dapat merampungkan hitungannya.

2.11 Kasus Terbaik, Kasus Terburuk, dan Kasus Rata-rata

Algoritma yang sama pada cacah elemen yang sama dapat menjadi cepat atau lambat bergantung pada penataan elemen itu. **Gambar 2.11** memperlihatkan bentuk keadaannya: untuk tiap ukuran masukan, waktu jalannya bukan satu titik melainkan sebuah pita, dan pita itu kita ringkas dengan tepi atasnya, tepi bawahnya, dan bagian tengahnya.



Gambar 2.11 Untuk suatu ukuran masukan terdapat banyak kemungkinan masukan dan karena itu banyak pula kemungkinan waktu jalan. Kasus terburuk adalah selubung atasnya, kasus terbaik selubung bawahnya, dan kasus rata-rata terletak di antara keduanya.

2.11.1 Kasus terbaik: larik yang sudah terurut

Andaikan larik masukannya sudah dalam urutan tak-menurun, misalnya $A = \langle 1,2,3,4,5,6 \rangle$.

Tinjau sembarang j. Kita tetapkan $key = A[j]$ dan $i = j - 1$, lalu menilai pengujian baris 4. Syarat pertama, $i > 0$, benar sebab $j \geq 2$. Syarat kedua menanyakan apakah $A[j-1] > key = A[j]$. Tetapi lariknya terurut, sehingga $A[j-1] \leq A[j]$, dan syarat itu **salah**. Pengujiannya dinilai sekali lalu langsung gagal. Karena itu

$$t_j = 1 \quad \text{untuk setiap } j,$$

dan dengan demikian $t_j - 1 = 0$: baris 5 dan 6 sama sekali tidak pernah dijalankan.

Sulihkan ke dalam rumus umum Persamaan (2.2). Kedua penjumlahannya menjadi

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} 1 = n - 1, \qquad \sum_{j=2}^{n} (t_j - 1) = 0,$$

di mana kesamaan yang pertama berlaku karena kita menjumlahkan angka 1 dengan dirinya sendiri sekali untuk tiap dari $n - 1$ nilai $j = 2,3, \ldots, n$.

Jadi

$$\begin{aligned} &T(n) \\ &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 \cdot 0 + c_6 \cdot 0 + c_7(n-1). \end{aligned}$$

Kumpulkan sukunya. Setiap tanda kurungnya adalah $(n-1) = n - 1$, sehingga

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_3 + c_4 + c_7)(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)\, n - (c_2 + c_3 + c_4 + c_7). \end{aligned}$$

Tuliskan $a = c_1 + c_2 + c_3 + c_4 + c_7$ dan $b = -(c_2 + c_3 + c_4 + c_7)$. Keduanya konstanta — jumlah beberapa konstanta tetaplah konstanta. Maka

$$T(n) = an + b. \tag{2.3}$$

> ✓ **WAWASAN KUNCI:** Pada kasus terbaik, waktu jalan Insertion Sort Persamaan (2.3) adalah fungsi **linear** dari n. Kita akan menuliskannya $\Theta(n)$ di Bab 3. Secara intuitif: algoritmanya menyusuri lariknya sekali, memastikan pada tiap langkah bahwa elemen barunya sudah berada di tempatnya, dan sama sekali tidak melakukan penggeseran. Ia mustahil lebih cepat daripada itu, sebab sekurang-kurangnya ia harus melihat setiap elemen untuk dapat mengetahui bahwa lariknya terurut.

2.11.2 Kasus terburuk: larik dengan urutan terbalik

Sekarang andaikan masukannya berada dalam urutan menurun ketat, misalnya $A = \langle 6,5,4,3,2,1 \rangle$.

Tinjau sembarang j. Setiap elemen di sebelah kiri posisi j lebih besar daripada $A[j]$ — itulah arti "menurun". Jadi pengujian baris 4 berhasil untuk $i = j - 1$, untuk $i = j - 2$, dan seterusnya sampai turun ke $i = 1$. Ia baru gagal ketika i mencapai 0, pada syarat pertamanya.

Hitung penilaiannya. Pengujiannya berhasil untuk $i = j - 1, j - 2, \ldots, 1$, yaitu $j - 1$ keberhasilan, lalu gagal sekali. Karena itu

$$t_j = (j - 1) + 1 = j.$$

Periksalah terhadap definisinya pada kasus kecil. Untuk $j = 2$ pengujiannya berjalan dua kali: sekali dengan $i = 1$ (berhasil, sebab $A[1] > A[2]$), sekali dengan $i = 0$ (gagal). Dan memang $t_2 = 2$. ✓

Sekarang kita harus menilai penjumlahannya. Pertama,

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = 2 + 3 + 4 + \cdots + n.$$

Ini adalah **deret aritmetika**, dan kita memerlukan bentuk tertutupnya. Alih-alih mengutipnya, turunkanlah — kiatnya layak diketahui dan dikaitkan dengan Gauss muda. Tuliskan jumlahnya maju dan mundur, lalu jumlahkan kedua barisnya suku demi suku:

$$\begin{array}{rcccccccc} S & = & 2 & + & 3 & + & \cdots & + & n \\ S & = & n & + & (n-1) & + & \cdots & + & 2 \\ \hline 2S & = & (n+2) & + & (n+2) & + & \cdots & + & (n+2) \end{array}$$

Setiap kolom berjumlah $n + 2$, sebab pada baris atas sukunya naik satu demi satu ketika Anda bergerak ke kanan dan pada baris bawah sukunya turun satu demi satu, sehingga jumlah tiap kolomnya tidak berubah. Ada berapa kolom? Satu untuk tiap nilai j dari 2 sampai n, yaitu $n - 1$ kolom. Karena itu

$$2S = (n - 1)(n + 2), \qquad S = \frac{(n - 1)(n + 2)}{2}.$$

Mari kita bentangkan lalu periksa terhadap bentuk yang lebih dikenal. Dengan mengalikan keluar, $(n - 1)(n + 2) = n^2 + n - 2$, sehingga

$$S = \frac{n^2 + n - 2}{2} = \frac{n(n+1)}{2} - 1.$$

Bentuk yang kedua itulah yang perlu diingat, dan ia punya pembacaan yang mudah: jumlah $1 + 2 + \cdots + n$ sama dengan $n(n+1)/2$, sedangkan jumlah kita mulai dari 2 dan bukan dari 1, jadi kita kurangi angka 1 yang hilang itu. Periksa kewarasannya dengan $n = 6$: jumlah kita adalah $2 + 3 + 4 + 5 + 6 = 20$, dan rumusnya memberi $6 \cdot 7/2 - 1 = 21 - 1 = 20$. ✓

Jadi

$$\sum_{j=2}^{n} t_j = \frac{n(n+1)}{2} - 1. \tag{2.4}$$

Penjumlahan kedua. Karena $t_j - 1 = j - 1$,

$$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1) = 1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2}, \tag{2.5}$$

dan langkah terakhirnya adalah rumus Gauss yang sama yang diterapkan pada $1 + 2 + \cdots + m$ dengan $m = n - 1$, yang memberi $m(m+1)/2 = (n-1)n/2$. Periksa dengan $n = 6$: $1 + 2 + 3 + 4 + 5 = 15$, dan $6 \cdot 5/2 = 15$. ✓

Sekarang sulihkan keduanya, Persamaan (2.4) dan Persamaan (2.5), ke dalam rumus umum Persamaan (2.2):

$$\begin{aligned} &T(n) \\ &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\frac{n(n-1)}{2} \\ &\qquad + c_6\frac{n(n-1)}{2} + c_7(n-1). \end{aligned}$$

Bentangkan bagian kuadratnya. Dengan memakai $\frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$ dan $\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$:

$$\begin{aligned} &T(n) \\ &= c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_4\frac{n^2}{2} + c_4\frac{n}{2} - c_4 + c_5\frac{n^2}{2} - c_5\frac{n}{2} + c_6\frac{n^2}{2} \\ &\qquad - c_6\frac{n}{2} + c_7 n - c_7. \end{aligned}$$

Kumpulkan menurut pangkat n. Suku-suku n^2:

$$\left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2.$$

Suku-suku n:

$$\left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right) n.$$

Suku-suku konstantanya:

$$-(c_2 + c_3 + c_4 + c_7).$$

Dengan menamai ketiga tanda kurung itu a, b, dan c — semuanya konstanta, sebab semuanya hanya dibangun dari konstanta $c_1, \ldots, c_7$ — kita telah menurunkan

$$\boxed{T(n) = an^2 + bn + c} \tag{2.6}$$

Dalam Persamaan (2.6), $a = (c_4 + c_5 + c_6)/2 > 0$. Waktu jalan kasus terburuk Insertion Sort adalah fungsi **kuadratik** dari n. Dalam bahasa Bab 3 ia $\Theta(n^2)$, dan penurunan di ataslah, bukan kata siapa pun, yang menjadi alasannya.

> ✓ **WAWASAN KUNCI:** Kasus terburuk Insertion Sort: $T(n)$ bersifat kuadratik. Artinya dalam praktik sungguh kejam. Mengurutkan larik terbalik berisi satu juta elemen memerlukan sekitar $10^{12}/2$ operasi penggeseran. Pada angka yang sangat optimistis, 10^9 operasi per detik, itu kira-kira 500 detik — lebih dari delapan menit — untuk mengurutkan senarai yang ditangani `std::sort` dalam waktu jauh di bawah satu detik.

> ⚠ **RENUNGKAN:** Perhatikan apa yang *tidak* dipakai oleh penurunan tadi: nilai c_1 sampai c_7. Apa pun kompilator Anda, apa pun prosesor Anda, jawabannya berbentuk $an^2 + bn + c$. Mesin mengubah a; ia tidak mengubah n^2. Itulah sebabnya analisis ini dapat kita kerjakan di atas kertas, bertahun-tahun sebelum programnya ditulis, pada perangkat keras yang belum ada.

2.11.3 Kasus rata-rata

Terbaik dan terburuk adalah dua ujung yang ekstrem. Apa yang terjadi pada larik yang *biasa*?

Untuk mengajukan pertanyaan itu secara persis, kita harus mengatakan apa arti "biasa", sebab rata-rata adalah rata-rata atas suatu sebaran dan anggapan yang berbeda memberikan jawaban yang berbeda. Anggapan bakunya adalah: **semua $n!$ penataan masukan sama mungkinnya.**

Tetapkan sebuah nilai j. Elemen $key = A[j]$ sedang disisipkan ke dalam awalan terurut $A[1 \ldots j-1]$, yang memuat $j-1$ elemen. Dengan anggapan kita, `key` sama mungkinnya untuk menempati salah satu dari j posisi yang tersedia: sebelum semua $j-1$ elemen itu, di antara yang pertama dan yang kedua, dan seterusnya, sampai sesudah semuanya. Cacah pengujian gelung dalam, t_j, ditentukan oleh posisi yang mana itu — mendarat di ujung paling kanan memberi $t_j = 1$, mendarat di ujung paling kiri memberi $t_j = j$.

Mari kita buat "secara rata-rata" itu persis. Jika `key` mendarat k tempat dari ujung kanan awalan itu ($k = 0,1, \ldots, j-1$), pengujian gelungnya berhasil k kali lalu gagal sekali, sehingga $t_j = k + 1$. Masing-masing dari j tempat pendaratan itu sama mungkinnya, jadi t_j sama mungkinnya bernilai salah satu dari $1,2, \ldots, j$, dan rata-ratanya adalah jumlah nilai-nilai itu dibagi banyaknya. Menurut rumus Gauss,

$$\text{rata-rata } t_j = \frac{1}{j} \cdot \frac{j(j+1)}{2} = \frac{j+1}{2} \approx \frac{j}{2}.$$

Dengan kata-kata: secara rata-rata kuncinya menempuh kira-kira separuh panjang awalan yang terurut.

Menyulihkannya ke dalam penjumlahan yang pertama, dengan memakai $\sum_{j=2}^{n} 1 = n - 1$:

$$\sum_{j=2}^{n} \frac{j+1}{2} = \frac{1}{2}\left(\sum_{j=2}^{n} j + (n-1)\right) = \frac{1}{2}\left(\frac{n(n+1)}{2} - 1 + n - 1\right) = \frac{n^2 + 3n - 4}{4} \approx \frac{n^2}{4}.$$

(Periksa dengan $n = 6$: rata-rata cacah pengujiannya $\frac{3}{2}, \frac{4}{2}, \frac{5}{2}, \frac{6}{2}, \frac{7}{2}$, yang jumlahnya 12.5, dan $(36 + 18 - 4)/4 = 12.5$.) Dengan cara yang sama, penjumlahan kedua secara rata-rata bernilai $\sum_{j=2}^{n} \frac{j-1}{2} = \frac{1}{2} \cdot \frac{n(n-1)}{2} \approx \frac{n^2}{4}$. Jadi baris 4, 5, dan 6 masing-masing berjalan sekitar $n^2/4$ kali pada masukan yang lazim, berhadapan dengan sekitar $n^2/2$ kali pada kasus terburuk: tepat separuhnya.

Dengan memasukkan cacah-cacah ini ke dalam Persamaan (2.2) dan mengumpulkan sukunya persis seperti di Subbab 2.11.2, koefisien n^2 menjadi $(c_4 + c_5 + c_6)/4$, yaitu separuh a kasus terburuk:

$$T(n) \approx \frac{a}{2} n^2 + b'n + c' \tag{2.7}$$

dengan a yang sama seperti dalam Persamaan (2.6), sedangkan b' dan c' adalah konstanta lain. Jadi kasus rata-rata Persamaan (2.7) **juga kuadratik** — kira-kira separuh kasus terburuk, tetapi dengan n^2 yang sama.

> ✓ **WAWASAN KUNCI:** Kasus rata-rata Insertion Sort memiliki *orde pertumbuhan* yang sama dengan kasus terburuknya: keduanya kuadratik. Memperkecil konstantanya menjadi separuh tidak menolong Anda pada skala besar. Jika delapan menit terlalu lambat untuk sejuta elemen, empat menit pun terlalu lambat. Hanya perubahan laju pertumbuhannya — dari n^2 menjadi $n\lg n$ — yang sungguh menyelesaikan persoalan itu, dan itulah yang dilakukan Merge Sort di Bab 3.

2.11.4 Menguji analisis itu pada mesin

Segala yang kita kerjakan sejauh ini dilakukan di atas kertas. Mari kita buktikan. **Kode 2.3** mengimplementasikan Insertion Sort dengan sebuah pencacah yang dinaikkan pada setiap penilaian pengujian baris 4 — artinya, ia mengukur $\sum_{j=2}^{n} t_j$ secara langsung — lalu menjalankannya pada masukan menaik, menurun, dan yang dikocok secara acak.

Kode 2.3 Menghitung $\sum_{j=2}^{n} t_j$ untuk masukan menaik, menurun, dan acak, bersanding dengan nilai kasus terburuk yang diramalkan.

```
// Menghitung kerja yang benar-benar dilakukan Insertion Sort, pada tiga kasus
// klasiknya. Yang kita hitung persis besaran yang disebut t_j di tabel biaya:
// berapa kali PENGUJIAN gelung while dinilai untuk tiap j.
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <iomanip>
using namespace std;

long long sortAndCountTests(vector<int> A) {      // A diterima per nilai: sebuah salinan
    int n = (int)A.size() - 1;                    // A[0] tak terpakai
    long long tests = 0;
    for (int j = 2; j <= n; ++j) {
        int key = A[j], i = j - 1;
        ++tests;                                  // pengujian yang bisa langsung gagal
        while (i > 0 && A[i] > key) {
            A[i + 1] = A[i]; --i;
            ++tests;                              // tiap pengujian ulang ikut dihitung
        }
        A[i + 1] = key;
    }
    return tests;
}

int main() {
    cout << left << setw(6) << "n" << setw(12) << "best"
         << setw(12) << "worst" << setw(12) << "random"
         << setw(14) << "n(n+1)/2-1" << '\n';
    mt19937 rng(12345);                           // benih tetap: dapat diulang
    for (int n : {8, 16, 32, 64, 128}) {
        vector<int> asc(n + 1), desc(n + 1), rnd(n + 1);
        for (int i = 1; i <= n; ++i) { asc[i] = i; desc[i] = n - i + 1; }
        rnd = asc;
        shuffle(rnd.begin() + 1, rnd.end(), rng);
        cout << left << setw(6) << n
             << setw(12) << sortAndCountTests(asc)
             << setw(12) << sortAndCountTests(desc)
             << setw(12) << sortAndCountTests(rnd)
             << setw(14) << ((long long)n * (n + 1) / 2 - 1) << '\n';
    }
    return 0;
}
```

Keluaran sesungguhnya adalah:

```
n     best        worst       random      n(n+1)/2-1
8     7           35          25          35
16    15          135         59          135
32    31          527         340         527
64    63          2079        1045        2079
128   127         8255        4453        8255
```

Bacalah tabel itu berdampingan dengan ketiga penurunan kita.

Kolom **best** persis $n - 1$ setiap kali: 7,15,31,63,127 untuk $n = 8,16,32,64,128$. Itulah $\sum t_j = n - 1$ dari Subbab 2.11.1, terkonfirmasi sampai ke digitnya.

Kolom **worst** identik dengan ramalan $n(n + 1)/2 - 1$ pada setiap barisnya. Bukan kira-kira — identik. Penurunan Subbab 2.11.2 bukanlah taksiran; ia cacah yang eksak, dan mesinnya sepakat.

Kolom **random** berada di dekat separuh kolom worst: $4453/8255 = 0.539$ pada $n = 128$, dan $1045/2079 = 0.503$ pada $n = 64$. Itulah faktor setengah dari Subbab 2.11.3, dalam batas derau yang wajar dari satu kali pengocokan. Kolom random tumbuh dengan cara yang sama dengan

kolom worst — melipatempat ketika n berlipat dua — dan itulah intinya: orde yang sama, konstanta yang lebih kecil.

Sekarang klaim yang sama, dengan pengukur waktu alih-alih pencacah. **Kode 2.4** mengukur kasus terburuk untuk $n = 5{,}000$ sampai $n = 80{,}000$, berlipat dua setiap kalinya, lalu mencetak nisbah tiap waktu terhadap waktu sebelumnya.

Kode 2.4 Mengukur waktu kasus terburuk Insertion Sort pada lima ukuran masukan, masing-masing dua kali lipat ukuran sebelumnya. Kolom `ratio` adalah pengganda biaya yang teramati ketika n dilipatduakan.

```
// Apakah kuadratiknya benar-benar muncul pada mesin sungguhan? Lipatduakan n,
// lalu perhatikan. Insertion Sort dijalankan pada masukan terurut terbalik,
// yaitu kasus terburuknya, sehingga cacah pengujian gelung while persis
// n(n+1)/2 - 1 setiap kalinya.
#include <iostream>
#include <vector>
#include <chrono>
#include <iomanip>
using namespace std;
using namespace std::chrono;

double timeWorstCase(int n) {
    vector<int> A(n + 1);
    for (int i = 1; i <= n; ++i) A[i] = n - i + 1;   // n, n-1, ..., 1
    auto t0 = steady_clock::now();
    for (int j = 2; j <= n; ++j) {
        int key = A[j], i = j - 1;
        while (i > 0 && A[i] > key) { A[i + 1] = A[i]; --i; }
        A[i + 1] = key;
    }
    auto t1 = steady_clock::now();
    return duration<double, milli>(t1 - t0).count();
}

int main() {
    cout << setw(8) << "n" << setw(14) << "time (ms)"
         << setw(12) << "ratio" << setw(18) << "n(n+1)/2-1" << '\n';
    double prev = 0;
    for (int n : {5000, 10000, 20000, 40000, 80000}) {
        double ms = timeWorstCase(n);
        cout << setw(8) << n << setw(14) << fixed << setprecision(2) << ms;
        if (prev > 0) cout << setw(12) << setprecision(2) << (ms / prev);
        else          cout << setw(12) << "-";
        cout << setw(18) << ((long long)n * (n + 1) / 2 - 1) << '\n';
        prev = ms;
    }
    return 0;
}
```

Pada mesin yang dipakai untuk menyiapkan buku ini, program itu mencetak:

```
       n     time (ms)       ratio        n(n+1)/2-1
    5000          4.10           -          12502499
   10000         16.09        3.93          50004999
   20000         64.06        3.98         200009999
   40000        256.48        4.00         800019999
   80000       1034.66        4.03        3200039999
```

Angka milidetiknya akan berbeda pada komputer Anda — justru itulah inti Subbab 2.6.4, dan Anda memang seharusnya mengharapkannya berbeda. Kolom **ratio**-lah bagian yang dapat dibawa ke mana-mana. Pada jalan di atas ia terbaca 3.93,3.98,4.00,4.03; pada mesin yang sedang sibuk Anda mungkin melihat apa saja dari sekitar 3,6 sampai 4,4, sebab pengukur waktu

mengukur sistem operasi Anda sekaligus program Anda. Ulangi larinya beberapa kali dan angka yang dikitarinya adalah empat: setiap pelipatduaan n mengalikan waktunya dengan empat.

Mengapa empat? Sebab jika $T(n) = an^2 + bn + c$ maka, untuk n yang besar, suku n^2 mendominasi, dan

$$\frac{T(2n)}{T(n)} \approx \frac{a(2n)^2}{an^2} = \frac{4an^2}{an^2} = 4. \tag{2.8}$$

Algoritma kuadratik melipatempat ketika masukannya dilipatduakan. Yang linear sekadar melipatdua. Yang logaritmik menambah sebuah konstanta. Mengukur nisbah Persamaan (2.8) adalah cara tercepat untuk mengetahui secara eksperimental berapa sebenarnya laju pertumbuhan kode Anda, dan itu teknik yang layak Anda genggam: jika Anda yakin kode Anda $O(n\lg n)$ lalu nisbahnya keluar 4, ada kutu pada penalaran Anda atau pada kode Anda.

2.11.5 Mengapa kita memusatkan perhatian pada kasus terburuk

Kasusnya ada tiga; hampir seluruh perhatian kita curahkan pada satu di antaranya. Inilah alasannya.

Ia sebuah jaminan. Waktu jalan kasus terburuk adalah *batas atas*: algoritmanya tidak akan pernah, pada masukan apa pun, memakan waktu lebih lama daripada itu. Tidak ada masukan yang dapat mengejutkan Anda. Rata-rata memberi tahu apa yang biasanya terjadi; kasus terburuk memberi tahu apa yang tidak mungkin tidak benar, dan hanya pernyataan jenis kedua yang dapat diandalkan.

Sebagian penerapan menuntutnya. Pada sistem pengendali lalu lintas udara, alat pacu jantung, pengendali rem anti-terkunci, atau robot bedah, "biasanya cukup cepat" bukanlah standar rekayasa. Anda harus sanggup menyebutkan batas yang berlaku setiap kali.

Kasus terburuk lebih sering muncul daripada yang Anda kira. Menyenangkan sekali menganggap masukan itu acak. Jarang sekali ia acak. Pencarian atas rekaman yang tidak ada persis mengenai kasus terburuk *linear search*; dan data nyata datang dalam keadaan sudah terurut, sudah terurut terbalik, atau penuh nilai kembar, jauh lebih sering daripada yang diduga peluang. Lebih buruk lagi, jika seorang lawan dapat memengaruhi masukan Anda — pengguna yang mengirim formulir, klien yang mengirim permintaan — mereka dapat *memilih* kasus terburuk Anda dengan sengaja. Serangan kompleksitas algoritmik bekerja persis dengan cara itu.

Kasus rata-ratanya pun sering berorde sama. Kita baru saja membuktikannya untuk Insertion Sort: yang terburuk an^2, yang rata-rata sekitar $\frac{a}{2}n^2$. Empat menit alih-alih delapan menit bukanlah penyelamatan.

Menghitung rata-rata yang sesungguhnya itu sukar, dan memerlukan anggapan yang biasanya tidak dapat Anda benarkan. Analisis kasus rata-rata di Subbab 2.11.3 bertumpu pada "semua $n!$ penataan sama mungkinnya". Benarkah itu untuk data Anda? Untuk nota laundry yang datang sepanjang hari, sama sekali tidak — nota itu datang hampir terurut. Untuk bacaan dari sebuah sensor, tidak juga. Tanpa sebarannya, kasus rata-rata tidak terdefinisi; kasus terburuk sama sekali tidak memerlukan sebaran.

> **Intuisi.** Perusahaan pengiriman yang berjanji "paket Anda tiba dalam 5 hari" telah memberi Anda sesuatu yang dapat Anda rencanakan. Perusahaan yang berkata "biasanya tiba dalam 3 hari" telah memberi Anda sebuah harapan. Analisis kasus terburuk menjual jaminan, bukan harapan.

2.12 Pertumbuhan Fungsi

Kita telah menurunkan tiga waktu jalan: $an + b$, $\frac{a}{2}n^2 + \cdots$, dan $an^2 + bn + c$. Subbab 2.11.4 memperlihatkan bahwa konstanta a, b, c bergantung pada mesinnya sedangkan pangkat-pangkat n tidak. Subbab ini memperlihatkan mengapa, untuk n yang besar, *hanya* pangkat-pangkat n yang berarti — dan itulah pembenaran bagi notasi asimtotik yang akan didefinisikan Bab 3 secara formal.

2.12.1 Logaritma, dengan kata-kata

Ada satu lambang yang perlu didefinisikan sebelum kita dapat membaca tabelnya. **Logaritma n dengan basis 2**, ditulis $\log_2 n$ atau, dalam buku ini, $\lg n$, menjawab pertanyaan ini: *2 harus saya pangkatkan dengan berapa supaya memperoleh n?* Secara setara, dan lebih berguna untuk algoritma: *berapa kali saya dapat membelah dua n sebelum mencapai 1?*

- $\lg 8 = 3$, sebab $2^3 = 8$; dan sebab $8 \to 4 \to 2 \to 1$ adalah tiga kali pembelahan dua.
- $\lg 1024 = 10$, sebab $2^{10} = 1024$.
- $\lg 1{,}000{,}000 \approx 19.93$, sebab $2^{20} = 1{,}048{,}576$, sedikit lebih dari sejuta.

Sifatnya yang menentukan adalah bahwa logaritma tumbuh dengan lambat yang menyiksa. Mengalikan n dengan seribu hanya menambahkan sekitar sepuluh pada $\lg n$. Algoritma yang ongkosnya $\lg n$ praktis gratis.

2.12.2 Tabel pembandingnya

Tabel 2.3 membandingkan lima laju pertumbuhan pada empat ukuran masukan. Setiap masukan tabel itu dihitung oleh program di **Kode 2.5** di bawah, sehingga Anda dapat memeriksa sendiri mana pun di antaranya.

Tabel 2.3 Bagaimana lima waktu jalan yang umum itu tumbuh. Nilai terakhirnya begitu besar sehingga sering ditulis begitu saja sebagai ∞; nilai terhitungnya berdigit lebih dari tiga ratus ribu.

n	$\lg n$	n	$n\lg n$	n^2	2^n
10	3.3	10	33	100	1,024
100	6.6	100	664	10,000	1.27×10^{30}
1,000	10.0	1,000	9,966	1,000,000	1.07×10^{301}
10^6	19.9	10^6	1.99×10^7	10^{12}	9.90×10^{301029}

Sekarang bacalah baris terakhirnya, $n = 1{,}000{,}000$, menyeberang ke kanan, dan biarkan ia meresap. Andaikan sebuah mesin melakukan 10^9 operasi sederhana per detik — angka yang wajar untuk satu inti prosesor modern yang menjalankan C++ terkompilasi.

- Algoritma $O(\lg n)$ melakukan **20** operasi. Jawabannya tiba sebelum jari Anda terangkat dari tombol.
- Algoritma $O(n)$ melakukan **satu juta** operasi: kira-kira satu milidetik.
- Algoritma $O(n\lg n)$ melakukan **dua puluh juta** operasi: kira-kira dua puluh milidetik. Masih seketika bagi manusia.
- Algoritma $O(n^2)$ melakukan **satu triliun** (10^{12}) operasi: kira-kira 1.000 detik, yaitu enam belas menit empat puluh detik. Pengguna Anda sudah menutup tabnya.
- Algoritma $O(2^n)$ melakukan sekitar 10^{301030} operasi. Tidak ada satuan waktu untuk ini. Para ilmuwan menaksir alam semesta teramati memuat antara 10^{78} dan 10^{82} atom — angka yang sering dikutip sebagai 10^{80} dan dikenal sebagai bilangan Eddington. Cacah operasi di sini

melampaui cacah atom di alam semesta dipangkatkan tiga ribu tujuh ratus. Mengalikan kecepatan prosesor Anda dengan satu miliar hanya akan membuang sembilan digit dari sebuah eksponen berangka 301.029. Algoritma eksponensial bukan lambat; ia mustahil.

Inilah program yang menghasilkan angka-angka itu. Perhatikan kiat dalam `powerOfTwo`: $2^{1000000}$ tidak dapat disimpan dalam tipe titik-kambang mana pun, jadi kita tidak pernah membentuknya, melainkan menghitung $\log_{10}(2^n) = n\log_{10}2$ lalu membelahnya menjadi mantisa dan eksponen.

Kode 2.5 Menghitung perbandingan laju pertumbuhan, termasuk 2^{10^6}, tanpa pernah menyimpan bilangan sebesar itu.

```
// Tabel laju pertumbuhan, dihitung alih-alih dikutip.
// log2 sebuah bilangan raksasa dihitung dalam logaritma supaya 2^n tidak
// pernah perlu disimpan: log10(2^n) = n * log10(2), lalu kita cetak mantisa
// dan eksponennya.
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

void powerOfTwo(double n) {                  // cetak 2^n sebagai m x 10^e
    double log10val = n * log10(2.0);
    double e = floor(log10val);
    double m = pow(10.0, log10val - e);      // digit terdepannya
    if (e < 6) cout << setw(14) << (long long)llround(pow(2.0, n));
    else       cout << setw(8) << fixed << setprecision(2) << m
                    << " x 10^" << (long long)e;
}

int main() {
    cout << setw(10) << "n" << setw(10) << "log2 n" << setw(14) << "n"
         << setw(14) << "n log2 n" << setw(16) << "n^2"
         << "   2^n\n";
    for (double n : {10.0, 100.0, 1000.0, 1e6}) {
        double lg = log2(n);
        cout << setw(10) << (long long)n
             << setw(10) << fixed << setprecision(1) << lg
             << setw(14) << (long long)n
             << setw(14) << (long long)llround(n * lg)
             << setw(16) << (long long)(n * n) << "   ";
        powerOfTwo(n);
        cout << '\n';
    }
    return 0;
}
```

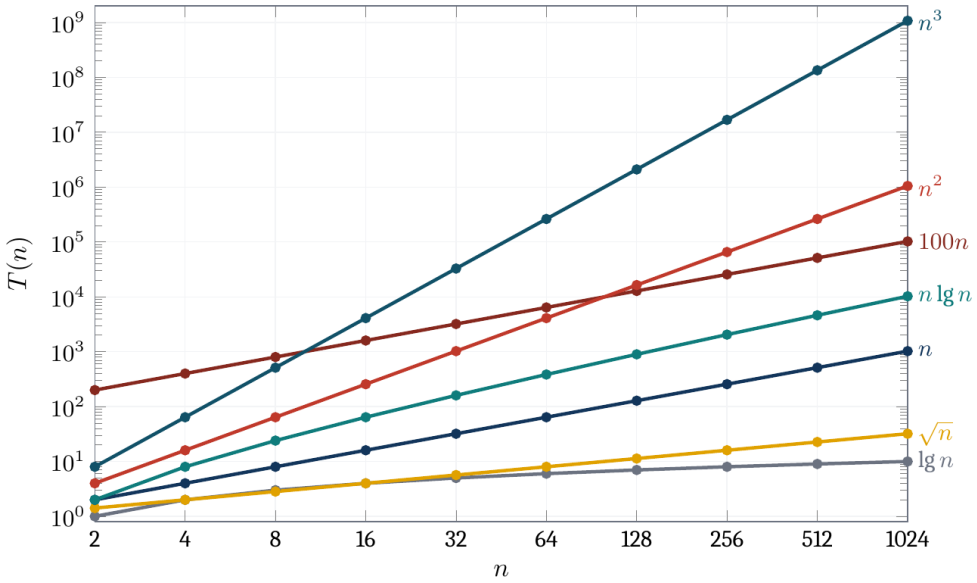
Keluaran sesungguhnya adalah:

```
         n    log2 n             n      n log2 n             n^2   2^n
        10       3.3            10            33             100             1024
       100       6.6           100           664           10000       1.27 x 10^30
      1000      10.0          1000          9966         1000000       1.07 x 10^301
   1000000      19.9       1000000      19931569   1000000000000       9.90 x 10^301029
```

✓ **WAWASAN KUNCI:** Ketika n tumbuh, **laju pertumbuhan** mendominasi segalanya. Konstanta dan suku berorde rendah menjadi tak berarti. Algoritma n^2 dengan konstanta yang menakjubkan, 0.001, tetap kalah terhadap algoritma $n\lg n$ dengan konstanta yang mengerikan, 1000 — hanya saja diperlukan n yang lebih besar untuk memperlihatkannya. Persis inilah sebabnya notasi asimtotik itu ada: ia membuang bagian jawaban yang bergantung pada perangkat keras Anda dan menyimpan bagian yang menentukan apakah program Anda selesai.

2.12.3 Melihatnya sebagai gambar

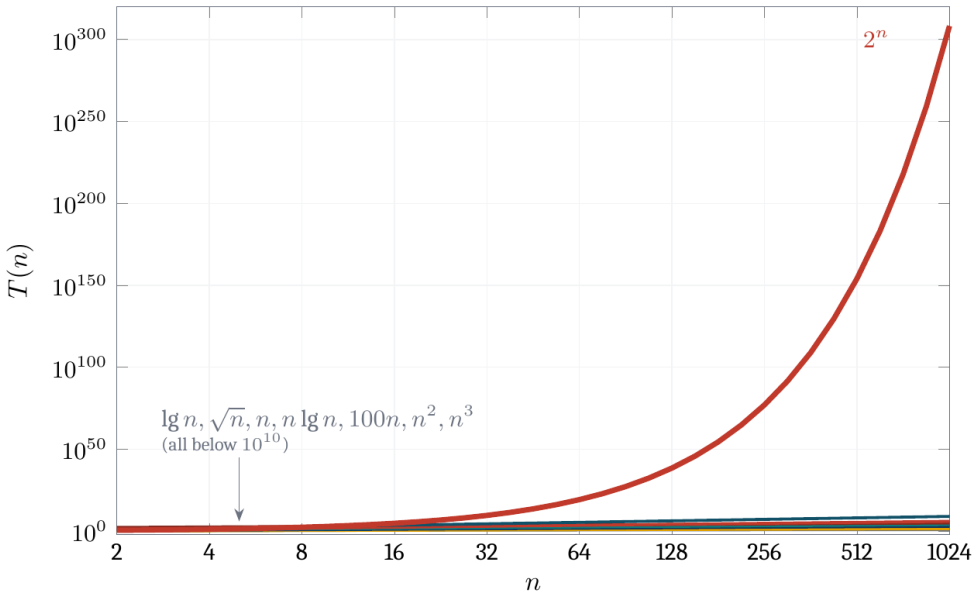
Dua grafik menyampaikan hal yang tidak dapat disampaikan sebuah tabel. **Gambar 2.12** memetakan waktu jalan polinomial dan logaritmik yang umum untuk n dari 2 sampai 1024, dengan *kedua* sumbunya logaritmik: di sepanjang sumbu datar tiap tanda melipatduakan n, dan di sepanjang sumbu tegak tiap garis kisi mengalikan $T(n)$ dengan sepuluh. (Kurva bertanda lg$n$ adalah logaritma basis 2, sehingga nilainya 10 pada $n = 1024$; logaritma dengan basis lain hanyalah kelipatan konstan darinya, dengan bentuk yang sama.)



Gambar 2.12 Pertumbuhan lg$n$, $\sqrt{n}$, n, $n$lg$n$, $100n$, n^2, dan n^3 dengan kedua sumbu logaritmik. Pada peta semacam itu fungsi n^k menjadi garis lurus yang kemiringannya k, sehingga makin curam berarti makin buruk.

Pada peta log–log, setiap fungsi berbentuk n^k menjadi garis lurus berkemiringan k, dan itu membuat gambarnya mudah dibaca: n^3 adalah garis yang paling curam, lalu n^2, lalu n, sedangkan lg$n$ hampir mendatar. Perhatikan garis bertanda $100n$. Pada $n = 2$ ia mulai dari 200, jauh di atas n^2 yang mulai dari 4. Mahasiswa yang membandingkan kedua algoritma itu pada masukan mungil akan menyimpulkan bahwa yang kuadratik lima puluh kali lebih baik. Tetapi kedua garis itu berpotongan di dekat $n = 100$, dan setelah itu n^2 memanjat menjauh selamanya. **Konstanta yang besar dapat menyembunyikan laju pertumbuhan yang buruk, tetapi hanya untuk n yang kecil.** Inilah juga sebabnya algoritma $O(n^2)$ sering kali benar-benar merupakan pilihan yang tepat ketika Anda tahu n akan selalu kecil.

Sekarang tambahkan yang eksponensial, dan seluruh gambarnya berubah. **Gambar 2.13** adalah peta yang sama dengan 2^n disertakan, dan sumbu tegaknya harus merentang sampai sekitar 10^{308} supaya muat, sebab $2^{1024} \approx 1.8 \times 10^{308}$. Sumbu tegaknya kembali logaritmik, dengan label berupa pangkat sepuluh.



Gambar 2.13 Laju pertumbuhan yang sama dengan 2^n ditambahkan. Sumbu tegaknya kini merentang sampai sekitar 10^{308}; setiap kurva polinomial terhimpit ke dasar bagan, tak terbedakan dari sumbunya.

Segala yang kita bandingkan dengan susah payah di **Gambar 2.12** — selisih antara n, $n\lg n$, n^2, dan n^3 — telah tergepeng menjadi satu coreng tunggal yang tak terbedakan di sepanjang dasar **Gambar 2.13**. Berhadapan dengan yang eksponensial, semua polinomial adalah hal yang sama: cepat. Itulah pelajaran terdalam dari kedua gambar ini, dan itulah sebabnya ilmu komputer teoretis menarik garis pemisah utamanya antara polinomial dan eksponensial, bukan antara n dan n^2.

2.12.4 Hierarki laju pertumbuhan

Dari yang paling lambat tumbuh ke yang paling cepat tumbuh — artinya, dari yang terbaik ke yang terburuk sebagai waktu jalan:

$$O(1) < O(\lg n) < O(\sqrt{n}) < O(n) < O(n\lg n) < O(n^2) < O(n^3) < O(2^n) < O(n!) \tag{2.9}$$

Di sini tanda $<$ di antara dua kelas berarti "tumbuh lebih lambat secara tegas, begitu n cukup besar". Persamaan (2.9) layak dihafal, bersama sebuah contoh sederhana untuk tiap kelasnya:

- $O(1)$, **konstan.** Sama sekali tidak bergantung pada n. Membaca `A[i]`; mendorong ke dalam tumpukan; aritmetika.
- $O(\lg n)$, **logaritmik.** *Binary search* (Subbab 2.14); penyisipan ke dalam pohon seimbang (Bab 6).
- $O(\sqrt{n})$, **akar.** Pembagian coba-coba untuk menguji apakah n prima.
- $O(n)$, **linear.** Satu kali sapuan atas sebuah larik: *linear search*, mencari nilai maksimum, menjumlahkan.
- $O(n\lg n)$, **linearitmik.** Titik paling manis untuk pengurutan: Merge Sort, Heapsort, Quicksort secara rata-rata.
- $O(n^2)$, **kuadratik.** Kasus terburuk Insertion Sort; membandingkan setiap pasangan dari n benda.
- $O(n^3)$, **kubik.** Perkalian matriks naif; lintasan terpendek semua pasangan Floyd–Warshall (Bab 15).

- $O(2^n)$, **eksponensial.** Mencacah setiap himpunan bagian dari n benda. Tak berpengharapan di luar $n \approx 30$.
- $O(n!)$, **faktorial.** Mencacah setiap penataan dari n benda — algoritma "urutkan dengan mencoba semua permutasi" di Subbab 2.4.3. Tak berpengharapan di luar $n \approx 12$.

2.12.5 Ancar-ancar yang praktis

Baik untuk pemrograman kompetitif maupun untuk pertimbangan rekayasa, bentuk yang berguna dari semua ini adalah sebuah tabel rujukan. Mesin modern melakukan kira-kira 10^8 sampai 10^9 operasi sederhana per detik. Dengan batas waktu yang lazim, satu atau dua detik, **Tabel 2.4** menyatakan kompleksitas mana yang terjangkau pada ukuran masukan mana.

Tabel 2.4 Ukuran masukan terbesar yang dapat ditangani tiap kelas kompleksitas dalam batas satu sampai dua detik. Bacalah mundur untuk memilih algoritma: pernyataan soal memberi Anda n, dan tabel ini memberi Anda kompleksitas yang harus Anda capai.

n Maks	Kompleksitas yang boleh	Operasi	Contoh
$n \leq 10$	$O(n!)$, $O(2^n)$	$\sim 10^6$	Gaya kasar / runut balik
$n \leq 1{,}000$	$O(n^2)$	$\sim 10^6$	Insertion Sort, Bubble Sort
$n \leq 100{,}000$	$O(n\lg n)$	$\sim 1.7 \times 10^6$	Merge Sort, Quicksort
$n \leq 10^6$	$O(n)$ atau $O(n\lg n)$	$\sim 10^7$	Sapuan linear, counting sort
$n \leq 10^{18}$	$O(\lg n)$ atau $O(1)$	~ 60	Binary search, rumus bentuk tertutup

Pakailah ia dalam arah yang memang berarti. Pernyataan soal memberi tahu Anda n maksimumnya sebelum Anda menulis satu baris kode pun. Carilah n itu di **Tabel 2.4**, bacalah kompleksitas yang diizinkan, baru kemudian mulailah merancang. Jika kendalanya menyatakan $n \leq 200{,}000$ dan gagasan Anda $O(n^2)$, Anda sudah tahu ia akan gagal — Anda baru saja menyelamatkan satu jam implementasi dan satu jawaban salah.

> ⚠ **RENUNGKAN:** Jika algoritma Anda $O(n^2)$ dan $n = 1{,}000{,}000$, ia memerlukan 10^{12} operasi — sekitar 1.000 detik, atau enam belas menit empat puluh detik, berhadapan dengan batas dua detik. Optimasi mikro sebanyak apa pun tidak menutup jurang selebar lima ratus kali. Anda tidak memerlukan kode yang lebih cepat; Anda memerlukan algoritma yang berbeda.

2.13 Pencarian: Linear Search

Pengurutan menata data. Pencarian menemukan sesuatu di dalamnya. Pencarian adalah problem mendasar kedua dalam bab ini, dan ia akan memperlihatkan sesuatu yang tidak diperlihatkan pengurutan: bahwa *struktur* datanya dapat mengubah apa yang dapat dicapai, bukan sekadar sebesar faktor konstan melainkan sebesar satu orde pertumbuhan penuh.

2.13.1 Problemnya

> **PROBLEM PENCARIAN.**
>
> *Masukan:* sebuah barisan $A = \langle a_1, a_2, \ldots, a_n \rangle$ dan sebuah nilai kueri q.
>
> *Keluaran:* sebuah indeks j sedemikian rupa sehingga $A[j] = q$, atau nilai khusus NIL jika q tidak muncul dalam A.

Gambar 2.14 menyatakan hal ini dalam bentuk masukan/keluaran, dengan dua instans konkret di bawahnya.

Gambar 2.14 Problem pencarian. Pada instans $A = \langle 2,5,4,10,7 \rangle$ dengan $q = 5$ jawabannya adalah indeks 2; dengan $q = 9$ jawabannya NIL, sebab 9 tidak muncul.

Dua rincian spesifikasi itu layak direnungkan sejenak.

Pertama, keluarannya adalah sebuah **indeks**, bukan nilainya. Mengembalikan nilainya tidak ada gunanya — pemanggilnya sudah memegang q; yang ia perlukan adalah *di mana* nilai itu berada, supaya ia dapat melihat rekaman yang tersimpan bersamanya.

Kedua, NIL bukan sebuah bilangan. Ia sebuah penanda yang berarti "tidak ada indeks semacam itu". Dalam C++ kita harus menyandikannya sebagai suatu nilai yang dapat dibedakan pemanggilnya dari jawaban yang sungguhan. Dengan pengindeksan berbasis-1 kita memakai 0, sebab 0 tidak pernah menjadi indeks yang sah dalam skema kita; dengan pengindeksan berbasis-0 kelazimannya adalah -1. Yang tidak boleh Anda lakukan adalah mengembalikan sesuatu seperti 0 dalam program berbasis-0, sebab di situ pemanggilnya tidak dapat membedakan "ditemukan di depan" dari "sama sekali tidak ditemukan".

2.13.2 Algoritmanya

Jika lariknya tidak terurut, kita tidak punya pilihan: kita harus melihat elemennya satu per satu sampai kita menemukan q atau kehabisan larik. **Pseudokode 2.3** melakukan persis hal itu.

Pseudokode 2.3 *Linear search*: periksa elemen dari kiri ke kanan sampai q ditemukan atau lariknya habis.

```
LINEAR-SEARCH(A, q)
1  j = 1
2  while j <= A.length and A[j] != q
3      j = j + 1
4  if j <= A.length
5      return j
6  else
7      return NIL
```

Sekali lagi urutan kedua syarat pada baris 2 menanggung beban: `j <= A.length` harus diuji **lebih dahulu**, supaya ketika j melewati ujungnya kita berhenti sebelum menilai $A[j]$ pada posisi yang tidak ada. Dalam C++, `&&` menjamin evaluasi hubung-singkat dari kiri ke kanan ini.

Baris 4 lalu menanyakan mengapa gelungnya berhenti. Jika ia berhenti karena syarat kedua gagal, maka j adalah indeks yang sah dan $A[j] = q$: kita menemukannya. Jika ia berhenti karena syarat pertama gagal, maka $j = n + 1$, yang berada di luar ujungnya, dan q memang tidak ada.

2.13.3 Invarian gelung untuk linear search

Teknik pembuktian yang sama berlaku, dan ia cukup pendek untuk dituntaskan dalam satu paragraf.

INVARIAN GELUNG (linear search). Pada awal tiap iterasi gelung `while`, tidak ada satu pun di antara $A[1], A[2], \ldots, A[j-1]$ yang sama dengan q.

Inisialisasi. Sebelum iterasi pertama $j = 1$, jadi klaimnya menyangkut $A[1 \ldots 0]$, sebuah jangkauan kosong. Tidak ada elemen jangkauan kosong yang sama dengan q, secara hampa. ✓

Pemeliharaan. Andaikan invariannya berlaku dan tubuh gelungnya berjalan. Tubuhnya berjalan hanya jika pengujiannya berhasil, yang menuntut $A[j] \neq q$. Jadi sekarang tidak ada satu pun di antara $A[1 \ldots j-1]$ yang sama dengan q *dan* $A[j] \neq q$, yang bersama-sama menyatakan bahwa tidak ada satu pun di antara $A[1 \ldots j]$ yang sama dengan q. Baris 3 menaikkan j, sehingga invarian dengan j yang baru berbunyi "tidak ada satu pun di antara $A[1 \ldots j-1]$ yang sama dengan q" — persis yang baru saja kita tegakkan. ✓

Terminasi. Ada dua pintu keluar, yang bersesuaian dengan baris 4–7. Jika gelungnya berhenti karena $A[j] = q$, kita mengembalikan j dan $A[j] = q$ adalah keluaran yang diminta. Jika ia berhenti karena $j > n$, maka $j = n + 1$ dan invariannya menyatakan tidak ada satu pun di antara $A[1 \ldots n]$ yang sama dengan q — yaitu seluruh lariknya — sehingga NIL adalah jawaban yang benar. ✓ ■

2.13.4 Analisisnya

Pengujian pada baris 2 adalah operasi yang menentukan; selebihnya adalah kelipatan konstan darinya.

Kasus terburuk. Jika q tidak muncul dalam A, gelungnya berjalan sampai $j = n + 1$, menilai pengujiannya $n + 1$ kali dan tubuhnya n kali. Jadi $T(n) = \Theta(n)$. Hal yang sama terjadi jika q berada tepat di posisi terakhir.

Kasus terbaik. Jika $A[1] = q$, pengujiannya dinilai sekali dan kita langsung kembali: $T(n) = \Theta(1)$.

Kasus rata-rata. Andaikan q ada dan sama mungkinnya berada di tiap posisi dari n posisi itu. Maka cacah pembandingannya adalah 1 dengan peluang $1/n$, 2 dengan peluang $1/n$, dan seterusnya, sehingga cacah harapannya adalah

$$\frac{1}{n}(1 + 2 + \cdots + n) = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}, \tag{2.10}$$

dengan memakai rumus Gauss yang kita turunkan di Subbab 2.11.2. Kira-kira separuh lariknya — dan nilai $\frac{n+1}{2}$ pada Persamaan (2.10) tetap saja $\Theta(n)$.

✓ **WAWASAN KUNCI:** *Linear search* pada larik yang **tidak terurut** adalah $\Theta(n)$ pada kasus terburuk, dan ini sudah optimal. Argumennya pendek dan layak dihayati: jika algoritmanya belum memeriksa posisi k, seorang lawan bebas menaruh q di sana, sehingga algoritma benar mana pun harus memeriksa setiap posisi sebelum ia boleh menjawab NIL. Anda tidak dapat mengalahkan $\Theta(n)$ pada data yang tidak terurut. Untuk lebih cepat, Anda harus mengubah datanya, bukan algoritmanya.

2.13.5 Linear search dalam C++

Kode 2.6 Linear search dalam C++, pada data yang sama dengan **Gambar 2.14**.

```
// Linear search pada larik yang TIDAK TERURUT, persis seperti pseudokodenya.
// Indeksnya berbasis-1; A[0] tak terpakai. NIL diwakili oleh 0, yang tidak
// pernah menjadi indeks yang sah, jadi ia tak dapat dikelirukan dengan
// jawaban yang sungguhan.
#include <iostream>
#include <vector>
using namespace std;

const int NIL = 0;

int linearSearch(const vector<int>& A, int n, int q) {
    int j = 1;
    while (j <= n && A[j] != q) ++j;    // berhenti pada kecocokan pertama, atau lewati
ujungnya
    if (j <= n) return j;               // j indeks sungguhan: q kita temukan di sana
    return NIL;                         // kita melewati ujungnya: q tidak ada
}

int main() {
    vector<int> A = {0, 2, 5, 4, 10, 7};   // data pada gambar problem pencarian
    int n = 5;
    for (int q : {5, 9}) {
        int j = linearSearch(A, n, q);
        cout << "q=" << q << " -> ";
        if (j == NIL) cout << "NIL\n"; else cout << j << '\n';
    }
    return 0;
}
```

Keluaran **Kode 2.6** adalah:

```
q=5 -> 2
q=9 -> NIL
```

yang persis dua jawaban yang tercetak pada kolom OUTPUT di **Gambar 2.14**.

2.13.6 Bisakah kita lebih cepat?

> ❓ **PERTANYAAN:** Kita baru saja membuktikan bahwa $\Theta(n)$ adalah yang terbaik yang mungkin pada larik yang tidak terurut. Tetapi bagaimana jika lariknya **terurut**? Apakah struktur tambahan itu membelikan kita sesuatu?

Ia membelikan kita sangat banyak. Inilah pertama kalinya dalam buku ini kita melihat pertukaran yang menjadi pusat bidang ini: *tanamkan modal pada struktur, dan Anda memperoleh kecepatan sebagai imbalannya.* Alasannya mudah dinyatakan. Pada larik yang tidak terurut, melihat $A[5]$ dan mendapati ia bukan q memberi tahu Anda persis satu hal: q tidak berada di posisi 5. Pada larik yang terurut, melihat $A[5]$ dan mendapati bahwa $A[5] > q$ memberi tahu Anda bahwa q tidak berada di posisi 5 — dan juga tidak di 6, 7, 8, atau di mana pun lebih ke kanan, sebab semua yang di sana lebih besar lagi. **Satu pembandingan menyingkirkan banyak posisi.** Itulah seluruh gagasan subbab berikutnya.

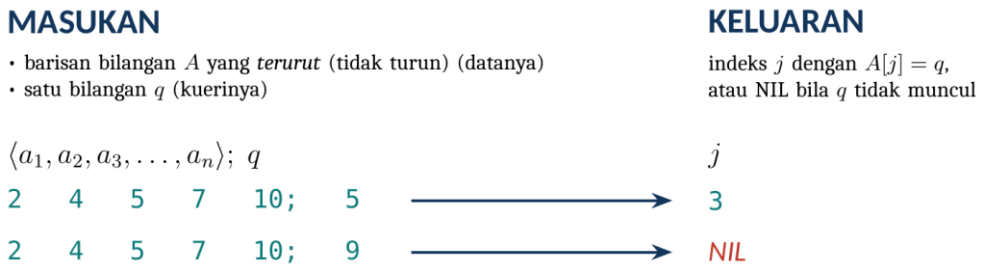
2.14 Binary Search

2.14.1 Problemnya

> **PROBLEM PENCARIAN TERURUT.**
>
> *Masukan:* sebuah barisan **terurut** tak-menurun $A = \langle a_1 \le a_2 \le \cdots \le a_n \rangle$ dan sebuah nilai kueri q.

> *Keluaran:* sebuah indeks j sedemikian rupa sehingga $A[j] = q$, atau NIL jika q tidak muncul dalam A.

Satu-satunya perubahan dari Subbab 2.13.1 adalah kata *terurut* pada baris masukannya, seperti diperlihatkan **Gambar 2.15**. Lihatlah dua instans di bagian bawah gambar itu dan bandingkan dengan instans di **Gambar 2.14**: lima bilangan yang sama, tetapi kini tertata, sehingga $q = 5$ dijawab dengan indeks 3 alih-alih indeks 2. Jawaban sebuah pencarian bergantung pada penataannya; *problem*-nya adalah problem yang berbeda.



Gambar 2.15 Problem pencarian terurut. Satu kata yang ditambahkan — *terurut* — pada spesifikasi masukan itulah yang memungkinkan percepatan sebesar eksponensial.

> ⚠ **RENUNGKAN:** *Binary search* pada larik yang tidak terurut bukan sekadar berjalan lambat; ia mengembalikan **jawaban yang salah**, tanpa suara. Ia akan dengan yakin melaporkan NIL untuk sebuah nilai yang sesungguhnya duduk di dalam larik itu. Prasyaratnya bukan nasihat, melainkan keharusan, dan tidak ada apa pun dalam kodenya yang memeriksakannya untuk Anda.

2.14.2 Gagasan intinya

Bandingkan q dengan elemen **tengah** lariknya. Tepat satu dari tiga hal berikut terjadi.

- Jika $A[mid] = q$, kita menemukannya. Kembalikan mid.
- Jika $A[mid] > q$, maka q pasti berada di paruh kiri, $A[1 \ldots mid - 1]$. Mengapa? Sebab lariknya terurut, sehingga setiap elemen dari mid ke kanan bernilai $\geq A[mid] > q$, dan tidak satu pun di antaranya dapat sama dengan q. Buang seluruh paruh kanan.
- Jika $A[mid] < q$, maka dengan argumen cerminannya q hanya dapat berada di $A[mid + 1 \ldots n]$. Buang seluruh paruh kiri.

Setiap pembandingan **membelah dua** ruang pencariannya. Inilah contoh nyata pertama kita untuk paradigma *divide and conquer* (bagi dan kuasai), dan Bab 3 akan membangun seluruh teori rekurensi di atas bentuk ini.

Ada versi manusiawinya yang akrab. Untuk mencari nama keluarga yang berawalan S dalam buku telepon cetak, Anda tidak mulai dari halaman 1. Anda membuka bukunya di dekat tengah, melihat bahwa Anda mendarat di huruf M, lalu membuang paruh pertamanya tanpa membaca satu nama pun di dalamnya. Pembuangan itulah algoritmanya.

2.14.3 Algoritmanya

Pseudokode 2.4 mengubah gagasan kunci tadi menjadi sebuah prosedur.

Pseudokode 2.4 *Binary search* pada larik terurut: bandingkan dengan elemen tengah dan buang paruh yang tidak mungkin memuat q.

```
BINARY-SEARCH(A, q)
1  left = 1
2  right = A.length
3  while left <= right
4      mid = (left + right) / 2        // pembagian bilangan bulat
5      if A[mid] == q then return mid
6      else if A[mid] > q then right = mid - 1
7      else left = mid + 1
8  return NIL
```

Pasangan $(left, right)$ mendefinisikan sebuah **jendela**: bagian larik yang di dalamnya q masih mungkin bersembunyi. Semula jendelanya adalah seluruh larik. Tiap iterasi menyusutkannya. Gelungnya berakhir ketika jendelanya menjadi kosong, yaitu ketika $left > right$, dan jendela kosong berarti q tidak ada di sana.

Invariannya adalah yang paling wajar, dan sekarang Anda semestinya sudah dapat menerkanya sebelum membacanya.

> **INVARIAN GELUNG (binary search).** Pada awal tiap iterasi, jika q muncul di mana pun dalam A, maka ia muncul pada suatu indeks di dalam jendela $[\,left, right\,]$.

Inisialisasi. Sebelum iterasi pertama $left = 1$ dan $right = n$, jadi jendelanya adalah seluruh larik. Jika q muncul dalam A, tentu ia muncul di dalam seluruh larik itu. ✓

Pemeliharaan. Anggaplah invariannya berlaku dan andaikan kita tidak kembali pada baris 5. Jika $A[mid] > q$, maka karena A terurut, $A[k] \geq A[mid] > q$ untuk setiap $k \geq mid$, sehingga q tidak muncul pada indeks mana pun yang $\geq mid$. Digabung dengan invariannya, jika q muncul sama sekali maka ia muncul di dalam $[\,left, mid - 1\,]$ — yang persis jendela baru yang ditetapkan baris 6. Kasus $A[mid] < q$ setangkup dengannya. ✓

Terminasi. Gelungnya keluar ketika $left > right$, yaitu ketika jendelanya kosong. Invariannya lalu menyatakan: jika q muncul dalam A, ia muncul dalam sebuah jangkauan kosong. Itu mustahil, jadi q tidak muncul dalam A, dan NIL pada baris 8 adalah jawaban yang benar. Pintu keluar yang lain adalah baris 5, yang mengembalikan mid setelah baru saja menguji $A[mid] = q$. ✓

Apakah gelungnya memang berhenti? Ya, dan itu layak diperiksa alih-alih dianggap begitu saja. Ukuran jendelanya adalah $right - left + 1$. Tiap iterasi menetapkan entah $right \leftarrow mid - 1$ atau $left \leftarrow mid + 1$, dan karena $left \leq mid \leq right$, kedua penetapan itu menyusutkan jendelanya secara ketat sekurang-kurangnya sebanyak satu. Besaran yang berupa bilangan bulat tak-negatif dan menurun ketat setiap iterasi pasti mencapai nol dalam waktu berhingga. ✓ ■

> ⚠ **PERINGATAN:** Tanda `+1` dan `-1` pada baris 6 dan 7 bukan hiasan. Keduanya mengecualikan mid itu sendiri, yang baru saja kita uji dan kita singkirkan. Tuliskan `right = mid` alih-alih `right = mid - 1`, lalu tinjaulah jendela dua elemen dengan $left = 1$, $right = 2$: maka $mid = 1$, dan jika $A[1] > q$ kita menetapkan $right = 1$, yang memberikan jendela [1,1] lagi — jendela yang sama, selamanya. Programnya menggantung. Tanda `-1`-lah yang menjamin kemajuan, dan karena itu pula yang membuat argumen terminasi di atas sahih.

2.14.4 Menelusuri binary search dengan tangan

Carilah $q = 7$ dalam $A = \langle 2,4,5,7,10 \rangle$, yang memiliki $n = 5$. **Tabel 2.5** mencatat telusurnya.

Tabel 2.5 Binary search untuk $q = 7$ dalam $A = \langle 2,4,5,7,10 \rangle$. Dua pembandingan sudah cukup, sedangkan linear search memerlukan empat.

Langkah	Left	Right	Mid	Tindakan
1	1	5	3	$A[3] = 5 < 7 \Rightarrow$ cari di paruh kanan: $left = 4$
2	4	5	4	$A[4] = 7 = 7 \Rightarrow$ ketemu! Kembalikan 4

Membentangkannya, satu langkah satu baris:

1. $left \leftarrow 1, right \leftarrow 5$. Jendelanya $[1,5]$, memuat $\langle 2,4,5,7,10 \rangle$.
2. Uji $left \leq right$: $1 \leq 5$, benar. Masuk ke dalam gelung.
3. $mid \leftarrow (1 + 5)/2 = 6/2 = 3$.
4. Apakah $A[3] = 7$? $A[3] = 5$, bukan.
5. Apakah $A[3] > 7$? $5 > 7$ bernilai salah, bukan.
6. Jadi $A[3] < 7$: tetapkan $left \leftarrow mid + 1 = 4$. Jendelanya sekarang $[4,5]$. Kita telah membuang posisi 1, 2, dan 3 — tiga dari lima — hanya dengan satu pembandingan.
7. Uji $left \leq right$: $4 \leq 5$, benar. Masuk ke dalam gelung lagi.
8. $mid \leftarrow (4 + 5)/2 = 9/2 = 4$ menurut pembagian bilangan bulat, yang memangkas ke arah nol.
9. Apakah $A[4] = 7$? $A[4] = 7$, ya. Kembalikan 4. ✓

Dua pembandingan alih-alih empat yang diperlukan *linear search* untuk mencapai posisi 4. Pada lima elemen, itu tak mengesankan. Pada satu juta, itu selisih antara 20 dan 1.000.000.

2.14.5 Mengapa ia $O(\lg n)$

> ❓ **PERTANYAAN:** Berapa kali Anda dapat memotong n menjadi separuh sebelum tidak ada lagi yang tersisa untuk dipotong?

Mari kita hitung. Mulailah dengan jendela berisi n elemen.

- Setelah 1 pembandingan, jendelanya memuat paling banyak $n/2$ elemen.
- Setelah 2 pembandingan, paling banyak $n/4$.
- Setelah 3 pembandingan, paling banyak $n/8$.
- Setelah k pembandingan, paling banyak $n/2^k$.

(Sebenarnya jendelanya menyusut sedikit lebih cepat daripada ini, sebab mid sendiri ikut dibuang, tetapi $n/2^k$ adalah batas atas yang sahih dan elemen tambahan itu tidak mengubah kesimpulannya.)

Gelungnya harus berhenti begitu jendelanya tidak dapat memuat elemen apa pun, dan pasti berhenti begitu $n/2^k \leq 1$. Selesaikan untuk k:

$$\frac{n}{2^k} \leq 1 \iff n \leq 2^k \iff \lg n \leq k. \tag{2.11}$$

Langkah tengah Persamaan (2.11) mengalikan kedua ruas dengan 2^k, yang positif sehingga arah ketaksamaannya tidak berubah; langkah terakhirnya mengambil $\log_2$ kedua ruas, yang sah sebab $\log_2$ adalah fungsi naik.

Jadi $k \geq \lg n$ pembandingan sudah mencukupi, dan waktu jalan kasus terburuknya adalah

$$T(n) = \Theta(\lg n).$$

Berilah ia angka. Untuk $n = 1{,}000{,}000$, $\lg n \approx 19.93$, jadi paling banyak **20 pembandingan** — berhadapan dengan sampai sejuta bagi *linear search*. Untuk $n = 10^{18}$, sebuah bilangan yang lebih besar daripada cacah butir pasir di Bumi, $\lg n \approx 60$. Enam puluh pembandingan. Inilah baris terbawah pada **Tabel 2.4**.

> ✓ **WAWASAN KUNCI:** *Binary search* berjalan dalam waktu $\Theta(\lg n)$. Setiap pelipatduaan masukan menambahkan tepat **satu** pembandingan. Inilah yang dibelikan oleh "memanfaatkan struktur": bukan konstanta yang lebih baik, melainkan fungsi yang berbeda. Dan semua itu datang dari satu kata — *terurut* — pada spesifikasi masukannya.

2.14.6 Binary search dalam C++, dan jebakan limpah

Kode 2.7 Binary search dalam C++, bentuk iteratif, dilengkapi pencatat agar mencetak jendelanya pada tiap langkah.

```
// Binary search pada larik TERURUT, dengan telusur left, right, mid yang
// dicetak. Indeks berbasis-1; NIL adalah 0. Titik tengahnya dihitung sebagai
// left + (right - left) / 2, tidak pernah (left + right) / 2 -- lihat teksnya.
#include <iostream>
#include <vector>
using namespace std;

const int NIL = 0;

int binarySearch(const vector<int>& A, int n, int q, bool trace) {
    int left = 1, right = n, step = 0;
    while (left <= right) {                 // jendela kosong berarti "tidak ada"
        int mid = left + (right - left) / 2;
        if (trace)
            cout << "  step " << ++step << ": left=" << left
                 << " right=" << right << " mid=" << mid
                 << " A[mid]=" << A[mid] << '\n';
        if (A[mid] == q) return mid;
        else if (A[mid] > q) right = mid - 1;   // q hanya mungkin di sebelah kiri
        else                 left  = mid + 1;   // q hanya mungkin di sebelah kanan
    }
    return NIL;
}

int main() {
    vector<int> A = {0, 2, 4, 5, 7, 10};    // terurut; A[0] tak terpakai
    int n = 5;
    cout << "searching for 7, with trace:\n";
    int j = binarySearch(A, n, 7, true);
    cout << "  result = " << j << "\n\n";
    for (int q : {5, 9}) {
        int r = binarySearch(A, n, q, false);
        cout << "q=" << q << " -> ";
        if (r == NIL) cout << "NIL\n"; else cout << r << '\n';
    }
    return 0;
}
```

Keluaran sesungguhnya adalah:

```
searching for 7, with trace:
  step 1: left=1 right=5 mid=3 A[mid]=5
  step 2: left=4 right=5 mid=4 A[mid]=7
  result = 4

q=5 -> 3
q=9 -> NIL
```

Kedua langkah yang tertelusur itu identik dengan **Tabel 2.5**, dan dua baris terakhirnya menirukan kolom OUTPUT pada **Gambar 2.15**: pada data yang terurut, $q = 5$ berada di indeks 3 dan $q = 9$ tidak ada.

Sekarang lihatlah sekali lagi baris 4 **Pseudokode 2.4**. Ia menyatakan $mid = (left + right)/2$, sedangkan programnya **tidak** menyatakan begitu. Programnya menyatakan `left + (right - left) / 2`. Kedua ungkapan itu setara dalam aritmetika biasa. Keduanya tidak setara dalam aritmetika `int`, dan selisihnya adalah sebuah kutu yang termasyhur: ia bersembunyi tanpa terdeteksi selama kira-kira dua dasawarsa di dalam *binary search* pada *Programming Pearls* [2], dan selama hampir satu dasawarsa di dalam *binary search* pustaka baku Java, yang menyalinnya.

Alasannya, $left + right$ dapat melampaui `int` terbesar yang dapat diwakilkan, padahal $left$, $right$, dan titik tengah yang sesungguhnya semuanya muat dengan nyaman. **Kode 2.8** memperagakannya alih-alih sekadar menegaskannya.

Kode 2.8 Mengapa titik tengah harus dihitung sebagai `left + (right - left) / 2`.

```
// Jebakan limpah pada titik tengah, diperagakan alih-alih sekadar ditegaskan.
// Kedua ungkapannya sama secara matematis. Dalam aritmetika 32 bit hanya satu
// di antaranya yang aman, sebab left + right dapat melampaui INT_MAX.
#include <iostream>
#include <climits>
using namespace std;

int main() {
    int left  = 1'500'000'000;
    int right = 2'000'000'000;
    cout << "INT_MAX            = " << INT_MAX << '\n';
    cout << "left               = " << left  << '\n';
    cout << "right              = " << right << '\n';
    cout << "true midpoint      = " << 1'750'000'000 << '\n';

    // Limpah bertanda adalah perilaku tak terdefinisi, jadi penjumlahannya kita
    // kerjakan dalam tipe yang lebih lebar lalu kita potong menjadi int -- itu
    // menirukan apa yang dilakukan mesin tanpa memberi kompilator kebebasan
    // menganggap bahwa hal itu tidak pernah terjadi.
    long long sum = (long long)left + right;
    int wrapped = (int)(unsigned int)sum;          // pola bit yang akan dipegang int
    cout << "(left+right)       = " << sum << " which does not fit in int\n";
    cout << "(left+right)/2     = " << wrapped / 2 << "   <-- wrong, negative\n";
    cout << "left+(right-left)/2 = " << left + (right - left) / 2 << "   <-- right\n";
    return 0;
}
```

Keluaran sesungguhnya adalah:

```
INT_MAX            = 2147483647
left               = 1500000000
right              = 2000000000
true midpoint      = 1750000000
(left+right)       = 3500000000 which does not fit in int
(left+right)/2     = -397483648   <-- wrong, negative
left+(right-left)/2 = 1750000000   <-- right
```

Rumus naifnya menghasilkan $-397{,}483{,}648$: sebuah indeks negatif, yang sebagai subskrip larik paling baik berujung pada kemacetan dan paling buruk pada kerusakan data tanpa suara. Rumus yang aman bekerja sebab $right - left$ paling banyak sebesar ukuran lariknya dan karena itu selalu kecil, sehingga tidak ada apa pun yang pernah melampaui jangkauannya.

> ⚠ **PERINGATAN:** Selalu tulis `mid = left + (right - left) / 2`. Ia tidak berongkos apa pun, ia terbaca hampir sama jelasnya, dan ia benar untuk setiap masukan. Jadikan ia kebiasaan

> sekarang, selagi Anda masih belajar, dan Anda tidak akan pernah perlu mengingat-ingat alasannya.

2.14.7 Binary search sebagai rekursi

Binary search adalah contoh pertama kita untuk *divide and conquer*, dan *divide and conquer* pada dasarnya rekursif. Menuliskannya dengan cara itu membuat strukturnya tampak, dan itulah bentuk yang akan dianalisis Bab 3 dengan sebuah rekurensi.

Kode 2.9 Binary search sebagai prosedur divide-and-conquer rekursif, yang mencetak jendelanya pada tiap aras rekursi.

```
// Binary search yang ditulis secara rekursif: bentuk divide-and-conquer-nya
// dibuat tampak. Rekursinya berjalan pada jendela [left..right]; jendela
// kosong adalah kasus dasarnya.
#include <iostream>
#include <vector>
using namespace std;

const int NIL = 0;

int bsRec(const vector<int>& A, int left, int right, int q, int depth) {
    cout << string(2 * depth, ' ') << "window [" << left << ".." << right << "]";
    if (left > right) { cout << " is empty -> NIL\n"; return NIL; }   // kasus dasar
    int mid = left + (right - left) / 2;
    cout << " mid=" << mid << " A[mid]=" << A[mid] << '\n';
    if (A[mid] == q) return mid;
    if (A[mid] > q) return bsRec(A, left, mid - 1, q, depth + 1);
    return               bsRec(A, mid + 1, right, q, depth + 1);
}

int main() {
    vector<int> A = {0, 1, 3, 5, 7, 9, 11, 13, 15};   // 8 elemen terurut
    int n = 8;
    for (int q : {13, 6}) {
        cout << "looking for " << q << ":\n";
        int r = bsRec(A, 1, n, q, 1);
        cout << "result: ";
        if (r == NIL) cout << "NIL\n\n"; else cout << r << "\n\n";
    }
    return 0;
}
```

Keluaran **Kode 2.9** adalah:

```
looking for 13:
  window [1..8] mid=4 A[mid]=7
    window [5..8] mid=6 A[mid]=11
      window [7..8] mid=7 A[mid]=13
result: 7

looking for 6:
  window [1..8] mid=4 A[mid]=7
    window [1..3] mid=2 A[mid]=3
      window [3..3] mid=3 A[mid]=5
        window [4..3] is empty -> NIL
result: NIL
```

Bacalah takuknya: tiap aras rekursi adalah satu pembandingan, dan jendelanya menyusut $8 \to 4 \to 2 \to 1 \to 0$. Dengan $n = 8$ kita memperoleh $\lg 8 = 3$, dan memang pencarian yang berhasil memakan 3 pembandingan dan yang tidak berhasil memakan 4 (tiga pembandingan ditambah pemeriksaan jendela kosong). Itu persis cocok dengan ramalan $\lg n$ pada Subbab 2.14.5.

Perhatikan pula bentuk rekursinya: **satu** pemanggilan rekursif, atas problem berukuran separuh, ditambah kerja konstan. Itulah rekurensi

$$T(n) = T(n/2) + \Theta(1), \tag{2.12}$$

yang penyelesaiannya $\Theta(\lg n)$. Rekurensi Persamaan (2.12) menyatakan: waktu untuk jendela berisi n elemen sama dengan waktu untuk jendela berukuran separuhnya, ditambah sebuah konstanta untuk satu pembandingan itu. Anda akan menyelesaikan rekurensi semacam ini secara mekanis dengan Teorema Master di Bab 3; untuk sekarang cukuplah melihat dari mana rekurensinya berasal.

2.14.8 Imbal baliknya: prapemrosesan

Binary search menuntut larik yang terurut. Ada yang harus mengurutkannya, dan pengurutan berongkos $\Theta(n\lg n)$ dengan algoritma yang baik. Jadi apakah *binary search* itu sepadan?

Semuanya bergantung pada berapa banyak pencarian yang Anda lakukan. Misalkan k adalah cacah pencariannya.

- **Linear search:** tanpa prapemrosesan, $\Theta(n)$ per pencarian. Total: $\Theta(kn)$.
- **Urutkan, lalu binary search:** $\Theta(n\lg n)$ sekali, lalu $\Theta(\lg n)$ per pencarian. Total: $\Theta(n\lg n + k\lg n)$.

Bandingkan keduanya untuk $n = 1{,}000{,}000$, dengan satuan hitungan 10^6 operasi:

- $k = 1$ **pencarian.** Linear: 1 satuan. Urut-lalu-cari: sekitar 20 satuan. **Linear menang** — jangan pernah mengurutkan sebuah larik untuk mencari di dalamnya satu kali saja.
- $k = 100$ **pencarian.** Linear: 100 satuan. Urut-lalu-cari: $20 + 100 \times 0.00002 \approx 20$ satuan. **Biner menang**, lima kali lipat.
- $k = 10{,}000$ **pencarian.** Linear: 10.000 satuan. Urut-lalu-cari: tetap sekitar 20 satuan. **Biner menang telak**, dengan faktor lima ratus.

Menyamakan kedua total itu lalu menyelesaikannya untuk k memperlihatkan bahwa titik impasnya kira-kira pada $k \approx \lg n$ pencarian — sekitar 20 untuk sejuta elemen. Di luar itu, pengurutannya membayar dirinya sendiri.

> ✓ **WAWASAN KUNCI:** Inilah kemunculan pertama sebuah pola yang berulang sepanjang buku ini: **bayar sekali untuk membangun struktur, lalu bertanyalah dengan murah selamanya.** Tabel hash, pohon pencarian seimbang, *heap*, larik jumlah-awalan, larik sufiks, dan prapemrosesan graf semuanya membuat tawar-menawar yang sama. Pertanyaannya tidak pernah "apakah strukturnya layak dibangun?" secara abstrak; pertanyaannya adalah "berapa banyak kueri yang akan mengamortisasinya?"

2.15 Menengok Kembali Paradigma Perancangan

Bab 1 menyebutkan nama paradigmanya. Sekarang kita sudah melihat dua di antaranya bekerja secara terperinci, dan dapat mengatakan apa yang membedakan keduanya.

2.15.1 Pendekatan inkremental

Bangunlah solusinya sepotong demi sepotong: setelah problemnya terselesaikan untuk $j - 1$ benda pertama, kerjakan sedikit hal untuk memperluas solusinya menjadi j benda. Ulangi sampai $j = n$.

Insertion Sort adalah contoh kanoniknya. Invarian gelungnya *adalah* pernyataan metode inkremental itu sendiri: "$A[1 \ldots j-1]$ sudah terurut", dan tiap iterasi memperluas bagian yang terurut itu sebanyak satu.

Kekuatannya. Mudah dipahami, mudah diimplementasikan, mudah dibuat benar. Sangat baik pada masukan kecil, dan di situ konstantanya yang rendah mengalahkan algoritma yang lebih canggih — inilah sebabnya pengurutan pustaka sungguhan seperti `std::sort` beralih ke Insertion Sort untuk sublarik berisi kira-kira enam belas elemen atau kurang. Sangat baik pula pada data yang hampir terurut, dan di situ ia mendekati kasus terbaiknya, $\Theta(n)$. Ia juga bersifat **daring**: ia dapat mengurutkan data sembari data itu berdatangan, tanpa mengetahui lebih dahulu akan ada berapa banyak.

Kelemahannya. Ia sering menghasilkan algoritma $\Theta(n^2)$, sebab memperluas solusinya sebanyak satu elemen bisa menuntut penyentuhan seluruh elemen yang sudah diproses: $1 + 2 + \cdots + n$ lagi.

2.15.2 Bagi dan kuasai

Pecahlah problemnya menjadi subproblem yang lebih kecil **dari jenis yang sama**, selesaikan subproblem itu secara rekursif, lalu gabungkan jawabannya. Selalu tiga langkah: *bagi*, *kuasai*, *gabung*.

Binary Search adalah contoh yang sesederhana-sederhananya. Bagi: hitung *mid* dan putuskan paruh mana yang dapat memuat q. Kuasai: cari di paruh itu secara rekursif. Gabung: tidak ada yang perlu dikerjakan — jawaban subproblemnya adalah jawabannya. Karena ia membuang satu paruh alih-alih menyelesaikan keduanya, ia kadang disebut *decrease and conquer* (kurangi dan kuasai).

Merge Sort, yang Anda jumpai di awal Bab 3, adalah versi utuhnya. Bagi: belah lariknya menjadi dua paruh. Kuasai: urutkan tiap paruh secara rekursif. Gabung: leburkan kedua paruh terurut itu dalam waktu linear. Waktu jalannya $\Theta(n\lg n)$ — jauh lebih baik daripada $\Theta(n^2)$ milik Insertion Sort untuk n yang besar.

Kekuatannya. Sering menghasilkan $\Theta(n\lg n)$ atau $\Theta(\lg n)$ di tempat pendekatan inkremental menghasilkan $\Theta(n^2)$. Rekursif secara alami, sehingga kodenya sering lebih pendek daripada padanan iteratifnya. Terparalelkan dengan baik, sebab subproblem yang saling bebas dapat diselesaikan serentak.

Kelemahannya. Rekursi punya biaya tambahan — pemanggilan fungsi tidaklah gratis — dan untuk masukan kecil biaya itu dapat melampaui penghematannya. Ia juga sering memerlukan memori tambahan untuk langkah penggabungannya: Merge Sort memerlukan ruang coret-coret $\Theta(n)$, sedangkan Insertion Sort tidak memerlukannya sama sekali.

2.15.3 Algoritma ad hoc

Frasa Latin *ad hoc* berarti "untuk ini" — solusi yang dirancang untuk satu problem tertentu saja, yang memanfaatkan struktur matematis yang khas bagi problem itu, dan yang tidak dapat dirampatkan ke apa pun yang lain.

Bab 1 memuat dua di antaranya:

- **Columns:** jawabannya $\max(n, \max_i a_i)$. Sebuah rumus langsung, yang dicapai dengan menyadari suatu sifat problemnya, bukan dengan menerapkan suatu paradigma.
- **Interesting Sequence:** jawabannya $\text{popcount}(m-1) \bmod 3$. Sebuah identitas manipulasi bit yang khas bagi cara barisan tertentu itu dibangun.

Solusi ad hoc biasanya adalah kode tercepat yang akan pernah Anda tulis — sering $O(1)$ — dan sekaligus yang paling sukar ditemukan, sebab tidak ada metodenya. Anda menemukannya dengan menatap kasus-kasus kecil sampai sebuah pola muncul, dan persis itulah yang diminta Fase 2 Kerangka Lima Fase dari Anda.

> ✓ **WAWASAN KUNCI:** Paradigma adalah apa yang Anda raih ketika Anda belum melihat kiatnya. Paradigma itu umum, dapat diajarkan, dan dapat diandalkan. Wawasan ad hoc lebih cepat ketika ia ada, tetapi Anda tidak dapat merencanakan untuk memperolehnya. Pelajarilah paradigmanya supaya Anda selalu punya langkah; teruslah mencari kiatnya supaya kadang-kadang Anda punya langkah yang lebih baik.

2.16 Menyambungkan Bab 1 dan Bab 2

Di Bab 1 Anda menyelesaikan enam problem dan berkenalan dengan Kerangka Lima Fase, tetapi klaim kompleksitas yang kita buat tentang solusi-solusi itu hanyalah penegasan. Sekarang Anda dapat membuktikan setiap satunya. **Tabel 2.6** menengok kembali keenamnya dengan perkakas bab ini diterapkan padanya.

Tabel 2.6 Keenam problem Bab 1, ditelaah ulang dengan perkakas analisis Bab 2. Setiap klaim kompleksitas yang dibuat secara tak resmi di Bab 1 kini menjadi sesuatu yang dapat Anda turunkan.

Problem Bab 1	Paradigma	Konsep Bab 2 yang dipakai	Kompleksitas
Columns	Greedy / langsung	Sapuan inkremental sekali jalan	$O(n)$ — linear, sebaik yang mungkin
Journey	Greedy	Minimum dan maksimum dalam satu sapuan	$O(n)$ — batas bawah untuk membaca masukannya
Unique Numbers	Teori bilangan	GCD Euclid, $O(\lg n)$ per pemanggilan	$O(k\lg n)$ kasus terburuk
Min/Max Numbers	Konstruksi greedy	Cetak digitnya secara inkremental	$O(N)$ per kasus uji
Interesting Sequence	Manipulasi bit	Instruksi CPU bawaan $O(1)$	$O(1)$ — waktu konstan
Recursive Function 1	Memoisasi / DP	Akses map $O(\lg D)$ per pencarian, D kunci berbeda	$O(\lg^2 n \cdot \lg\lg n)$

Tiga baris di antaranya pantas memperoleh komentar.

Columns dan Journey adalah $O(n)$ dan tidak dapat diperbaiki, dengan alasan yang kita buktikan tentang *linear search* di Subbab 2.13.4: algoritma benar mana pun sekurang-kurangnya harus membaca masukannya, dan masukannya berisi n bilangan. Membaca n bilangan berongkos n langkah. Argumen "batas bawah" semacam ini — bahwa tidak ada algoritma apa pun yang dapat lebih baik — adalah salah satu hal paling berguna yang dapat Anda tegakkan, sebab ia memberi tahu kapan harus berhenti mengoptimasi.

Interesting Sequence adalah $O(1)$ sebab `__builtin_popcount` terkompilasi menjadi satu instruksi mesin tunggal pada prosesor modern mana pun. Inilah paradigma ad hoc di Subbab 2.15.3 pada bentuknya yang paling ekstrem: jawabannya sama sekali tidak bergantung pada n.

Baris Recursive Function 1 memperlihatkan mengapa ukuran masukan harus didefinisikan dengan cermat (Subbab 2.7.1). Setiap argumen yang pernah dijumpai rekursinya berbentuk $\lfloor n/(2^a 3^b) \rfloor$ dengan $a \le \lg n$ dan $b \le \log_3 n$, sehingga tabel memonya memuat $D = O(\lg^2 n)$ kunci yang berbeda — 1,012 kunci untuk $n = 10^{18}$, sebagaimana diukur di Subbab 1.11.2. Setiap kunci dihitung sekali, dan tiap pencarian `std::map` berongkos $O(\lg D) = O(\lg\lg n)$. Kalikan semuanya untuk memperoleh isi tabelnya. Tanpa memoisasi, rekursi yang sama melakukan $2f(n) - 1$

pemanggilan, sekitar 4.5×10^{14} untuk $n = 10^{18}$ — selisih antara seketika dan lima puluh dua hari.

2.16.1 Letak bab ini di dalam buku

Tabel 2.7 memetakan topik-topik buku ini ke CLRS [1], sehingga Anda selalu tahu ke mana harus membaca lebih jauh.

Tabel 2.7 Topik-topik buku ini dipetakan ke *Introduction to Algorithms*, edisi ke-4 (geometri komputasional hanya ada pada edisi ke-3). Bab ini adalah baris satu dan dua.

Topik	Bab CLRS	Konsep kunci
Pengantar dan memulai	1–2	Insertion Sort, invarian gelung
Kebenaran dan analisis asimtotik	2–4	Big-O, Θ, Ω, menyelesaikan rekurensi
Heapsort dan Quicksort	6–7	Struktur data heap, pemartisian
Pohon AVL	13	BST seimbang, rotasi
Pemrograman dinamis	14	Substruktur optimal, memoisasi
Algoritma greedy	15	Pemilihan aktivitas, pengodean Huffman
Graf	20	BFS, DFS, pengurutan topologis
MST dan lintasan terpendek	21–23	Prim, Kruskal, Dijkstra, Bellman–Ford, Floyd–Warshall
Analisis amortisasi	16	Metode agregat, akuntansi, potensial
Geometri komputasional (bacaan lanjutan)	33 pada edisi ke-3	Selubung konveks, perpotongan garis

Di Bab 3 kita membuat persis segala yang dikerjakan bab ini secara tak resmi: definisi formal O, Θ, dan Ω; cara *membuktikan* bahwa $f(n) = O(g(n))$ alih-alih menegaskannya; cara menyelesaikan rekurensi dengan penyulihan, pohon rekursi, dan Teorema Master; serta analisis Merge Sort, $T(n) = 2T(n/2) + \Theta(n) \Rightarrow \Theta(n\lg n)$, yang akhirnya mengalahkan kuadratik yang kita turunkan di sini.

2.17 Profitina dalam Praktik

Profitina adalah platform visibilitas AI: ia memindai sebuah situs web, memberi skor seberapa siap tiap halaman untuk **dikutip** oleh mesin jawab AI, dan menjelaskan setiap sinyal di balik skor itu. Produknya hidup di app.profitina.com, dibangun dengan Python di atas PostgreSQL, dengan model Qwen2.5 lokal yang dilayani melalui Ollama untuk mesin kontennya. Tidak satu pun dari semua itu terdengar seperti bahan Bab 2. Padahal ia bahan Bab 2, pada empat tempat.

Ukuran masukan adalah bahasa yang dipakai menuliskan kendala Anda. Situs sebuah merek bisa berisi 40 halaman atau 40.000. Setiap pengukuran dalam pemindainya harus diperikan sebagai fungsi dari cacah halaman n, panjang halaman rata-rata m, atau keduanya — pelajaran Subbab 2.7.1, langsung dari buku teks. Pemeriksaan sinyal yang $\Theta(n)$ berjalan pada setiap pemindaian tanpa ada yang menyadarinya. Pemeriksaan yang membandingkan tiap halaman dengan setiap halaman lain adalah $\Theta(n^2)$: pada 40 halaman itu 1.600 pembandingan dan tak terlihat; pada 40.000 halaman itu 1,6 miliar, dan menurut **Tabel 2.4** itu berarti beberapa menit CPU untuk tiap pemindaian. Karena itu pemindainya memiliki anggaran kompleksitas yang tertulis untuk tiap sinyal, dan sinyal yang tidak dapat dihitung di dalam anggaran itu tidak dikirim ke produksi. Inilah **Tabel 2.4** yang dipakai persis seperti yang diuraikan Subbab 2.12.5: kendalanya diketahui lebih dahulu, dan kendala itulah yang memutuskan algoritmanya.

Data yang hampir terurut membuat metode inkremental menjadi metode yang tepat. Papan analitik historisnya menyimpan tren skor mingguan untuk tiap merek. Skornya datang dalam urutan waktu, hampir selalu — tetapi tidak selalu, sebab pemindaian ulang yang gagal lalu diulang, atau pengisian mundur setelah gangguan, dapat mengantarkan satu minggu di luar urutannya. Memelihara deret itu dengan menyisipkan tiap titik baru ke tempatnya adalah metode inkremental di Subbab 2.15.1, dan pada masukan yang hampir terurut ia berjalan mendekati kasus terbaiknya, $\Theta(n)$ (Subbab 2.11.1), bukan kasus terburuknya, $\Theta(n^2)$. Memilih pengurutan $\Theta(n\lg n)$ yang "lebih baik" di sini justru akan lebih lambat dalam praktik: inilah kasus yang digambarkan **Gambar 2.12**, tempat konstantanya yang menentukan.

Binary search-lah yang membuat penjadwalnya terjangkau. Penjadwal pemindaian ulang harus berulang kali bertanya "merek mana yang jatuh tempo sekarang?" terhadap sebuah daftar yang dijaga terurut menurut waktu jatuh tempo berikutnya. Mengurutkan sekali lalu mencari berkali-kali adalah tawar-menawar di Subbab 2.14.8, dengan $k \gg \lg n$ — penjadwalnya bertanya jauh lebih sering daripada daftarnya berubah, sehingga prapemrosesannya membayar dirinya sendiri berkali-kali setiap harinya.

Kasus terburuk bukan pesimisme; ia adalah SLA. Sebuah pemindaian punya batas waktu. Sebuah kueri tren mingguan punya anggaran waktu muat halaman. Argumen Subbab 2.11.5 adalah argumen operasionalnya: waktu pindai rata-rata memberi tahu apa yang biasanya terjadi, dan batas kasus terburuk memberi tahu apa yang dapat Anda janjikan. Dan karena sebuah pemindaian berjalan atas situs yang *dikendalikan orang lain*, masukannya bersifat lawan sejak awalnya — sebuah halaman dapat sepanjang apa pun, bersarang sedalam apa pun, atau penuh bagian yang nyaris kembar. Persis itulah keadaan yang membuat argumen kasus rata-rata tidak berharga.

> **Sepatah kata tentang tanggung jawab.** Apa pun yang dihitung Profitina tentang sebuah halaman adalah pengukuran dan penjelasan, tidak pernah jaminan tentang bagaimana sebuah mesin AI akan berperilaku. Aplikasinya menasihati; penggunanya yang memutuskan. Dan setiap pemindaian yang menyentuh data pribadi terikat pada UU PDP 27/2022 — persetujuannya harus tersurat dan dipahami, bukan sekadar diklik.

2.17.1 Studi kasus yang dapat dijalankan: Profitina Laundry

Studi kasus praktik dalam buku ini selalu **Profitina Laundry**, sebuah usaha kecil untuk pengajaran, tidak pernah produk yang sesungguhnya. Inilah bahan bab ini saat bekerja di dalamnya.

Profitina Laundry menerima buntelan cucian sepanjang hari. Tiap pelanggan memperoleh nota bernomor. Buku loketnya harus dijaga terurut menurut nomor nota, sebab ketika pelanggan kembali untuk mengambil cuciannya, petugas harus menemukan notanya dengan cepat. Nota dikeluarkan berurutan, sehingga kedatangannya *hampir* terurut — tetapi sesekali ada pelanggan yang mengganti layanan lalu dibukukan ulang, dan itu membuat satu nota keluar dari urutannya.

Persis itulah keadaan yang menjadi alasan lahirnya metode inkremental. **Kode 2.10** menjaga bukunya tetap terurut dengan menyisipkan tiap kedatangan ke tempatnya, dan menemukan nota dengan *binary search*. Ia menghitung kedua ongkosnya sehingga keduanya dapat dibandingkan.

Kode 2.10 Buku loket Profitina Laundry: penyisipan inkremental untuk menjaganya tetap terurut, binary search untuk menemukan sebuah nota.

```
// Profitina Laundry: buku loket hari itu.
// Pesanan berdatangan sepanjang hari, masing-masing dengan nomor nota. Buku
// itu dijaga selalu terurut menurut nomor nota dengan metode inkremental
// (gelung dalam Insertion Sort), dan pelanggan yang mengambil pesanannya
// ditemukan dengan binary search. Kerjanya kita hitung supaya kedua ongkosnya
// dapat dibandingkan.
#include <iostream>
#include <vector>
#include <string>
#include <iomanip>
using namespace std;

struct Order {
    int    ticket;       // nomor yang tercetak pada nota pelanggan
    double kg;           // berat buntelannya
    string service;      // "reguler", "kilat", "setrika"
};

long long shifts = 0;    // total perpindahan data oleh seluruh penyisipan

// Sisipkan satu pesanan ke dalam buku yang sudah terurut. Ini persis tubuh
// gelung luar Insertion Sort, dipakai satu kedatangan pada satu waktu.
void insertOrder(vector<Order>& book, const Order& o) {
    book.push_back(o);                        // sediakan tempat di ujungnya
    int i = (int)book.size() - 2;             // elemen terakhir bagian yang terurut
    while (i >= 0 && book[i].ticket > o.ticket) {
        book[i + 1] = book[i];                // geser ia ke kanan
        --i;
        ++shifts;
    }
    book[i + 1] = o;
}

// Temukan sebuah nota dalam buku yang terurut. Mengembalikan posisinya, atau
// -1 bila tidak ada.
int findTicket(const vector<Order>& book, int ticket, int& probes) {
    int left = 0, right = (int)book.size() - 1;
    probes = 0;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        ++probes;
        if (book[mid].ticket == ticket) return mid;
        if (book[mid].ticket > ticket) right = mid - 1;
        else                           left  = mid + 1;
    }
    return -1;
```

Kode 2.10 (Lanjutan) *Buku loket Profitina Laundry: penyisipan inkremental untuk menjaganya tetap terurut, binary search untuk menemukan sebuah nota.*

```
}

int main() {
    // Nota dibagikan berurutan, tetapi dua pelanggan kembali untuk mengganti
    // layanannya lalu dibukukan ulang, sehingga 1104 dan 1109 datang di luar
    // urutannya.
    vector<Order> arrivals = {
        {1101, 3.5, "reguler"}, {1102, 2.0, "kilat"},   {1103, 6.25, "reguler"},
        {1105, 1.5, "setrika"}, {1106, 4.0, "reguler"}, {1104, 8.0,  "kilat"},
        {1107, 2.75,"setrika"}, {1108, 5.0, "reguler"}, {1110, 3.0,  "kilat"},
        {1109, 7.5, "reguler"}
    };

    vector<Order> book;
    for (const Order& o : arrivals) insertOrder(book, o);

    cout << "the day's book, sorted by ticket:\n";
    double total = 0;
    for (const Order& o : book) {
        cout << "  " << o.ticket << "  " << fixed << setprecision(2)
             << setw(5) << o.kg << " kg  " << o.service << '\n';
        total += o.kg;
    }
    cout << "total weight: " << total << " kg\n";
    cout << "shifts done by all insertions: " << shifts
         << "  (worst case would be " << (arrivals.size() * (arrivals.size() - 1) / 2)
         << ")\n\n";

    for (int t : {1104, 1110, 1120}) {
        int probes;
        int at = findTicket(book, t, probes);
        cout << "collect ticket " << t << ": ";
        if (at < 0) cout << "not in today's book";
        else cout << book[at].kg << " kg " << book[at].service;
        cout << "  (" << probes << " probes)\n";
    }
    return 0;
}
```

Keluaran sesungguhnya adalah:

```
the day's book, sorted by ticket:
  1101   3.50 kg  reguler
  1102   2.00 kg  kilat
  1103   6.25 kg  reguler
  1104   8.00 kg  kilat
  1105   1.50 kg  setrika
  1106   4.00 kg  reguler
  1107   2.75 kg  setrika
  1108   5.00 kg  reguler
  1109   7.50 kg  reguler
  1110   3.00 kg  kilat
total weight: 43.50 kg
shifts done by all insertions: 3  (worst case would be 45)

collect ticket 1104: 8.00 kg kilat  (4 probes)
collect ticket 1110: 3.00 kg kilat  (4 probes)
collect ticket 1120: not in today's book  (4 probes)
```

Dua angka pada keluaran itu adalah seluruh bab ini dalam bentuk mungil.

Tiga penggeseran, berhadapan dengan kasus terburuk 45. Sepuluh pesanan yang datang dalam urutan yang sungguh-sungguh acak akan berongkos sekitar $n(n-1)/4 \approx 22$ penggeseran,

dan pada kasus terburuknya 45 (Subbab 2.11.2). Karena notanya hampir berurutan, kita hanya membayar tiga. Analisis kasus terbaik di Subbab 2.11.1 bukanlah keingintahuan teoretis — ia justru kasus yang benar-benar terjadi di loket laundry, dan itulah sebabnya algoritma sederhana adalah algoritma yang tepat di sini.

Empat penyelidikan untuk sepuluh pesanan. $\lg 10 \approx 3.3$, dan cacah pembandingan kasus terburuk yang persis, yang diturunkan di Penyelesaian 2.10, yaitu $\lceil \lg(n+1) \rceil$, memberi 4 pembandingan. Perhatikan bahwa nota 1120 yang *tidak ada* pun memakan empat penyelidikan, bukan sepuluh: *binary search* yang gagal berongkos sama dengan yang berhasil, sedangkan *linear search* yang gagal justru mencapai maksimumnya. Dengan 5.000 nota sepanjang sebulan, bukunya paling banyak memerlukan 13 penyelidikan; *linear search* akan memerlukan sampai 5.000.

2.18 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Mengacaukan algoritma dengan program. "Algoritma saya lambat karena C++ lambat" hampir tidak pernah benar. Jika program Anda seribu kali terlalu lambat, Anda memakai algoritma n^2 di tempat yang menuntut $n\lg n$, dan pergantian bahasa tidak akan menyelamatkan Anda. *Perbaikannya:* hitung operasinya sebelum Anda menyalahkan perkakasnya.

Lupa bahwa gelung bukan satu langkah. Ini kesalahan analisis yang paling umum, jauh melampaui yang lain: membaca sebuah fungsi dengan satu gelung bersarang lalu menyebutnya $O(n)$. *Perbaikannya:* terapkan **Tabel 2.2** secara mekanis. Tuliskan barisnya, tuliskan cacahnya di sebelahnya, lalu kalikan.

Melaporkan waktu jam dinding seolah-olah ia kompleksitas. "Ia memakan 0,4 detik, jadi ia linear" tidak menyatakan apa pun. *Perbaikannya:* pakai uji pelipatduaan di Subbab 2.11.4. Lipatduakan n lalu lihat nisbahnya — 2 berarti linear, 4 berarti kuadratik, sedikit di atas 2 berarti $n\lg n$.

Kehilangan kuncinya pada Insertion Sort. Menghilangkan `key = A[j]` lalu menulis `A[i+1] = A[j]` di akhir akan menghasilkan larik penuh nilai kembar, sebab penggeserannya sudah menimpa $A[j]$. *Perbaikannya:* invarian di Subbab 2.9.4 klausa (c) menyebut $A[i+1]$ sebagai sebuah *lubang* — dan lubang itu ada justru karena Anda menyimpan isinya lebih dahulu.

Menyisipkan di $A[i]$ alih-alih $A[i+1]$. Gelung dalamnya selalu menurunkan i satu langkah melewati elemen terakhir yang digesernya. *Perbaikannya:* telusuri iterasi terakhir gelungnya di atas kertas satu kali, seperti yang kita lakukan di Subbab 2.8.3, langkah 14–16, dan Anda tidak akan pernah menulisnya keliru lagi.

Menguji $A[i] > key$ sebelum $i > 0$. Ini membaca $A[0]$, di luar lariknya. Dalam C++ ia bisa tampak bekerja selama berbulan-bulan lalu merusak memori. *Perbaikannya:* pengujian batas selalu didahulukan; `&&` berhubung-singkat dari kiri ke kanan.

Menulis `mid = (left + right) / 2`. Benar dalam matematika, melimpah dalam `int`, seperti diperlihatkan **Kode 2.8**. *Perbaikannya:* `mid = left + (right - left) / 2`, selalu, sebagai gerak refleks.

Menulis `right = mid` alih-alih `right = mid - 1`. Jendelanya berhenti menyusut dan programnya menggantung pada jendela dua elemen. *Perbaikannya:* setiap kali Anda menulis *binary search*, tanyakan "apakah jendelanya menyusut secara ketat pada setiap cabang?" Jika ada kemungkinan tidak, Anda memegang sebuah gelung tak berujung.

Menjalankan binary search pada data yang tidak terurut. Ia mengembalikan NIL untuk nilai yang sebenarnya ada, tanpa suara dan dengan penuh keyakinan. *Perbaikannya:* prasyaratnya ditempatkan pada komentar di atas fungsinya dan di dalam benak pemanggilnya; tidak ada apa pun dalam kodenya yang akan memeriksanya untuk Anda.

Mengembalikan -1 untuk "tidak ditemukan" dalam program berbasis-1, atau 0 dalam program berbasis-0. *Perbaikannya:* pilihlah penanda yang mustahil menjadi indeks yang sah dalam skema Anda, lalu sebutkan yang mana itu dalam sebuah komentar.

Menganggap "terurut" itu gratis. Mengurutkan demi memungkinkan satu kali *binary search* lebih lambat daripada satu kali sapuan linear. *Perbaikannya:* Subbab 2.14.8 — hitung kueri Anda sebelum Anda membeli strukturnya.

Mengoptimasi konstanta padahal laju pertumbuhannya yang salah. Menulis ulang gelung $\Theta(n^2)$ agar dua kali lebih cepat mengubah enam belas menit menjadi delapan menit. *Perbaikannya:* **Tabel 2.3**. Ubah eksponennya, bukan koefisiennya.

Membuktikan kebenaran dengan pengujian. Sepuluh kasus uji yang lolos membuktikan bahwa sepuluh masukan bekerja. *Perbaikannya:* nyatakan invariannya, lalu periksa inisialisasi, pemeliharaan, dan terminasinya. Itu memakan lima belas menit dan mencakup tak berhingga banyaknya masukan.

2.19 Ringkasan Bab

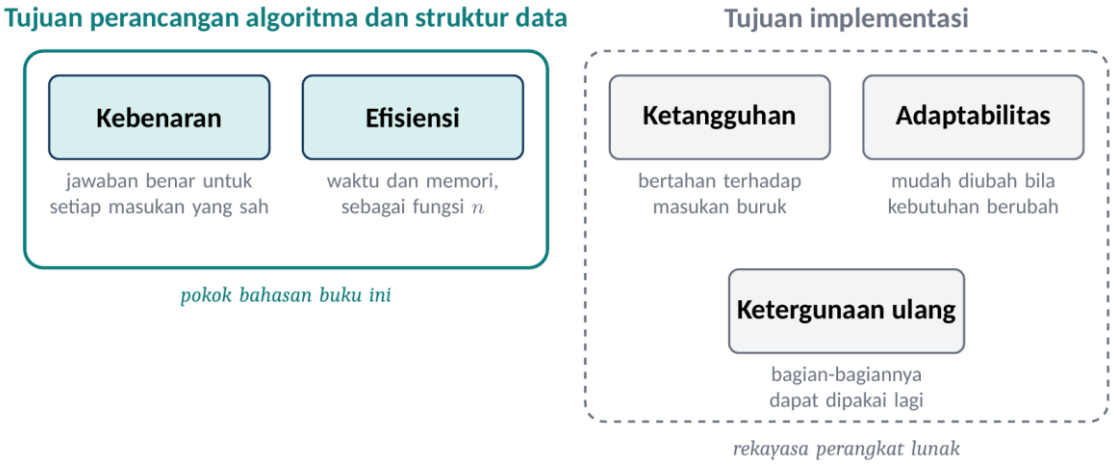
1. Sebuah **algoritma** adalah prosedur langkah demi langkah yang mengubah masukan menjadi keluaran; sebuah **program** adalah implementasinya dalam suatu bahasa; sebuah **struktur data** adalah cara datanya ditata. Sebuah **problem** menetapkan hubungan masukan/keluaran; sebuah **instans** adalah satu masukan tertentu. Kebenaran adalah sifat algoritma atas semua instans; waktu jalan biasanya bergantung pada instansnya.
2. Sebuah algoritma harus memenuhi lima sifat: masukan, keluaran, kepastian, keberhinggaan, keterlaksanaan. Menjadi algoritma adalah ambang yang rendah; menjadi algoritma yang efisien barulah tujuannya.
3. **Lima tujuan perancangan** adalah kebenaran, efisiensi, kekukuhan, keteradaptasian, dan keterpakaian ulang — dua yang pertama adalah tujuan perancangan dan menjadi pokok bahasan buku ini, tiga yang terakhir adalah tujuan implementasi yang termasuk rekayasa perangkat lunak. **Gambar 2.16** adalah petanya.
4. Kita menganalisis dalam **model RAM**: satu prosesor, akses memori seragam, tiap instruksi sederhana satu langkah. Ia penyederhanaan yang disengaja, dipilih sebab ia tidak bergantung pada mesin dan tetap meramalkan kinerja yang sesungguhnya.
5. Untuk mencari $T(n)$: berikan biaya konstan pada tiap baris, hitung berapa kali tiap baris berjalan, lalu jumlahkan hasil kalinya. Gelung tidak pernah satu langkah.
6. **Insertion Sort** benar, terbukti lewat invarian gelung "$A[1 \dots j-1]$ memuat elemen asli $A[1 \dots j-1]$ dalam keadaan terurut" melalui inisialisasi, pemeliharaan, dan terminasi.
7. Insertion Sort berjalan dalam $\Theta(n)$ pada kasus terbaik (sudah terurut, $t_j = 1$), dan $\Theta(n^2)$ pada kasus terburuk (terurut terbalik, $t_j = j$, sehingga $\sum_{j=2}^{n} j = n(n+1)/2 - 1$). Kasus rata-ratanya juga $\Theta(n^2)$, dengan konstanta kira-kira separuh konstanta kasus terburuknya.

8. Kita menganalisis **kasus terburuk** sebab ia sebuah jaminan, sebab masukan yang bersifat lawan dan masukan yang berstruktur itu lazim, dan sebab kasus rata-ratanya biasanya berorde sama juga.

9. **Laju pertumbuhan mendominasi segalanya** ketika n tumbuh: $O(1) < O(\lg n) < O(\sqrt{n}) < O(n) < O(n\lg n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$. Pada $n = 10^6$, algoritma $n\lg n$ memakan beberapa milidetik dan algoritma n^2 memakan enam belas menit.

10. **Linear search** adalah $\Theta(n)$ dan optimal pada data yang tidak terurut. **Binary search** adalah $\Theta(\lg n)$ pada data yang terurut — sebuah perbaikan sebesar eksponensial yang dibeli sepenuhnya oleh struktur, dan dibayar sekali dengan pengurutan $\Theta(n\lg n)$.

11. Paradigma **inkremental** memperluas solusi parsial satu elemen pada satu waktu (Insertion Sort). **Bagi dan kuasai** membelah menjadi subproblem sejenis (binary search, Merge Sort). Solusi **ad hoc** memanfaatkan struktur yang khas bagi satu problem.



Gambar 2.16 Kelima tujuan itu sekali lagi. Segala isi bab ini mengabdi pada dua yang di sebelah kiri — kebenaran, yang dibuktikan dengan invarian gelung, dan efisiensi, yang diturunkan dengan menghitung.

"An algorithm must be seen to be believed." — Donald Knuth [3]. *Telusurilah setiap algoritma dengan tangan. Pemahaman lahir dari pengerjaan.*

2.20 Soal Latihan

2.1. Telusurilah Insertion Sort dengan tangan pada $A = \langle 31,41,59,26,41,58 \rangle$. Berikan lariknya sesudah tiap iterasi gelung luar, dan nyatakan t_j untuk tiap j.

2.2. Untuk larik pada Soal 2.1, hitunglah $\sum_{j=2}^{6} t_j$ dan $\sum_{j=2}^{6}(t_j - 1)$. Bagaimana keduanya dibandingkan dengan kasus terbaik $(n - 1)$ dan kasus terburuk $(n(n + 1)/2 - 1)$ untuk $n = 6$?

2.3. Tulislah ulang **Kode 2.1** dengan pengindeksan berbasis-0, tanpa elemen yang tak terpakai. Nyatakan batas gelung yang baru dan pengujian gelung dalam yang baru, lalu jelaskan pembandingan mana yang berubah dan mengapa.

2.4. Ubahlah Insertion Sort agar mengurutkan ke dalam urutan **tak-naik**. Satu karakter mana yang berubah? Tuliskan invarian gelungnya yang sudah dimodifikasi.

2.5. Buktikanlah, dengan memakai invarian gelung, bahwa prosedur pada **Pseudokode 2.5** menghitung jumlah $A[1 \ldots n]$ dengan benar.

Pseudokode 2.5 Menjumlahkan elemen-elemen sebuah larik.

```
SUM(A, n)
s = 0
for i = 1 to n
    s = s + A[i]
return s
```

2.6. Sebuah **inversi** pada larik A adalah pasangan indeks (i, j) dengan $i < j$ dan $A[i] > A[j]$. Daftarkan semua inversi pada $\langle 2,3,8,6,1 \rangle$. Berapa cacah inversi terbanyak yang mungkin pada larik berisi n elemen yang berbeda-beda, dan larik yang mana yang mencapainya?

2.7. Buktikanlah bahwa total cacah penggeseran yang dilakukan Insertion Sort — yaitu $\sum_{j=2}^{n}(t_j - 1)$ — sama persis dengan cacah inversi pada masukannya. Pakailah hal ini untuk memberikan penurunan kedua bagi waktu jalan kasus terburuknya.

2.8. Tulislah sebuah program C++ yang memverifikasi klaim Soal 2.7 secara empiris pada larik-larik acak.

2.9. Insertion Sort menemukan titik penyisipannya dengan menyapu secara linear. Karena $A[1 \ldots j-1]$ sudah terurut, kita dapat saja menemukan titik itu dengan *binary search*. Apakah hal ini mengurangi waktu jalan kasus terburuk pengurutannya? Benarkan jawaban Anda dengan menghitung pembandingan dan perpindahan secara terpisah.

2.10. Berikan cacah pembandingan yang persis yang dilakukan *binary search*, pada kasus terburuk, atas larik berisi n elemen, untuk $n = 1,2,3, \ldots ,8$. Nyatakan jawaban umumnya sebagai sebuah rumus.

2.11. Larik $A = \langle 1,3,3,3,3,7,9,9,11 \rangle$ memuat nilai kembar. Menjalankan **Kode 2.7** dengan $q = 3$ mengembalikan *suatu* indeks yang memuat 3, tetapi yang mana tidak ditentukan. Tulislah sebuah varian yang mengembalikan kemunculan **pertama**, dan satu lagi yang mengembalikan kemunculan **terakhir**, keduanya tetap dalam $\Theta(\lg n)$.

2.12. Seorang mahasiswa menulis `while (left < right)` alih-alih `while (left <= right)` pada *binary search*. Berikan sebuah larik dan kueri konkret yang membuat versi itu mengembalikan jawaban yang salah, lalu telusurilah untuk memperlihatkan mengapa.

2.13. Anda harus memilih antara *linear search* dan "urutkan sekali, lalu *binary search*" untuk larik berisi $n = 2^{20}$ elemen. Turunkan cacah kueri k yang membuat pilihan kedua menjadi lebih murah, dengan menganggap pengurutannya berongkos $n \lg n$ dan tiap *binary search* berongkos $\lg n$.

2.14. Sebuah pernyataan soal menjamin $1 \leq n \leq 200{,}000$ dan mengizinkan dua detik. Kompleksitas mana saja dari **Tabel 2.4** yang dapat diterima? Jika gagasan pertama Anda adalah algoritma $O(n^2)$, berapa operasi yang akan dilakukannya pada n maksimum, dan berapa kali lipat ia melampaui anggarannya?

2.15. Insertion Sort adalah $\Theta(n^2)$ dan Merge Sort adalah $\Theta(n \lg n)$, namun pengurutan pustaka tetap memakai Insertion Sort untuk sublarik yang kecil. Andaikan Insertion Sort berongkos $8n^2$ operasi dan Merge Sort berongkos $64 n \lg n$. Untuk nilai n yang mana Insertion Sort menang?

2.21 Penyelesaian Soal Latihan

Penyelesaian 2.1. Lariknya adalah $\langle 31,41,59,26,41,58 \rangle$, $n = 6$.

- $j = 2$: $key = 41$. Uji $A[1] = 31 > 41$? Tidak. Jadi $t_2 = 1$, tanpa penggeseran. Larik: $\langle 31,41,59,26,41,58 \rangle$.

- $j = 3$: $key = 59$. Uji $A[2] = 41 > 59$? Tidak. $t_3 = 1$. Larik tak berubah.
- $j = 4$: $key = 26$. $A[3] = 59 > 26$ ya, geser. $A[2] = 41 > 26$ ya, geser. $A[1] = 31 > 26$ ya, geser. $i = 0$, pengujiannya gagal. Empat pengujian, jadi $t_4 = 4$, tiga penggeseran. Larik: $\langle 26,31,41,59,41,58 \rangle$.
- $j = 5$: $key = 41$. $A[4] = 59 > 41$ ya, geser. $A[3] = 41 > 41$? Tidak — pengujiannya ketat, sehingga elemen yang sama menghentikan sapuannya. Dua pengujian, $t_5 = 2$, satu penggeseran. Larik: $\langle 26,31,41,41,59,58 \rangle$.
- $j = 6$: $key = 58$. $A[5] = 59 > 58$ ya, geser. $A[4] = 41 > 58$? Tidak. $t_6 = 2$, satu penggeseran. Larik: $\langle 26,31,41,41,58,59 \rangle$.

Akhirnya: $\langle 26,31,41,41,58,59 \rangle$. Perhatikan pada $j = 5$ bahwa tanda `>` yang ketat itulah yang menjaga kedua angka 41 tetap pada urutan relatif aslinya — sifat itu disebut **kestabilan** (*stability*), dan itulah sebabnya yang tertulis pada baris 4 adalah `>` dan bukan `>=`.

Penyelesaian 2.2. $\sum t_j = 1 + 1 + 4 + 2 + 2 = 10$ dan $\sum(t_j - 1) = 0 + 0 + 3 + 1 + 1 = 5$. Untuk $n = 6$ kasus terbaiknya memberi $\sum t_j = n - 1 = 5$ dan kasus terburuknya memberi $n(n+1)/2 - 1 = 20$. Nilai kita, 10, terletak di antara keduanya dan dekat dengan ramalan kasus rata-rata, yaitu kira-kira separuh kasus terburuk, yang berarti 10. Masukannya tidak terurut secara sedang-sedang saja.

Penyelesaian 2.3. Dengan indeks berbasis-0, awalan yang terurut adalah $A[0 \ldots j-1]$. **Kode 2.11** adalah program lengkapnya, yang mengurutkan larik yang sama dengan **Kode 2.1**.

Kode 2.11 Insertion Sort dengan pengindeksan berbasis-0 yang biasa, tanpa elemen yang dibiarkan kosong.

```
// Soal 2.3: Insertion Sort dengan pengindeksan berbasis-0 yang biasa.
// Awalan yang terurut kini A[0..j-1], dan indeks 0 adalah posisi yang sah.
#include <iostream>
#include <vector>
using namespace std;

void insertionSort0(vector<int>& A) {
    int n = (int)A.size();
    for (int j = 1; j < n; ++j) {          // sebelumnya: j = 2 sampai n
        int key = A[j], i = j - 1;
        while (i >= 0 && A[i] > key) {     // sebelumnya: i > 0
            A[i + 1] = A[i]; --i;
        }
        A[i + 1] = key;
    }
}

int main() {
    vector<int> A = {5, 2, 4, 6, 1, 3};    // kali ini tanpa elemen yang dibiarkan kosong
    insertionSort0(A);
    for (size_t i = 0; i < A.size(); ++i) cout << A[i] << (i + 1 < A.size() ? ' ' : '\n');
    return 0;
}
```

Dikompilasi dengan `g++ -std=c++17 -O2 -Wall` lalu dijalankan, program itu mencetak:

```
1 2 3 4 5 6
```

Ada dua perubahan. Gelung luarnya mulai dari $j = 1$ (elemen kedua kini berada di indeks 1, bukan 2) dan berjalan selama $j < n$. Penjaga gelung dalamnya menjadi $i \geq 0$ alih-alih $i > 0$, sebab indeks 0 kini posisi yang sah dan harus diperiksa. Selebihnya, termasuk `A[i+1] = key`, tidak

berubah — dan itu pertanda yang berguna bahwa yang ditangkap invariannya adalah logikanya, bukan pengindeksannya.

Penyelesaian 2.4. Ubah `A[i] > key` menjadi `A[i] < key` pada baris 4: satu karakter. Invariannya menjadi "pada awal tiap iterasi, $A[1 \ldots j-1]$ terdiri atas elemen yang semula berada di $A[1 \ldots j-1]$ dalam urutan **tak-naik**", dan setiap langkah pembuktian di Subbab 2.9 berlaku dengan ketaksamaannya dibalik.

Penyelesaian 2.5. *Invarian:* pada awal tiap iterasi gelung di baris 2, $s = \sum_{k=1}^{i-1} A[k]$.

Inisialisasi: sebelum iterasi pertama $i = 1$, jadi klaimnya adalah $s = \sum_{k=1}^{0} A[k]$, sebuah penjumlahan kosong, yang menurut definisi bernilai 0. Baris 1 menetapkan $s = 0$. ✓

Pemeliharaan: andaikan $s = \sum_{k=1}^{i-1} A[k]$. Baris 3 menetapkan $s \leftarrow s + A[i] = \sum_{k=1}^{i-1} A[k] + A[i] = \sum_{k=1}^{i} A[k]$. Gelungnya lalu menaikkan i, sehingga dengan nilai i yang baru invariannya kembali berbunyi $s = \sum_{k=1}^{i-1} A[k]$. ✓

Terminasi: gelungnya berakhir dengan $i = n + 1$, sehingga $s = \sum_{k=1}^{n} A[k]$, yaitu jumlah seluruh lariknya. Baris 4 mengembalikannya. ✓ ■

Penyelesaian 2.6. Untuk $\langle 2,3,8,6,1 \rangle$ dengan indeks berbasis-1, inversinya adalah (1,5) sebab $2 > 1$; (2,5) sebab $3 > 1$; (3,4) sebab $8 > 6$; (3,5) sebab $8 > 1$; dan (4,5) sebab $6 > 1$. Itu **lima** inversi.

Maksimumnya adalah $\binom{n}{2} = \frac{n(n-1)}{2}$, yang dicapai oleh larik yang menurun ketat: setiap satu dari $\binom{n}{2}$ pasangan $i < j$ lalu memenuhi $A[i] > A[j]$, dan tidak ada larik yang dapat memiliki pasangan lebih banyak daripada cacah pasangan yang dimilikinya.

Penyelesaian 2.7. Tiap eksekusi baris 5 memindahkan satu elemen tepat satu posisi ke kanan, melewati kuncinya. Sebuah elemen $A[i]$ dipindahkan melewati $key = A[j]$ persis ketika $i < j$ dan $A[i] > key$ — artinya, persis ketika (i, j) adalah sebuah inversi. Sebaliknya, tiap inversi (i, j) menyebabkan tepat satu penggeseran: ketika gelung luarnya mencapai j, elemen yang semula berada di i masih berada di sebelah kiri tempat mendaratnya kunci dan lebih besar daripadanya, sehingga ia digeser sekali dan tidak pernah lagi. Padanannya satu-satu, sehingga total cacah penggeserannya sama dengan cacah inversinya.

Penurunan kedua bagi kasus terburuk: menurut Soal 2.6 cacah inversi terbanyak adalah $n(n-1)/2$, yang dicapai oleh larik terurut terbalik. Jadi cacah penggeseran terbanyak adalah $n(n-1)/2 = \Theta(n^2)$, dan karena tiap penggeseran berongkos $\Theta(1)$ sedangkan kerja sisanya $\Theta(n)$, waktu jalan kasus terburuknya adalah $\Theta(n^2)$ — kesimpulan yang sama dengan Subbab 2.11.2, yang dicapai tanpa menjumlahkan sebuah deret.

Penyelesaian 2.8.

Kode 2.12 Memverifikasi bahwa cacah penggeseran Insertion Sort sama dengan cacah inversi pada masukannya.

```
// Soal: cacah penggeseran yang dilakukan Insertion Sort persis sama dengan
// cacah inversi pada masukannya. Program ini menghitung keduanya lalu
// membandingkannya. Sebuah inversi adalah pasangan posisi (i, j) dengan
// i < j dan A[i] > A[j].
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
using namespace std;

long long countInversions(const vector<int>& A, int n) {
    long long inv = 0;
    for (int i = 1; i <= n; ++i)
        for (int j = i + 1; j <= n; ++j)
            if (A[i] > A[j]) ++inv;          // definisinya secara gaya kasar
    return inv;
}

long long insertionShifts(vector<int> A, int n) {   // per nilai: kita ubah salinannya
    long long shifts = 0;
    for (int j = 2; j <= n; ++j) {
        int key = A[j], i = j - 1;
        while (i > 0 && A[i] > key) { A[i + 1] = A[i]; --i; ++shifts; }
        A[i + 1] = key;
    }
    return shifts;
}

int main() {
    vector<int> demo = {0, 2, 3, 8, 6, 1};              // contoh dari CLRS
    cout << "A = [2,3,8,6,1]: inversions = " << countInversions(demo, 5)
         << ", shifts = " << insertionShifts(demo, 5) << '\n';

    mt19937 rng(2024);
    bool allEqual = true;
    for (int trial = 0; trial < 200; ++trial) {
        int n = 1 + (int)(rng() % 40);
        vector<int> A(n + 1);
        for (int i = 1; i <= n; ++i) A[i] = (int)(rng() % 100);
        if (countInversions(A, n) != insertionShifts(A, n)) allEqual = false;
    }
    cout << "200 random trials agree: " << (allEqual ? "yes" : "no") << '\n';
    return 0;
}
```

Keluaran **Kode 2.12** adalah:

```
A = [2,3,8,6,1]: inversions = 5, shifts = 5
200 random trials agree: yes
```

Kelima inversi yang ditemukan program itu adalah kelima inversi yang didaftarkan dengan tangan di Penyelesaian 2.6. Perhatikan bahwa larik pada percobaan acaknya memuat nilai kembar, dan identitasnya tetap berlaku — sebab tanda `>` yang ketat, baik pada definisi inversi maupun pada pengujian gelungnya, memperlakukan elemen yang sama dengan cara yang sama pula.

Penyelesaian 2.9. Tidak: kasus terburuknya tetap $\Theta(n^2)$.

Hitunglah kedua jenis kerjanya secara terpisah. **Pembandingannya** memang turun: *binary search* menemukan titik penyisipan di antara $j-1$ elemen terurut dengan $\Theta(\lg j)$ pembandingan alih-alih sampai j pembandingan, sehingga totalnya turun dari $\Theta(n^2)$ menjadi $\sum_{j=2}^{n} \Theta(\lg j) = \Theta(n \lg n)$. Tetapi **perpindahannya** sama sekali tidak berubah. Begitu titik penyisipannya diketahui, setiap

elemen di sebelah kanannya tetap harus digeser satu tempat, dan pada kasus terburuk itu $j - 1$ perpindahan untuk tiap j, yang berjumlah $n(n-1)/2 = \Theta(n^2)$ seperti sebelumnya.

Waktu jalannya adalah jumlah keduanya, jadi ia tetap $\Theta(n^2)$. Pelajarannya bersifat umum dan penting: mempercepat satu bagian sebuah algoritma tidak mengubah apa pun jika bagian lain berorde sama. Varian ini baru berharga ketika pembandingan jauh lebih mahal daripada perpindahan — misalnya ketika yang dibandingkan adalah untai yang panjang sedangkan yang dipindahkan hanyalah penunjuknya.

Penyelesaian 2.10. Misalkan $C(n)$ adalah cacah pembandingan kasus terburuk pada n elemen. Tiap pembandingan menyusutkan jendelanya dari n menjadi paling banyak $\lfloor n/2 \rfloor$:

Tabel 2.8 Cacah pembandingan kasus terburuk untuk binary search pada n elemen.

n	1	2	3	4	5	6	7	8
$C(n)$	1	2	2	3	3	3	3	4

Membaca **Tabel 2.8**, $C(n)$ naik satu tingkat persis pada $n = 1,2,4,8$ — yaitu pada pangkat-pangkat dua. Rumusnya adalah $C(n) = \lfloor \lg n \rfloor + 1$, yang setara dengan $\lceil \lg(n+1) \rceil$. Periksa: $n = 5$ memberi $\lfloor \lg 5 \rfloor + 1 = 2 + 1 = 3$ ✓, dan $n = 10^6$ memberi $\lfloor 19.93 \rfloor + 1 = 20$ ✓, yaitu angka yang dikutip di Subbab 2.14.5.

Penyelesaian 2.11. Perbaikannya adalah dengan *tidak* langsung kembali begitu terjadi kecocokan. Catat posisinya, lalu teruslah mencari di sisi yang masih mungkin memuat kemunculan yang lebih awal (atau lebih akhir).

Kode 2.13 Binary search untuk kemunculan pertama dan terakhir sebuah nilai, ketika ada nilai kembar.

```
// Soal: temukan kemunculan PERTAMA q pada larik terurut yang bernilai kembar.
// Binary search biasa bisa mendarat di tengah-tengah deretan nilai yang sama,
// jadi gelungnya harus terus bergerak ke kiri setelah sebuah kecocokan alih-
// alih langsung kembali.
#include <iostream>
#include <vector>
using namespace std;

// Mengembalikan indeks i terkecil (berbasis-0) dengan A[i] == q, atau -1 bila tidak ada.
int firstOccurrence(const vector<int>& A, int q) {
    int left = 0, right = (int)A.size() - 1, answer = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (A[mid] == q) { answer = mid; right = mid - 1; }  // ingat, lalu terus ke kiri
        else if (A[mid] > q) right = mid - 1;
        else                 left  = mid + 1;
    }
    return answer;
}

// Mengembalikan indeks i terbesar (berbasis-0) dengan A[i] == q, atau -1 bila tidak ada.
int lastOccurrence(const vector<int>& A, int q) {
    int left = 0, right = (int)A.size() - 1, answer = -1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (A[mid] == q) { answer = mid; left = mid + 1; }   // terus ke kanan
        else if (A[mid] > q) right = mid - 1;
        else                 left  = mid + 1;
    }
    return answer;
}

int main() {
    vector<int> A = {1, 3, 3, 3, 3, 7, 9, 9, 11};
    for (int q : {3, 9, 5}) {
        int f = firstOccurrence(A, q), l = lastOccurrence(A, q);
        cout << "q=" << q << ": first=" << f << " last=" << l
             << " count=" << (f < 0 ? 0 : l - f + 1) << '\n';
    }
    return 0;
}
```

Keluaran **Kode 2.13** adalah:

```
q=3: first=1 last=4 count=4
q=9: first=6 last=7 count=2
q=5: first=-1 last=-1 count=0
```

Kedua variannya tetap $\Theta(\lg n)$: jendelanya tetap sekurang-kurangnya membelah dua setiap iterasi, sebab `right = mid - 1` maupun `left = mid + 1` sama-sama mengecualikan mid. Kerja tambahannya adalah bahwa kita tidak pernah keluar lebih awal, sehingga kita selalu membayar penuh $\lfloor \lg n \rfloor + 1$ pembandingan alih-alih kadang-kadang lebih sedikit — biaya sebesar faktor konstan untuk memperoleh jawaban yang eksak. Mengurangkan kedua hasilnya memberikan cacah kemunculannya dalam $\Theta(\lg n)$, dan begitulah cara menghitung nilai kembar pada larik terurut tanpa menyapunya. Inilah rutin yang dalam pustaka baku C++ disebut `std::lower_bound` dan `std::upper_bound` [4].

Penyelesaian 2.12. Ambil $A = \langle 5 \rangle$ dengan $n = 1$ dan $q = 5$, memakai indeks berbasis-1.

Dengan `while (left < right)`: semula $left = 1$, $right = 1$. Pengujian $1 < 1$ bernilai **salah**, sehingga tubuh gelungnya tidak pernah berjalan dan fungsinya mengembalikan NIL — padahal $A[1] = 5 = q$. Jawaban yang salah.

Kutunya tidak terbatas pada $n = 1$. Ambil $A = \langle 2,4 \rangle$ dan $q = 4$: $left = 1$, $right = 2$, pengujian $1 < 2$ benar, $mid = 1$, $A[1] = 2 < 4$, sehingga $left = 2$. Sekarang pengujian $2 < 2$ salah dan gelungnya keluar dengan NIL, padahal $A[2] = 4$. Secara umum `<` gagal setiap kali jawabannya adalah elemen terakhir yang masih berada di dalam jendela, sebab jendela satu elemen tidak pernah diperiksa. Versi `<=` memeriksanya dan karena itu benar — dan inilah persis golongan kesalahan selisih-satu yang menurut laporan Bentley menjangkiti percobaan pertama sebagian besar pemrogram pada *binary search* [2].

Penyelesaian 2.13. Dengan $n = 2^{20} = 1{,}048{,}576$ kita memperoleh $\lg n = 20$.

- Linear search: k kueri berongkos $kn = 1{,}048{,}576\,k$.
- Urutkan lalu binary search: $n\lg n + k\lg n = 20{,}971{,}520 + 20k$.

Samakan keduanya:

$$1{,}048{,}576\,k = 20{,}971{,}520 + 20k \;\Rightarrow\; 1{,}048{,}556\,k = 20{,}971{,}520 \;\Rightarrow\; k \approx 20.0.$$

Jadi mulai dari sekitar 20 kueri ke atas, pengurutannya membayar dirinya sendiri. Secara umum titik impasnya adalah

$$k \approx \frac{n\lg n}{n - \lg n} \approx \lg n, \tag{2.13}$$

dan Persamaan (2.13) adalah aturan yang menyenangkan: **pengurutan sepadan begitu Anda memperkirakan akan ada lebih dari sekitar** $\lg n$ **pencarian.** Untuk sejuta elemen itu berarti dua puluh pencarian — ambang yang sangat rendah, dan itulah sebabnya dalam praktik orang hampir selalu mengurutkan.

Penyelesaian 2.14. Dari **Tabel 2.4**, $n \leq 200{,}000$ berada pada pita $O(n\lg n)$, sehingga $O(n\lg n)$, $O(n)$, $O(\lg n)$, dan $O(1)$ dapat diterima; $O(n^2)$ tidak.

Algoritma $O(n^2)$ pada $n = 200{,}000$ melakukan $(2 \times 10^5)^2 = 4 \times 10^{10}$ operasi. Pada angka optimistis 10^9 operasi per detik, itu 40 detik berhadapan dengan batas 2 detik: **20 kali lipat melampaui anggaran**. Tidak ada optimasi faktor konstan yang dapat memulihkan faktor dua puluh secara andal. Sebaliknya $O(n\lg n)$ memberi $2 \times 10^5 \times 17.6 \approx 3.5 \times 10^6$ operasi — beberapa milidetik saja, dengan kelonggaran empat orde besaran. Inilah hitungan yang harus dilakukan *sebelum* menulis kode, seperti yang ditekankan baik oleh Kerangka Lima Fase di Bab 1 maupun oleh tabel kendala CP4 [5].

Penyelesaian 2.15. Insertion Sort menang ketika $8n^2 < 64n\lg n$. Bagilah kedua ruas dengan $8n$, yang bernilai positif:

$$n < 8\lg n.$$

Selesaikan dengan pemeriksaan, sebab persamaan $n = 8\lg n$ tidak punya bentuk tertutup:

- $n = 8$: apakah $8 < 8 \times 3 = 24$? Ya.
- $n = 16$: apakah $16 < 8 \times 4 = 32$? Ya.
- $n = 32$: apakah $32 < 8 \times 5 = 40$? Ya.
- $n = 43$: $\lg 43 \approx 5.43$, sehingga $8\lg n \approx 43.4$. Apakah $43 < 43.4$? Ya, nyaris.
- $n = 44$: $\lg 44 \approx 5.46$, sehingga $8\lg n \approx 43.7$. Apakah $44 < 43.7$? Tidak.

Jadi Insertion Sort lebih cepat untuk $n \leq 43$ dan Merge Sort mulai dari $n \geq 44$. Persis itulah sebabnya pengurutan di lingkungan produksi bersifat hibrida: bergulunglah dengan algoritma yang asimtotiknya lebih baik sampai sublariknya kecil, lalu beralihlah ke algoritma yang konstantanya lebih baik. Implementasi nyata memakai ambang di kisaran 16 sampai 32, sedikit di bawah 43 yang kita idealkan, sebab konstanta yang sesungguhnya bukanlah angka bulat seperti pada soal ini [4].

Bacaan lanjutan

Bab 1 sampai 4 CLRS [1] membahas materi ini dalam bentuk kanoniknya dan sebaiknya menjadi perhentian pertama Anda; Jilid 1 Knuth [3] adalah sumber bagi kelima sifat algoritma, dan Jilid 3 [6] memuat pembahasan yang paling lengkap tentang pengurutan sisip dan *binary search*, termasuk catatan bahwa meskipun gagasannya diterbitkan pada 1946, implementasi yang *benar* untuk larik dengan panjang sembarang baru terbit pada 1962. Bentley [2] menceritakan betapa sukarnya hal yang terakhir itu sebenarnya. Untuk sudut pandang pemrograman kompetitif atas tabel kendala di Subbab 2.12.5, lihat CP4 [5].

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] J. Bentley, *Programming Pearls*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2000.

[3] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[4] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[5] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, 4th ed. Raleigh, NC, USA: Lulu Press, 2020.

[6] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

Bab 3

Notasi Asimtotik, Rekurensi, dan Teorema Master

Di akhir bab ini Anda akan sanggup mengambil algoritma apa pun yang memanggil dirinya sendiri, menuliskan persamaan yang menggambarkan waktu jalannya, lalu menyelesaikan persamaan itu dengan tiga cara yang berbeda — dengan menebak lalu membuktikan, dengan menggambar pohon, dan dengan sebuah resep yang menjawab sebagian besar persamaan semacam itu dalam waktu kira-kira tiga puluh detik. Anda akan tahu persis apa arti O, Ω, Θ, o, dan ω — bukan sebagai slogan yang kabur, melainkan sebagai pernyataan tentang konstanta yang dapat Anda sebut namanya dan Anda tuliskan angkanya. Anda akan sanggup membuktikan bahwa sebuah batas berlaku, dan — ini sama pentingnya — membuktikan bahwa sebuah batas *tidak* berlaku. Anda akan sanggup menyatakan *Master Theorem* (Teorema Master) dari ingatan, menerapkan ketiga kasusnya, memeriksa syarat keteraturannya, serta mengenali dua celah tempat teorema itu diam-diam menolak menjawab. Dan Anda akan menurunkan Merge Sort dari nol sama sekali, membuktikan langkah penggabungannya benar dengan *loop invariant* (invarian loop), dan menunjukkan bahwa ia berjalan dalam waktu $\Theta(n\lg n)$.

3.1 Dari Mana Kita Melanjutkan

Bab 2 memberi kita dua hal: sebuah mesin untuk menghitung, dan sebuah metode untuk dipercaya.

Mesin itu adalah **model RAM** — *random-access machine*, mesin akses acak. Ia adalah komputer pura-pura dengan sekumpulan instruksi sederhana yang tetap (tambah, kurang, kali, banding, baca sebuah sel memori, tulis sebuah sel memori, lompat), dan setiap instruksi itu memakan satu satuan waktu. Tidak ada cache, tidak ada pipeline, tidak ada sistem operasi yang mencuri siklus. Kita menghitung instruksi, bukan detik. Alasannya bukan kemalasan; alasannya adalah bahwa detik merupakan sifat sebuah laptop tertentu pada suatu hari Selasa tertentu, sedangkan cacah instruksi merupakan sifat algoritmanya sendiri. Jika bengkel becak Anda mengganti komputer berusia sepuluh tahun dengan komputer baru yang empat kali lebih cepat, setiap algoritma yang Anda miliki menjadi empat kali lebih cepat, dan tidak satu pun yang Anda pelajari tentang algoritma mana yang lebih baik berubah karenanya.

Metode itu adalah **loop invariant**. Sebuah algoritma disebut *benar* jika, untuk setiap masukan yang sah, ia berhenti dan menghasilkan keluaran yang dikehendaki. Kita membuktikan Insertion Sort benar dengan menemukan sebuah pernyataan — "pada awal setiap iterasi gelung luar untuk indeks j, subarray $A[1..j-1]$ memuat elemen-elemen yang semula berada di $A[1..j-1]$, tetapi dalam keadaan terurut" — lalu menunjukkan tiga hal tentangnya:

1. **Inisialisasi.** Pernyataan itu benar sebelum iterasi pertama. Ketika $j = 2$, subarray $A[1..1]$ panjangnya satu elemen, dan satu elemen sudah pasti terurut.
2. **Pemeliharaan.** Jika pernyataan itu benar sebelum sebuah iterasi, ia masih benar sebelum iterasi berikutnya. Gelung `while` di dalam menggeser $A[j-1], A[j-2], \ldots$ satu posisi ke kanan, tanpa mengganggu urutan di antara mereka sendiri, sampai ia menemukan tempat yang pas

bagi kunci, lalu menjatuhkan kunci itu ke sana. Yang tadinya terurut di $A[1..j-1]$ kini terurut di $A[1..j]$.

3. **Terminasi.** Ketika gelung akhirnya berhenti — pada $j = n + 1$ — pernyataan itu berbunyi "$A[1..n]$ memuat elemen-elemen semula dalam keadaan terurut", dan itu persis yang kita inginkan.

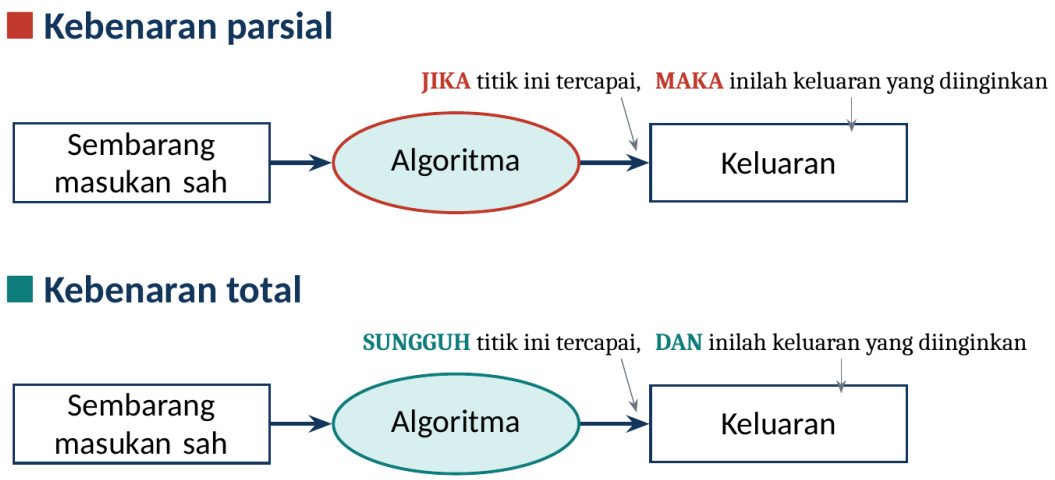
Itu adalah induksi matematis yang sedang mengenakan pakaian lain. Inisialisasi adalah kasus basis, pemeliharaan adalah langkah induktif, dan terminasi adalah kesimpulan yang Anda tuai di ujung.

3.1.1 Kebenaran parsial dan kebenaran total

Bab 2 membuktikan bahwa Insertion Sort *bekerja*. Bab itu belum memisahkan dua janji berbeda yang tersembunyi di dalam kata tersebut, dan perbedaan keduanya menjadi penting begitu kita mulai menulis algoritma yang memanggil dirinya sendiri. **Gambar 3.1** menggambar pembedaan itu dalam satu gambar.

> **DEFINISI.** Sebuah algoritma disebut **benar secara parsial** jika, *setiap kali ia berhenti*, ia menghasilkan keluaran yang benar. Kebenaran parsial sama sekali tidak mengatakan apa pun tentang apakah algoritma itu pernah berhenti.
>
> **DEFINISI.** Sebuah algoritma disebut **benar secara total** jika (1) ia selalu berhenti pada setiap masukan yang sah, *dan* (2) ketika ia berhenti, ia menghasilkan keluaran yang benar. Singkatnya: **kebenaran total = kebenaran parsial + terminasi.**

Kebenaran parsial adalah janji "*jika* juru masak selesai memasak, hidangannya akan lezat." Itu janji yang sungguhan dan berguna, tetapi janji itu tidak mencegah si juru masak berdiri di dapur selamanya. Kebenaran total menambahkan janji kedua: juru masaknya selesai.



Gambar 3.1 Kebenaran parsial dibandingkan kebenaran total. Pada diagram atas, jaminannya bersyarat — **JIKA** kendali sampai ke kotak keluaran, **MAKA** keluarannya adalah yang dikehendaki. Pada diagram bawah, jaminannya tanpa syarat — kendali **SUNGGUH** sampai ke kotak keluaran, **DAN** keluarannya adalah yang dikehendaki.

Mengapa kita tidak bisa begitu saja menulis program yang membaca algoritma apa pun lalu memutuskan apakah algoritma itu benar? Karena program semacam itu tidak mungkin ada. Pertanyaan "apakah algoritma ini berhenti pada masukan ini?" adalah **Halting Problem** (persoalan penghentian), dan Alan Turing membuktikan pada 1936 bahwa tidak ada algoritma yang memutuskannya secara umum. Karena itu kebenaran adalah sesuatu yang kita buktikan dengan tangan, satu algoritma pada satu waktu, dengan perkakas seperti loop invariant dan induksi. Ia bukan sesuatu yang bisa kita serahkan pada mesin.

Untuk sebagian besar algoritma dalam buku ini, terminasi sudah jelas dengan sendirinya — sebuah pencacah gelung merangkak menuju batas yang tetap, atau sebuah panggilan rekursif dilakukan pada array yang benar-benar lebih kecil — dan pekerjaan yang sulit sekaligus menarik justru terletak pada pembuktian kebenaran parsial.

3.1.2 Apa yang masih kurang

Bab 2 meninggalkan tiga lubang, dan bab ini menambal ketiganya.

Lubang pertama adalah **bahasa**. Bab 2 memperkenalkan pertumbuhan fungsi secara tidak resmi dan mengatakan bahwa Insertion Sort memakan "kira-kira n^2 langkah pada kasus terburuk" dan "kira-kira n langkah pada kasus terbaik". Apa sebenarnya arti "kira-kira" itu? Jika satu implementasi memakan $3n^2 + 5n + 7$ langkah dan implementasi lain memakan $100n^2$ langkah, apakah keduanya sama atau berbeda? Kita membutuhkan definisi yang cukup tepat untuk diperdebatkan, bukan kata sifat — definisi yang di dalamnya ada konstanta yang dapat Anda tuliskan dan Anda periksa.

Lubang kedua adalah **hitungan**. Menghitung kerja gelung bersarang menghasilkan jumlahan — $1 + 2 + \cdots + (n - 1)$, atau $1 + \frac{3}{4} + \frac{9}{16} + \cdots$ — dan kita membutuhkan bentuk tertutup bagi jumlahan itu, lengkap dengan penurunannya, bukan sekadar hasilnya.

Lubang ketiga adalah **rekursi**. Insertion Sort adalah sebuah gelung, jadi sebuah jumlahan menggambarkannya. Tetapi Merge Sort memanggil dirinya sendiri, dua kali, pada dua paruh. Tidak ada jumlahan yang menggambarkan hal itu secara langsung. Yang dibutuhkan adalah sebuah *recurrence* (rekurensi): persamaan yang mendefinisikan sebuah fungsi lewat dirinya sendiri pada masukan yang lebih kecil. Sebagian besar isi bab ini adalah tentang cara menyelesaikannya.

3.2 Mengapa Kita Memerlukan Bahasa untuk Pertumbuhan

Misalkan Anda mengukur waktu dua program pengurutan di mesin Anda sendiri dan memperoleh ini:

- Program A mengurutkan 1,000 rekaman dalam 0.8 detik.
- Program B mengurutkan 1,000 rekaman dalam 0.1 detik.

Program B delapan kali lebih cepat, jadi B menang. Sekarang jalankan keduanya pada 1,000,000 rekaman. A memakan sekitar 27 menit. B memakan sekitar 28 jam. B kalah, dan kalahnya telak, karena waktu jalan A tumbuh seperti $n\lg n$ sedangkan waktu jalan B tumbuh seperti n^2, dan pada $n = 1{,}000{,}000$ selisih itu menenggelamkan faktor delapan tadi. Inilah hitungannya. Mengalikan n dengan 1000 berarti mengalikan $n\lg n$ dengan $1000 \times \frac{\lg 10^6}{\lg 10^3} = 1000 \times 2 = 2000$, sehingga A memerlukan $0.8 \times 2000 = 1600$ detik, sekitar 27 menit. Langkah yang sama mengalikan n^2 dengan $1000^2 = 10^6$, sehingga B memerlukan $0.1 \times 10^6 = 100{,}000$ detik, sekitar 28 jam.

Inilah seluruh alasan keberadaan analisis asimtotik, dalam satu contoh. **Faktor konstanta** — angka 8 itu — adalah sifat mesin, kompilator, ketelitian pemrogram dalam menata memori, dan fase bulan. **Laju pertumbuhan** — $n\lg n$ lawan n^2 — adalah sifat algoritmanya, dan itulah yang menentukan siapa yang menang pada data sungguhan.

> **WAWASAN KUNCI:** Analisis asimtotik membuang persis informasi yang bergantung pada mesin, dan menyimpan persis informasi yang bergantung pada algoritma. Dua aturan melakukan seluruh pembuangan itu: **buang faktor konstanta** ($3n^2 \to n^2$) dan **buang suku**

> **berorde lebih rendah** ($5n^2 + 3n + 7 \to n^2$). Yang tersisa adalah bagaimana waktu jalan tumbuh ketika masukan tumbuh tanpa batas.

3.2.1 Angkanya, supaya Anda percaya

Slogan itu murah. **Kode 3.1** mencetak nilai sesungguhnya dari fungsi-fungsi pertumbuhan yang akan kita temui, agar pembaca dapat melihat sendiri betapa berbedanya perilaku mereka.

Kode 3.1 Sebuah program C++ yang menabelkan fungsi-fungsi pertumbuhan baku. Simpan sebagai `growth.cpp`, kompilasi dengan `g++ -std=c++17 -O2 -o growth growth.cpp`, lalu jalankan `./growth`.

```
// growth.cpp - Seberapa cepat sesungguhnya fungsi-fungsi pertumbuhan baku tumbuh?
// Kompilasi: g++ -std=c++17 -O2 -o growth growth.cpp
#include <iostream>
#include <iomanip>
#include <string>
#include <cmath>
#include <sstream>
using namespace std;

// Kembalikan bentuk yang terbaca manusia: eksak bila kecil, ilmiah bila raksasa.
static string pretty(double v) {
    if (v < 0) return "overflow";
    ostringstream os;
    if (v < 1e15) { os << fixed << setprecision(0) << v; }
    else          { os << scientific << setprecision(2) << v; }
    return os.str();
}

int main() {
    const long long sizes[] = {8, 16, 32, 64, 1024, 1048576};
    cout << left << setw(10) << "n"
         << right << setw(10) << "lg n"
         << setw(14) << "n"
         << setw(16) << "n lg n"
         << setw(18) << "n^2"
         << setw(14) << "2^n" << "\n";
    cout << string(82, '-') << "\n";
    for (long long n : sizes) {
        double lg = log2((double)n);
        // 2^n besar secara astronomis melewati n = 64, jadi cetak dalam bentuk ilmiah.
        double pow2 = (n <= 1024) ? pow(2.0, (double)n) : INFINITY;
        cout << left << setw(10) << n
             << right << setw(10) << pretty(lg)
             << setw(14) << pretty((double)n)
             << setw(16) << pretty(n * lg)
             << setw(18) << pretty((double)n * (double)n)
             << setw(14) << (isinf(pow2) ? string("astronomical") : pretty(pow2)) << "\n";
    }
    return 0;
}
```

Menjalankannya menghasilkan persis ini:

```
n               lg n             n          n lg n               n^2           2^n
--------------------------------------------------------------------------------
8                   3             8              24                64           256
16                  4            16              64               256         65536
32                  5            32             160              1024    4294967296
64                  6            64             384              4096      1.84e+19
1024               10          1024           10240           1048576  astronomical
1048576            20       1048576        20971520     1099511627776  astronomical
```

Bacalah baris terakhir. Pada satu juta rekaman, algoritma $\lg n$ melakukan sekitar 20 satuan kerja; algoritma n melakukan satu juta; algoritma $n\lg n$ melakukan dua puluh satu juta; algoritma n^2 melakukan *sejuta juta*. Mesin yang melakukan satu miliar satuan kerja per detik menyelesaikan pekerjaan $n\lg n$ dalam kira-kira seperlima puluh detik dan pekerjaan n^2 dalam kira-kira delapan belas menit. Dan 2^n pada $n = 64$ sudah bernilai 1.84×10^{19}, yang pada satu miliar per detik setara dengan sekitar 1.84×10^{10} detik — hampir enam ratus tahun.

Tabel 3.1 merangkum cerita yang sama dalam kata-kata, lengkap dengan nama yang disandang tiap kelas pertumbuhan dan sebuah algoritma dari buku ini yang termasuk ke dalamnya.

Tabel 3.1 Kelas-kelas pertumbuhan, apa yang terjadi pada masing-masing ketika ukuran masukan digandakan, dan di mana tiap kelas muncul dalam buku ini.

Pertumbuhan	Nama	Menggandakan n ...	Contoh dari buku ini
$\Theta(1)$	konstan	tidak mengubah apa pun	membaca $A[i]$
$\Theta(\lg n)$	logaritmik	menambah satu langkah	pencarian biner
$\Theta(n)$	linear	menggandakan kerjanya	memindai array satu kali
$\Theta(n\lg n)$	linearitmik	sedikit lebih dari menggandakannya	Merge Sort, Heapsort
$\Theta(n^2)$	kuadratik	mengalikan kerja dengan 4	Insertion Sort, kasus terburuk
$\Theta(n^3)$	kubik	mengalikan kerja dengan 8	perkalian matriks naif
$\Theta(2^n)$	eksponensial	*mengkuadratkan* kerjanya	pencarian himpunan bagian secara brutal
$\Theta(n!)$	faktorial	lebih buruk lagi	permutasi secara brutal

Kolom "menggandakan" itulah yang perlu dihafal, karena kolom itulah yang bisa Anda rasakan. Jika sebuah algoritma kuadratik, maka beranjak dari 10,000 pelanggan ke 20,000 pelanggan tidak membuat pekerjaan malam hari dua kali lebih lambat — ia membuatnya empat kali lebih lambat. Itulah bedanya antara pekerjaan yang selesai sebelum toko buka dan pekerjaan yang tidak.

> **Peringatan tentang masukan kecil.** Analisis asimtotik menggambarkan perilaku ketika n tumbuh *tanpa batas*. Untuk n yang kecil, konstanta bisa mendominasi sepenuhnya: Insertion Sort, yang $\Theta(n^2)$, sungguh-sungguh mengalahkan Merge Sort, yang $\Theta(n\lg n)$, pada array berisi sepuluh atau dua puluh elemen, karena faktor konstantanya mungil dan ia menyentuh memori dengan pola yang bersahabat. Rutin pengurutan pustaka sungguhan memanfaatkan hal ini — mereka menjalankan Merge Sort atau Quicksort sampai blok berukuran sekitar 16 elemen, lalu menyelesaikannya dengan Insertion Sort. Asimtotik memberi tahu Anda algoritma mana yang menang *pada akhirnya*; pengukuran memberi tahu Anda di mana "pada akhirnya" itu dimulai.

3.3 Big-O: Batas Atas Asimtotik

Sekarang kita membuat "tumbuh tidak lebih cepat daripada" menjadi tepat. Sebelum definisinya, inilah setiap lambang yang dipakainya, dalam kata-kata.

- n adalah **ukuran masukan** — banyaknya rekaman yang harus diurutkan, banyaknya karakter dalam sebuah string, banyaknya halaman web yang harus dinilai. Ia bilangan bulat tak negatif.
- $f(n)$ adalah **fungsi yang sedang kita pelajari**, biasanya waktu jalan sebuah algoritma yang diukur dalam langkah model RAM.
- $g(n)$ adalah **fungsi sederhana yang kita jadikan pembanding** — sesuatu seperti n, n^2, atau $n\lg n$.

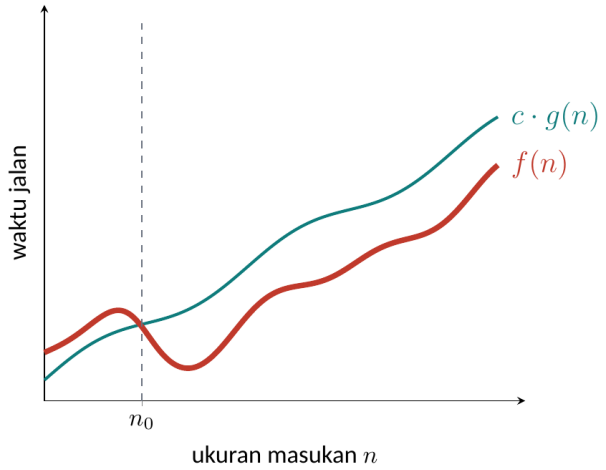
- c adalah **pengali konstanta positif**. Ia satu bilangan tetap, dipilih sekali, yang tidak berubah ketika n berubah. Ia menyerap setiap faktor yang bergantung pada mesin.
- n_0 adalah **ambang**, sebuah bilangan bulat tak negatif yang tetap. Kita tidak menjanjikan apa pun tentang apa yang terjadi di bawahnya; kita menjanjikan segalanya tentang apa yang terjadi pada dan di atasnya.

DEFINISI (Big-O). $f(n) = O(g(n))$ jika terdapat konstanta positif c dan n_0 sedemikian sehingga

$$0 \leq f(n) \leq c \cdot g(n) \quad \text{untuk semua } n \geq n_0. \tag{3.1}$$

Dalam kata-kata sederhana, Persamaan (3.1) berkata: **pada akhirnya f bertahan di bawah suatu kelipatan tetap dari g, dan bertahan di sana selamanya.** Anda boleh memilih kelipatannya. Anda boleh mengabaikan masukan yang kecil. Anda tidak boleh mengubah pilihan Anda seiring n membesar.

Gambar 3.2 adalah gambarnya. Kurva tebal adalah $f(n)$. Kurva tipis adalah $c \cdot g(n)$ — yaitu g yang diregangkan secara tegak sebesar konstanta c yang Anda pilih. Di sebelah kiri n_0, kurva tebal boleh berkeliaran sesukanya dan boleh saja duduk di atas kurva tipis; tak seorang pun peduli. Dari n_0 ke kanan, kurva tebal terkurung di bawah.



Gambar 3.2 Big-O sebagai batas atas. Di luar ambang n_0, waktu jalan $f(n)$ tidak pernah naik melampaui $c \cdot g(n)$. Apa yang terjadi di sebelah kiri n_0 tidak relevan bagi klaim ini.

Big-O adalah yang kita pakai untuk **batas atas kasus terburuk**: "algoritma ini tidak akan pernah memakan waktu lebih dari sekitar sekian."

3.3.1 Sebuah pembuktian, dikerjakan tuntas: $7n - 3 = O(n)$

Inilah contoh pertama yang klasik, dan ia contoh pertama yang tepat karena seluruh pembuktiannya muat dalam dua baris — tetapi kita akan menghabiskan satu halaman untuk dua baris itu, sebab *bentuk* argumennya itulah yang penting.

Klaim. $7n - 3 = O(n)$.

Apa yang harus kita hasilkan. Dua bilangan: sebuah konstanta $c > 0$ dan sebuah ambang n_0. Lalu kita harus menunjukkan bahwa pertidaksamaan $7n - 3 \leq c \cdot n$ berlaku untuk setiap $n \geq n_0$. Hanya itulah isi sebuah pembuktian klaim O, selalu: tunjukkan saksinya, periksa pertidaksamaannya.

Memilih saksinya. Perhatikan $7n - 3$. Ia adalah $7n$ dikurangi sesuatu. Mengurangi membuat sesuatu menjadi lebih kecil. Jadi $7n - 3 \leq 7n$ sudah berlaku, untuk setiap n. Itu menyarankan $c = 7$.

Pemeriksaannya, satu langkah satu baris.

$$7n - 3 \leq 7n \qquad \text{karena } -3 \leq 0$$

$$7n = 7 \cdot n \qquad \text{jadi kita boleh mengambil } c = 7$$

$$\text{dan pertidaksamaannya berlaku untuk setiap } n \geq 1, \text{ jadi } n_0 = 1.$$

Karena itu $7n - 3 \leq 7n$ untuk semua $n \geq 1$, dengan $c = 7$ dan $n_0 = 1$, dan menurut definisi $7n - 3 = O(n)$. ■

Perhatikan apa yang *tidak* perlu kita lakukan. Kita tidak perlu mencari c yang terkecil. Kita tidak perlu mencari n_0 yang terkecil. Pasangan mana pun yang berhasil sudah merupakan pembuktian yang lengkap. $c = 100$ dan $n_0 = 50$ pun akan berhasil; klaimnya tidak menjadi lebih lemah.

3.3.2 Sebuah pembuktian yang ambangnya bekerja sungguhan: $3n^2 + 5n + 7 = O(n^2)$

Di sini pilihan $c = 3$ tidak akan cukup, karena $3n^2 + 5n + 7$ benar-benar lebih besar daripada $3n^2$ untuk setiap n. Kita harus membayar suku berorde lebih rendah itu dari kantong konstanta.

Langkah 1 — batasi tiap suku dengan kelipatan n^2. Untuk $n \geq 1$ kita punya $n \leq n^2$ dan $1 \leq n^2$. Karena itu

$$3n^2 + 5n + 7 \leq 3n^2 + 5n^2 + 7n^2 = 15n^2 \qquad \text{untuk semua } n \geq 1.$$

Jadi $c = 15$, $n_0 = 1$ sudah merupakan pasangan saksi yang sah, dan kita selesai. Tetapi ada gunanya bertanya seberapa kecil c bisa dibuat.

Langkah 2 — menekan c turun. Coba $c = 4$. Kita perlu $3n^2 + 5n + 7 \leq 4n^2$, yaitu $5n + 7 \leq n^2$. Pada $n = 6$: $37 \leq 36$ salah. Pada $n = 7$: $42 \leq 49$ benar, dan karena n^2 menyalip $5n + 7$ sejak saat itu, pertidaksamaannya berlaku untuk semua $n \geq 7$. Jadi $c = 4$, $n_0 = 7$ juga berhasil.

Inilah pertukaran yang memang disediakan oleh ambang: konstanta yang lebih kecil, dibeli dengan ambang yang lebih besar. **Kode 3.2** melakukan persis pencarian ini secara mekanis.

3.3.3 Sebuah penyangkalan: $n^2 \neq O(n)$

Membuktikan bahwa sesuatu *bukan* O dari sesuatu yang lain adalah jenis argumen yang berbeda, dan mahasiswa yang belum pernah melihatnya ditulis lengkap cenderung kebingungan. Bentuknya adalah **pembuktian dengan kontradiksi**: andaikan saksinya ada, lalu turunkan sesuatu yang mustahil.

Klaim. $n^2 \neq O(n)$.

Pembuktian. Andaikan, demi kontradiksi, bahwa $n^2 = O(n)$. Maka menurut definisi terdapat konstanta positif c dan n_0 dengan

$$n^2 \leq c \cdot n \qquad \text{untuk semua } n \geq n_0.$$

Karena $n \geq n_0 \geq 0$ dan kita boleh mengandaikan $n \geq 1$, bagilah kedua ruas dengan n (sebuah besaran positif, sehingga arah pertidaksamaan terjaga):

$$n \leq c \qquad \text{untuk semua } n \geq n_0.$$

Tetapi ini berarti bilangan bulat n tidak pernah melampaui bilangan tetap c, sebesar apa pun n menjadi. Ambil $n = \lfloor c \rfloor + 1$, yang pasti paling sedikit n_0 bila kita juga mengambil n paling sedikit n_0 — konkretnya, ambil $n = \max(n_0, \lfloor c \rfloor + 1)$. Untuk n itu kita punya $n > c$, bertentangan dengan $n \leq c$.

Andaian tadi mestilah salah, jadi $n^2 \neq O(n)$. ■

> **WAWASAN KUNCI:** Untuk *membuktikan* $f = O(g)$, tunjukkan satu pasangan (c, n_0) yang berhasil. Untuk *menyangkalnya*, tunjukkan bahwa *setiap* pasangan gagal — biasanya dengan mengandaikan sebuah pasangan ada lalu memproduksi sebuah n di luar n_0 yang mematahkan pertidaksamaannya. Membuktikan bersifat eksistensial; menyangkal bersifat universal. Ketaksetangkupan itulah yang membuat penyangkalan terasa lebih sulit.

3.3.4 Membiarkan mesin yang memburu saksinya

Pembuktian tetaplah pembuktian, tetapi sangat membantu bila kita bisa *melihat* konstantanya. **Kode 3.2** mencari bilangan bulat c terkecil, lalu n_0 terkecil yang cocok, yang menjadi saksi sebuah klaim pada suatu jangkauan berhingga — dan melapor dengan jujur ketika tidak ada saksi yang muncul.

Kode 3.2 Memburu secara numerik konstanta c dan n_0 dalam definisi Big-O.

```
// witness.cpp - Mencari konstanta c dan n0 yang menjadi saksi sebuah klaim O,
// dan menyaksikan pencarian itu gagal ketika klaimnya salah.
// Kompilasi: g++ -std=c++17 -O2 -o witness witness.cpp
#include <iostream>
#include <functional>
#include <string>
#include <cmath>
using namespace std;

// Coba tiap c di 1..cMax; untuk masing-masingnya, cari n0 terkecil sedemikian
// sehingga f(n) <= c*g(n) berlaku untuk tiap n di [n0, nMax]. Laporkan keberhasilan
pertama.
bool findWitness(const string& claim,
                 function<double(double)> f,
                 function<double(double)> g,
                 int cMax, int nMax) {
    for (int c = 1; c <= cMax; ++c) {
        int n0 = -1;
        // Pindai turun dari nMax: n terakhir yang melanggar batasnya menetapkan n0.
        for (int n = nMax; n >= 1; --n) {
            if (f(n) > c * g(n)) { n0 = n + 1; break; }
            if (n == 1) n0 = 1;
        }
        if (n0 != -1 && n0 <= nMax) {
            cout << claim << " : witnessed by c = " << c
                 << ", n0 = " << n0 << "\n";
            return true;
        }
    }
    cout << claim << " : NO witness with c <= " << cMax
         << " on n <= " << nMax << " -- the claim looks false\n";
    return false;
}

int main() {
    findWitness("7n - 3 = O(n)      ", [](double n){ return 7*n - 3; },
                                       [](double n){ return n; }, 20, 100000);
    findWitness("3n^2+5n+7 = O(n^2) ", [](double n){ return 3*n*n + 5*n + 7; },
                                       [](double n){ return n*n; }, 20, 100000);
    findWitness("2n+10 = O(n)       ", [](double n){ return 2*n + 10; },
                                       [](double n){ return n; }, 20, 100000);
    findWitness("n^2 = O(n)         ", [](double n){ return n*n; },
                                       [](double n){ return n; }, 20, 100000);
    findWitness("2^n = O(n^10)      ", [](double n){ return (n<1000)?pow(2.0,n):1e300; },
                                       [](double n){ return pow(n,10); }, 20, 200);
    return 0;
}
```

Program itu mencetak:

```
7n - 3 = O(n)       : witnessed by c = 7, n0 = 1
3n^2+5n+7 = O(n^2)  : witnessed by c = 4, n0 = 7
2n+10 = O(n)        : witnessed by c = 3, n0 = 10
n^2 = O(n)          : NO witness with c <= 20 on n <= 100000 -- the claim looks false
2^n = O(n^10)       : NO witness with c <= 20 on n <= 200 -- the claim looks false
```

Baris pertama adalah pembuktian tangan kita, terkonfirmasi. Baris kedua menemukan $c = 4, n_0 = 7$ — persis pasangan yang kita hitung di §3.3.2. Baris ketiga memperlihatkan pertukaran itu lagi: $2n + 10 \leq 3n$ menuntut $10 \leq n$, jadi $n_0 = 10$. Dua baris terakhir adalah peringatan sekaligus hasil: **sebuah program hanya bisa menyarankan bahwa sebuah klaim salah, tidak pernah membuktikannya**, karena ia memeriksa n yang berhingga banyaknya. Pembuktian di §3.3.3-lah yang menuntaskan perkara itu.

3.4 Big-Ω: Batas Bawah Asimtotik

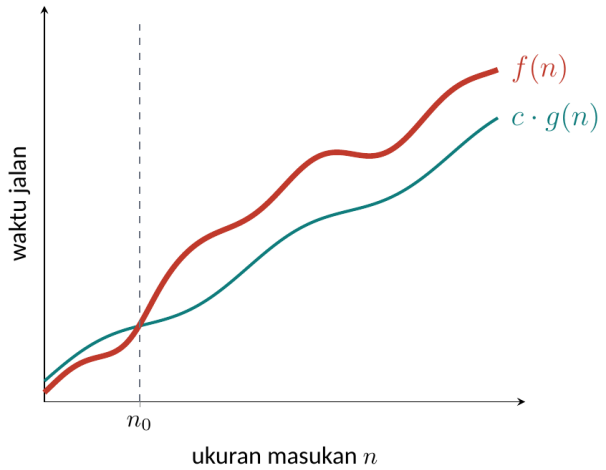
Semuanya berbalik.

DEFINISI (Big-Ω). $f(n) = \Omega(g(n))$ jika terdapat konstanta positif c dan n_0 sedemikian sehingga

$$0 \leq c \cdot g(n) \leq f(n) \quad \text{untuk semua } n \geq n_0. \tag{3.2}$$

Dalam kata-kata sederhana, Persamaan (3.2) berkata: **pada akhirnya f bertahan di atas suatu kelipatan tetap dari g, dan bertahan di sana selamanya.** Konstanta c dan ambang n_0 berarti persis seperti sebelumnya; hanya arah pertidaksamaannya yang berbalik.

Gambar 3.3 menggambarkannya. Kurva tebal $f(n)$ sempat menukik di bawah kurva tipis $c \cdot g(n)$ untuk n yang kecil — dan itu diizinkan — lalu memanjat ke atasnya pada n_0 dan tidak pernah turun lagi.



Gambar 3.3 Big-Ω sebagai batas bawah. Di luar n_0, waktu jalan $f(n)$ tidak pernah jatuh di bawah $c \cdot g(n)$.

Contoh terkerjakan. Tunjukkan $7n - 3 = \Omega(n)$.

Kita memerlukan $c \cdot n \leq 7n - 3$ untuk n yang besar. Coba $c = 1$:

$$n \leq 7n - 3 \Leftrightarrow 3 \leq 6n \Leftrightarrow n \geq \frac{1}{2}.$$

Jadi $c = 1$ dan $n_0 = 1$ berhasil, dan $7n - 3 = \Omega(n)$. ■

Sebuah c yang lebih ambisius: coba $c = 6$. Kita memerlukan $6n \leq 7n - 3$, yaitu $3 \leq n$, jadi $c = 6, n_0 = 3$ juga berhasil. Kita tidak boleh mengambil $c = 7$, karena $7n \leq 7n - 3$ salah untuk setiap n.

Contoh terkerjakan. Tunjukkan $\frac{1}{2}n^2 - 3n = \Omega(n^2)$.

Coba $c = \frac{1}{4}$. Kita memerlukan $\frac{1}{4}n^2 \leq \frac{1}{2}n^2 - 3n$, yaitu $3n \leq \frac{1}{4}n^2$, yaitu $12 \leq n$. Jadi $c = \frac{1}{4}, n_0 = 12$ berhasil. ■

Big-Ω adalah yang kita pakai untuk **pernyataan kasus terbaik** ("algoritma ini selalu memakan waktu paling sedikit sekian") dan, yang jauh lebih penting, untuk **batas bawah sebuah persoalan** ("*tidak ada* algoritma yang dapat menyelesaikan persoalan ini dalam waktu kurang dari sekian"). Mencari di dalam array tak terurut berisi n elemen adalah $\Omega(n)$ dalam pengertian kedua yang lebih kuat ini: setiap algoritma yang benar, pada kasus terburuk, harus melihat setiap elemen, sebab elemen yang tidak pernah ia lihat bisa jadi justru elemen yang sedang dicarinya.

3.5 Big-Θ: Batas Ketat Asimtotik

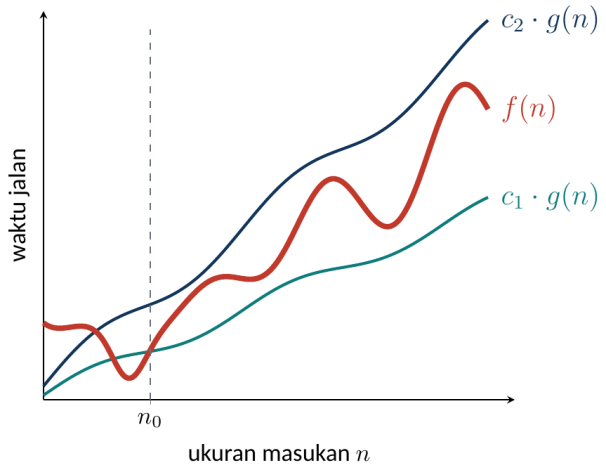
Sekarang kita menjepit dari kedua sisi sekaligus. Ini memerlukan **dua** konstanta.

DEFINISI (Big-Θ). $f(n) = \Theta(g(n))$ jika terdapat konstanta positif c_1, c_2, dan n_0 sedemikian sehingga

$$0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n) \quad \text{untuk semua } n \ge n_0. \tag{3.3}$$

Dalam Persamaan (3.3), c_1 adalah **pengali bawah** dan c_2 adalah **pengali atas**; n_0 adalah ambang tunggal yang di luarnya kedua pertidaksamaan berlaku. Dalam kata-kata sederhana: **pada akhirnya f terjepit di antara dua kelipatan tetap dari g** — ia tumbuh dengan laju yang persis sama dengan g, sampai pada faktor konstanta.

Gambar 3.4 memperlihatkan jepitan itu: $f(n)$ berjalan di lorong antara $c_1 \cdot g(n)$ di bawah dan $c_2 \cdot g(n)$ di atas.



Gambar 3.4 Big-Θ sebagai batas ketat. Setelah n_0, fungsi $f(n)$ terkurung di lorong antara $c_1 \cdot g(n)$ dan $c_2 \cdot g(n)$, sehingga f dan g tumbuh dengan laju yang sama.

TEOREMA. $f(n) = \Theta(g(n))$ **jika dan hanya jika** $f(n) = O(g(n))$ dan $f(n) = \Omega(g(n))$.

Pembuktian. ($\Rightarrow$) Jika Θ berlaku dengan konstanta c_1, c_2, n_0, maka $f(n) \le c_2 g(n)$ untuk $n \ge n_0$ memberikan O dengan $c = c_2$, dan $c_1 g(n) \le f(n)$ untuk $n \ge n_0$ memberikan Ω dengan $c = c_1$. ($\Leftarrow$) Jika $f = O(g)$ dengan konstanta c', n_1 dan $f = \Omega(g)$ dengan konstanta c'', n_2, ambil $c_2 = c'$, $c_1 = c''$, dan $n_0 = \max(n_1, n_2)$. Di luar ambang tunggal itu kedua pertidaksamaan berlaku sekaligus. ■

Teorema inilah cara pembuktian Θ hampir selalu dikerjakan dalam praktik: buktikan batas atasnya, buktikan batas bawahnya, gabungkan.

Contoh terkerjakan, selengkapnya. Tunjukkan $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Batas atas. Untuk $n \geq 1$, berlaku $-3n \leq 0$, karena itu

$$\frac{1}{2}n^2 - 3n \leq \frac{1}{2}n^2, \qquad \text{jadi } c_2 = \frac{1}{2} \text{ berlaku mulai } n \geq 1.$$

Batas bawah. Kita menghendaki $c_1 n^2 \leq \frac{1}{2}n^2 - 3n$. Setelah ditata ulang,

$$\left(\frac{1}{2} - c_1\right)n^2 \geq 3n \Leftrightarrow \left(\frac{1}{2} - c_1\right)n \geq 3 \Leftrightarrow n \geq \frac{3}{\frac{1}{2} - c_1}.$$

Pilih $c_1 = \frac{1}{4}$. Maka tuntutannya menjadi $n \geq 3/\frac{1}{4} = 12$.

Gabungkan. Dengan $c_1 = \frac{1}{4}$, $c_2 = \frac{1}{2}$, dan $n_0 = 12$, kedua pertidaksamaan berlaku untuk semua $n \geq 12$. Karena itu $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. ■

PIKIRKAN. O sangat sering dipakai padahal yang dimaksud adalah Θ. Pernyataan $7n = O(n^5)$ sepenuhnya **benar** — ambil $c = 1$, $n_0 = 1$ — dan sepenuhnya **tidak berguna**, karena ia membuang hampir semua yang sudah Anda ketahui. Selalu laporkan batas paling ketat yang dapat Anda buktikan. Mengatakan "Insertion Sort adalah $O(n^2)$ pada kasus terburuk" itu benar; mengatakan ia $\Theta(n^2)$ pada kasus terburuk itu benar *sekaligus* informatif, sebab pernyataan itu juga memberi tahu Anda bahwa batas tersebut tidak dapat diperbaiki lagi.

3.6 Little-o, Little-ω, dan Analogi Bilangan

Big-O mengizinkan f dan g tumbuh dengan laju yang sama: $2n^2 = O(n^2)$ itu benar. Kadang kita ingin mengatakan bahwa f tumbuh **benar-benar** lebih lambat. Itulah little-o.

DEFINISI (little-o). $f(n) = o(g(n))$ jika untuk **setiap** konstanta positif c terdapat sebuah ambang n_0 (yang boleh bergantung pada c) sedemikian sehingga

$$0 \leq f(n) < c \cdot g(n) \quad \text{untuk semua } n \geq n_0.$$

(3.4)

Secara setara, bila limitnya ada, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Bacalah pengkuantifikasi dalam Persamaan (3.4) baik-baik, sebab di situlah seluruh perbedaannya. Pada Big-O kita mengatakan "**terdapat** sebuah c". Pada little-o kita mengatakan "**untuk setiap** c". Sekecil apa pun Anda menyusutkan pengalinya — $c = 0.001$, $c = 10^{-9}$ — pada akhirnya f menyelinap ke bawahnya. Itu hanya mungkin terjadi jika f sungguh-sungguh dapat diabaikan secara asimtotik dibandingkan g.

DEFINISI (little-ω). $f(n) = \omega(g(n))$ jika untuk **setiap** konstanta positif c terdapat n_0 sedemikian sehingga

$$0 \leq c \cdot g(n) < f(n) \quad \text{untuk semua } n \geq n_0.$$

Secara setara, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$. Ia adalah versi ketat dari Ω, dan $f(n) = \omega(g(n))$ jika dan hanya jika $g(n) = o(f(n))$.

Contoh-contoh terkerjakan.

- $2n = o(n^2)$. Diberikan sembarang $c > 0$, kita memerlukan $2n < cn^2$, yaitu $n > 2/c$. Ambil $n_0 = \lceil 2/c \rceil + 1$. Selesai — dan perhatikan bagaimana n_0 bergantung pada c, dan itu memang diizinkan.
- $2n^2 \neq o(n^2)$. Pilih $c = 1$. Kita akan memerlukan $2n^2 < n^2$, yaitu $n^2 < 0$, yang mustahil. Satu c yang buruk sudah cukup untuk mematikan sebuah klaim little-o.
- $n^2 = \omega(n)$, karena $n^2/n = n \to \infty$.
- $\lg n = o(n)$, karena $\lg n/n \to 0$. Logaritma benar-benar lebih lambat daripada pangkat positif berapa pun dari n.
- $n^{100} = o(2^n)$. Setiap polinomial benar-benar lebih lambat daripada setiap eksponensial dengan basis di atas 1.

3.6.1 Analogi bilangan

Kelima notasi itu berpadanan dengan kelima perbandingan antara bilangan biasa, dan inilah gambaran mental yang paling berguna di seluruh bab ini. **Tabel 3.2** membuat padanan itu gamblang.

Tabel 3.2 Kelima notasi asimtotik dan perbandingan bilangan yang ditiru masing-masingnya.

Notasi	Arti bagi pertumbuhan	Analogi bilangan
$f(n) = O(g(n))$	f tumbuh tidak lebih cepat daripada g	$f \leq g$
$f(n) = \Omega(g(n))$	f tumbuh setidaknya secepat g	$f \geq g$
$f(n) = \Theta(g(n))$	f tumbuh dengan laju yang sama dengan g	$f = g$
$f(n) = o(g(n))$	f tumbuh benar-benar lebih lambat daripada g	$f < g$
$f(n) = \omega(g(n))$	f tumbuh benar-benar lebih cepat daripada g	$f > g$

Analogi itu pelayan yang baik dan majikan yang buruk. Ia dapat diandalkan dalam satu hal penting dan tidak dapat diandalkan dalam hal lain, dan §3.7.6 menjelaskan di mana ia patah.

3.6.2 Penyalahgunaan notasi yang kita semua lakukan

Secara ketat, $O(g(n))$ adalah sebuah **himpunan fungsi** — himpunan semua f yang untuknya terdapat c dan n_0 yang cocok. Cara jujur menuliskan relasinya adalah

$$f(n) \in O(g(n)),$$

dengan lambang keanggotaan $\in$. Semua orang, termasuk buku ini, menulis $f(n) = O(g(n))$ dengan tanda sama dengan. Itu dipahami semua orang dan ceroboh menurut semua orang. Dua akibat kecerobohan itu perlu ditandai sekarang:

1. **Tanda sama dengan itu searah.** $2n = O(n^2)$ benar; "$O(n^2) = 2n$" tidak bermakna. Jangan pernah membaca persamaan asimtotik dari kanan ke kiri.
2. **Asimtotik di dalam sebuah rumus berarti "suatu fungsi".** Ketika kita menulis $T(n) = 2T(n/2) + \Theta(n)$, lambang $\Theta(n)$ itu mewakili *suatu fungsi anonim* yang kebetulan $\Theta(n)$. Ketika kita menulis $n^2 + O(n)$, lambang $O(n)$ itu mewakili suatu fungsi anonim yang terbatas di atas oleh suatu kelipatan n.

3.7 Sifat-Sifat Notasi Tersebut

Kelima notasi itu menuruti sebuah aljabar kecil. Mengetahuinya menghemat kerja luar biasa banyak, sebab ia memungkinkan Anda memanipulasi batas tanpa kembali ke definisi setiap kali.

3.7.1 Ketransitifan

Kelimanya transitif.

$$\begin{aligned} f(n) = O(g(n)) \text{ dan } g(n) = O(h(n)) &\implies f(n) = O(h(n)), \\ f(n) = \Omega(g(n)) \text{ dan } g(n) = \Omega(h(n)) &\implies f(n) = \Omega(h(n)), \\ f(n) = \Theta(g(n)) \text{ dan } g(n) = \Theta(h(n)) &\implies f(n) = \Theta(h(n)), \end{aligned}$$

dan demikian pula untuk o dan ω.

Pembuktian untuk O. Andaikan $f(n) \le c_1 g(n)$ untuk $n \ge n_1$ dan $g(n) \le c_2 h(n)$ untuk $n \ge n_2$. Untuk setiap $n \ge \max(n_1, n_2)$, keduanya berlaku, sehingga

$$f(n) \le c_1 g(n) \le c_1(c_2 h(n)) = (c_1 c_2)\, h(n).$$

Ambil $c = c_1 c_2$ dan $n_0 = \max(n_1, n_2)$. ■

Ketransitifan inilah yang membuat rantai penalaran bekerja: jika langkah penggabungan adalah $O(n)$ dan n adalah $O(n\lg n)$, maka langkah penggabungan adalah $O(n\lg n)$.

3.7.2 Kerefleksifan

$f(n) = O(f(n))$, $f(n) = \Omega(f(n))$, dan $f(n) = \Theta(f(n))$, selalu. Ambil $c = 1$ (atau $c_1 = c_2 = 1$) dan $n_0 = 1$.

Little-o dan little-ω **tidak** refleksif: $f(n) \ne o(f(n))$, sebab $f/f = 1 \ne 0$. Ini cocok dengan analogi bilangan — $x \le x$ benar, $x < x$ salah.

3.7.3 Kesetangkupan dan kesetangkupan transpos

Kesetangkupan hanya berlaku bagi Θ:

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n)).$$

Pembuktian. Dari $c_1 g \le f \le c_2 g$ untuk $n \ge n_0$, bagilah seluruhnya. Pertidaksamaan kanan memberi $g \ge f/c_2$, dan yang kiri memberi $g \le f/c_1$. Jadi $\frac{1}{c_2} f \le g \le \frac{1}{c_1} f$, yang merupakan definisi Θ dengan peran tertukar dan konstanta $\frac{1}{c_2}, \frac{1}{c_1}$. ■

Kesetangkupan transpos menghubungkan notasi atas dengan notasi bawah:

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n)), \qquad f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n)).$$

Pembuktian yang pertama. $f(n) \le c\, g(n)$ untuk $n \ge n_0$ secara harfiah adalah pernyataan yang sama dengan $g(n) \ge \frac{1}{c} f(n)$ untuk $n \ge n_0$, yaitu Ω dengan konstanta $1/c$. ■

Dalam analogi bilangan: $f \le g$ mengatakan hal yang sama dengan $g \ge f$. Tepat sekali.

3.7.4 Jumlah, hasil kali, dan konstanta

Misalkan $f_1(n) = O(g_1(n))$ dan $f_2(n) = O(g_2(n))$. Maka:

- **Aturan jumlah.** $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$.
- **Aturan hasil kali.** $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.
- **Aturan konstanta.** $k \cdot f_1(n) = O(g_1(n))$ untuk sembarang konstanta $k > 0$.

Pembuktian aturan jumlah. Andaikan $f_1 \le c_1 g_1$ untuk $n \ge n_1$ dan $f_2 \le c_2 g_2$ untuk $n \ge n_2$. Tulis $g(n) = \max(g_1(n), g_2(n))$. Untuk $n \ge \max(n_1, n_2)$,

$$\begin{aligned} f_1(n) + f_2(n) \le c_1 g_1(n) + c_2 g_2(n) &\le c_1 g(n) + c_2 g(n) \\ &= (c_1 + c_2)\, g(n). \qquad \blacksquare \end{aligned}$$

Pembuktian aturan hasil kali. Untuk $n \ge \max(n_1, n_2)$, karena semua besaran tak negatif,

$$f_1(n) f_2(n) \le (c_1 g_1(n))(c_2 g_2(n)) = (c_1 c_2)\, g_1(n) g_2(n). \qquad \blacksquare$$

Aturan jumlah itulah asal-usul "buang suku berorde lebih rendah", dan aturan konstanta itulah asal-usul "buang faktor konstanta". Mari kita lihat hal itu diurai satu per satu.

3.7.5 Mengapa kita membuang konstanta dan suku berorde lebih rendah — dibuktikan, bukan diklaim

Konstanta. Andaikan $f(n) = 3n^2$. Apakah $f(n) = \Theta(n^2)$? Ambil $c_1 = c_2 = 3$ dan $n_0 = 1$; maka $3n^2 \le 3n^2 \le 3n^2$ secara sepele. Jadi ya. Konstantanya sekadar menjadi bagian dari saksinya. Lebih umum, untuk sembarang $k > 0$, $k \cdot g(n) = \Theta(g(n))$, sebab definisinya memang sudah memberi kita pengali sembarang.

Alasan yang lebih dalam adalah alasan dari §3.2: faktor konstanta adalah fakta tentang mesin, bukan tentang algoritma. Ganti kompilatornya, konstantanya berubah. Ganti algoritmanya, kelas pertumbuhannya berubah.

Suku berorde lebih rendah. Misalkan $p(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0$ dengan $a_d > 0$. Kita mengklaim $p(n) = \Theta(n^d)$.

Batas atas. Untuk $n \ge 1$, setiap pangkat n^i dengan $i \le d$ memenuhi $n^i \le n^d$. Karena itu

$$p(n) \le (|a_d| + |a_{d-1}| + \cdots + |a_0|) n^d = A n^d,$$

dengan A adalah jumlah nilai mutlak koefisien-koefisiennya — sebuah bilangan tetap. Jadi $c_2 = A$ berhasil mulai dari $n_0 = 1$.

Batas bawah. Tulis $p(n) = a_d n^d + R(n)$ dengan $R(n) = a_{d-1} n^{d-1} + \cdots + a_0$. Dengan batas yang sama, $|R(n)| \le B n^{d-1}$ untuk $n \ge 1$, dengan $B = |a_{d-1}| + \cdots + |a_0|$. Maka

$$p(n) \ge a_d n^d - B n^{d-1} = n^d \left(a_d - \frac{B}{n} \right).$$

Begitu $n \ge 2B/a_d$ kita punya $B/n \le a_d/2$, sehingga $p(n) \ge \frac{a_d}{2} n^d$. Ambil $c_1 = a_d/2$ dan $n_0 = \max(1, \lceil 2B/a_d \rceil)$.

Kesimpulan. $p(n) = \Theta(n^d)$. **Setiap polinomial adalah Θ dari pangkat tertingginya.** $\blacksquare$

Diterapkan pada tiga contoh yang lazim:

- $50n\log n \to \Theta(n\log n)$: angka 50 diserap oleh konstanta.
- $7n - 3 \to \Theta(n)$: derajat 1, koefisien tertinggi 7.

- $8n^2\log n + 5n^2 + n \to \Theta(n^2\log n)$: di sini $n^2\log n$ menyalip n^2, yang menyalip n, jadi suku pertamalah yang berkuasa. (Ini bukan polinomial, tetapi argumen aturan jumlah yang sama tetap berlaku: suku terbesar yang menang.)

3.7.6 Di mana analogi bilangan patah: ketrikotomian gagal

Untuk bilangan real, tepat satu di antara $a < b$, $a = b$, $a > b$ berlaku. Inilah **trikotomi**, dan inilah yang membuat bilangan dapat dibandingkan.

Pertumbuhan asimtotik **tidak** memiliki trikotomi. Ada pasangan fungsi yang untuknya tidak satu pun dari $f = O(g)$, $f = \Theta(g)$, $f = \Omega(g)$ berlaku. Contoh bakunya adalah

$$f(n) = n, \qquad g(n) = n^{1+\sin n}.$$

Pangkat $1 + \sin n$ berayun antara 0 dan 2 selamanya. Ketika $\sin n$ dekat 1, g dekat n^2 dan menjulang di atas f; ketika $\sin n$ dekat -1, g dekat $n^0 = 1$ dan f yang menjulang di atas g. Tidak ada fungsi yang pada akhirnya selalu di atas atau selalu di bawah yang lain, jadi baik O maupun Ω tidak berlaku ke arah mana pun.

MITOS YANG SERING DIPERCAYA. "Setiap dua fungsi dapat dibandingkan secara asimtotik." Tidak bisa. Fungsi yang berayun bisa tidak terbandingkan. Dalam praktik, waktu jalan algoritma sungguhan tersusun dari polinomial, logaritma, dan eksponensial, yang semuanya saling terbandingkan — jadi analogi itu bekerja dari hari ke hari. Tetapi ia bukan teorema, dan pengecualiannya justru jenis $f(n)$ yang membuat kasus ketiga Teorema Master gagal.

3.8 Notasi Baku dan Fungsi-Fungsi yang Sering Muncul

Setiap waktu jalan dalam buku ini dirakit dari sebuah daftar pendek berisi balok-balok penyusun. Bagian ini memperkenalkan masing-masingnya, lengkap dengan kesamaan-kesamaan yang benar-benar akan kita pakai.

3.8.1 Kemonotonan

Sebuah fungsi f disebut **naik monoton** jika $m \le n$ mengakibatkan $f(m) \le f(n)$; **turun monoton** jika $m \le n$ mengakibatkan $f(m) \ge f(n)$. Ia disebut naik atau turun **secara ketat** jika pertidaksamaannya ketat. Hampir setiap fungsi waktu jalan yang kita temui naik monoton — lebih banyak masukan, lebih banyak kerja — dan kita memakai kenyataan ini terus-menerus tanpa menyebutnya.

3.8.2 Lantai dan langit-langit

Untuk sebuah bilangan real x:

- $\lfloor x \rfloor$, **lantai** (*floor*), adalah bilangan bulat terbesar yang $\le x$. $\lfloor 3.7 \rfloor = 3$, $\lfloor 3 \rfloor = 3$, $\lfloor -3.2 \rfloor = -4$.
- $\lceil x \rceil$, **langit-langit** (*ceiling*), adalah bilangan bulat terkecil yang $\ge x$. $\lceil 3.2 \rceil = 4$, $\lceil 3 \rceil = 3$.

Kesamaan yang kita andalkan:

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1, \qquad \lfloor x \rfloor + \lceil x \rceil = 2x \text{ ketika } x \in \mathbb{Z},$$

$$\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n \quad \text{untuk setiap bilangan bulat } n.$$

Kesamaan yang terakhir itulah yang membuat pembelahan pada Merge Sort bekerja: berapa pun n, ganjil atau genap, kedua paruhnya dijumlahkan kembali menjadi tepat n. Lantai dan langit-langit keduanya naik monoton.

Rekurensi dari bagi dan kuasai sesungguhnya berbentuk $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$. Kita hampir selalu akan menulis $T(n) = 2T(n/2) + \Theta(n)$ sebagai gantinya, dan §3.10.3 menjelaskan mengapa hal itu tidak berbahaya.

3.8.3 Aritmetika modular

Untuk bilangan bulat a dan $n > 0$, $a \bmod n$ adalah sisanya: $a \bmod n = a - n\lfloor a/n \rfloor$, yang selalu berada dalam jangkauan $0 \le a \bmod n < n$. Tabel hash, penyangga siklik, dan "putaran balik" pada banyak gelung memakainya.

3.8.4 Polinomial

Polinomial dalam n berderajat d adalah $p(n) = \sum_{i=0}^{d} a_i n^i$ dengan $a_d \neq 0$. Kita sudah membuktikan di §3.7.5 bahwa jika $a_d > 0$ maka $p(n) = \Theta(n^d)$.

Sebuah fungsi disebut **terbatas secara polinomial** jika $f(n) = O(n^k)$ untuk suatu konstanta k. Pangkat-pangkat yang berguna:

$$n^0 = 1, \quad n^1 = n, \quad n^{-1} = \frac{1}{n}, \quad n^{1/2} = \sqrt{n}, \quad n^a n^b = n^{a+b}, \quad (n^a)^b = n^{ab}.$$

3.8.5 Eksponensial

Untuk $a > 0$ real dan bilangan bulat m, n:

$$a^0 = 1, \qquad a^{-1} = \frac{1}{a}, \qquad (a^m)^n = a^{mn}, \qquad a^m a^n = a^{m+n}.$$

Fakta tunggal yang paling penting tentang eksponensial dalam analisis algoritma:

TEOREMA. Untuk setiap konstanta $a > 1$ dan setiap konstanta $d > 0$,

$$\lim_{n\to\infty} \frac{n^d}{a^n} = 0, \qquad \text{yaitu,} \qquad n^d = o(a^n).$$

Setiap eksponensial dengan basis lebih besar dari 1 pada akhirnya mengalahkan setiap polinomial, seberapa pun besar derajatnya. Jadi $n^{1000} = o(1.001^n)$, walaupun polinomialnya unggul selama rentang waktu yang tak masuk akal lamanya.

Inilah sebabnya algoritma berwaktu eksponensial bersifat bencana dan algoritma berwaktu polinomial (pada asasnya) dapat diterima, dan inilah alasan seluruh bidang ini menarik garis utamanya di antara keduanya.

Berguna pula, dari definisi $e^x = \sum_{i=0}^{\infty} x^i / i!$:

$$e^x \ge 1 + x \quad \text{untuk semua bilangan real } x, \qquad \lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x.$$

3.8.6 Logaritma — dan notasi yang akan kita pakai di mana-mana

Logaritma menjawab pertanyaan "berapa pangkat yang harus saya kenakan pada basis untuk memperoleh bilangan ini?" Jika $b^y = x$ maka $\log_b x = y$.

Tiga singkatan sudah baku dan kita memakai ketiganya:

$$\lg n = \log_2 n \quad \text{(biner)}, \qquad \ln n = \log_e n \quad \text{(natural)}, \qquad \lg^k n = (\lg n)^k, \qquad \lg\lg n = \lg(\lg n).$$

Perhatikan baik-baik bahwa $\lg^2 n$ berarti $(\lg n)^2$, **bukan** $\lg(\lg n)$. Hal ini terus-menerus menjatuhkan orang; ia akan berarti penting di §3.13 ketika sebuah kasus celah menghasilkan $\Theta(n\lg^2 n)$.

Tabel 3.3 mendaftar setiap kesamaan yang akan kita pakai dalam bab ini, lengkap dengan satu baris alasan bagi masing-masingnya.

Tabel 3.3 Kesamaan logaritma yang dipakai dalam bab ini. Dua yang terakhir adalah yang membuat Teorema Master bekerja.

Kesamaan	Mengapa ia benar
$\log_b(xy) = \log_b x + \log_b y$	pangkat bertambah ketika perpangkatan dikalikan
$\log_b(x/y) = \log_b x - \log_b y$	pangkat berkurang ketika perpangkatan dibagi
$\log_b(x^a) = a\log_b x$	$(b^y)^a = b^{ya}$
$\log_b x = \dfrac{\log_c x}{\log_c b}$	penggantian basis; $\log_c b$ adalah konstanta
$b^{\log_b x} = x$	definisinya, dibaca mundur
$x^{\log_b y} = y^{\log_b x}$	ambil $\log_b$ kedua ruas: keduanya memberi $\log_b x \cdot \log_b y$
$a^{\log_b n} = n^{\log_b a}$	baris sebelumnya, dengan $x = a$, $y = n$
$\log_b a = \dfrac{1}{\log_a b}$	penggantian basis dengan $c = a$

Kesamaan **penggantian basis** punya akibat yang pantas mendapat barisnya sendiri: karena $\log_c b$ adalah konstanta, semua logaritma hanya berbeda oleh faktor konstanta, sehingga

$$\log_2 n = \Theta(\log_{10} n) = \Theta(\ln n) = \Theta(\log_b n) \text{ untuk sembarang nilai tetap } b > 1. \tag{3.5}$$

Menurut Persamaan (3.5), di dalam O, Ω, atau Θ, **basis sebuah logaritma tidak berarti apa-apa** dan biasanya tidak dituliskan. Di luar ketiganya — dalam cacah yang eksak, atau dalam sebuah pangkat seperti $n^{\log_b a}$ — basisnya berarti sangat banyak.

Kesamaan $a^{\log_b n} = n^{\log_b a}$ adalah jantung aljabar Teorema Master, jadi mari kita buktikan pelan-pelan. Ambil $\log_b$ dari ruas kiri:

$$\log_b(a^{\log_b n}) = (\log_b n)(\log_b a).$$

Sekarang ambil $\log_b$ dari ruas kanan:

$$\log_b(n^{\log_b a}) = (\log_b a)(\log_b n).$$

Perkalian bersifat komutatif, jadi kedua logaritma itu sama; dan karena $\log_b$ adalah fungsi yang naik secara ketat (sehingga satu-satu), kedua besaran asalnya sama. ■

Terakhir, dua fakta pertumbuhan:

$$\lg n = o(n^{\varepsilon}) \text{ untuk setiap konstanta } \varepsilon > 0, \qquad \lg(n!) = \Theta(n\lg n).$$

Yang pertama mengatakan bahwa logaritma lebih lambat daripada pangkat positif *mana pun* dari n, sekecil apa pun pangkat itu — dan justru itulah sebabnya $\lg n$ tidak pernah dapat menyediakan pemisahan polinomial yang dituntut oleh Kasus 1 dan Kasus 3 Teorema Master.

3.8.7 Faktorial dan hampiran Stirling

$n! = 1 \cdot 2 \cdot 3 \cdots n$, dengan $0! = 1$ menurut kesepakatan. Ia mencacah permutasi n objek. Batas kasarnya adalah $n! \le n^n$, dan batas bawah kasarnya adalah $n! \ge 2^{n-1}$. **Hampiran Stirling** jauh lebih tajam:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right). \tag{3.6}$$

Tiga akibat Persamaan (3.6) yang kita pakai: $n! = o(n^n)$, $n! = \omega(2^n)$, dan dengan mengambil logaritma, $\lg(n!) = \Theta(n\lg n)$. Persamaan terakhir itulah alasan mengapa tidak ada pengurutan berbasis perbandingan yang dapat mengalahkan $n\lg n$ — sebuah fakta yang dibuktikan di Bab 5.

3.8.8 Logaritma teriterasi

$$\lg^* n = \min\{\, i \ge 0 : \lg^{(i)} n \le 1 \,\},$$

dengan $\lg^{(i)}$ berarti "kenakan lg tepat i kali". Dalam kata-kata: **berapa kali Anda harus mengambil logaritma sebelum bilangannya turun ke** 1 **atau di bawahnya?** Ia tumbuh begitu lambat sehingga praktis konstan: $\lg^* 2 = 1$, $\lg^* 4 = 2$, $\lg^* 16 = 3$, $\lg^* 65536 = 4$, dan $\lg^* 2^{65536} = 5$ — padahal 2^{65536} punya sekitar 20,000 angka desimal, jauh lebih banyak daripada jumlah atom di alam semesta teramati.

3.8.9 Bilangan Fibonacci

$$F_0 = 0, \qquad F_1 = 1, \qquad F_i = F_{i-1} + F_{i-2} \text{ untuk } i \ge 2,$$

yang memberi 0,1,1,2,3,5,8,13,21,34, ... Dengan rasio emas $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$ dan sekawannya $\hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.61803$,

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}, \qquad \text{dan karena } |\hat{\phi}| < 1, \qquad F_i = \left\lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \right\rfloor.$$

Jadi bilangan Fibonacci tumbuh **secara eksponensial**, dengan laju ϕ^n. Inilah persisnya sebab mengapa program Fibonacci rekursif naif dari Bab 1 tidak dapat dipakai, dan mengapa Bab 7 menulisnya ulang dengan memoisasi.

Kode 3.3 membuat tiga di antara fakta-fakta ini terlihat.

Kode 3.3 Logaritma teriterasi, hampiran Stirling, dan bentuk tertutup Fibonacci, dihitung dan dibandingkan.

```
// functions.cpp - Fungsi-fungsi baku Bab 3 dilihat secara numerik:
// logaritma teriterasi, hampiran Stirling untuk n!, dan bentuk tertutup
// bagi bilangan Fibonacci.
// Kompilasi: g++ -std=c++17 -O2 -o functions functions.cpp
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

// lg*(n): berapa kali kita harus mengambil lg sebelum nilainya turun ke <= 1?
int lgStar(double n) {
    int count = 0;
    while (n > 1.0) { n = log2(n); ++count; }
    return count;
}

int main() {
    cout << "-- The iterated logarithm lg*(n) --\n";
    double vals[] = {1, 2, 4, 16, 65536, pow(2.0, 65536.0)};
    const char* names[] = {"1", "2", "4", "16", "65536", "2^65536"};
    for (int i = 0; i < 6; ++i) {
        if (isinf(vals[i])) { cout << "  n = " << setw(10) << names[i] << "  lg*(n) = 5 (by
definition, one more than lg*(65536))\n"; continue; }
        cout << "  n = " << setw(10) << names[i] << "  lg*(n) = " << lgStar(vals[i]) <<
"\n";
    }

    cout << "\n-- Stirling: n! vs sqrt(2 pi n) (n/e)^n --\n";
    cout << fixed;
    double fact = 1.0;
    for (int n = 1; n <= 20; ++n) {
        fact *= n;
        if (n == 1 || n == 5 || n == 10 || n == 15 || n == 20) {
            double st = sqrt(2 * M_PI * n) * pow(n / M_E, n);
            cout << "  n = " << setw(3) << n
                 << "  n! = " << scientific << setprecision(6) << setw(14) << fact
                 << "  Stirling = " << setw(14) << st
                 << "  ratio = " << fixed << setprecision(6) << fact / st << "\n";
        }
    }
```

Kode 3.3 (Lanjutan) *Logaritma teriterasi, hampiran Stirling, dan bentuk tertutup Fibonacci, dihitung dan dibandingkan.*

```
    cout << "\n-- Fibonacci F(n) vs phi^n / sqrt(5), phi = (1+sqrt 5)/2 --\n";
    double phi = (1.0 + sqrt(5.0)) / 2.0;
    long long a = 0, b = 1;
    for (int n = 1; n <= 40; ++n) {
        long long t = a + b; a = b; b = t;   // sesudah badan gelung, a = F(n)
        if (n == 1 || n == 5 || n == 10 || n == 20 || n == 40)
            cout << "  n = " << setw(3) << n << "  F(n) = " << setw(12) << a
                 << "  phi^n/sqrt5 = " << setprecision(3) << pow(phi, n) / sqrt(5.0) <<
"\n";
    }
    cout << "\n  phi = " << setprecision(9) << phi
         << "  so F(n) grows like " << setprecision(3) << phi << "^n\n";
    return 0;
}
```

Keluarannya:

```
-- The iterated logarithm lg*(n) --
  n =          1  lg*(n) = 0
  n =          2  lg*(n) = 1
  n =          4  lg*(n) = 2
  n =         16  lg*(n) = 3
  n =      65536  lg*(n) = 4
  n =    2^65536  lg*(n) = 5 (by definition, one more than lg*(65536))

-- Stirling: n! vs sqrt(2 pi n) (n/e)^n --
  n =   1  n! =   1.000000e+00  Stirling =   9.221370e-01  ratio = 1.084438
  n =   5  n! =   1.200000e+02  Stirling =   1.180192e+02  ratio = 1.016784
  n =  10  n! =   3.628800e+06  Stirling =   3.598696e+06  ratio = 1.008365
  n =  15  n! =   1.307674e+12  Stirling =   1.300431e+12  ratio = 1.005570
  n =  20  n! =   2.432902e+18  Stirling =   2.422787e+18  ratio = 1.004175
```

Dua hal yang perlu diperhatikan. Beranjak dari $n = 65536$ ke $n = 2^{65536}$ — sebuah lompatan yang tak terbayangkan — menaikkan $\lg^* n$ tepat sebesar satu. Dan nisbah Stirling melangkah mantap menuju 1: ia 1.0844 pada $n = 1$ dan 1.0042 pada $n = 20$, sejalan dengan suku koreksi $1 + \Theta(1/n)$.

Paruh Fibonacci dari jalannya program yang sama mencetak:

```
-- Fibonacci F(n) vs phi^n / sqrt(5), phi = (1+sqrt 5)/2 --
  n =   1  F(n) =            1  phi^n/sqrt5 = 0.724
  n =   5  F(n) =            5  phi^n/sqrt5 = 4.960
  n =  10  F(n) =           55  phi^n/sqrt5 = 55.004
  n =  20  F(n) =         6765  phi^n/sqrt5 = 6765.000
  n =  40  F(n) =    102334155  phi^n/sqrt5 = 102334155.000

  phi = 1.618033989  so F(n) grows like 1.618^n
```

Pada $n = 20$ bentuk tertutupnya sudah eksak sampai angka-angka yang tercetak — koreksi $\hat{\phi}^i$ telah menyusut di bawah 10^{-4}. **Tabel 3.4** mengumpulkan seluruh keluarga itu dalam urutan pertumbuhan.

Tabel 3.4 Fungsi-fungsi yang sering muncul, dalam urutan pertumbuhan menaik. Setiap baris adalah $o(\cdot)$ dari setiap baris di bawahnya.

Fungsi	Kelas pertumbuhan	Nilai pada n=1024
1	konstan	1
$\lg^* n$	log teriterasi	4
$\lg\lg n$	log log	≈ 3.3
$\lg n$	logaritmik	10
$\sqrt{n} = n^{1/2}$	akar	32
n	linear	1024
$n\lg n$	linearitmik	10240
n^2	kuadratik	1048576
n^3	kubik	$\approx 1.07 \times 10^9$
2^n	eksponensial	besar secara astronomis
$n!$	faktorial	lebih besar lagi
n^n	superfaktorial	lebih besar lagi daripada itu

3.9 Jumlahan: Aritmetika Gelung

Sebuah gelung yang berjalan n kali dan melakukan kerja t_i pada iterasi ke-i menelan biaya $\sum_{i=1}^{n} t_i$. Gelung bersarang memberi jumlahan bersarang. Pohon rekursi, nanti di bab ini, memberi jumlahan atas tingkat-tingkatnya. Karena itu jumlahan adalah aritmetika kerja sehari-hari dalam analisis algoritma, dan bagian ini menurunkan empat jumlahan yang kita butuhkan.

Pertama, notasinya sendiri. Lambang

$$\sum_{i=1}^{n} a_i$$

dibaca "jumlah dari a_i, ketika i berjalan dari 1 sampai n", dan ia sekadar berarti $a_1 + a_2 + \cdots + a_n$. Huruf i adalah **indeks**; 1 adalah **batas bawah**; n adalah **batas atas**; a_i adalah **suku yang dijumlahkan**. Jika batas atasnya berada di bawah batas bawahnya, jumlahannya **kosong** dan bernilai 0 menurut kesepakatan. Dua aturan berikut mengalir langsung dari aritmetika biasa dan terus-menerus dipakai:

$$\sum_{i=1}^{n} (c \cdot a_i) = c \sum_{i=1}^{n} a_i \qquad \text{(kelinearan: konstanta dapat dikeluarkan)},$$

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i \qquad \text{(kelinearan: jumlahan dapat dipecah)}.$$

3.9.1 Deret aritmetika

DEFINISI (deret aritmetika).

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = \Theta(n^2).$$

(3.7)

Gambar 3.5 menyatakan Persamaan (3.7) dalam bentuk yang sedikit berbeda; perhatikan bahwa di sana indeksnya dimulai dari $i = 0$, yang tidak mengubah apa pun karena suku $i = 0$ menyumbang nol, dan bentuk tertutupnya ditulis sebagai $(n^2 + n)/2$, yang sama saja dengan $n(n+1)/2$.

$$\sum_{i=0}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n^2 + n}{2}$$

Gambar 3.5 Deret aritmetika dan bentuk tertutupnya, $\sum_{i=0}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n^2+n}{2}$.

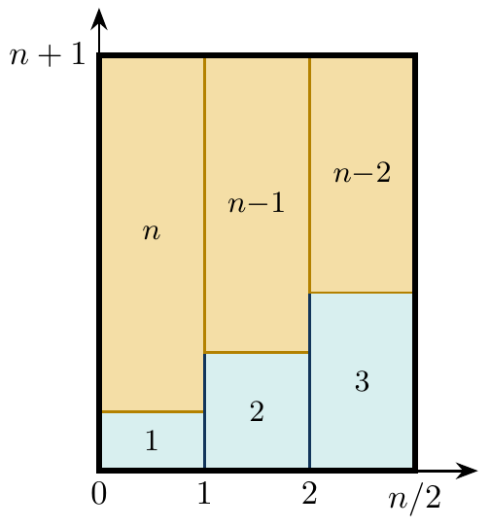
Trik Gauss, penurunannya. Ceritanya, Gauss menemukan ini pada usia kira-kira sembilan tahun ketika gurunya menyuruh kelas menjumlahkan 1 sampai 100. Tuliskan jumlahannya maju, lalu tuliskan lagi mundur di bawahnya:

$$\begin{array}{llllll} S = 1 & +\,2 & +\,3 & +\cdots & +\,(n-1) & +\,n \\ S = n & +(n-1) & +(n-2) & +\cdots & +\,2 & +\,1 \end{array}$$

Sekarang jumlahkan kedua baris itu **kolom demi kolom**. Kolom pertama memberi $1 + n$. Kolom kedua memberi $2 + (n - 1) = n + 1$. Kolom ketiga memberi $3 + (n - 2) = n + 1$. Setiap kolom memberi total yang sama, yaitu $n + 1$, karena ketika satu suku naik satu, suku lainnya turun satu. Ada n kolom. Jadi

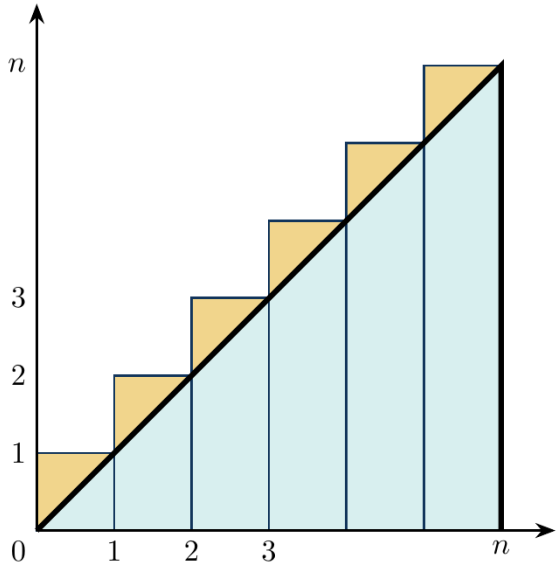
$$2S = n(n + 1), \qquad \text{maka} \qquad S = \frac{n(n+1)}{2}. \qquad \blacksquare$$

Gambar 3.6 adalah gambar dari argumen yang sama: tangga batang setinggi $1,2,\ldots,n$ dilipat menjadi sebuah persegi panjang padat selebar $n/2$ dan setinggi $n + 1$, sehingga luasnya $\frac{n}{2}(n + 1)$.



Gambar 3.6 Pemasangan ala Gauss dibuat geometris. Memasangkan batang pertama dengan yang terakhir, yang kedua dengan yang kedua dari belakang, dan seterusnya, mengemas ulang tangga itu menjadi persegi panjang selebar $n/2$ dan setinggi $n + 1$, yang luasnya $n(n + 1)/2$.

Gambar kedua memberi jawaban yang sama lewat jalan yang berbeda, dan gambar itu layak dimiliki karena ia dapat diperumum. Pada **Gambar 3.7**, batang-batang setinggi $1,2,3,\ldots,n$ berdiri berdampingan di bawah garis lurus bergradien 1 yang melalui titik asal. Luas di bawah garis itu dari 0 sampai n adalah segitiga $\frac{1}{2}n \cdot n = \frac{n^2}{2}$. Batang-batang itu menjorok melewati garisnya tepat setengah satuan persegi masing-masing, sebanyak n buah, sehingga menambahkan $\frac{n}{2}$. Totalnya: $\frac{n^2}{2} + \frac{n}{2} = \frac{n(n+1)}{2}$, sama seperti tadi.



Gambar 3.7 Deret aritmetika sebagai luas. Batang-batang setinggi 1,2, … , n duduk mengangkangi garis bergradien 1; luas totalnya $n(n+1)/2$ adalah segitiga $n^2/2$ ditambah n setengah-persegi yang menjorok melewatinya.

Mengapa $\Theta(n^2)$? Karena $\frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$ adalah polinomial berderajat 2 dengan koefisien tertinggi positif, dan §3.7.5 sudah membuktikan bahwa setiap polinomial semacam itu adalah Θ dari pangkat tertingginya. Konkretnya, $\frac{n^2}{2} \leq \frac{n(n+1)}{2} \leq n^2$ untuk $n \geq 1$, sehingga $c_1 = 1/2$, $c_2 = 1$, $n_0 = 1$.

Di mana ia muncul. Kasus terburuk Insertion Sort. Gelung `while` di dalam untuk indeks j dapat menggeser sebanyak-banyaknya $j - 1$ elemen, jadi jumlah pergeseran seluruhnya paling banyak

$$\sum_{j=2}^{n}(j-1) = 1 + 2 + \cdots + (n-1) = \frac{(n-1)n}{2} = \Theta(n^2).$$

Itulah seluruh analisis kasus terburuk Insertion Sort, dalam satu baris, sekarang setelah kita punya rumusnya.

3.9.2 Hasil yang sama lewat induksi

Bentuk tertutup itu juga dapat dibuktikan dengan **induksi matematis**, dan ada baiknya kita melakukannya sekali karena induksi adalah mesin penggerak metode substitusi di §3.11. Resepnya punya dua bagian:

1. **Basis.** Buktikan klaim $P(n_0)$ untuk kasus terkecil.
2. **Langkah induktif.** Andaikan $P(k)$ untuk semua $n_0 \leq k \leq n-1$ (ini induksi *kuat*), lalu buktikan $P(n)$.

Klaim. $S(n) = \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ untuk semua $n \geq 1$.

Basis ($n = 1$). Ruas kiri adalah $S(1) = 0 + 1 = 1$. Ruas kanan adalah $\frac{1 \cdot 2}{2} = 1$. Keduanya cocok. ✓

Langkah induktif. Andaikan $S(k) = \frac{k(k+1)}{2}$ untuk semua $k \leq n-1$. Maka

$$S(n) = \sum_{i=0}^{n} i = \left(\sum_{i=0}^{n-1} i\right) + n = S(n-1) + n$$

$$= \frac{(n-1)((n-1)+1)}{2} + n \qquad \text{(hipotesis induksi pada } k = n-1\text{)}$$

$$= \frac{(n-1)n}{2} + n = \frac{n^2 - n}{2} + \frac{2n}{2} = \frac{n^2 - n + 2n}{2} = \frac{n^2 + n}{2} = \frac{n(n+1)}{2}. \checkmark$$

Menurut induksi, rumus itu berlaku untuk setiap $n \geq 1$. ■ **Gambar 3.8** memperlihatkan rantai kesamaan ini dituliskan selengkapnya.

$$S(k) = \sum_{i=0}^{k} i = \frac{k(k+1)}{2} \quad \text{untuk } 1 \leq k \leq n-1,$$

$$S(n) = \sum_{i=0}^{n} i = \sum_{i=0}^{n-1} i + n = S(n-1) + n$$

$$= \frac{(n-1)(n-1+1)}{2} + n = \frac{n^2 - n + 2n}{2}$$

$$= \frac{n(n+1)}{2}.$$

Gambar 3.8 Langkah induktif ditulis lengkap: andaikan $S(k) = k(k+1)/2$ untuk $1 \leq k \leq n-1$, lalu bentangkan $S(n) = S(n-1) + n$ dan sederhanakan menjadi $n(n+1)/2$.

3.9.3 Deret geometri

DEFINISI (deret geometri). Untuk $a \neq 1$ real,

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{1 - a^{n+1}}{1 - a}.$$

(3.8)

Gambar 3.9 menyatakannya dalam bentuk yang sama.

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{1 - a^{n+1}}{1 - a} \qquad (a \neq 1)$$

Gambar 3.9 Deret geometri berhingga, $\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{1-a^{n+1}}{1-a}$.

Penurunan Persamaan (3.8)**, dengan trik yang sama seperti Gauss tetapi memakai perkalian alih-alih pembalikan.** Misalkan $S = 1 + a + a^2 + \cdots + a^n$. Kalikan setiap sukunya dengan a:

$$aS = a + a^2 + a^3 + \cdots + a^{n+1}.$$

Kurangkan baris kedua dari baris pertama. Semua yang di tengah saling menghapus — a menghapus a, a^2 menghapus a^2, dan seterusnya sampai a^n — sehingga tersisa hanya suku pertama S dan suku terakhir aS:

$$S - aS = 1 - a^{n+1}, \qquad \text{jadi} \qquad S(1-a) = 1 - a^{n+1}, \qquad S = \frac{1-a^{n+1}}{1-a}. \qquad \blacksquare$$

(Langkah itu membagi dengan $1 - a$, dan itulah sebabnya $a \neq 1$ disyaratkan. Ketika $a = 1$ setiap sukunya adalah 1 dan jumlahannya sekadar $n + 1$.)

Tiga wilayah, dan ketiganya adalah seluruh isi analisis pohon rekursi:

- **Jika $a < 1$ (konvergen).** Ketika $n \to \infty$, $a^{n+1} \to 0$, sehingga jumlahannya menuju $\frac{1}{1-a}$, sebuah **konstanta**. Deretnya adalah $\Theta(1)$, dan **suku pertama yang berkuasa** — seluruh ekor tak hingganya bernilai tidak lebih dari suatu kelipatan konstanta suku pertama. Contoh: $1 + 1/2 + 1/4 + \cdots = 2$.
- **Jika $a > 1$ (divergen).** Maka a^{n+1} mengerdilkan angka 1, dan $\frac{a^{n+1}-1}{a-1} = \Theta(a^n)$. **Suku terakhir yang berkuasa.** Contoh: $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1 = \Theta(2^n)$.
- **Jika $a = 1$.** Jumlahannya adalah $n + 1 = \Theta(n)$. **Setiap suku menyumbang sama banyak.**

> **WAWASAN KUNCI:** Sebuah jumlahan geometri selalu dikuasai oleh sukunya yang terbesar, sampai pada faktor konstanta. Jika rasionya di bawah 1, suku terbesarnya adalah yang pertama; jika di atas 1, yang terakhir; jika tepat 1, tidak ada satu suku terbesar dan Anda mengalikan satu suku dengan banyaknya suku. Mengenali Anda sedang berada di wilayah yang mana adalah *seluruh keterampilan* analisis pohon rekursi — dan, seperti akan ditunjukkan §3.13, itu juga persis ketiga kasus Teorema Master.

3.9.4 Deret harmonik

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \ln n + \gamma + O\left(\frac{1}{n}\right) = \Theta(\lg n). \tag{3.9}$$

Dalam Persamaan (3.9), $\gamma \approx 0.5772156649$ adalah konstanta Euler–Mascheroni. Deret harmonik divergen — ia tumbuh tanpa batas — tetapi hanya selambat logaritma. Ia muncul dalam analisis kasus rata-rata Quicksort (Bab 5) dan dalam algoritma acak pada umumnya.

Pembuktian satu baris bahwa $H_n = \Omega(\lg n)$. Kelompokkan suku-sukunya dalam blok yang panjangnya berlipat dua:

$$1 + \underbrace{\frac{1}{2}}_{\geq 1/2} + \underbrace{\left(\frac{1}{3} + \frac{1}{4}\right)}_{\geq 2 \cdot \frac{1}{4} = 1/2} + \underbrace{\left(\frac{1}{5} + \cdots + \frac{1}{8}\right)}_{\geq 4 \cdot \frac{1}{8} = 1/2} + \cdots$$

Setiap blok berisi 2^k suku bernilai paling sedikit $2^k \cdot 2^{-(k+1)} = 1/2$, dan ada sekitar $\lg n$ blok, sehingga $H_n \geq 1/2 \lg n$. $\blacksquare$

3.9.5 Jumlahan teleskopis

Sebuah jumlahan disebut **teleskopis** ketika suku-suku yang berturutan saling menghapus:

$$\sum_{i=1}^{n}(a_i - a_{i-1}) = a_n - a_0.$$

Semua yang di tengah muncul sekali dengan tanda tambah dan sekali dengan tanda kurang. Contoh klasiknya:

$$\sum_{i=1}^{n}\frac{1}{i(i+1)} = \sum_{i=1}^{n}\left(\frac{1}{i} - \frac{1}{i+1}\right) = 1 - \frac{1}{n+1},$$

karena $\frac{1}{i} - \frac{1}{i+1} = \frac{(i+1)-i}{i(i+1)} = \frac{1}{i(i+1)}$. Teleskoping juga merupakan cara terbersih menyelesaikan rekurensi sederhana: jika $T(n) - T(n-1) = f(n)$, maka menjumlahkan kedua ruas dari 1 sampai n memberi $T(n) - T(0) = \sum_{i=1}^{n} f(i)$.

3.9.6 Membatasi sebuah jumlahan dengan integral

Ketika tidak ada bentuk tertutup, bandingkan jumlahannya dengan sebuah integral. Jika f **naik monoton**, maka setiap suku $f(i)$ paling banyak sebesar luas f pada $[i, i+1]$ dan paling sedikit sebesar luasnya pada $[i-1, i]$, sehingga

$$\int_{m-1}^{n} f(x)\,dx \;\le\; \sum_{i=m}^{n} f(i) \;\le\; \int_{m}^{n+1} f(x)\,dx. \tag{3.10}$$

Jika f **turun** monoton, pertidaksamaan pada Persamaan (3.10) bertukar. **Gambar 3.7** adalah tepat gambar ini untuk $f(x) = x$: batang-batangnya adalah jumlahannya, garisnya adalah fungsi yang diintegralkan.

Contoh 1 — deret aritmetika lagi. Dengan $f(x) = x$ yang naik,

$$\sum_{i=1}^{n} i \;\le\; \int_{1}^{n+1} x\;dx = \left[\frac{x^2}{2}\right]_1^{n+1} = \frac{(n+1)^2 - 1}{2} = \frac{n^2 + 2n}{2} = O(n^2).$$

Contoh 2 — deret harmonik. Dengan $f(x) = 1/x$ yang turun,

$$\sum_{i=1}^{n}\frac{1}{i} \;\le\; 1 + \int_{1}^{n}\frac{dx}{x} = 1 + \ln n = O(\lg n),$$

yang bersama §3.9.4 memberi $H_n = \Theta(\lg n)$.

Contoh 3 — $\lg(n!)$**.** Karena $\lg(n!) = \sum_{i=1}^{n} \lg i$ dan lg naik,

$$\sum_{i=1}^{n} \lg i \;\le\; \int_{1}^{n+1} \lg x\,dx = \Theta(n\lg n), \qquad \sum_{i=1}^{n} \lg i \;\ge\; \int_{1}^{n} \lg x\,dx = \Theta(n\lg n),$$

sehingga $\lg(n!) = \Theta(n\lg n)$, yang menegaskan §3.8.7 tanpa Stirling.

Tabel 3.5 adalah kartu rujukannya.

Tabel 3.5 Jumlahan-jumlahan yang dipakai di sepanjang buku ini, dengan bentuk tertutup dan kelas asimtotiknya.

Jumlahan	Bentuk tertutup	Asimtotik
$\sum_{i=1}^{n} 1$	n	$\Theta(n)$
$\sum_{i=1}^{n} i$	$\frac{n(n+1)}{2}$	$\Theta(n^2)$
$\sum_{i=1}^{n} i^2$	$\frac{n(n+1)(2n+1)}{6}$	$\Theta(n^3)$
$\sum_{i=1}^{n} i^k$	—	$\Theta(n^{k+1})$ untuk $k \geq 0$
$\sum_{i=0}^{n} a^i,\ a \neq 1$	$\frac{1-a^{n+1}}{1-a}$	$\Theta(1)$ jika $a < 1$; $\Theta(n)$ jika $a = 1$; $\Theta(a^n)$ jika $a > 1$
$\sum_{i=0}^{\infty} a^i$ dengan $\lvert a \rvert < 1$	$\frac{1}{1-a}$	$\Theta(1)$
$\sum_{i=1}^{n} \frac{1}{i}$	$\ln n + \gamma + O(1/n)$	$\Theta(\lg n)$
$\sum_{i=1}^{n} \lg i$	$\lg(n!)$	$\Theta(n \lg n)$
$\sum_{i=1}^{n} (a_i - a_{i-1})$	$a_n - a_0$	teleskopis

Kode 3.4 memeriksa empat di antara baris-baris itu secara numerik, terhadap penjumlahan brutal.

Kode 3.4 Penjumlahan brutal diperiksa terhadap setiap bentuk tertutup yang diturunkan di §3.9.

```
// sums.cpp - Memeriksa bentuk tertutup empat jumlahan yang akan kita pakai
// berulang-ulang: aritmetika, geometri, harmonik, teleskopis.
// Kompilasi: g++ -std=c++17 -O2 -o sums sums.cpp
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main() {
    cout << fixed << setprecision(6);

    cout << "-- Arithmetic: sum of i for i = 1..n, against n(n+1)/2 --\n";
    for (long long n : {5LL, 10LL, 100LL, 1000LL}) {
        long long brute = 0;
        for (long long i = 1; i <= n; ++i) brute += i;
        long long closed = n * (n + 1) / 2;
        cout << "  n = " << setw(5) << n << "  brute = " << setw(8) << brute
             << "  closed = " << setw(8) << closed
             << (brute == closed ? "  MATCH\n" : "  DIFFER\n");
    }

    cout << "\n-- Geometric with ratio a = 1/2 (convergent): sum a^i, i = 0..n --\n";
    double s = 0.0;
    for (int n = 0; n <= 30; ++n) {
        s += pow(0.5, n);
        if (n == 0 || n == 3 || n == 10 || n == 30)
            cout << "  n = " << setw(3) << n << "  sum = " << s
                 << "   limit 1/(1-a) = 2.000000\n";
    }

    cout << "\n-- Geometric with ratio a = 2 (divergent): sum 2^i, i = 0..n --\n";
    for (int n : {3, 5, 10, 20}) {
        double t = 0.0;
        for (int i = 0; i <= n; ++i) t += pow(2.0, i);
        cout << "  n = " << setw(3) << n << "  sum = " << setw(12) << t
             << "   2^(n+1)-1 = " << pow(2.0, n + 1) - 1 << "\n";
    }
```

Kode 3.4 (Lanjutan) *Penjumlahan brutal diperiksa terhadap setiap bentuk tertutup yang diturunkan di §3.9.*

```
    cout << "\n-- Harmonic: H(n) = sum 1/i, against ln n + 0.5772156649 --\n";
    double h = 0.0;
    int next = 1;
    for (int i = 1; i <= 1000000; ++i) {
        h += 1.0 / i;
        if (i == next) {
            cout << "  n = " << setw(8) << i << "  H(n) = " << h
                 << "   ln n + gamma = " << log((double)i) + 0.5772156649 << "\n";
            next *= 100;
        }
    }

    cout << "\n-- Telescoping: sum 1/(i(i+1)) for i = 1..n equals 1 - 1/(n+1) --\n";
    for (int n : {1, 5, 50, 5000}) {
        double t = 0.0;
        for (int i = 1; i <= n; ++i) t += 1.0 / ((double)i * (i + 1));
        cout << "  n = " << setw(6) << n << "  sum = " << t
             << "   1 - 1/(n+1) = " << 1.0 - 1.0 / (n + 1) << "\n";
    }
    return 0;
}
```

Keluarannya:

```
-- Arithmetic: sum of i for i = 1..n, against n(n+1)/2 --
  n =     5  brute =       15  closed =       15  MATCH
  n =    10  brute =       55  closed =       55  MATCH
  n =   100  brute =     5050  closed =     5050  MATCH
  n =  1000  brute =   500500  closed =   500500  MATCH

-- Geometric with ratio a = 1/2 (convergent): sum a^i, i = 0..n --
  n =   0  sum = 1.000000   limit 1/(1-a) = 2.000000
  n =   3  sum = 1.875000   limit 1/(1-a) = 2.000000
  n =  10  sum = 1.999023   limit 1/(1-a) = 2.000000
  n =  30  sum = 2.000000   limit 1/(1-a) = 2.000000

-- Geometric with ratio a = 2 (divergent): sum 2^i, i = 0..n --
  n =   3  sum =    15.000000   2^(n+1)-1 = 15.000000
  n =   5  sum =    63.000000   2^(n+1)-1 = 63.000000
  n =  10  sum =  2047.000000   2^(n+1)-1 = 2047.000000
  n =  20  sum = 2097151.000000   2^(n+1)-1 = 2097151.000000

-- Harmonic: H(n) = sum 1/i, against ln n + 0.5772156649 --
  n =        1  H(n) = 1.000000   ln n + gamma = 0.577216
  n =      100  H(n) = 5.187378   ln n + gamma = 5.182386
  n =    10000  H(n) = 9.787606   ln n + gamma = 9.787556
  n =  1000000  H(n) = 14.392727   ln n + gamma = 14.392726

-- Telescoping: sum 1/(i(i+1)) for i = 1..n equals 1 - 1/(n+1) --
  n =      1  sum = 0.500000   1 - 1/(n+1) = 0.500000
  n =      5  sum = 0.833333   1 - 1/(n+1) = 0.833333
  n =     50  sum = 0.980392   1 - 1/(n+1) = 0.980392
  n =   5000  sum = 0.999800   1 - 1/(n+1) = 0.999800
```

Perhatikan jawaban terkenal Gauss yang duduk di baris ketiga: 5050. Perhatikan pula bahwa jumlahan geometri yang konvergen masih di bawah 2 setelah tiga puluh suku dan akan selalu begitu, sedangkan yang divergen sudah melewati dua juta setelah dua puluh suku — itulah bedanya antara "akar yang berkuasa" dan "daun yang berkuasa", yang sebentar lagi akan kita jumpai lagi dengan nama yang berbeda. Dan setelah *sejuta* suku, jumlahan harmoniknya baru mencapai 14.39, sejalan dengan $\ln n + \gamma$ sampai enam angka.

3.10 Rekurensi: Persamaan yang Menggambarkan Rekursi

3.10.1 Apa itu rekurensi

> **DEFINISI.** Sebuah **rekurensi** adalah persamaan atau pertidaksamaan yang menggambarkan sebuah fungsi lewat nilainya pada masukan yang lebih kecil, bersama satu atau lebih **syarat batas** (disebut juga kasus basis) yang memberikan nilainya secara langsung untuk masukan terkecil.

Waktu jalan algoritma mana pun yang memanggil dirinya sendiri secara wajar digambarkan oleh sebuah rekurensi, sebab algoritmanya sendiri memang digambarkan begitu: "untuk mengurutkan n benda, urutkan dua kelompok berisi $n/2$ benda, lalu habiskan $\Theta(n)$ untuk menggabungkannya."

Inilah kalimat itu yang diubah menjadi persamaan. **Gambar 3.10** menyatakannya sebagai definisi dua kasus.

$$T(n) = \begin{cases} \Theta(1) & \text{jika } n = 1, \\ 2T(n/2) + \Theta(n) & \text{jika } n > 1. \end{cases}$$

Gambar 3.10 Rekurensi Merge Sort sebagai definisi dua kasus: $T(n) = \Theta(1)$ ketika $n = 1$, dan $T(n) = 2T(n/2) + \Theta(n)$ ketika $n > 1$.

Bacalah keras-keras: *ketika array berisi satu elemen, mengurutkannya menelan biaya sejumlah konstanta; selain itu ia menelan dua pengurutan rekursif atas array berukuran separuh, ditambah kerja linear.* Kasus basisnya bukan hiasan. Tanpanya, rekurensi itu tidak mendefinisikan apa-apa sama sekali, sebab pembentangannya tidak pernah berhenti.

3.10.2 Bentuk umum bagi dan kuasai

Sebagian besar rekurensi dalam buku ini berbentuk

$$T(n) = a\,T(n/b) + f(n), \tag{3.11}$$

dan ada baiknya kita menamai tiap bagian Persamaan (3.11) dengan kata-kata sebelum melangkah lebih jauh.

- $a \geq 1$ adalah **berapa banyak submasalah** yang dibuat algoritmanya. Ia sebuah cacah, jadi ia paling sedikit 1.
- $b > 1$ adalah **faktor penyusutan masukannya**. Setiap submasalah berukuran n/b. Ia harus melebihi 1, sebab kalau tidak, subpersoalannya tidak pernah mengecil dan rekursinya tidak pernah berakhir.
- $f(n)$ adalah **biaya segala sesuatu yang bukan panggilan rekursif**: membelah masukan, dan menggabungkan jawaban. Ia harus positif secara asimtotik — positif untuk semua n yang cukup besar.

Perhatikan bahwa a dan b **saling bebas**. Anda mungkin membelah menjadi dua paruh lalu berekursi pada keduanya ($a = 2, b = 2$, Merge Sort); membelah menjadi dua paruh lalu berekursi hanya pada satu ($a = 1, b = 2$, pencarian biner); atau memotong matriks $n \times n$ menjadi empat blok bersisi $n/2$ lalu melakukan tujuh panggilan rekursif pada blok-blok semacam itu ($a = 7, b = 2$, perkalian matriks Strassen, §3.14.10).

3.10.3 Tiga kerewelan yang boleh kita abaikan

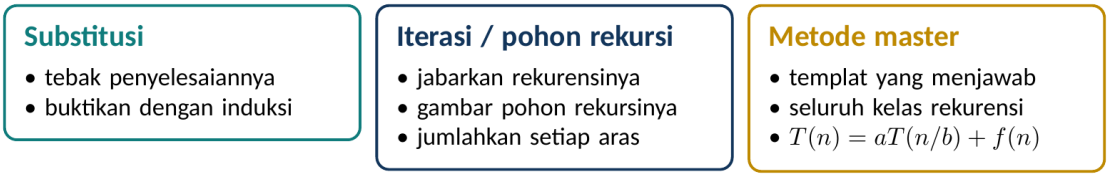
Lantai dan langit-langit. Rekurensi Merge Sort yang jujur adalah $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$. Kita menulis $2T(n/2)$. Dapat dibuktikan bahwa untuk rekurensi-rekurensi dalam bab ini, lantai dan langit-langit mengubah jawabannya paling banyak sebesar faktor konstanta, dan Θ menyerapnya. Jadi kita membuangnya, dan bernalar seolah-olah n selalu tepat sebuah pangkat dari b.

Syarat batas. Kita biasanya menghilangkan kasus basisnya dan hanya menulis $T(n) = 2T(n/2) + \Theta(n)$. Pembenarannya adalah bahwa $T(n) = \Theta(1)$ untuk semua n di bawah suatu konstanta, dan mengubah sejumlah konstanta nilai berukuran konstanta tidak dapat mengubah jawaban asimtotiknya. Ketika kita memerlukan angka konkret untuk sebuah perhitungan terkerjakan, kita cukup menetapkan $T(1) = 1$.

Persamaan dan pertidaksamaan. Rekurensi yang dinyatakan dengan $\leq$ memberi batas O; yang dinyatakan dengan $\geq$ memberi batas Ω; yang dinyatakan dengan $=$ memberi Θ. Ingat baik-baik Anda sedang memegang yang mana.

3.10.4 Tiga metodenya

Gambar 3.11 mendaftar tiga teknik yang diajarkan bab ini, dalam urutan pengajarannya.



Gambar 3.11 Tiga metode untuk menyelesaikan rekurensi: substitusi (tebak, lalu sahkan dengan induksi), iterasi atau pohon rekursi (bentangkan rekurensinya dan gambar), dan metode master (sebuah cetakan yang menjawab sekelas rekurensi sekaligus).

Ketiganya membagi kerja dengan rapi:

- **pohon rekursi** *menemukan* sebuah jawaban, secara tak resmi tetapi cepat;
- **substitusi** *membuktikan* jawaban itu, secara ketat;
- **metode master** *melewati* keduanya, kapan pun rekurensinya pas dengan cetakannya.

Dalam praktik Anda mencoba metode master lebih dulu. Jika rekurensinya tidak pas — atau jatuh ke salah satu celahnya — Anda menggambar pohon untuk memperoleh tebakan, lalu memakai substitusi untuk menjadikannya pembuktian.

3.11 Metode 1: Metode Substitusi

Substitution method (metode substitusi) punya tepat dua langkah.

1. **Tebak** bentuk penyelesaiannya.
2. **Sahkan** tebakan itu dengan induksi matematis, dan sambil melakukannya, temukan konstanta yang membuatnya bekerja.

Namanya agak menyesatkan: yang Anda substitusikan adalah hipotesis induktif ke dalam rekurensinya. Ia satu-satunya dari ketiga metode itu yang menghasilkan pembuktian sejati, jadi ia pula yang harus diraih ketika sebuah batas harus kedap udara.

3.11.1 Sebuah contoh lengkap: $T(n) = 4T(n/2) + n$ adalah $O(n^3)$

Tebakannya. $T(n) \le c\,n^3$ untuk suatu konstanta $c > 0$ dan semua $n \ge n_0$.

Hipotesis induktifnya. Andaikan batas itu berlaku untuk semua masukan yang lebih kecil: $T(k) \le c\,k^3$ untuk setiap $k < n$. Khususnya ia berlaku untuk $k = n/2$.

Langkah induktifnya, satu baris satu langkah.

$$T(n) = 4\,T(n/2) + n$$

$$\le 4 \cdot c\left(\frac{n}{2}\right)^3 + n \qquad \text{(menyulihkan hipotesis pada } k = n/2\text{)}$$

$$= 4 \cdot c \cdot \frac{n^3}{8} + n \qquad \text{karena } (n/2)^3 = n^3/8$$

$$= \frac{c\,n^3}{2} + n$$

$$= c\,n^3$$

$$-\left(\frac{c\,n^3}{2} - n\right) \qquad \text{(menulis ulang, untuk menampakkan kelonggarannya)}$$

$$\leq c\,n^3 \qquad \text{asalkan } \frac{c\,n^3}{2} - n \geq 0.$$

Menemukan konstantanya. Syarat $\frac{cn^3}{2} \geq n$ ditata ulang menjadi $c \geq \frac{2}{n^2}$. Karena $n \geq 1$ memberi $\frac{2}{n^2} \leq 2$, pilihan tunggal $c \geq 2$ berhasil untuk setiap $n \geq 1$.

Kasus basisnya. Kita juga harus memeriksa bahwa induksinya bermula. Dengan $T(1) = 1$ kita memerlukan $1 \leq c \cdot 1^3 = c$, yang benar untuk $c \geq 2$. ✓

Kesimpulan. $T(n) \leq 2n^3$ untuk semua $n \geq 1$, jadi $T(n) = O(n^3)$. ■

Langkah menulis ulang itu — mengubah $\frac{cn^3}{2} + n$ menjadi $cn^3 - \left(\frac{cn^3}{2} - n\right)$ — adalah gerakan yang harus dipelajari. Ia bukan kepiawaian aljabar; ia pembukuan. Ia menaruh jawaban yang Anda inginkan di sebelah kiri dan segala sesuatu yang harus Anda buang dalam satu tanda kurung yang kelihatan.

3.11.2 Percobaan yang gagal: apakah ia $O(n^2)$?

Batas $O(n^3)$ itu benar tetapi longgar. Pohon rekursi di §3.12 akan memperlihatkan bahwa jawaban sejatinya adalah $\Theta(n^2)$. Maka mari kita coba membuktikan $O(n^2)$ secara langsung — dan menyaksikannya gagal, sebab kegagalannya mengajarkan sesuatu.

Tebakannya. $T(n) \leq c\,n^2$.

$$T(n) = 4T(n/2) + n \leq 4c\left(\frac{n}{2}\right)^2 + n = 4c \cdot \frac{n^2}{4} + n = c\,n^2 + n.$$

Kita memerlukan $cn^2 + n \leq cn^2$, yang menuntut $n \leq 0$. **Mustahil**, untuk setiap pilihan $c > 0$. **Gambar 3.12** mencatat jalan buntu ini.

$$\begin{aligned}
&\text{Akan ditunjukkan } T(n) = O(n^2).\\
&\text{Asumsikan } T(k) \leq ck^2 \text{ untuk } k < n.\\
T(n) &= 4T(n/2) + n\\
&\leq 4c(n/2)^2 + n\\
&= cn^2 + n\\
&\not\leq cn^2 \quad \text{untuk } \textit{tidak ada} \text{ pilihan } c > 0.
\end{aligned}$$

Gambar 3.12 Metode substitusi yang gagal. Mengandaikan $T(k) \leq ck^2$ menghasilkan $T(n) \leq cn^2 + n$, dan $cn^2 + n \leq cn^2$ tidak berlaku untuk pilihan $c > 0$ **mana pun**.

> **JEBAKAN YANG SERING TERJADI.** Sampai di $T(n) \leq cn^2 + n$ lalu menyatakan "$cn^2 + n$ adalah $O(n^2)$, jadi kita selesai" itu **salah**, dan itulah kekeliruan tunggal yang paling sering terjadi dalam pembuktian substitusi. Induksinya harus menghasilkan kembali hipotesisnya *persis*, dengan konstanta c yang *sama*. Sisa positif apa pun, serendah apa pun ordenya,

> mematahkan induksinya: terapkan langkah yang ceroboh itu n kali dan sisa-sisanya menumpuk menjadi sesuatu yang sama sekali bukan $O(n^2)$.

3.11.3 Perbaikannya: perkuat hipotesisnya

Obatnya berlawanan dengan naluri: **buktikan sesuatu yang lebih kuat**. Kurangkan sebuah suku berorde lebih rendah dari tebakannya, supaya induksinya punya kelonggaran untuk menyerap sisanya.

Tebakan yang diperkuat. $T(n) \le c_1 n^2 - c_2 n$ untuk konstanta $c_1, c_2 > 0$.

Langkah induktifnya, satu baris satu langkah.

$$T(n) = 4T(n/2) + n$$

$$\le 4\left(c_1\left(\frac{n}{2}\right)^2 - c_2\left(\frac{n}{2}\right)\right) + n \qquad \text{(hipotesis pada } k = n/2\text{)}$$

$$= 4\left(\frac{c_1 n^2}{4} - \frac{c_2 n}{2}\right) + n$$

$$= c_1 n^2 - 2c_2 n + n$$

$$= c_1 n^2 - c_2 n - (c_2 n - n) \qquad \text{(menampakkan kelonggarannya lagi)}$$

$$\le c_1 n^2 - c_2 n \qquad \text{asalkan } c_2 n - n \ge 0\text{, yaitu } c_2 \ge 1.$$

Induksinya kini menutup. **Gambar 3.13** menuliskan rantai ini selengkapnya.

$$\begin{aligned}
&\text{Asumsikan } T(k) \le c_1 k^2 - c_2 k \text{ untuk } k < n.\\
T(n) &= 4T(n/2) + n\\
&\le 4\big(c_1(n/2)^2 - c_2(n/2)\big) + n\\
&= c_1 n^2 - 2c_2 n + n\\
&= c_1 n^2 - c_2 n - (c_2 n - n)\\
&\le c_1 n^2 - c_2 n \quad \text{jika } c_2 \ge 1.
\end{aligned}$$

Gambar 3.13 Hipotesis yang diperkuat $T(k) \le c_1 k^2 - c_2 k$ menutup induksinya: tambahan $-c_2 n$ menyerap sisa $+n$ kapan pun $c_2 \ge 1$.

Mengapa membuktikan *lebih banyak* justru membuatnya *lebih mudah*? Karena hipotesis induktif adalah apa yang boleh Anda andaikan, sekaligus apa yang harus Anda buktikan. Hipotesis yang lebih kuat memberi Anda andaian yang lebih kuat pada $k = n/2$, dan di sini tambahan $-c_2(n/2)$, dikalikan keempat subpersoalannya, menjadi $-2c_2 n$ — dua kali lipat kelonggaran negatif yang harus Anda hasilkan. Kelebihan $-c_2 n$ itulah yang membayar $+n$.

Akhirnya, karena $c_1 n^2 - c_2 n \le c_1 n^2$, batas yang diperkuat itu mengakibatkan $T(n) = O(n^2)$, yang memang kita inginkan sejak awal.

> **WAWASAN KUNCI:** Ketika induksi meninggalkan Anda dengan (batas yang diinginkan) + (sisa positif), kurangkan sebuah suku berorde sama dengan sisanya dari hipotesisnya, lalu coba lagi. **Jangan** menambahkan sebuah suku — itu membuat hipotesisnya lebih lemah dan sisanya lebih buruk.

3.11.4 Dua teknik substitusi lagi

Waspadai kasus basis. Kadang langkah induktifnya bekerja tetapi kasus basisnya tidak. Untuk $T(n) = 2T(\lfloor n/2 \rfloor) + n$ dengan $T(1) = 1$, tebakan $T(n) \leq cn\lg n$ gagal pada $n = 1$, karena $c \cdot 1 \cdot \lg 1 = 0 < 1$. Perbaikannya adalah memindahkan kasus basisnya: batas asimtotik hanya perlu berlaku untuk $n \geq n_0$, jadi buktikanlah basisnya pada $n = 2$ dan $n = 3$ sebagai gantinya. Kita punya $T(2) = 4$ dan $T(3) = 5$; tuntutannya adalah $4 \leq 2c\lg 2 = 2c$ dan $5 \leq 3c\lg 3 \approx 4.75c$, keduanya dipenuhi oleh $c \geq 2$. Langkah induktif untuk $n \geq 4$ hanya pernah merujuk kembali ke $\lfloor n/2 \rfloor \geq 2$, jadi $n = 1$ tidak pernah dibutuhkan.

Mengganti peubah. Rekurensi $T(n) = 2\,T\left(\sqrt{n}\right) + \lg n$ tampak tidak seperti apa pun yang dapat kita tangani. Ambil $m = \lg n$, sehingga $n = 2^m$ dan $\sqrt{n} = 2^{m/2}$:

$$T(2^m) = 2\,T(2^{m/2}) + m.$$

Sekarang definisikan $S(m) = T(2^m)$, yang memberi $S(m) = 2S(m/2) + m$ — rekurensi Merge Sort, yang jawabannya sudah kita ketahui yaitu $\Theta(m\lg m)$. Mengganti kembali, $m = \lg n$:

$$T(n) = S(\lg n) = \Theta(\lg n \cdot \lg\lg n).$$

3.11.5 Memeriksa aljabarnya terhadap angka sesungguhnya

Pembuktian substitusi menghasilkan sebuah pertidaksamaan. Menenangkan hati — dan, bagi mahasiswa yang baru pertama kali menjumpainya, mencerahkan — bila kita menghitung rekurensinya secara eksak lalu menyaksikan pertidaksamaan itu berlaku. **Kode 3.5** melakukannya untuk kedua rekurensi di bagian ini.

Kode 3.5 Menghitung dua rekurensi secara eksak, lalu membandingkannya dengan batas yang dibuktikan lewat substitusi.

```
// verify.cpp - Menyelesaikan rekurensi secara brutal, agar batas yang dibuktikan di
// atas kertas dapat diperiksa terhadap angka yang seharusnya ia batasi.
// Kompilasi: g++ -std=c++17 -O2 -o verify verify.cpp
#include <iostream>
#include <iomanip>
#include <map>
#include <cmath>
using namespace std;

// T(n) = 4T(n/2) + n, dengan T(1) = 1, dihitung hanya pada pangkat dua.
double T4(long long n, map<long long,double>& memo) {
    if (n <= 1) return 1.0;
    auto it = memo.find(n);
    if (it != memo.end()) return it->second;
    double v = 4.0 * T4(n / 2, memo) + (double)n;
    memo[n] = v;
    return v;
}

// T(n) = 2T(n/2) + n, dengan T(1) = 1: rekurensi Merge Sort.
double T2(long long n, map<long long,double>& memo) {
    if (n <= 1) return 1.0;
    auto it = memo.find(n);
    if (it != memo.end()) return it->second;
    double v = 2.0 * T2(n / 2, memo) + (double)n;
    memo[n] = v;
    return v;
}

int main() {
    cout << "T(n) = 4T(n/2) + n,  T(1) = 1\n";
    cout << left << setw(10) << "n" << setw(16) << "T(n)"
         << setw(16) << "2n^3" << setw(20) << "2n^2 - n"
         << "T(n)/n^2\n";
    cout << string(74, '-') << "\n";
    map<long long,double> m4;
    for (long long n = 1; n <= 1024; n *= 2) {
        double t = T4(n, m4);
        cout << left << setw(10) << n << fixed << setprecision(0)
             << setw(16) << t
             << setw(16) << 2.0 * n * n * n
             << setw(20) << 2.0 * n * n - n
             << setprecision(4) << t / ((double)n * n) << "\n";
    }
```

Kode 3.5 (Lanjutan) *Menghitung dua rekurensi secara eksak, lalu membandingkannya dengan batas yang dibuktikan lewat substitusi.*

```
    cout << "\nT(n) = 2T(n/2) + n,  T(1) = 1   (Merge Sort)\n";
    cout << left << setw(10) << "n" << setw(16) << "T(n)"
         << setw(20) << "n lg n + n" << "T(n)/(n lg n)\n";
    cout << string(66, '-') << "\n";
    map<long long,double> m2;
    for (long long n = 2; n <= 1024; n *= 2) {
        double t = T2(n, m2);
        double lg = log2((double)n);
        cout << left << setw(10) << n << fixed << setprecision(0)
             << setw(16) << t
             << setw(20) << n * lg + n
             << setprecision(4) << t / (n * lg) << "\n";
    }
    return 0;
}
```

Program itu mencetak:

```
T(n) = 4T(n/2) + n,  T(1) = 1
n         T(n)            2n^3            2n^2 - n            T(n)/n^2
---------------------------------------------------------------------------
1         1               2               1                   1.0000
2         6               16              6                   1.5000
4         28              128             28                  1.7500
8         120             1024            120                 1.8750
16        496             8192            496                 1.9375
32        2016            65536           2016                1.9688
64        8128            524288          8128                1.9844
128       32640           4194304         32640               1.9922
256       130816          33554432        130816              1.9961
512       523776          268435456       523776              1.9980
1024      2096128         2147483648      2096128             1.9990

T(n) = 2T(n/2) + n,  T(1) = 1   (Merge Sort)
n         T(n)            n lg n + n          T(n)/(n lg n)
-----------------------------------------------------------------
2         4               4                   2.0000
4         12              12                  1.5000
8         32              32                  1.3333
16        80              80                  1.2500
32        192             192                 1.2000
64        448             448                 1.1667
128       1024            1024                1.1429
256       2304            2304                1.1250
512       5120            5120                1.1111
1024      11264           11264               1.1000
```

Perhatikan kolom ketiga dan keempat pada tabel pertama. Batas $O(n^3)$ berupa $2n^3$ itu benar tetapi luar biasa longgar — pada $n = 1024$ ia seribu kali lebih besar daripada $T(n)$. Batas yang diperkuat $2n^2 - n$ bukan sekadar benar, ia **eksak**: $T(n) = 2n^2 - n$ pada setiap barisnya. Pilihan kita $c_1 = 2$, $c_2 = 1$ ternyata yang terbaik yang mungkin. Kolom terakhir, $T(n)/n^2$, memanjat mantap menuju 2 dan tidak pernah mencapainya, dan itulah $\Theta(n^2)$ yang dibuat kasatmata.

Tabel kedua sama bersihnya: $T(n) = n\lg n + n$ secara eksak, untuk rekurensi Merge Sort dengan $T(1) = 1$. Nisbah $T(n)/(n\lg n)$ turun pelan menuju 1, menegaskan $\Theta(n\lg n)$ dan memperlihatkan mengapa suku $+\,n$ itu berarti untuk masukan kecil dan lenyap untuk masukan besar.

3.12 Metode 2: Iterasi dan Pohon Rekursi

Metode substitusi memerlukan sebuah tebakan sebelum ia dapat mulai. *Recursion tree* (pohon rekursi) adalah tempat tebakan itu berasal.

Gagasannya sekadar **membentangkan rekurensinya berulang-ulang** — substitusikan definisi T ke dalam dirinya sendiri, lagi dan lagi — lalu menjumlahkan semua yang berjatuhan. Digambar sebagai gambar, tiap pembentangan menjadi satu tingkat baru sebuah pohon: akarnya adalah persoalan aslinya, anak-anaknya adalah subpersoalannya, anak-anak mereka adalah sub-subpersoalannya, dan daun-daunnya adalah kasus basisnya. **Biaya yang tertulis di dalam sebuah simpul** adalah kerja tak rekursif f yang dilakukan di simpul itu, tidak termasuk anak-anaknya.

Dua bilangan menentukan segalanya:

1. **Ada berapa tingkat** — yaitu, berapa kali rekurensinya harus diiterasi sebelum argumennya mencapai syarat batasnya?
2. **Berapa biaya tiap tingkat seluruhnya** — jumlah biaya semua simpul pada kedalaman itu?

Jumlahkan biaya tiap tingkatnya, dan Anda memperoleh $T(n)$.

3.12.1 Iterasi murni: $T(n) = n + 3T(\lfloor n/4 \rfloor)$

Mari kita kerjakan satu rekurensi sepenuhnya dengan aljabar lebih dulu, tanpa gambar, supaya mekaniknya jelas. **Gambar 3.14** memperlihatkan pembentangan yang sama dituliskan selengkapnya.

$$\begin{aligned}
T(n) &= n + 3T(\lfloor n/4 \rfloor) \\
&= n + 3\big(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)\big) \\
&= n + 3\big(\lfloor n/4 \rfloor + 3\big(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor)\big)\big) \\
&= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor) \\
&\le n + \tfrac{3}{4}n + \tfrac{9}{16}n + \tfrac{27}{64}n + \cdots + 3^{\log_4 n}\, T(1) \\
&\le n \sum_{i=0}^{\infty} \left(\tfrac{3}{4}\right)^i + \Theta\left(n^{\log_4 3}\right) \\
&= 4n + o(n) \\
&= O(n).
\end{aligned}$$

Gambar 3.14 Mengiterasi $T(n) = n + 3T(\lfloor n/4 \rfloor)$ sebanyak tiga kali, mengumpulkan deret geometri $n\sum(3/4)^i$, dan menyimpulkan $T(n) = O(n)$.

Pembentangan, satu substitusi satu baris.

$$T(n) = n + 3\,T(n/4)$$

$$= n + 3\left(\frac{n}{4} + 3\,T(n/16)\right) = n + \frac{3n}{4} + 9\,T(n/16)$$

$$= n + \frac{3n}{4} + 9\left(\frac{n}{16} + 3\,T(n/64)\right) = n + \frac{3n}{4} + \frac{9n}{16} + 27\,T(n/64)$$

$$= \cdots = n\left(1 + \frac{3}{4} + \frac{9}{16} + \frac{27}{64} + \cdots\right) + 3^{\log_4 n}\,T(1).$$

Mengenali deretnya. Isi tanda kurung itu geometri dengan rasio $3/4$:

$$T(n) \le n \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}).$$

Membatasinya. Karena $3/4 < 1$, rentangkan jumlahannya sampai tak hingga — ia hanya menjadi lebih besar — dan pakai $\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$:

$$\sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i = \frac{1}{1 - 3/4} = 4.$$

Menangani suku daunnya. Ada $3^{\log_4 n} = n^{\log_4 3} \approx n^{0.793}$ daun, masing-masing berbiaya $\Theta(1)$, sehingga suku daunnya adalah $\Theta(n^{0.793}) = o(n)$ — dapat diabaikan di sebelah $4n$.

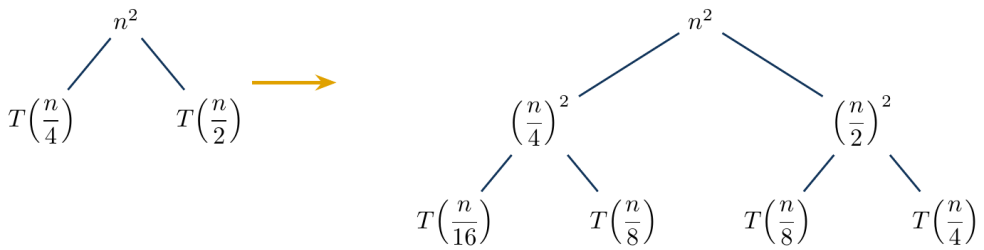
Kesimpulan. $T(n) \leq 4n + o(n) = O(n)$. ■

> **WAWASAN KUNCI:** Rasio di bawah 1 ⇒ deret geometrinya konvergen ⇒ **akarnya** yang berkuasa ⇒ $T(n) = \Theta(f(n))$. Di sini $f(n) = n$, dan memang $T(n) = \Theta(n)$.

3.12.2 Menggambar pohonnya: $T(n) = T(n/4) + T(n/2) + n^2$

Rekurensi ini punya submasalah berukuran **tidak sama**, jadi Teorema Master tidak dapat menyentuhnya. Pohonnya bisa. **Gambar 3.15** memperlihatkan dua pembentangan pertama: akarnya, berlabel n^2, menumbuhkan anak berukuran $n/4$ dan $n/2$; masing-masing anak itu menumbuhkan dua anaknya sendiri, berukuran $n/16, n/8$ dan $n/8, n/4$.

$$T(n) = T(n/4) + T(n/2) + n^2$$



Gambar 3.15 Membangun pohon rekursi untuk $T(n) = T(n/4) + T(n/2) + n^2$. Tiap pembentangan mengganti sebuah daun $T(\cdot)$ dengan sebuah simpul yang membawa biaya tak rekursifnya sendiri beserta dua anak baru.

Biaya tiap tingkat, dihitung satu tingkat satu kali.

Tingkat 0. Satu simpul berukuran n. Biaya $= n^2$.

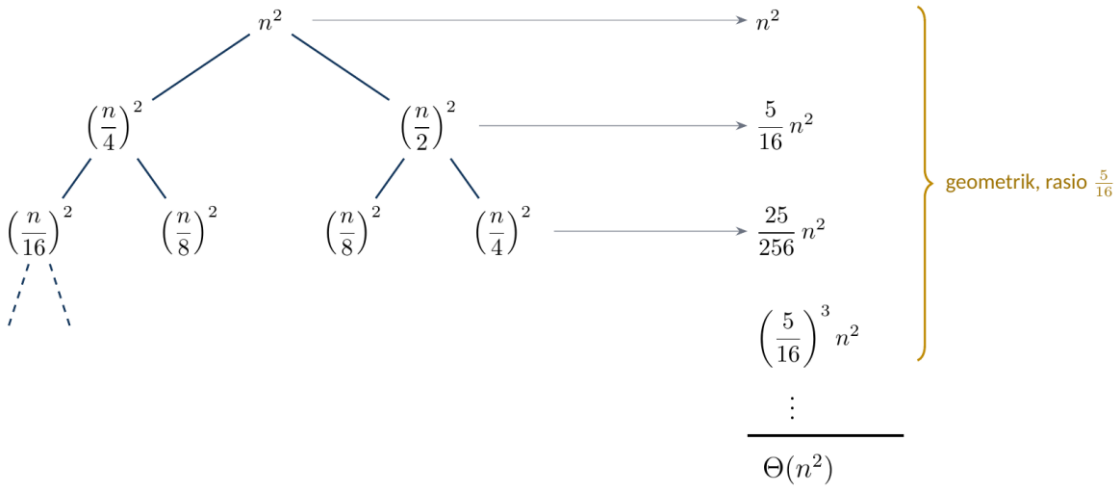
Tingkat 1. Dua simpul, berukuran $n/4$ dan $n/2$. Biaya

$$\left(\frac{n}{4}\right)^2 + \left(\frac{n}{2}\right)^2 = \frac{n^2}{16} + \frac{n^2}{4} = \frac{n^2}{16} + \frac{4n^2}{16} = \frac{5n^2}{16}.$$

Tingkat 2. Empat simpul, berukuran $n/16, n/8, n/8, n/4$. Biaya

$$\frac{n^2}{256} + \frac{n^2}{64} + \frac{n^2}{64} + \frac{n^2}{16} = \frac{n^2 + 4n^2 + 4n^2 + 16n^2}{256} = \frac{25n^2}{256} = \left(\frac{5}{16}\right)^2 n^2.$$

Polanya tak mungkin keliru: **tiap tingkat berbiaya tepat $\frac{5}{16}$ dari tingkat di atasnya.** Sebabnya adalah pengkuadratan bersifat multiplikatif, sehingga rasio antartingkatnya adalah $(1/4)^2 + (1/2)^2 = \frac{5}{16}$, tingkat demi tingkat. **Gambar 3.16** memperlihatkan pohon yang sudah lengkap dengan total tiap tingkatnya tertulis di sampingnya.



Gambar 3.16 Pohon rekursi yang sudah lengkap untuk $T(n) = T(n/4) + T(n/2) + n^2$. Total tiap tingkatnya $n^2, \frac{5}{16}n^2, \frac{25}{256}n^2, \ldots$ membentuk deret geometri berasio $5/16 < 1$, yang berjumlah $\Theta(n^2)$.

Menjumlahkan.

$$T(n) \le n^2 \sum_{i=0}^{\infty} \left(\frac{5}{16}\right)^i = n^2 \cdot \frac{1}{1 - \frac{5}{16}} = n^2 \cdot \frac{1}{\frac{11}{16}} = \frac{16}{11} n^2 \approx 1.4545\, n^2.$$

Dan karena tingkat 0 saja sudah berbiaya n^2, kita punya $n^2 \le T(n) \le \frac{16}{11}n^2$, yang memberi

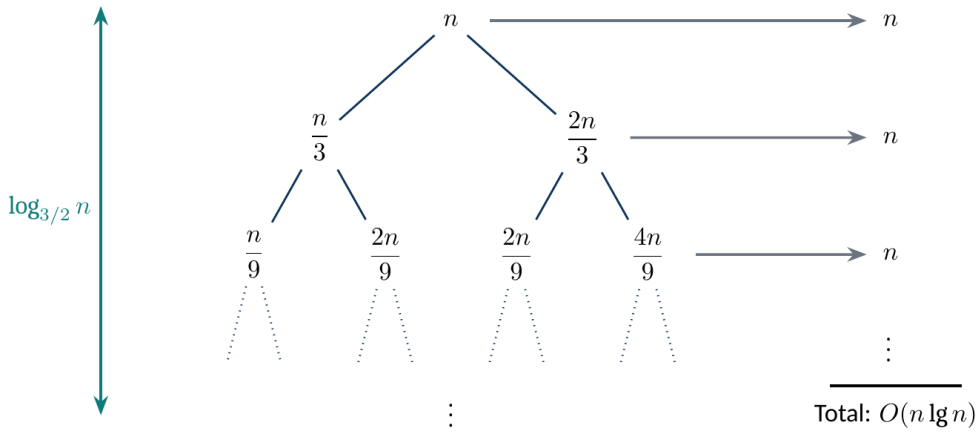
$$T(n) = \Theta(n^2). \qquad \blacksquare$$

Perhatikan betapa murahnya rekursi itu: akarnya membayar n^2 dan segala sesuatu di bawahnya, sampai ke dasar, hanya menambahkan 45% lagi.

3.12.3 Ketika tiap tingkat berbiaya sama: $T(n) = T(n/3) + T(2n/3) + n$

Di sini subpersoalannya berat sebelah — sepertiga dan dua pertiga — tetapi keduanya dijumlahkan kembali menjadi utuh. **Gambar 3.17** adalah pohonnya.

$$T(n) = T(n/3) + T(2n/3) + n$$



Gambar 3.17 Pohon rekursi untuk $T(n) = T(n/3) + T(2n/3) + n$. Tiap tingkat berjumlah tepat n, dan lintasan akar-ke-daun yang terpanjang memiliki $\log_{3/2} n$ tingkat, sehingga totalnya $O(n\lg n)$.

Biaya tiap tingkat.

Tingkat 0. n.

Tingkat 1. $\frac{n}{3} + \frac{2n}{3} = n$.

Tingkat 2. $\frac{n}{9} + \frac{2n}{9} + \frac{2n}{9} + \frac{4n}{9} = \frac{9n}{9} = n$.

Tiap tingkat berjumlah n. Itu bukan kebetulan: ukuran anak-anaknya dijumlahkan menjadi ukuran induknya pada setiap simpul, dan fungsi biaya $f(n) = n$ bersifat linear, jadi tidak ada yang hilang dalam pembelahannya.

Ada berapa tingkat? Pohonnya tidak seimbang. Lintasan terpendek ke bawah membagi dengan 3 tiap kali dan mencapai basisnya setelah $\log_3 n$ langkah; yang terpanjang membagi dengan 3/2 tiap kali dan mencapai basisnya setelah $\log_{3/2} n$ langkah. Dengan memakai yang lebih dalam sebagai batas atas:

$$T(n) \leq n \cdot \log_{3/2} n = n \cdot \frac{\lg n}{\lg(3/2)} = \frac{1}{\lg 1.5}\, n\lg n \approx 1.71\, n\lg n = O(n\lg n).$$

Memakai yang lebih dangkal sebagai batas bawah memberi $T(n) \geq n\log_3 n = \Omega(n\lg n)$, sehingga sesungguhnya $T(n) = \Theta(n\lg n)$.

> **WAWASAN KUNCI:** Rasio antartingkat tepat 1 ⇒ tiap tingkat berbiaya sama ⇒ kalikan biaya satu tingkat dengan banyaknya tingkat ⇒ $T(n) = \Theta(f(n)\log n)$.

3.12.4 Mencacah daunnya: dari mana $n^{\log_b a}$ berasal

Untuk bentuk seimbang $T(n) = a\,T(n/b) + f(n)$, pohonnya berbentuk kaku, dan ada baiknya bentuk itu diturunkan dengan saksama sebab besaran pusat Teorema Master jatuh langsung darinya.

Ukurannya. Akarnya berukuran n. Tingkat 1 berukuran n/b. Tingkat 2 berukuran n/b^2. Tingkat i berukuran n/b^i.

Kedalamannya. Rekursinya berhenti ketika ukurannya mencapai 1:

$$\frac{n}{b^i} = 1 \Leftrightarrow b^i = n \Leftrightarrow i = \log_b n.$$

Jadi pohonnya punya $\log_b n + 1$ tingkat, bernomor 0 sampai $\log_b n$.

Cacah simpulnya. Akarnya satu simpul. Tiap simpul punya a anak, jadi tingkat i memuat a^i simpul.

Daunnya. Tingkat terbawah adalah tingkat $\log_b n$, jadi banyaknya daun adalah

$$a^{\log_b n} = n^{\log_b a}, \tag{3.12}$$

menurut kesamaan yang dibuktikan di §3.8.6. **Persamaan** Persamaan (3.12) **adalah rumus tunggal yang paling penting dalam bab ini.**

Biaya tiap tingkat. Tingkat i memuat a^i simpul, masing-masing berukuran n/b^i, masing-masing berbiaya $f(n/b^i)$. Jadi

$$\begin{aligned} \text{biaya aras } i \quad &= \quad a^i f\left(\frac{n}{b^i}\right), \qquad \text{dan} \qquad T(n) \\ \underbrace{\sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)}_{\text{pembagian dan penggabungan, semua aras}} \quad &+ \underbrace{\Theta(n^{\log_b a})}_{\text{semua daun}}. \end{aligned} \tag{3.13}$$

Suku pertama Persamaan (3.13) adalah total biaya pembelahan dan penggabungan di seluruh pohon. Suku kedua adalah biaya menyelesaikan seluruh $n^{\log_b a}$ submasalah berukuran 1 — segenap kerja yang terdorong ke daun-daunnya.

> **CONTOH:** Merge Sort punya $a = 2$, $b = 2$, sehingga cacah daunnya adalah $n^{\log_2 2} = n^1 = n$. Benar: mengurutkan n benda berakhir di n array berisi satu elemen. Pencarian biner punya $a = 1$, $b = 2$, sehingga cacah daunnya adalah $n^{\log_2 1} = n^0 = 1$. Benar juga: satu rantai panggilan tunggal berakhir di satu daun tunggal.

3.12.5 Tiga bentuk pohon

Segala yang di atas mengerucut menjadi tiga kasus, yang diperlihatkan dalam **Tabel 3.6**.

Tabel 3.6 Tiga bentuk yang dapat diambil sebuah pohon rekursi, dan apa yang diakibatkan masing-masingnya. Ketiganya berpadanan persis dengan Kasus 3, 2, dan 1 Teorema Master.

Jenis	Rasio antartingkat	Dikuasai oleh	Hasil
Menurun	rasio <1	akarnya (tingkat teratas)	$\Theta(f(n))$
Setara	rasio $=1$	semua tingkat sama rata	$\Theta(f(n)\log n)$
Menaik	rasio >1	daunnya (tingkat terbawah)	$\Theta(n^{\log_b a})$

Kode 3.6 membangun ketiga bentuk itu secara numerik dan mencetak biaya tiap tingkatnya, sehingga rasionya dapat dibaca langsung.

Kode 3.6 Menjumlahkan pohon rekursi tingkat demi tingkat untuk ketiga bentuk pada **Tabel 3.6**, sekaligus memeriksa kesamaan cacah daun.

```
// tree.cpp - Menjumlahkan pohon rekursi tingkat demi tingkat, tiga bentuk berbeda.
// Kompilasi: g++ -std=c++17 -O2 -o tree tree.cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
using namespace std;

// Bentangkan pohon yang simpul berukuran s-nya berbiaya cost(s) dan menumbuhkan anak
// berukuran menurut 'shrink'. Cetak total biaya tiap tingkatnya.
void levels(const char* title, double n, const vector<double>& shrink,
            double (*cost)(double), int maxLevel) {
    cout << title << "\n";
    vector<double> sizes = {n};
    double running = 0.0;
    for (int L = 0; L <= maxLevel && !sizes.empty(); ++L) {
        double levelCost = 0.0;
        for (double s : sizes) levelCost += cost(s);
        running += levelCost;
        cout << "  level " << setw(2) << L
             << "  nodes = " << setw(6) << sizes.size()
             << "  level cost = " << setw(14) << fixed << setprecision(2) << levelCost
             << "  running total = " << setw(14) << running << "\n";
        vector<double> next;
        for (double s : sizes)
            for (double r : shrink)
                if (s * r >= 1.0) next.push_back(s * r);
        sizes.swap(next);
    }
    cout << "\n";
}

static double square(double s) { return s * s; }
static double linear(double s) { return s; }

int main() {
    // Bentuk A: rasio < 1, akarnya yang berkuasa.
    levels("T(n) = T(n/4) + T(n/2) + n^2,  n = 1024  (level ratio 5/16 < 1)",
           1024, {0.25, 0.5}, square, 6);

    // Bentuk B: rasio = 1, tiap tingkat berbiaya sama.
    levels("T(n) = T(n/3) + T(2n/3) + n,   n = 1024  (level ratio 1)",
           1024, {1.0/3.0, 2.0/3.0}, linear, 6);
```

Kode 3.6 (Lanjutan) *Menjumlahkan pohon rekursi tingkat demi tingkat untuk ketiga bentuk pada Tabel 3.6, sekaligus memeriksa kesamaan cacah daun.*

```
    // Bentuk C: rasio > 1, daunnya yang berkuasa.
    levels("T(n) = 4T(n/2) + n,            n = 1024  (level ratio 2 > 1)",
           1024, {0.5, 0.5, 0.5, 0.5}, linear, 6);

    // Kesamaan cacah daun a^(log_b n) = n^(log_b a), diperiksa secara numerik.
    cout << "Leaf-count identity  a^(log_b n) = n^(log_b a):\n";
    for (auto ab : vector<pair<double,double>>{{2,2},{4,2},{3,4},{9,3},{7,2}}) {
        double a = ab.first, b = ab.second, n = 4096;
        double lhs = pow(a, log(n)/log(b));
        double rhs = pow(n, log(a)/log(b));
        cout << "  a = " << a << ", b = " << b << ", n = 4096 : "
             << scientific << setprecision(6) << lhs << "  vs  " << rhs << "\n";
    }
    return 0;
}
```

Keluarannya — program itu membentangkan tujuh tingkat dari tiap pohon, yang sudah cukup untuk memperlihatkan rasionya:

```
T(n) = T(n/4) + T(n/2) + n^2,  n = 1024  (level ratio 5/16 < 1)
  level  0  nodes =      1  level cost =     1048576.00  running total =     1048576.00
  level  1  nodes =      2  level cost =      327680.00  running total =     1376256.00
  level  2  nodes =      4  level cost =      102400.00  running total =     1478656.00
  level  3  nodes =      8  level cost =       32000.00  running total =     1510656.00
  level  4  nodes =     16  level cost =       10000.00  running total =     1520656.00
  level  5  nodes =     32  level cost =        3125.00  running total =     1523781.00
  level  6  nodes =     57  level cost =         975.00  running total =     1524756.00

T(n) = T(n/3) + T(2n/3) + n,   n = 1024  (level ratio 1)
  level  0  nodes =      1  level cost =        1024.00  running total =        1024.00
  level  1  nodes =      2  level cost =        1024.00  running total =        2048.00
  level  2  nodes =      4  level cost =        1024.00  running total =        3072.00
  level  3  nodes =      8  level cost =        1024.00  running total =        4096.00
  level  4  nodes =     16  level cost =        1024.00  running total =        5120.00
  level  5  nodes =     32  level cost =        1024.00  running total =        6144.00
  level  6  nodes =     64  level cost =        1024.00  running total =        7168.00

T(n) = 4T(n/2) + n,            n = 1024  (level ratio 2 > 1)
  level  0  nodes =      1  level cost =        1024.00  running total =        1024.00
  level  1  nodes =      4  level cost =        2048.00  running total =        3072.00
  level  2  nodes =     16  level cost =        4096.00  running total =        7168.00
  level  3  nodes =     64  level cost =        8192.00  running total =       15360.00
  level  4  nodes =    256  level cost =       16384.00  running total =       31744.00
  level  5  nodes =   1024  level cost =       32768.00  running total =       64512.00
  level  6  nodes =   4096  level cost =       65536.00  running total =      130048.00

Leaf-count identity  a^(log_b n) = n^(log_b a):
  a = 2.00, b = 2.00, n = 4096 : 4.096000e+03  vs  4.096000e+03
  a = 4.000000e+00, b = 2.000000e+00, n = 4096 : 1.677722e+07  vs  1.677722e+07
  a = 3.000000e+00, b = 4.000000e+00, n = 4096 : 7.290000e+02  vs  7.290000e+02
  a = 9.000000e+00, b = 3.000000e+00, n = 4096 : 1.677722e+07  vs  1.677722e+07
  a = 7.000000e+00, b = 2.000000e+00, n = 4096 : 1.384129e+10  vs  1.384129e+10
```

Setiap klaim di bagian ini terlihat pada angka-angka itu. Pada pohon pertama, $327680/1048576 = 0.3125 = \frac{5}{16}$ tepat, dan total berjalannya merangkak menuju $\frac{16}{11} \cdot 1048576 \approx 1525201$ — ia sudah mencapai 1524756 pada tingkat 6. Pada pohon kedua, tiap tingkat berbiaya tepat 1024. Pada pohon ketiga, tiap tingkat berbiaya dua kali tingkat di atasnya, sehingga total berjalannya dikuasai oleh tingkat *terakhir*, bukan yang pertama. Dan kesamaan cacah daunnya berlaku sampai enam angka berarti pada kelima kasus ujinya.

3.13 Metode 3: Teorema Master

Pohon rekursi di §3.12.4 selalu berbentuk sama ketika rekurensinya berbentuk seimbang $T(n) = a\,T(n/b) + f(n)$. Karena bentuknya selalu sama, jawabannya dapat dikerjakan **sekali**, secara umum, lalu tinggal dilihat di tabel. Tabel lihat-cepat itulah Teorema Master.

3.13.1 Pernyataannya

> **TEOREMA MASTER.** Misalkan $a \geq 1$ dan $b > 1$ adalah konstanta, misalkan $f(n)$ adalah fungsi yang positif secara asimtotik, dan misalkan $T(n)$ didefinisikan pada bilangan bulat tak negatif oleh

$$T(n) = a\,T(n/b) + f(n),$$

dengan n/b ditafsirkan sebagai $\lfloor n/b \rfloor$ atau $\lceil n/b \rceil$. Maka $T(n)$ memiliki batas-batas asimtotik berikut.

Kasus 1. Jika $f(n) = O(n^{\log_b a - \varepsilon})$ untuk suatu konstanta $\varepsilon > 0$, maka $T(n) = \Theta(n^{\log_b a})$.

Kasus 2. Jika $f(n) = \Theta(n^{\log_b a})$, maka $T(n) = \Theta(n^{\log_b a} \lg n)$.

Kasus 3. Jika $f(n) = \Omega(n^{\log_b a+\varepsilon})$ untuk suatu konstanta $\varepsilon > 0$, **dan** jika $a\,f(n/b) \leq c\,f(n)$ untuk suatu konstanta $c < 1$ dan semua n yang cukup besar (*syarat keteraturan*), maka $T(n) = \Theta(f(n))$.

> Itu paragraf yang padat. Sisa bagian ini membongkar setiap kepingnya.

3.13.2 Apa arti tiap kepingnya, dalam kata-kata

- a — **faktor percabangan.** Berapa banyak panggilan rekursif yang dibuat tiap pemanggilan. Merge Sort membuat dua, jadi $a = 2$.
- b — **faktor penyusutan.** Tiap panggilan memperoleh masukan yang b kali lebih kecil. Merge Sort membagi dua, jadi $b = 2$.
- $f(n)$ — **kerja setempat.** Segala yang dilakukan sebuah pemanggilan selain panggilan rekursifnya: membelah, menggabungkan, memadukan. Langkah penggabungan Merge Sort adalah $\Theta(n)$, jadi $f(n) = \Theta(n)$.
- $n^{\log_b a}$ — **besaran kritisnya, sekaligus cacah daunnya.** §3.12.4 membuktikan lewat Persamaan (3.12) bahwa pohonnya punya $a^{\log_b n} = n^{\log_b a}$ daun; tiap daun adalah kasus basis berbiaya $\Theta(1)$; jadi $n^{\log_b a}$ adalah *total biaya semua daunnya*.

> **SATU GAGASANNYA.** Teorema Master membandingkan dua besaran: biaya di **akar**, yaitu $f(n)$, terhadap biaya di **daun**, yaitu $n^{\log_b a}$. Siapa pun yang lebih besar secara polinomial menang dan menjadi jawabannya. Jika keduanya seri, tidak ada yang menang dan Anda membayar seluruh $\lg n$ tingkatnya.

> Itulah seluruh teoremanya. Ketiga kasusnya adalah: daun menang, seri, akar menang.

3.13.3 Peran ε: mengapa harus "secara polinomial"

> Kasus 1 dan Kasus 3 tidak mengatakan bahwa $f(n)$ *lebih kecil* atau *lebih besar* daripada $n^{\log_b a}$. Keduanya mengatakan ia **lebih kecil atau lebih besar secara polinomial**, dan lambang ε — sebuah konstanta positif kecil, dibaca "epsilon" — adalah cara hal itu dituliskan.
>
> $f(n) = O(n^{\log_b a-\varepsilon})$ berarti: ada suatu besaran positif tetap ε sedemikian sehingga f bertahan di bawah n yang dipangkatkan dengan bilangan yang utuh ε *lebih kecil* daripada pangkat kritisnya. Lebih kecil sebesar faktor $n^{0.0001}$ itu terhitung. Lebih kecil hanya sebesar faktor $\lg n$ **tidak** terhitung, sebab §3.8.6 sudah menunjukkan $\lg n = o(n^{\varepsilon})$ untuk setiap $\varepsilon > 0$ — sebuah logaritma lebih kecil daripada setiap pangkat positif dari n, sehingga ia tidak pernah dapat menyediakan jarak yang dituntut ε.
>
> Untuk menemukan ε dalam praktik, tuliskan $f(n)$ dan $n^{\log_b a}$ sama-sama sebagai pangkat dari n lalu kurangkan pangkatnya.

- Kasus 1 menuntut: (pangkat dari f) $< \log_b a$. Maka $\varepsilon = \log_b a -$ (pangkat dari f).
- Kasus 3 menuntut: (pangkat dari f) $> \log_b a$. Maka $\varepsilon =$ (pangkat dari f) $- \log_b a$.

3.13.4 Syarat keteraturan, dalam bahasa sederhana

> Kasus 3 membawa satu hipotesis tambahan:

$$a\,f\left(\frac{n}{b}\right) \leq c\,f(n) \qquad \text{untuk suatu konstanta } c < 1 \text{ dan semua nilai besar } n.$$

> (3.14)

Apa yang dikatakan Persamaan (3.14)**.** Ruas kirinya adalah total kerja setempat yang dilakukan oleh a anak sebuah simpul. Ruas kanannya adalah suatu pecahan dari kerja setempat yang dilakukan simpul itu sendiri. Jadi syarat itu berkata: **kerja anak-anaknya, seluruhnya disatukan, paling banyak sebesar suatu pecahan tetap dari kerja induknya.**

Mengapa ia dibutuhkan. Jika syarat itu berlaku, maka tingkat 1 berbiaya paling banyak c kali tingkat 0, tingkat 2 paling banyak c^2 kali tingkat 0, dan seterusnya. Biaya antartingkatnya membentuk deret geometri berasio paling banyak $c < 1$, yang berjumlah paling banyak $\frac{1}{1-c} f(n) = \Theta(f(n))$. Tanpa keteraturan, biaya tiap tingkatnya mungkin tidak menyusut secara geometri, dan akarnya mungkin akhirnya tidak berkuasa.

Kapan ia berlaku. Kapan pun $f(n) = n^k$ dengan $k > \log_b a$ — yaitu, untuk setiap f polinomial pada Kasus 3 — sebab kemudian

$$af(n/b) = a\left(\frac{n}{b}\right)^k = \frac{a}{b^k}\, n^k = \frac{a}{b^k}\, f(n),$$

dan $k > \log_b a$ berarti $b^k > b^{\log_b a} = a$, sehingga $c = a/b^k < 1$. **Untuk f yang polinomial, syarat itu otomatis**, dan itulah sebabnya soal-soal buku teks hampir tidak pernah melihatnya gagal.

Kapan ia gagal. Ia gagal untuk f yang berayun liar. Contoh penyangkal bakunya adalah

$$T(n) = T(n/2) + n\,(2 - \cos n).$$

Di sini $a = 1$, $b = 2$, $n^{\log_2 1} = n^0 = 1$, dan $f(n) = n(2 - \cos n)$ jelas $\Omega(n^{0+1})$, sebab $2 - \cos n \geq 1$ selalu. Jadi syarat *polinomial* Kasus 3 berlaku. Tetapi keteraturan menuntut $f(n/2) \leq c\, f(n)$, yaitu

$$\frac{n}{2}\left(2 - \cos\frac{n}{2}\right) \leq c\, n\,(2 - \cos n).$$

Pilihlah n yang membuat $\cos n \approx 1$ dan $\cos(n/2) \approx -1$ — n semacam itu ada, sebesar apa pun yang dikehendaki. Maka ruas kirinya kira-kira $\frac{n}{2} \cdot 3 = 1.5n$ dan ruas kanannya kira-kira $c \cdot n \cdot 1 = cn$, dan $1.5n \leq cn$ memaksa $c \geq 1.5$. Tidak ada konstanta $c < 1$ yang berhasil. Teorema Master tidak berlaku.

3.13.5 Kartu ringkasannya

Tabel 3.7 adalah kartu yang harus dihafal.

Tabel 3.7 Tiga kasus Teorema Master untuk $T(n) = a\,T(n/b) + f(n)$, beserta gagasan yang diungkapkan masing-masingnya.

Kasus	Syarat atas $f(n)$	Kesimpulan	Gagasannya
1	$f(n) = O(n^{\log_b a - \varepsilon})$ untuk suatu $\varepsilon > 0$	$T(n) = \Theta(n^{\log_b a})$	daun berkuasa
2	$f(n) = \Theta(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \lg n)$	tiap tingkat berbiaya sama
3	$f(n) = \Omega(n^{\log_b a + \varepsilon})$ untuk suatu $\varepsilon > 0$, **dan** $af(n/b) \leq cf(n)$ dengan $c < 1$	$T(n) = \Theta(f(n))$	akar berkuasa

3.13.6 Di mana teorema itu menolak menjawab: celah-celahnya

Teorema Master adalah fungsi parsial. Ia tidak mencakup setiap rekurensi berbentuk $T(n) = aT(n/b) + f(n)$, dan penting untuk tahu persis di mana ia membisu.

Celah antara Kasus 1 dan Kasus 2. $f(n)$ lebih kecil daripada $n^{\log_b a}$, tetapi tidak lebih kecil *secara polinomial*. Contoh: $f(n) = n^{\log_b a}/\lg n$. Konkretnya,

$$T(n) = 2T(n/2) + \frac{n}{\lg n}.$$

Di sini $n^{\log_2 2} = n$ dan $f(n) = n/\lg n$. Ia bukan $\Theta(n)$, jadi Kasus 2 gagal. Apakah ia $O(n^{1-\varepsilon})$? Itu akan menuntut $\frac{n/\lg n}{n^{1-\varepsilon}} = \frac{n^{\varepsilon}}{\lg n}$ tetap terbatas, padahal besaran itu menuju tak hingga untuk setiap $\varepsilon > 0$. Jadi Kasus 1 pun gagal. Jawaban sejatinya, yang dapat diperoleh lewat pohon rekursi, adalah $\Theta(n \lg\lg n)$.

Celah antara Kasus 2 dan Kasus 3. $f(n)$ lebih besar daripada $n^{\log_b a}$, tetapi tidak lebih besar *secara polinomial*. Inilah contoh celah yang klasik, dan yang paling sering dijawab keliru:

$$T(n) = 2T(n/2) + n \lg n.$$

Di sini $a = 2$, $b = 2$, $n^{\log_2 2} = n^1 = n$, dan $f(n) = n \lg n$.

Apakah ini Kasus 2? Kita akan memerlukan $n \lg n = \Theta(n)$. Tetapi $\frac{n \lg n}{n} = \lg n \to \infty$, jadi tidak.

Apakah ini Kasus 3? Kita akan memerlukan $n \lg n = \Omega(n^{1+\varepsilon})$ untuk suatu $\varepsilon > 0$, yaitu $n \lg n \geq c\, n^{1+\varepsilon}$, yaitu $\lg n \geq c\, n^{\varepsilon}$. Tetapi $\lg n$ tumbuh lebih lambat daripada n^{ε} untuk setiap $\varepsilon > 0$. Jadi tidak.

> **JEBAKAN YANG SERING TERJADI.** $T(n) = 2T(n/2) + n \lg n$ jatuh di celah antara Kasus 2 dan Kasus 3, dan Teorema Master **tidak dapat** menyelesaikannya. Menjawab "Kasus 3, jadi $\Theta(n \lg n)$" salah dua kali: kasusnya tidak berlaku, dan jawabannya keliru. Jawaban yang benar, dari sebuah pohon rekursi, adalah $\Theta(n \lg^2 n)$ — yaitu, $\Theta(n (\lg n)^2)$.

Mengapa jawaban yang benar adalah $\Theta(n \lg^2 n)$. Tingkat i punya 2^i simpul berukuran $n/2^i$, masing-masing berbiaya $\frac{n}{2^i} \lg \frac{n}{2^i}$, sehingga tingkat i berbiaya $n \lg(n/2^i) = n(\lg n - i)$. Menjumlahkan sepanjang $\lg n$ tingkatnya:

$$T(n) = \sum_{i=0}^{\lg n - 1} n\,(\lg n - i) = n \sum_{j=1}^{\lg n} j = n \cdot \frac{\lg n(\lg n + 1)}{2} = \Theta(n \lg^2 n). \tag{3.15}$$

(Langkah tengahnya mengganti nama $j = \lg n - i$: ketika i berjalan dari 0 naik sampai $\lg n - 1$, j berjalan dari $\lg n$ turun sampai 1.) Persamaan (3.15) adalah deret aritmetika dari §3.9.1, kembali bekerja dengan berguna.

Alasan-alasan lain teorema itu tidak berlaku.

- $f(n)$ **tidak positif secara asimtotik.** $T(n) = 4T(n/2) - n^2$ berada di luar jangkauan teoremanya.
- a **atau** b **bukan konstanta.** $T(n) = 2^n T(n/2) + n^n$ punya faktor percabangan yang bergantung pada n.

- **Subpersoalannya berukuran berbeda.** $T(n) = T(n/3) + T(2n/3) + n$ bukan berbentuk $aT(n/b)$; pakailah pohon rekursi (§3.12.3).
- **Rekurensinya mengurangi alih-alih membagi.** $T(n) = T(n-1) + n$ bukan berbentuk $aT(n/b)$; selesaikan dengan teleskoping, yang memberi $\Theta(n^2)$.
- $a < 1$**.** Anda tidak mungkin punya submasalah kurang dari satu.

3.13.7 Strategi lima langkah

1. **Tarik** a, b, dan $f(n)$ dari rekurensinya. Periksa $a \geq 1$, $b > 1$, dan f positif secara asimtotik.
2. **Hitung** pangkat kritisnya $\log_b a$, lalu tuliskan $n^{\log_b a}$.
3. **Bandingkan** $f(n)$ dengan $n^{\log_b a}$ dengan membandingkan pangkatnya.
4. **Putuskan** kasusnya — atau kenali bahwa itu sebuah celah.
5. **Jika Kasus 3, periksa keteraturannya** sebelum menuliskan jawaban.

3.14 Dua Puluh Soal Teorema Master, Dikerjakan Tuntas

Semua yang di bawah ini mengikuti lima langkah §3.13.7. Hitungannya diperlihatkan satu baris satu langkah; tidak ada yang ditinggalkan untuk diisi sendiri oleh pembaca.

3.14.1 Pencarian biner: $T(n) = T(n/2) + 1$

Pencarian biner membagi dua selang pencariannya lalu berekursi pada satu paruh, sambil melakukan sejumlah konstanta kerja pembandingan pada tiap langkah. **Pseudokode 3.1** memperlihatkan algoritmanya dalam pseudokode. Uji pertamanya mengakhiri pencarian yang gagal begitu irisan $A[p..r]$ kosong, dengan mengembalikan NIL.

Pseudokode 3.1 Pencarian biner rekursif untuk s pada irisan terurut $A[p..r]$. Satu panggilan rekursif ($a = 1$) pada selang berukuran separuh ($b = 2$), ditambah kerja $\Theta(1)$ per panggilan, memberi $T(n) = T(n/2) + \Theta(1)$.

```
BINARY-SEARCH(A, p, r, s)
    if p > r then
        return NIL
    q = floor((p + r) / 2)
    if A[q] == s then
        return q
    else if A[q] > s then
        return BINARY-SEARCH(A, p, q-1, s)
    else
        return BINARY-SEARCH(A, q+1, r, s)
```

Langkah 1. $a = 1, b = 2, f(n) = 1$.

Langkah 2. $\log_b a = \log_2 1 = 0$, karena $2^0 = 1$. Jadi $n^{\log_b a} = n^0 = 1$.

Langkah 3. $f(n) = 1$ dan $n^{\log_b a} = 1$. Keduanya sama, jadi $f(n) = \Theta(n^{\log_b a})$.

Langkah 4. **Kasus 2.**

Langkah 5. Tidak diperlukan.

$$T(n) = \Theta(n^0 \lg n) = \Theta(\lg n). \qquad \checkmark$$

Uji kewarasan. Membagi dua n berulang-ulang sampai mencapai 1 memakan $\lg n$ langkah, dan tiap langkah berbiaya konstanta. Benar.

3.14.2 $T(n) = 9T(n/3) + n$

Langkah 1. $a = 9, b = 3, f(n) = n$.

Langkah 2. $\log_3 9 = 2$, karena $3^2 = 9$. Jadi $n^{\log_b a} = n^2$.

Langkah 3. $f(n) = n = n^1$. Pangkat 1 lebih kecil daripada 2. Ambil $\varepsilon = 2 - 1 = 1$; maka $f(n) = n = O(n^{2-1})$. ✓

Langkah 4. **Kasus 1.**

$$T(n) = \Theta(n^2). \qquad \checkmark$$

Mengapa. Daunnya berbiaya n^2 dan akarnya hanya n; dasar pohonnya menenggelamkan segalanya.

3.14.3 Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$

Langkah 1. $a = 2, b = 2, f(n) = \Theta(n)$.

Langkah 2. $\log_2 2 = 1$. Jadi $n^{\log_b a} = n^1 = n$.

Langkah 3. $f(n) = \Theta(n) = \Theta(n^{\log_b a})$. Seri persis.

Langkah 4. **Kasus 2.**

$$T(n) = \Theta(n\lg n). \qquad \checkmark$$

3.14.4 $T(n) = 3T(n/4) + n\lg n$

Langkah 1. $a = 3, b = 4, f(n) = n\lg n$.

Langkah 2. $\log_4 3 = \frac{\ln 3}{\ln 4} = \frac{1.0986}{1.3863} \approx 0.7925$. Jadi $n^{\log_b a} \approx n^{0.7925}$.

Langkah 3. Apakah $f(n) = \Omega(n^{0.7925+\varepsilon})$? Coba $\varepsilon = 0.2$, sehingga sasarannya $n^{0.9925}$. Maka

$$\frac{n\lg n}{n^{0.9925}} = n^{0.0075}\lg n \to \infty,$$

jadi $n\lg n$ memang $\Omega(n^{0.9925})$. ✓

Langkah 4. **Calon Kasus 3.**

Langkah 5 — keteraturan. Kita memerlukan $af(n/b) \le cf(n)$ dengan $c < 1$:

$$3f\left(\frac{n}{4}\right) = 3 \cdot \frac{n}{4}\lg\frac{n}{4} = \frac{3n}{4}(\lg n - 2) \le \frac{3n}{4}\lg n = \frac{3}{4}f(n).$$

Jadi $c = 3/4 < 1$. ✓

$$T(n) = \Theta(n\lg n). \qquad \checkmark$$

3.14.5 $T(n) = 4T(n/2) + n^3$

Langkah 1. $a = 4, b = 2, f(n) = n^3$.

Langkah 2. $\log_2 4 = 2$. Jadi $n^{\log_b a} = n^2$.

Langkah 3. Pangkat f adalah $3 > 2$. Ambil $\varepsilon = 1$: $f(n) = n^3 = \Omega(n^{2+1})$. ✓

Langkah 4. **Calon Kasus 3.**

Langkah 5 — keteraturan.

$$4f\left(\frac{n}{2}\right) = 4\left(\frac{n}{2}\right)^3 = 4 \cdot \frac{n^3}{8} = \frac{n^3}{2} = \frac{1}{2} f(n),$$

jadi $c = 1/2 < 1$. ✓

$$T(n) = \Theta(n^3). \qquad ✓$$

3.14.6 Celahnya: $T(n) = 2T(n/2) + n\lg n$

Dikerjakan tuntas di §3.13.6. $a = 2$, $b = 2$, $n^{\log_2 2} = n$, $f(n) = n\lg n$. Baik Kasus 2 maupun Kasus 3 tidak berlaku: $n\lg n$ lebih besar daripada n tetapi tidak lebih besar *secara polinomial*.

Teorema Master tidak berlaku. Dengan pohon rekursi: $T(n) = \Theta(n\lg^2 n)$.

3.14.7 $T(n) = 4T(n/2) + n$

Langkah 1. $a = 4$, $b = 2$, $f(n) = n$.

Langkah 2. $\log_2 4 = 2$, jadi $n^{\log_b a} = n^2$.

Langkah 3. Pangkat f adalah $1 < 2$; ambil $\varepsilon = 1$, yang memberi $f(n) = O(n^{2-1})$. ✓

Langkah 4. **Kasus 1.**

$$T(n) = \Theta(n^2). \qquad ✓$$

Inilah rekurensi yang kita buktikan lewat substitusi di §3.11.3, dan **Kode 3.5** memperlihatkan $T(n) = 2n^2 - n$ secara eksak. Teorema Master memberikan jawaban yang sama dalam empat baris, bukan satu halaman.

3.14.8 $T(n) = 4T(n/2) + n^2$

Langkah 1. $a = 4$, $b = 2$, $f(n) = n^2$. *Langkah 2.* $n^{\log_2 4} = n^2$.

Langkah 3. $f(n) = n^2 = \Theta(n^2)$. Seri. *Langkah 4.* **Kasus 2.**

$$T(n) = \Theta(n^2 \lg n). \qquad ✓$$

Bandingkan dengan §3.14.7 dan §3.14.5: a dan b yang sama, tiga f yang berbeda, tiga kasus yang berbeda. Trio ini layak dihafal sebagai satu kesatuan.

3.14.9 $T(n) = 8T(n/2) + n^2$

Langkah 1. $a = 8$, $b = 2$, $f(n) = n^2$.

Langkah 2. $\log_2 8 = 3$, jadi $n^{\log_b a} = n^3$.

Langkah 3. Pangkat $2 < 3$; ambil $\varepsilon = 1$. **Kasus 1.**

$$T(n) = \Theta(n^3). \qquad \checkmark$$

Inilah perkalian matriks bagi dan kuasai yang naif: delapan perkalian rekursif atas matriks berukuran separuh, ditambah $\Theta(n^2)$ penjumlahan. Ia tidak lebih baik daripada gelung rangkap tiga ala buku sekolah.

3.14.10 Strassen: $T(n) = 7T(n/2) + n^2$

Langkah 1. $a = 7, b = 2, f(n) = n^2$.

Langkah 2. $\log_2 7 = \frac{\ln 7}{\ln 2} = \frac{1.9459}{0.6931} \approx 2.807$. Jadi $n^{\log_b a} \approx n^{2.807}$.

Langkah 3. Pangkat f adalah $2 < 2.807$. Ambil $\varepsilon = 0.8$, sehingga sasarannya $n^{2.007}$, dan $n^2 = O(n^{2.007})$. ✓

Langkah 4. **Kasus 1.**

$$T(n) = \Theta(n^{\lg 7}) \approx \Theta(n^{2.807}). \qquad \checkmark$$

Wawasan Strassen adalah mengganti delapan perkalian dengan tujuh perkalian dengan biaya lebih banyak penjumlahan — dan karena $f(n)$ sudah berada di Kasus 1, penjumlahan tambahan itu tidak berbiaya apa pun secara asimtotik, sedangkan a yang berkurang memangkas pangkatnya dari 3 menjadi 2.807.

3.14.11 $T(n) = 8T(n/2) + n^3$

Langkah 1. $a = 8, b = 2, f(n) = n^3$. *Langkah 2.* $n^{\log_2 8} = n^3$.

Langkah 3. $f(n) = n^3 = \Theta(n^3)$. Seri. *Langkah 4.* **Kasus 2.**

$$T(n) = \Theta(n^3 \lg n). \qquad \checkmark$$

3.14.12 $T(n) = 2T(n/4) + \sqrt{n}$

Langkah 1. $a = 2, b = 4, f(n) = \sqrt{n} = n^{1/2}$.

Langkah 2. $\log_4 2 = 1/2$, karena $4^{1/2} = 2$. Jadi $n^{\log_b a} = n^{1/2} = \sqrt{n}$.

Langkah 3. $f(n) = \sqrt{n} = \Theta(\sqrt{n})$. Seri. *Langkah 4.* **Kasus 2.**

$$T(n) = \Theta(\sqrt{n} \lg n). \qquad \checkmark$$

3.14.13 $T(n) = 3T(n/3) + n/2$

Langkah 1. $a = 3, b = 3, f(n) = n/2$.

Langkah 2. $\log_3 3 = 1$, jadi $n^{\log_b a} = n$.

Langkah 3. $f(n) = n/2 = \Theta(n)$ — konstanta $1/2$ terserap. Seri. *Langkah 4.* **Kasus 2.**

$$T(n) = \Theta(n\lg n). \qquad \checkmark$$

Pelajarannya. Faktor konstanta di dalam $f(n)$ tidak pernah mengubah kasusnya. Jangan biarkan $1/2$ itu mengalihkan perhatian Anda.

3.14.14 $T(n) = 16T(n/4) + n^2$

Langkah 1. $a = 16, b = 4, f(n) = n^2$.

Langkah 2. $\log_4 16 = 2$, karena $4^2 = 16$. Jadi $n^{\log_b a} = n^2$.

Langkah 3. Seri. *Langkah 4.* **Kasus 2.**

$$T(n) = \Theta(n^2 \lg n). \qquad \checkmark$$

3.14.15 $T(n) = T(n/2) + n$

Langkah 1. $a = 1, b = 2, f(n) = n$.

Langkah 2. $\log_2 1 = 0$, jadi $n^{\log_b a} = 1$.

Langkah 3. Pangkat f adalah $1 > 0$; ambil $\varepsilon = 1$, yang memberi $f(n) = \Omega(n^{0+1})$. ✓

Langkah 4. **Calon Kasus 3.**

Langkah 5 — keteraturan. $1 \cdot f(n/2) = \frac{n}{2} = 1/2\, f(n)$, jadi $c = 1/2 < 1$. ✓

$$T(n) = \Theta(n). \qquad \checkmark$$

Uji kewarasan. Kerjanya adalah $n + \frac{n}{2} + \frac{n}{4} + \cdots < 2n$. Benar.

3.14.16 Celah yang satu lagi: $T(n) = 2T(n/2) + n/\lg n$

Langkah 1. $a = 2, b = 2, f(n) = n/\lg n$. *Langkah 2.* $n^{\log_2 2} = n$.

Langkah 3. Apakah $f = \Theta(n)$? Tidak: $f(n)/n = 1/\lg n \to 0$. Apakah $f = O(n^{1-\varepsilon})$? Itu menuntut $\frac{n/\lg n}{n^{1-\varepsilon}} = \frac{n^{\varepsilon}}{\lg n}$ terbatas, padahal ia divergen untuk setiap $\varepsilon > 0$.

Celah antara Kasus 1 dan 2 — teoremanya tidak berlaku. Jawaban sebe

3.14.17 $T(n) = 5T(n/2) + n^2$

Langkah 1. $a = 5, b = 2, f(n) = n^2$.

Langkah 2. $\log_2 5 = \frac{\ln 5}{\ln 2} = \frac{1.6094}{0.6931} \approx 2.3219$.

Langkah 3. Pangkat $2 < 2.3219$; ambil $\varepsilon = 0.3$, sasarannya $n^{2.0219}$, dan $n^2 = O(n^{2.0219})$. ✓
Langkah 4. **Kasus 1.**

$$T(n) = \Theta(n^{\lg 5}) \approx \Theta(n^{2.322}). \qquad \checkmark$$

3.14.18 $T(n) = T(2n/3) + 1$

Langkah 1. $a = 1, b = 3/2, f(n) = 1$.

Langkah 2. $\log_{3/2} 1 = 0$, jadi $n^{\log_b a} = 1$.

Langkah 3. $f(n) = 1 = \Theta(1)$. Seri. *Langkah 4.* **Kasus 2.**

$$T(n) = \Theta(\lg n). \qquad \checkmark$$

Catatan. b tidak harus bilangan bulat. Menyusut dengan faktor konstanta berapa pun yang lebih besar daripada 1 memberi tingkat sebanyak logaritmik.

3.14.19 Keteraturan yang benar-benar bekerja: $T(n) = 2T(n/2) + n^2/\lg n$

Langkah 1. $a = 2, b = 2, f(n) = n^2/\lg n$. *Langkah 2.* $n^{\log_2 2} = n$.

Langkah 3. Apakah $f = \Omega(n^{1+\varepsilon})$? Dengan $\varepsilon = 1/2$:

$$\frac{n^2/\lg n}{n^{1.5}} = \frac{n^{0.5}}{\lg n} \to \infty. \checkmark$$

Langkah 4. **Calon Kasus 3.**

Langkah 5 — keteraturan, dihitung dengan saksama.

$$2f\left(\frac{n}{2}\right) = 2 \cdot \frac{(n/2)^2}{\lg(n/2)} = 2 \cdot \frac{n^2/4}{\lg n - 1} = \frac{n^2}{2(\lg n - 1)}.$$

Bagilah dengan $f(n) = n^2/\lg n$:

$$\frac{2f(n/2)}{f(n)} = \frac{n^2}{2(\lg n - 1)} \cdot \frac{\lg n}{n^2} = \frac{\lg n}{2(\lg n - 1)} \to \frac{1}{2}.$$

Untuk $n \geq 16$ kita punya $\lg n \geq 4$, sehingga $\frac{\lg n}{2(\lg n - 1)} \leq \frac{4}{2 \cdot 3} = 2/3 < 1$. Ambil $c = 2/3$. ✓

$$T(n) = \Theta\left(\frac{n^2}{\lg n}\right). \qquad \checkmark$$

Pelajarannya. Ketika f bukan pangkat murni dari n, keteraturan adalah perhitungan yang sungguhan, bukan formalitas.

3.14.20 Keteraturan yang gagal: $T(n) = T(n/2) + n(2 - \cos n)$

Dikerjakan di §3.13.4. Bagian polinomial Kasus 3 berlaku, tetapi tidak ada konstanta $c < 1$ yang memenuhi $f(n/2) \leq cf(n)$, karena f berayun. **Teorema Master tidak berlaku**, walaupun rekurensinya berbentuk persis seperti yang dituntut. Inilah contoh baku yang memperlihatkan bahwa syarat keteraturan bukan syarat yang berlebihan.

3.14.21 Memeriksa semuanya dengan mesin

Kode 3.7 mengimplementasikan strategi lima langkah itu secara langsung: ia menerima a, b, dan sebuah f berbentuk $n^p(\lg n)^q$, menghitung $\log_b a$, membandingkan p dengannya, menamai kasusnya, lalu memeriksa keteraturannya.

Kode 3.7 Sebuah penyelesai Teorema Master dalam C++, diterapkan pada delapan belas rekurensi.

```
// master.cpp - Penyelesai Teorema Master untuk T(n) = a T(n/b) + f(n)
// dengan f(n) = n^p (lg n)^q. Ia menghitung pangkat kritis log_b(a),
// membandingkannya dengan p, menamai kasusnya, memeriksa keteraturan, lalu mencetak
Theta(.).
// Kompilasi: g++ -std=c++17 -O2 -o master master.cpp
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
#include <vector>
#include <cmath>
using namespace std;

struct Rec { double a, b, p; int q; string name; };

// Cetak pangkat real secara ringkas: "2" bila ia bilangan bulat, "2.807" bila tidak.
static string expo(double e) {
    ostringstream os;
    if (fabs(e - llround(e)) < 1e-9) os << llround(e);
    else os << fixed << setprecision(3) << e;
    return os.str();
}

// Gambarkan n^e, dengan n^0 diciutkan menjadi 1 dan n^1 menjadi n.
static string npow(double e) {
    if (fabs(e) < 1e-9) return "1";
    if (fabs(e - 1.0) < 1e-9) return "n";
    return "n^" + expo(e);
}

static string lgpow(int q) {
    if (q == 0) return "";
    if (q == 1) return " lg n";
    return " lg^" + to_string(q) + " n";
}

int main() {
    vector<Rec> rs = {
        {1, 2,   0, 0, "T(n) = T(n/2) + 1"},
        {9, 3,   1, 0, "T(n) = 9T(n/3) + n"},
        {2, 2,   1, 0, "T(n) = 2T(n/2) + n"},
        {3, 4,   1, 1, "T(n) = 3T(n/4) + n lg n"},
        {4, 2,   3, 0, "T(n) = 4T(n/2) + n^3"},
        {2, 2,   1, 1, "T(n) = 2T(n/2) + n lg n"},
        {4, 2,   1, 0, "T(n) = 4T(n/2) + n"},
        {4, 2,   2, 0, "T(n) = 4T(n/2) + n^2"},
        {8, 2,   2, 0, "T(n) = 8T(n/2) + n^2"},
```

Kode 3.7 (Lanjutan) *Sebuah penyelesai Teorema Master dalam C++, diterapkan pada delapan belas rekurensi.*

```
        {7, 2,   2, 0, "T(n) = 7T(n/2) + n^2"},
        {8, 2,   3, 0, "T(n) = 8T(n/2) + n^3"},
        {2, 4, 0.5, 0, "T(n) = 2T(n/4) + sqrt(n)"},
        {3, 3,   1, 0, "T(n) = 3T(n/3) + n/2"},
        {16,4,   2, 0, "T(n) = 16T(n/4) + n^2"},
        {1, 2,   1, 0, "T(n) = T(n/2) + n"},
        {2, 2,   1,-1, "T(n) = 2T(n/2) + n/lg n"},
        {5, 2,   2, 0, "T(n) = 5T(n/2) + n^2"},
        {1, 1.5, 0, 0, "T(n) = T(2n/3) + 1"},
    };

    cout << left << setw(30) << "recurrence" << setw(10) << "log_b a"
         << setw(16) << "f(n)" << setw(10) << "case" << "T(n)\n";
    cout << string(96, '-') << "\n";

    const double EPS = 1e-9;
    for (const Rec& r : rs) {
        double crit = log(r.a) / log(r.b);
        string f = npow(r.p) + lgpow(r.q);
        if (r.q == -1) f = "n/lg n";
        string kase, ans;

        if (r.p < crit - EPS) {                    // lebih kecil secara polinomial
            kase = "1";
            ans = "Theta(" + npow(crit) + ")";
        } else if (fabs(r.p - crit) < EPS) {       // orde polinomial yang sama
            if (r.q == 0) { kase = "2"; string base = npow(crit);
                            ans = "Theta(" + (base == "1" ? "lg n" : base + " lg n") + ")";
}
            else if (r.q > 0) { kase = "gap 2-3"; ans = "Theta(" + npow(crit) + lgpow(r.q +
1) + ") by other means"; }
            else { kase = "gap <2"; ans = "not covered (answer Theta(n lg lg n))"; }
        } else {                                   // lebih besar secara polinomial
            double c = r.a / pow(r.b, r.p);        // konstanta keteraturan
            if (c < 1.0 - 1e-12) {
                kase = "3";
                ostringstream os; os << fixed << setprecision(3) << c;
                ans = "Theta(" + f + ")   [regularity c = " + os.str() + " < 1]";
            } else { kase = "3?"; ans = "regularity fails"; }
        }
        cout << left << setw(30) << r.name << setw(10) << expo(crit)
             << setw(16) << f << setw(10) << kase << ans << "\n";
    }
    return 0;
}
```

Keluarannya persis tabel jawaban yang diturunkan dengan tangan di atas:

```
recurrence                    log_b a   f(n)            case      T(n)
-----------------------------------------------------------------------------------------
-----
T(n) = T(n/2) + 1             0         1               2         Theta(lg n)
T(n) = 9T(n/3) + n            2         n               1         Theta(n^2)
T(n) = 2T(n/2) + n            1         n               2         Theta(n lg n)
T(n) = 3T(n/4) + n lg n       0.792     n lg n          3         Theta(n lg n)
[regularity c = 0.750 < 1]
T(n) = 4T(n/2) + n^3          2         n^3             3         Theta(n^3)   [regularity
c = 0.500 < 1]
T(n) = 2T(n/2) + n lg n       1         n lg n          gap 2-3   Theta(n lg^2 n) by other
means
T(n) = 4T(n/2) + n            2         n               1         Theta(n^2)
T(n) = 4T(n/2) + n^2          2         n^2             2         Theta(n^2 lg n)
T(n) = 8T(n/2) + n^2          3         n^2             1         Theta(n^3)
T(n) = 7T(n/2) + n^2          2.807     n^2             1         Theta(n^2.807)
T(n) = 8T(n/2) + n^3          3         n^3             2         Theta(n^3 lg n)
T(n) = 2T(n/4) + sqrt(n)      0.500     n^0.500         2         Theta(n^0.500 lg n)
T(n) = 3T(n/3) + n/2          1         n               2         Theta(n lg n)
T(n) = 16T(n/4) + n^2         2         n^2             2         Theta(n^2 lg n)
T(n) = T(n/2) + n             0         n               3         Theta(n)   [regularity c
= 0.500 < 1]
T(n) = 2T(n/2) + n/lg n       1         n/lg n          gap <2    not covered (answer
Theta(n lg lg n))
T(n) = 5T(n/2) + n^2          2.322     n^2             1         Theta(n^2.322)
T(n) = T(2n/3) + 1            0         1               2         Theta(lg n)
```

Setiap konstanta keteraturan yang dihitung dengan tangan muncul kembali: $3/4 = 0.750$ di §3.14.4 dan $1/2 = 0.500$ di §3.14.5 dan §3.14.15. Program itu juga menegaskan kedua celahnya, dan — ini penting — ia *tahu* bahwa keduanya celah, bukan diam-diam mengembalikan kasus yang keliru. **Tabel 3.8** mengumpulkan hasilnya.

Tabel 3.8 Kedua puluh soal terkerjakan §3.14 dalam satu pandangan, lengkap dengan kasus yang menaungi masing-masingnya.

Rekurensi	$\log_b a$	$f(n)$	Kasus	$T(n)$
$T(n) = T(n/2) + 1$	0	1	2	$\Theta(\lg n)$
$T(n) = 9T(n/3) + n$	2	n	1	$\Theta(n^2)$
$T(n) = 2T(n/2) + n$	1	n	2	$\Theta(n\lg n)$
$T(n) = 3T(n/4) + n\lg n$	0.7925	$n\lg n$	3	$\Theta(n\lg n)$
$T(n) = 4T(n/2) + n^3$	2	n^3	3	$\Theta(n^3)$
$T(n) = 2T(n/2) + n\lg n$	1	$n\lg n$	celah	$\Theta(n\lg^2 n)$
$T(n) = 4T(n/2) + n$	2	n	1	$\Theta(n^2)$
$T(n) = 4T(n/2) + n^2$	2	n^2	2	$\Theta(n^2\lg n)$
$T(n) = 8T(n/2) + n^2$	3	n^2	1	$\Theta(n^3)$
$T(n) = 7T(n/2) + n^2$	2.807	n^2	1	$\Theta(n^{\lg 7})$
$T(n) = 8T(n/2) + n^3$	3	n^3	2	$\Theta(n^3\lg n)$
$T(n) = 2T(n/4) + \sqrt{n}$	0.5	$\sqrt{n}$	2	$\Theta(\sqrt{n}\lg n)$
$T(n) = 3T(n/3) + n/2$	1	$n/2$	2	$\Theta(n\lg n)$
$T(n) = 16T(n/4) + n^2$	2	n^2	2	$\Theta(n^2\lg n)$
$T(n) = T(n/2) + n$	0	n	3	$\Theta(n)$
$T(n) = 2T(n/2) + n/\lg n$	1	$n/\lg n$	celah	$\Theta(n\lg\lg n)$
$T(n) = 5T(n/2) + n^2$	2.322	n^2	1	$\Theta(n^{\lg 5})$
$T(n) = T(2n/3) + 1$	0	1	2	$\Theta(\lg n)$
$T(n) = 2T(n/2) + n^2/\lg n$	1	$n^2/\lg n$	3	$\Theta(n^2/\lg n)$
$T(n) = T(n/2) + n(2 - \cos n)$	0	berayun	tidak ada	keteraturan gagal

3.15 Merge Sort, Diturunkan dari Nol

Kita kini punya setiap perkakas yang dijanjikan bab ini. Mari kita pakai pada sebuah algoritma sungguhan, dari gagasan pertama sampai program C++ yang berjalan beserta pembuktian bahwa ia benar dan cepat.

3.15.1 Cetak biru bagi dan kuasai

> **DEFINISI.** Sebuah algoritma **divide and conquer** (bagi dan kuasai) menyelesaikan sebuah persoalan dalam tiga langkah. **BAGI** persoalannya menjadi submasalah yang lebih kecil *dan sejenis*. **KUASAI** subpersoalannya dengan menyelesaikannya secara rekursif; jika sebuah submasalah sudah cukup kecil, selesaikan langsung. **PADUKAN** penyelesaian subpersoalannya menjadi penyelesaian bagi persoalan aslinya.

Ucapan Sun Tzu dalam *Seni Perang* (*The Art of War*) — *"mengendalikan pasukan besar sama asasnya dengan mengendalikan sedikit orang: itu semata-mata soal membagi jumlah mereka"* — adalah seluruh siasat itu dalam satu kalimat.

Terapkan pada pengurutan.

- **Bagi.** Belah array menjadi dua. Tidak ada yang pintar-pintar: potong saja di titik tengahnya. Ini berbiaya $\Theta(1)$ bila kita mengoper indeks alih-alih menyalin.

- **Kuasai.** Urutkan tiap paruhnya dengan memanggil diri kita sendiri. Array berisi satu elemen sudah terurut, jadi itulah kasus basisnya.
- **Padukan.** Ambil dua paruh yang sudah terurut lalu selang-selingkan menjadi satu keutuhan yang terurut. Inilah **penggabungan** (*merge*), dan di sinilah seluruh kerja yang sesungguhnya bersemayam.

3.15.2 Algoritmanya

Pseudokode 3.2 memberikan algoritmanya dalam pseudokode.

Pseudokode 3.2 Merge Sort pada irisan $A[p..r]$, dengan MERGE diuraikan dalam kata-kata; **Kode 3.8** memberikan kode lengkapnya.

```
MERGE-SORT(A, p, r)
    if p < r then
        q = floor((p + r) / 2)
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q+1, r)
        MERGE(A, p, q, r)

MERGE(A, p, q, r)
    Ambil yang lebih kecil di antara dua elemen terdepan
    A[p..q] dan A[q+1..r], taruh ke dalam hasil. Ulangi
    sampai keduanya kosong. Salin hasilnya kembali ke A[p..r].
```

Di sini p adalah indeks elemen pertama potongannya, r indeks elemen terakhirnya, dan q titik tengahnya. Uji `if p < r` adalah kasus basis yang sedang menyamar: ketika $p = r$ potongannya berisi satu elemen dan tidak ada yang perlu dikerjakan; ketika $p > r$ potongannya kosong.

3.15.3 Penelusuran lengkap atas $A = [1,5,2,4,6,3,2,6]$

Fase bagi (pembelahan dari atas ke bawah).

```
Tingkat 0: [1, 5, 2, 4 | 6, 3, 2, 6]
Tingkat 1: [1, 5 | 2, 4]   [6, 3 | 2, 6]
Tingkat 2: [1|5] [2|4]     [6|3] [2|6]
Tingkat 3: [1][5] [2][4]   [6][3] [2][6]      <- tunggalan: kasus-kasus basisnya
```

Fase padu (penggabungan dari bawah ke atas).

```
Tingkat 3 -> 2: [1]+[5]=[1,5]   [2]+[4]=[2,4]   [6]+[3]=[3,6]   [2]+[6]=[2,6]
Tingkat 2 -> 1: [1,5]+[2,4]=[1,2,4,5]           [3,6]+[2,6]=[2,3,6,6]
Tingkat 1 -> 0: [1,2,4,5]+[2,3,6,6] = [1, 2, 2, 3, 4, 5, 6, 6]
```

Ikutilah satu penggabungan pelan-pelan, yang terakhir. Senarai kirinya adalah [1,2,4,5] dan senarai kanannya [2,3,6,6]. Bandingkan yang terdepan: $1 < 2$, ambil 1. Sekarang [2,4,5] lawan [2,3,6,6]: yang terdepan seri pada 2, dan kita mengambil yang **kiri** (pilihan inilah yang membuat pengurutannya *stabil*). Sekarang [4,5] lawan [2,3,6,6]: $2 < 4$, ambil 2. Lalu $3 < 4$, ambil 3. Lalu $4 < 6$, ambil 4; lalu $5 < 6$, ambil 5. Senarai kirinya habis, jadi salin sisa kanannya: 6,6. Hasilnya: [1,2,2,3,4,5,6,6]. ✓

3.15.4 Membuktikan MERGE benar dengan loop invariant

Gelung penggabungan berulang kali menyalin yang lebih kecil di antara dua elemen terdepan. Mengapa itu menghasilkan array yang terurut? Inilah pembuktiannya.

Misalkan L adalah salinan paruh kiri yang terurut $A[p..q]$ dan R salinan paruh kanan yang terurut $A[q+1..r]$. Misalkan i menunjuk elemen L berikutnya yang belum terpakai, j elemen R

berikutnya yang belum terpakai, dan k posisi berikutnya yang harus diisi di A. Mula-mula $i = j = 0$ dan $k = p$.

> **INVARIAN LOOP.** Pada awal tiap iterasi gelung penggabungan: (a) subarray $A[p..k-1]$ memuat $k-p$ elemen terkecil dari $L \cup R$, dalam keadaan terurut; dan (b) $L[i]$ dan $R[j]$ adalah elemen terkecil dari L dan R yang belum disalin kembali.

Inisialisasi. Sebelum iterasi pertama, $k = p$, sehingga $A[p..k-1]$ kosong — ia memuat 0 elemen terkecil, yang terurut secara hampa, dan (a) berlaku. Karena $i = j = 0$ dan baik L maupun R terurut, $L[0]$ dan $R[0]$ adalah elemen terkecil mereka, sehingga (b) berlaku. ✓

Pemeliharaan. Andaikan invariannya berlaku di puncak sebuah iterasi. Ada dua kasus.

Jika $L[i] \leq R[j]$, maka $L[i]$ adalah elemen terkecil yang belum disalin — ia mengalahkan segala yang tersisa di L karena L terurut, dan ia mengalahkan segala yang tersisa di R karena ia $\leq R[j]$, yang merupakan yang terkecil di antara sisa itu. Baris `A[k++] = L[i++]` menaruhnya di posisi k. Sekarang $A[p..k]$ memuat $k-p+1$ elemen terkecil dalam keadaan terurut, sebab kita menambahkan sebuah elemen yang setidaknya sebesar tiap elemen yang sudah ada di sana. Menaikkan i membuat $L[i]$ menjadi elemen L terkecil berikutnya yang belum terpakai. Jadi (a) dan (b) berlaku dengan k dan i maju satu. ✓

Jika $L[i] > R[j]$, argumen yang sama berlaku dengan peran L dan R tertukar. ✓

Terminasi. Gelungnya berakhir ketika i mencapai $|L|$ atau j mencapai $|R|$ — katakanlah L yang habis. Menurut (a), $A[p..k-1]$ memuat $k-p$ elemen terkecil dalam keadaan terurut; elemen yang masih kurang adalah persis $R[j..]$, yang terurut dan semuanya setidaknya sebesar segala yang sudah ditempatkan (sebab segala yang sudah ditempatkan, pada saat ia ditempatkan, tidak lebih besar daripada elemen terdepan R waktu itu). Kedua gelung pembersihan menambahkannya dalam urutan. Karena itu array $A[p..r]$ terurut dan merupakan permutasi dari yang semula. Itulah pascakondisinya. ■

Terminasi gelungnya sendiri. Tiap iterasi menaikkan tepat satu di antara i atau j, yang keduanya terbatas di atas, sehingga gelungnya berjalan paling banyak $|L| + |R| = r - p + 1$ kali dan pasti berhenti. Digabungkan dengan kebenaran parsial, MERGE **benar secara total**, dan ia berjalan dalam waktu $\Theta(n)$ pada potongan sepanjang n, sebab ia melakukan sejumlah konstanta kerja per elemen yang disalin.

Kebenaran MERGE-SORT sendiri kemudian menyusul lewat induksi kuat atas panjang potongannya: potongan sepanjang 1 sudah terurut (kasus basis); potongan sepanjang $m > 1$ dibelah menjadi dua potongan yang benar-benar lebih pendek, masing-masing diurutkan menurut hipotesis induktif, dan MERGE memadukan dua potongan terurut menjadi satu keutuhan yang terurut. ■

3.15.5 Rekurensinya, dan tiga cara menyelesaikannya

Membagi berbiaya $\Theta(1)$; menguasai berbiaya $2T(n/2)$; memadukan berbiaya $\Theta(n)$. Totalnya:

$$T(n) = \begin{cases} \Theta(1) & \text{jika } n = 1, \\ 2\,T(n/2) + \Theta(n) & \text{jika } n > 1, \end{cases} \tag{3.16}$$

Rekurensi Persamaan (3.16) persis sama dengan **Gambar 3.10**.

Lewat pohon rekursi. Tulis biaya pemaduannya sebagai cn untuk suatu konstanta c. Akarnya berbiaya cn. Tingkat 1 punya dua simpul berukuran $n/2$, berbiaya $2 \cdot c\frac{n}{2} = cn$. Tingkat 2 punya empat simpul berukuran $n/4$, berbiaya $4 \cdot c\frac{n}{4} = cn$. **Tiap tingkat berbiaya** cn**.** Pohonnya punya $\lg n + 1$ tingkat, sebab membagi dua n mencapai 1 setelah $\lg n$ kali pembagian. Karena itu

$$T(n) = cn(\lg n + 1) = cn\lg n + cn = \Theta(n\lg n).$$

Lewat Teorema Master. $a = 2$, $b = 2$, $n^{\log_2 2} = n$, $f(n) = \Theta(n)$: Kasus 2, jadi $T(n) = \Theta(n\lg n)$ (§3.14.3).

Lewat substitusi. Tebak $T(n) \leq cn\lg n$ lalu sahkan:

$$\begin{aligned} T(n) = 2T(n/2) + n \leq 2 \cdot c\frac{n}{2}\lg\frac{n}{2} + n &= cn(\lg n - 1) + n \\ &= cn\lg n - cn + n \leq cn\lg n \text{ ketika } c \geq 1. \end{aligned}$$

Tiga metode, satu jawaban. Dan **Kode 3.5** sudah memperlihatkan bahwa penyelesaian eksaknya adalah $T(n) = n\lg n + n$ ketika $T(1) = 1$ — yang cocok persis dengan $cn\lg n + cn$ untuk $c = 1$.

3.15.6 Programnya

Kode 3.8 adalah implementasi lengkap yang dapat dikompilasi, dengan penelusuran §3.15.3 dicetak oleh programnya sendiri serta cacah pembandingan yang diukur terhadap ramalan $n\lg n$.

Kode 3.8 Merge Sort dalam C++, dengan penelusuran tercetak dan cacah pembandingan terukur.

```
// mergesort.cpp - Merge Sort selengkapnya: langkah penggabungan, pengurutan rekursif,
// penelusuran tercetak atas fase bagi dan fase padu, serta cacah
// pembandingan yang sungguh dilakukan, terhadap ramalan n lg n.
// Kompilasi: g++ -std=c++17 -O2 -o mergesort mergesort.cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <random>
#include <cmath>
using namespace std;

long long comparisons = 0;   // tiap pembandingan kunci-ke-kunci yang dilakukan
penggabungan

static string show(const vector<int>& a, int p, int r) {
    string s = "[";
    for (int i = p; i <= r; ++i) { s += to_string(a[i]); if (i < r) s += ", "; }
    return s + "]";
}

// MERGE: A[p..q] dan A[q+1..r] masing-masing terurut; sesudahnya A[p..r] terurut.
// Invarian loop: di puncak tiap iterasi, A[p..k-1] memuat k-p
// elemen terkecil dari L dan R dalam keadaan terurut, dan L[i] serta R[j] adalah
// elemen terkecil array masing-masing yang belum disalin kembali.
void merge(vector<int>& A, int p, int q, int r) {
    vector<int> L(A.begin() + p, A.begin() + q + 1);
    vector<int> R(A.begin() + q + 1, A.begin() + r + 1);
    size_t i = 0, j = 0;
    int k = p;
    while (i < L.size() && j < R.size()) {
        ++comparisons;
        if (L[i] <= R[j]) A[k++] = L[i++];   // <= menjaga pengurutannya tetap stabil
        else              A[k++] = R[j++];
    }
    while (i < L.size()) A[k++] = L[i++];
    while (j < R.size()) A[k++] = R[j++];
}
```

Kode 3.8 (Lanjutan) *Merge Sort dalam C++, dengan penelusuran tercetak dan cacah pembandingan terukur.*

```
void mergeSort(vector<int>& A, int p, int r, int depth, bool trace) {
    if (p >= r) return;                          // satu elemen: sudah terurut
    int q = p + (r - p) / 2;                     // floor((p+r)/2) tanpa limpahan
    if (trace) cout << string(2 * depth, ' ') << "divide " << show(A, p, r)
                    << " -> " << show(A, p, q) << " " << show(A, q + 1, r) << "\n";
    mergeSort(A, p, q, depth + 1, trace);
    mergeSort(A, q + 1, r, depth + 1, trace);
    merge(A, p, q, r);
    if (trace) cout << string(2 * depth, ' ') << "merge  -> " << show(A, p, r) << "\n";
}

int main() {
    vector<int> A = {1, 5, 2, 4, 6, 3, 2, 6};
    cout << "input  " << show(A, 0, (int)A.size() - 1) << "\n\n";
    mergeSort(A, 0, (int)A.size() - 1, 0, true);
    cout << "\nsorted " << show(A, 0, (int)A.size() - 1) << "\n";
    cout << "comparisons for n = 8: " << comparisons << "\n\n";

    cout << "n          comparisons      n lg n    ratio\n";
    cout << "--------------------------------------------\n";
    mt19937 rng(12345);
    for (int n : {16, 64, 256, 1024, 4096, 65536}) {
        vector<int> B(n);
        for (int& x : B) x = (int)(rng() % 1000000);
        comparisons = 0;
        mergeSort(B, 0, n - 1, 0, false);
        double nlgn = n * log2((double)n);
        cout << left << setw(11) << n << setw(17) << comparisons
             << setw(12) << fixed << setprecision(0) << nlgn
             << setprecision(3) << comparisons / nlgn << "\n";
    }
    return 0;
}
```

Program itu mencetak:

```
input  [1, 5, 2, 4, 6, 3, 2, 6]

divide [1, 5, 2, 4, 6, 3, 2, 6] -> [1, 5, 2, 4] [6, 3, 2, 6]
  divide [1, 5, 2, 4] -> [1, 5] [2, 4]
    divide [1, 5] -> [1] [5]
    merge  -> [1, 5]
    divide [2, 4] -> [2] [4]
    merge  -> [2, 4]
  merge  -> [1, 2, 4, 5]
  divide [6, 3, 2, 6] -> [6, 3] [2, 6]
    divide [6, 3] -> [6] [3]
    merge  -> [3, 6]
    divide [2, 6] -> [2] [6]
    merge  -> [2, 6]
  merge  -> [2, 3, 6, 6]
merge  -> [1, 2, 2, 3, 4, 5, 6, 6]

sorted [1, 2, 2, 3, 4, 5, 6, 6]
comparisons for n = 8: 16

n          comparisons      n lg n    ratio
--------------------------------------------
16         47               64        0.734
64         307              384       0.799
256        1725             2048      0.842
1024       8962             10240     0.875
4096       43970            49152     0.895
65536      965956           1048576   0.921
```

Penelusurannya menghasilkan kembali §3.15.3 baris demi baris, berakhir di [1,2,2,3,4,5,6,6]. Tabel di bawahnya adalah klaim asimtotik yang ditegaskan oleh pengukuran: ketika n tumbuh sebesar empat ribu kali lipat, nisbah pembandingan sesungguhnya terhadap $n\lg n$ tetap terpaku antara 0.73 dan 0.93 — tidak pernah melenceng ke mana-mana, dan itulah persis makna $\Theta(n\lg n)$. (Nisbahnya duduk di bawah 1 karena penggabungan dua senarai yang masing-masing sepanjang m selesai setelah paling banyak $2m - 1$ pembandingan, bukan $2m$: blok terakhirnya disalin tanpa dibandingkan.)

3.15.7 Berapa harga Merge Sort, dan apa yang dibelinya

Merge Sort adalah $\Theta(n\lg n)$ pada kasus **terbaik, rata-rata, dan terburuk** sama saja — ia selalu membelah tepat di tengah dan selalu menggabungkan dengan cara yang sama, jadi tidak ada masukan yang membuatnya lambat. Itu jaminan yang sungguhan, dan Insertion Sort tidak dapat menawarkannya.

Harganya adalah ruang. Langkah penggabungannya memerlukan tempat untuk menampung kedua paruhnya, jadi Merge Sort memakai $\Theta(n)$ memori tambahan dan **tidak bekerja di tempat**. Bab 5 memperkenalkan Heapsort, yang $\Theta(n\lg n)$ *dan* bekerja di tempat, serta Quicksort, yang $\Theta(n^2)$ pada kasus terburuk tetapi biasanya lebih cepat daripada keduanya dalam praktik. Merge Sort menyimpan satu keunggulan atas keduanya: ia **stabil**, artinya elemen yang sama nilainya mempertahankan urutan relatif aslinya — dan justru itulah sebabnya tanda `<=` pada pembandingan penggabungannya berarti penting, serta sebabnya Merge Sort menjadi algoritma pilihan ketika rekaman membawa lebih banyak hal daripada kunci yang Anda pakai mengurutkan.

3.16 Profitina dalam Praktik

Profitina adalah platform **AI Visibility (GEO)**: ia memindai situs web sebuah merek, menilai seberapa siap situs itu untuk *dikutip oleh mesin penjawab AI*, dan menjelaskan tiap sinyal yang diukurnya. Generative Engine Optimization adalah bagi mesin penjawab seperti halnya optimasi mesin pencari dahulu bagi daftar tautan. Segala isi bab ini menanggung beban di dalam produk itu, dan bentuk persoalannya — bukan isi perut penilaiannya yang bersifat kepemilikan — layak diuraikan.

Pemindaiannya adalah sebuah rekurensi, bukan sebuah gelung. Sebuah situs adalah pohon: halaman beranda menautkan ke halaman bagian, yang menautkan ke halaman artikel. Pemindaian situs berisi n halaman mengunjungi tiap halaman, menarik teks dan data terstrukturnya, lalu berekursi ke halaman-halaman yang ditautkannya. Jika muka jelajahnya dibelah menjadi a pecahan yang ditangani secara mandiri dan tiap pecahan b kali lebih kecil, biayanya menuruti $T(n) = a\,T(n/b) + f(n)$, dengan $f(n)$ adalah kerja per kelompok untuk mengambil dan mengurai. Pertanyaan rekayasanya — *apakah menambah pecahan sungguh membantu?* — persis pertanyaan tentang Anda sedang berada di kasus Teorema Master yang mana. Jika $f(n)$ yang berkuasa (Kasus 3), memecah tidak membeli apa pun, sebab hambatannya ada pada penguraian, bukan pada rekursinya. Jika daunnya yang berkuasa (Kasus 1), memecah adalah seluruh permainannya.

Pengurutan muncul di mana-mana, dan ia harus $n\lg n$. Papan pantau analitik historis memperlihatkan tren skor mingguan per merek atas tabel pengukuran yang terus bertambah. Menggambar sebuah tren berarti mengurutkan pengukuran menurut cap waktu; membuang halaman yang nyaris kembar berarti mengurutkan menurut hash isi lalu memindai deretan kunci

yang sama; memeringkat halaman berskor terburuk sebuah merek berarti mengurutkan menurut skor. Masing-masingnya adalah pengurutan $O(n\lg n)$ diikuti satu kali lintasan $O(n)$. Melakukan salah satunya dengan metode $O(n^2)$ akan tak kasatmata selama pengembangan, pada merek uji berisi empat puluh halaman, dan menjadi bencana di produksi pada merek berisi empat puluh ribu halaman.

Kestabilan adalah kebutuhan produk, bukan kemewahan. Ketika dua halaman berskor sama persis, papan pantau harus mendaftarnya dalam urutan yang stabil dan dapat diulang; kalau tidak, pelanggan yang menyegarkan halamannya melihat barisnya melompat-lompat dan kehilangan kepercayaan pada angkanya. Kestabilan Merge Sort — tanda `<=` dari §3.15.6 — itulah yang memberikan hal tersebut.

Penjadwal pemindaian ulangnya adalah persoalan antrean prioritas. Tiap merek dipindai ulang secara otomatis pada irama yang bergantung pada tingkat langganannya. Memilih merek mana yang dipindai berikutnya berarti "ambil waktu jatuh tempo terkecil berikutnya", yang dikerjakan oleh timbunan biner Bab 5 dalam $O(\lg n)$. Memindai daftar merek yang jatuh tempo dengan berulang kali mencari di seluruh daftar akan berbiaya $O(n)$ per pemilihan dan $O(n^2)$ per putaran.

Dan disiplin menghadapi kasus celah. Kurva biaya terukur di dunia nyata jarang mendarat tepat pada kasus buku teks. Sebuah langkah yang biayanya $n\lg n$ di dalam rekurensi yang membagi dua — wilayah celah antara Kasus 2 dan Kasus 3 — sungguh-sungguh $\Theta(n\lg^2 n)$, bukan $\Theta(n\lg n)$, dan selisihnya muncul sebagai kekeliruan perencanaan kapasitas bila Anda membulatkannya ke kasus terdekat. Mengetahui di mana teoremanya berhenti menjawab sama bergunanya dengan mengetahui apa yang dijawabnya.

> **Dua aturan ketika materi ini menyentuh produk nyata.** Pertama, Profitina memberi saran; pemilik merek yang memutuskan. Sebuah skor adalah pengukuran yang disertai penjelasan, bukan jaminan akan dikutip. Kedua, apa pun yang menyentuh data pribadi tunduk pada **UU PDP 27/2022** — persetujuan yang tegas, ekspor data, penghapusan akun sepenuhnya.

3.16.1 Studi kasus terkerjakan: Profitina Laundry

Studi kasus yang dapat dijalankan dalam buku ini selalu **Profitina Laundry**, sebuah usaha untuk keperluan pengajaran, bukan produk yang sebenarnya. Kedai itu menerima keranjang cucian sepanjang pagi, masing-masing dengan nomor tiket, waktu pengambilan yang dijanjikan, dan berat. Petugas konter memasukkannya sesuai urutan pelanggan datang, dan urutan itu bukan urutan mereka harus diselesaikan.

Dua pertanyaan yang sungguh ditanyakan pemiliknya:

1. *Dalam urutan apa kita harus menjalankan mesinnya?* — Urutkan keranjangnya menurut waktu pengambilan yang dijanjikan. Itu Merge Sort, $\Theta(n\lg n)$.
2. *Seberapa kacau urutan antrean sebagaimana dimasukkan?* — Cacahlah **inversi**: pasangan keranjang yang keranjang masuk lebih awalnya justru jatuh tempo *lebih lambat* daripada keranjang yang masuk belakangan. Antrean dengan nol inversi sudah berada dalam urutan yang benar; antrean dengan inversi maksimum $\binom{n}{2}$ persis terbalik. Angka ini memberi tahu pemiliknya seberapa mahal urutan penerimaan di konter itu bagi mereka.

Cara naif mencacah inversi adalah membandingkan setiap pasangan: $\binom{n}{2} = \Theta(n^2)$. Tetapi langkah penggabungannya sudah menemukan inversi secara cuma-cuma. Ketika penggabungan mengambil sebuah elemen dari paruh kanan, setiap elemen yang masih tersisa di paruh kiri lebih

besar daripadanya — dan masing-masingnya adalah sebuah inversi. Jadi kita menambahkan $|L| - i$ ke sebuah pencacah, dalam $O(1)$, dan memperoleh semuanya. Rekurensinya tidak berubah:

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n\lg n),$$

sehingga kita mencacah inversi secepat kita mengurutkan. **Kode 3.9** melakukan keduanya dalam satu kali lintasan.

Kode 3.9 Profitina Laundry: Merge Sort menurut waktu pengambilan yang dijanjikan, sambil mencacah inversi dalam lintasan $\Theta(n\lg n)$ yang sama.

```
// laundry.cpp - Profitina Laundry: urutkan keranjang hari itu menurut waktu
// pengambilan yang dijanjikan dengan Merge Sort, dan, dalam lintasan bagi-dan-kuasai yang
sama,
// cacah berapa pasang yang kini tidak berurutan (inversi).
// Keduanya berjalan dalam T(n) = 2T(n/2) + Theta(n) = Theta(n lg n).
// Kompilasi: g++ -std=c++17 -O2 -o laundry laundry.cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <sstream>
using namespace std;

struct Order {
    string ticket;      // mis. "PL-014"
    int    minutes;     // pengambilan yang dijanjikan, menit setelah 06:00
    double kilograms;
};

static string clockOf(int minutes) {          // 0 -> "06:00"
    int h = 6 + minutes / 60, m = minutes % 60;
    ostringstream os;
    os << setw(2) << setfill('0') << h << ':' << setw(2) << setfill('0') << m;
    return os.str();
}

// Gabungkan A[p..q] dengan A[q+1..r], keduanya sudah terurut menurut 'minutes',
// lalu kembalikan banyaknya inversi yang terhapus: pasangan (i, j) dengan i di
// paruh kiri, j di paruh kanan, dan A[i].minutes > A[j].minutes.
long long mergeCount(vector<Order>& A, int p, int q, int r) {
    vector<Order> L(A.begin() + p, A.begin() + q + 1);
    vector<Order> R(A.begin() + q + 1, A.begin() + r + 1);
    size_t i = 0, j = 0; int k = p; long long inv = 0;
    while (i < L.size() && j < R.size()) {
        if (L[i].minutes <= R[j].minutes) A[k++] = L[i++];
        else {
            // L[i..] semuanya melebihi R[j], jadi L.size()-i inversi lenyap sekaligus.
            inv += (long long)(L.size() - i);
            A[k++] = R[j++];
        }
    }
    while (i < L.size()) A[k++] = L[i++];
    while (j < R.size()) A[k++] = R[j++];
    return inv;
}
```

Kode 3.9 (Lanjutan) *Profitina Laundry: Merge Sort menurut waktu pengambilan yang dijanjikan, sambil mencacah inversi dalam lintasan \Theta(n\lg n) yang sama.*

```
long long sortCount(vector<Order>& A, int p, int r) {
    if (p >= r) return 0;
    int q = p + (r - p) / 2;
    long long inv = sortCount(A, p, q);
    inv += sortCount(A, q + 1, r);
    inv += mergeCount(A, p, q, r);
    return inv;
}

int main() {
    vector<Order> day = {
        {"PL-014", 300, 4.5}, {"PL-002",  90, 2.0}, {"PL-021", 420, 7.25},
        {"PL-008", 150, 3.0}, {"PL-017", 240, 5.5}, {"PL-005", 120, 1.5},
        {"PL-030", 480, 9.0}, {"PL-011", 210, 6.0}, {"PL-026", 360, 3.75},
        {"PL-019", 270, 2.25}, {"PL-003",  60, 8.0}, {"PL-023", 330, 5.0}
    };

    cout << "Queue as the counter entered it (" << day.size() << " baskets)\n";
    for (const Order& o : day)
        cout << "  " << o.ticket << "  due " << clockOf(o.minutes)
             << "  " << fixed << setprecision(2) << setw(5) << o.kilograms << " kg\n";

    long long inversions = sortCount(day, 0, (int)day.size() - 1);

    cout << "\nQueue after Merge Sort, earliest promise first\n";
    double total = 0.0;
    for (const Order& o : day) {
        total += o.kilograms;
        cout << "  " << o.ticket << "  due " << clockOf(o.minutes)
             << "  " << setw(5) << o.kilograms << " kg\n";
    }
    cout << "\ntotal load          : " << total << " kg\n";
    cout << "out-of-order pairs  : " << inversions
         << "  (maximum possible = " << day.size() * (day.size() - 1) / 2 << ")\n";
    return 0;
}
```

Keluarannya:

```
Queue as the counter entered it (12 baskets)
  PL-014  due 11:00   4.50 kg
  PL-002  due 07:30   2.00 kg
  PL-021  due 13:00   7.25 kg
  PL-008  due 08:30   3.00 kg
  PL-017  due 10:00   5.50 kg
  PL-005  due 08:00   1.50 kg
  PL-030  due 14:00   9.00 kg
  PL-011  due 09:30   6.00 kg
  PL-026  due 12:00   3.75 kg
  PL-019  due 10:30   2.25 kg
  PL-003  due 07:00   8.00 kg
  PL-023  due 11:30   5.00 kg

Queue after Merge Sort, earliest promise first
  PL-003  due 07:00   8.00 kg
  PL-002  due 07:30   2.00 kg
  PL-005  due 08:00   1.50 kg
  PL-008  due 08:30   3.00 kg
  PL-011  due 09:30   6.00 kg
  PL-017  due 10:00   5.50 kg
  PL-019  due 10:30   2.25 kg
  PL-014  due 11:00   4.50 kg
  PL-023  due 11:30   5.00 kg
  PL-026  due 12:00   3.75 kg
  PL-021  due 13:00   7.25 kg
  PL-030  due 14:00   9.00 kg
```

lalu ringkasannya:

```
total load          : 57.75 kg
out-of-order pairs  : 32  (maximum possible = 66)
```

Tiga puluh dua inversi dari kemungkinan enam puluh enam: urutan penerimaannya kurang lebih seacak kocokan sembarang, yang rata-rata menghasilkan 33 inversi (setiap pasangan dari 66 pasangan itu terbalik dengan peluang setengah). Pemiliknya kini punya angka yang dapat ditindaklanjuti — dan angka itu dihitung dalam lintasan $\Theta(n\lg n)$ yang sama yang menghasilkan jadwalnya, tanpa tambahan biaya asimtotik sama sekali.

3.17 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. Memakai O padahal yang dimaksud Θ. "Merge Sort adalah $O(n^2)$" adalah pernyataan yang benar sekaligus tidak berguna. O adalah batas atas, dan fungsi mana pun yang lebih longgar juga merupakan batas atas. *Perbaikan:* nyatakan batas paling ketat yang dapat Anda buktikan, dan katakan Θ bila Anda punya kedua arahnya.

2. Membaca persamaan asimtotik secara terbalik. $2n = O(n^2)$ tidak apa-apa; $O(n^2) = 2n$ omong kosong. *Perbaikan:* bacalah tanda $=$ di sini sebagai "adalah anggota dari", selalu dari kiri ke kanan.

3. Meninggalkan sisa positif dalam pembuktian substitusi. Sampai di $T(n) \le cn^2 + n$ lalu menyimpulkan "$cn^2 + n$ adalah $O(n^2)$, selesai" adalah kekeliruan tunggal yang paling sering terjadi dalam bab ini (§3.11.2). Induksinya harus menghasilkan kembali hipotesisnya *persis*. *Perbaikan:* kurangkan sebuah suku berorde lebih rendah dari tebakannya lalu coba lagi (§3.11.3).

4. Melupakan kasus basis. Langkah induktif yang bekerja untuk semua $n \ge 2$ tidak membuktikan apa-apa bila klaimnya gagal pada n yang terkecil. *Perbaikan:* periksa basisnya secara tegas; jika ia gagal pada $n = 1$, naikkan basisnya ke $n = 2$ dan $n = 3$, dan notasi asimtotik mengizinkan hal itu (§3.11.4).

5. Mengira "lebih besar" sama dengan "lebih besar secara polinomial". Inilah jebakan klasik Teorema Master. $n\lg n$ lebih besar daripada n, tetapi tidak lebih besar sebesar faktor n^{ε} untuk $\varepsilon > 0$ mana pun, sehingga $T(n) = 2T(n/2) + n\lg n$ **bukan** Kasus 3. *Perbaikan:* bandingkan pangkatnya, bukan fungsinya. Jika satu-satunya perbedaan adalah faktor logaritmik, Anda sedang berada di sebuah celah.

6. Melewatkan syarat keteraturan pada Kasus 3. Ia otomatis untuk f yang polinomial, dan itu meninabobokan orang untuk tidak pernah memeriksanya. Ia *tidak* otomatis secara umum (§3.13.4, §3.14.20). *Perbaikan:* tuliskan pemeriksaannya setiap kali; ongkosnya cuma dua baris.

7. Menerapkan Teorema Master pada rekurensi yang bukan bentuknya. $T(n) = T(n/3) + T(2n/3) + n$ punya submasalah yang tidak sama; $T(n) = T(n-1) + n$ mengurangi alih-alih membagi; $T(n) = 2^n T(n/2) + n^n$ punya a yang tidak konstan. Tidak satu pun dari ketiganya berbentuk $aT(n/b) + f(n)$ dengan konstanta $a \geq 1$ dan $b > 1$. *Perbaikan:* periksa bentuknya sebelum meraih resepnya.

8. Mengacaukan $\lg^2 n$ dengan $\lg\lg n$. $\lg^2 n = (\lg n)^2$; $\lg\lg n$ adalah logaritma dari logaritma. Pada $n = 65536$ yang pertama bernilai 256 dan yang kedua bernilai 4. *Perbaikan:* bacalah pangkat di atas itu sebagai pangkat atas keseluruhan $\lg n$.

9. Mencemaskan basis sebuah logaritma di dalam Θ. $\Theta(\log_2 n) = \Theta(\log_{10} n) = \Theta(\ln n)$, sebab penggantian basis hanya mengalikan dengan sebuah konstanta. Tetapi di dalam $n^{\log_b a}$ basisnya menentukan. *Perbaikan:* basis tidak berarti apa-apa di dalam notasi asimtotik dan selalu berarti di dalam sebuah pangkat.

10. Mengandaikan setiap dua fungsi dapat dibandingkan. Trikotomi gagal (§3.7.6). *Perbaikan:* ingatlah n lawan $n^{1+\sin n}$.

11. Percaya bahwa asimtotik meramalkan perilaku pada masukan kecil. Ia tidak. Insertion Sort mengalahkan Merge Sort di bawah kira-kira dua puluh elemen. *Perbaikan:* pakailah asimtotik untuk memilih algoritmanya dan pengukuran untuk memilih titik potongnya.

12. Membuang sebuah konstanta yang ternyata bukan konstanta. Dalam $T(n) = aT(n/b) + f(n)$, a dan b tidak boleh bergantung pada n. *Perbaikan:* jika faktor percabangannya tumbuh seiring n, yang Anda punya adalah rekurensi jenis lain.

3.18 Ringkasan Bab

1. **Kebenaran lebih dahulu.** Sebuah algoritma disebut benar jika ia berhenti dan memberikan jawaban yang tepat untuk *setiap* masukan yang sah. Kebenaran parsial ditambah terminasi sama dengan kebenaran total. Loop invariant — inisialisasi, pemeliharaan, terminasi — adalah induksi yang diterapkan pada gelung.
2. **Lima notasi.** O adalah batas atas, Ω batas bawah, Θ batas ketat, o benar-benar lebih kecil, ω benar-benar lebih besar. Masing-masingnya adalah pernyataan tentang konstanta c (atau c_1, c_2) dan ambang n_0 yang dapat Anda tuliskan. Untuk membuktikan sebuah batas, tunjukkan konstantanya; untuk menyangkalnya, tunjukkan bahwa setiap pilihan gagal.
3. **Aturan aljabarnya.** Kelima notasi itu transitif; O, Ω, Θ refleksif; hanya Θ yang setangkup; O dan Ω setangkup transpos. Jumlah mengambil yang terbesar, hasil kali saling mengalikan, konstanta lenyap. Setiap polinomial adalah Θ dari pangkat tertingginya, dan itulah *sebab* kita membuang konstanta dan suku berorde lebih rendah.

4. **Fungsi-fungsi yang sering muncul,** dalam urutan: 1, $\lg^* n$, $\lg\lg n$, $\lg n$, $\sqrt{n}$, n, $n\lg n$, n^2, n^3, 2^n, $n!$, n^n. Logaritma kalah terhadap setiap pangkat positif dari n; polinomial kalah terhadap setiap eksponensial; $\lg(n!) = \Theta(n\lg n)$.
5. **Jumlahan-jumlahannya.** Aritmetika $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$, lewat pemasangan ala Gauss. Geometri $\sum_{i=0}^{n} a^i = \frac{1-a^{n+1}}{1-a}$, dikuasai suku pertamanya ketika $a < 1$, suku terakhirnya ketika $a > 1$, dan bernilai $n + 1$ ketika $a = 1$. Harmonik $H_n = \Theta(\lg n)$. Teleskopis saling menghapus. Ketika semua cara gagal, batasi dengan integral.
6. **Tiga metode untuk rekurensi.** Substitusi membuktikan; pohon rekursi menyarankan; metode master menuntaskan. Pakailah pohonnya untuk menebak, substitusi untuk membuktikan, dan Teorema Master kapan pun bentuknya pas.
7. **Teorema Master.** Untuk $T(n) = aT(n/b) + f(n)$, bandingkan $f(n)$ dengan cacah daun $n^{\log_b a}$. Lebih kecil secara polinomial $\Rightarrow \Theta(n^{\log_b a})$; sama $\Rightarrow \Theta(n^{\log_b a}\lg n)$; lebih besar secara polinomial *dan* teratur $\Rightarrow \Theta(f(n))$. Ada dua celah, dan teoremanya membisu di dalamnya.
8. **Merge Sort.** Bagi di titik tengah, kuasai secara rekursif, padukan lewat penggabungan. $T(n) = 2T(n/2) + \Theta(n) = \Theta(n\lg n)$ pada setiap kasus, dibuktikan dengan tiga cara. Benar lewat loop invariant, stabil, tetapi tidak bekerja di tempat.

Tabel 3.9 adalah rujukan satu halaman untuk seluruh bab ini, dan **Tabel 3.10** mengikat algoritma-algoritma Bab 1–3 dengan rekurensinya masing-masing.

Tabel 3.9 Segala isi Bab 3, dalam satu halaman.

Topik	Perkakas kunci	Kegunaan	Kapan dipakai	Contoh
Kebenaran	loop invariant	membuktikan algoritmanya bekerja	algoritma iteratif mana pun	Insertion Sort, MERGE
Batas atas	Big-O	jaminan kasus terburuk	membatasi sebuah waktu jalan	$7n - 3 = O(n)$
Batas bawah	Big-Ω	kasus terbaik, batas persoalan	menunjukkan sebuah batasan	pencarian tak terurut adalah $\Omega(n)$
Batas ketat	Big-Θ	laju pertumbuhan eksak	analisis yang presisi	$\frac{n(n+1)}{2} = \Theta(n^2)$
Rekurensi (membuktikan)	substitusi	pembuktian yang ketat	Anda sudah punya tebakan	$4T(n/2) + n$
Rekurensi (menjajaki)	pohon rekursi	penggambaran yang kasatmata	rekurensi baru, butuh tebakan	$T(n/3) + T(2n/3) + n$
Rekurensi (seketika)	Teorema Master	jawaban langsung	bentuk $aT(n/b) + f(n)$	Merge Sort, pencarian biner

Tabel 3.10 Algoritma-algoritma yang sudah dijumpai sejauh ini, rekurensinya, dan cara masing-masing diselesaikan.

Algoritma	Rekurensi	Penyelesaian
Merge Sort	$T(n) = 2T(n/2) + \Theta(n)$	Master Kasus 2: $\Theta(n\lg n)$
Pencarian biner	$T(n) = T(n/2) + \Theta(1)$	Master Kasus 2: $\Theta(\lg n)$
Insertion Sort	tidak rekursif — pakai jumlahan	$\sum_{j=2}^{n}(j-1) = \Theta(n^2)$ kasus terburuk
Perkalian matriks naif (bagi dan kuasai)	$T(n) = 8T(n/2) + \Theta(n^2)$	Master Kasus 1: $\Theta(n^3)$
Strassen	$T(n) = 7T(n/2) + \Theta(n^2)$	Master Kasus 1: $\Theta(n^{\lg 7})$
Fibonacci rekursif (Bab 1)	$T(n) = T(n-1) + T(n-2) + \Theta(1)$	bukan bentuk master: $\Theta(\phi^n)$ — pakai memoisasi

3.18.1 Menengok ke depan

Bab 4, *Evaluasi dan Pembahasan I*, mengajukan soal-soal tentang Bab 1–3 dan mengerjakan semuanya secara tuntas. Bab 5 kembali ke pengurutan dengan **Heapsort**, yang mencapai $\Theta(n\lg n)$ *di tempat*, dan **Quicksort**, yang kasus terburuknya $\Theta(n^2)$ tetapi kasus rata-ratanya $\Theta(n\lg n)$ dengan konstanta yang begitu kecil sehingga ia biasanya menang dalam praktik. Kedua analisisnya adalah rekurensi, dan Anda kini punya segala yang diperlukan untuk menyelesaikannya.

3.19 Soal Latihan

3.1 Buktikan dari definisinya bahwa $10n + 5 = O(n)$. Berikan c dan n_0 secara tegas, lalu carilah bilangan bulat c terkecil yang berhasil dengan $n_0 = 1$.

3.2 Buktikan bahwa $3n^2 + 2n = \Theta(n^2)$, dengan memberikan c_1, c_2, dan n_0.

3.3 Buktikan bahwa $2^n \neq O(n^2)$.

3.4 Urutkan menurut pertumbuhan, yang paling lambat lebih dahulu: $\lg^2 n$, $n^{1/3}$, $2^{\lg n}$, $n\lg n$, $4^{\lg n}$, n^3, 2^n, $n!$.

3.5 Benar atau salah, masing-masing dengan pembenaran satu baris: (a) $n^2 = O(n^3)$; (b) $n^3 = O(n^2)$; (c) $2^{n+1} = O(2^n)$; (d) $2^{2n} = O(2^n)$; (e) $\lg(n^2) = \Theta(\lg n)$.

3.6 Buktikan $\sum_{i=1}^{n}(2i-1) = n^2$ dua kali: sekali dengan rumus deret aritmetika, sekali dengan induksi.

3.7 Hitunglah $\sum_{i=0}^{n} 3^i$ dalam bentuk tertutup dan berikan kelas Θ-nya.

3.8 Selesaikan $T(n) = 2T(n/2) + n^3$ dengan Teorema Master, termasuk pemeriksaan keteraturannya.

3.9 Selesaikan $T(n) = T(9n/10) + n$.

3.10 Selesaikan $T(n) = 16T(n/4) + n$.

3.11 Selesaikan $T(n) = 7T(n/3) + n^2$, termasuk keteraturannya.

3.12 Selesaikan $T(n) = 2T(n/4) + n^{0.51}$, termasuk keteraturannya.

3.13 Selesaikan $T(n) = T(n/2) + n/\lg n$. (Hati-hati: ini *bukan* kasus celah §3.14.16.)

3.14 Selesaikan $T(n) = T(n-1) + n$ dengan $T(1) = 1$. Mengapa Teorema Master tidak dapat dipakai?

3.15 Pakailah pohon rekursi untuk menyelesaikan $T(n) = 3T(n/2) + n$, lalu tegaskan dengan Teorema Master.

3.16 Buktikan $T(n) = 2T(n/2) + 1$ adalah $O(n)$ lewat substitusi. Tebakan yang jelas akan gagal; perkuatlah.

3.17 Menggabungkan dua senarai terurut yang masing-masing berisi m elemen: berapa cacah pembandingan terkecil yang dapat dilakukan penggabungan pada **Kode 3.8**, dan berapa yang terbesar? Berikan masukan yang mencapai masing-masingnya.

3.20 Penyelesaian Soal Latihan

3.1 Untuk $n \geq 1$ kita punya $5 \leq 5n$, sehingga $10n + 5 \leq 10n + 5n = 15n$. Ambil $c = 15$, $n_0 = 1$. Untuk c terkecil dengan $n_0 = 1$: pada $n = 1$ ruas kirinya adalah 15 dan kita memerlukan $15 \leq c \cdot 1$, jadi $c \geq 15$; dan $c = 15$ berhasil untuk semua $n \geq 1$ seperti yang sudah ditunjukkan. c **terkecilnya adalah** 15. (Dengan ambang yang lebih besar, c yang lebih kecil berhasil: $c = 11$ menuntut $5 \leq n$, jadi $n_0 = 5$.)

3.2 *Atas:* untuk $n \geq 1$, $2n \leq 2n^2$, sehingga $3n^2 + 2n \leq 5n^2$; ambil $c_2 = 5$. *Bawah:* $3n^2 + 2n \geq 3n^2$ untuk semua $n \geq 0$; ambil $c_1 = 3$. Keduanya berlaku mulai $n_0 = 1$. Karena itu $3n^2 + 2n = \Theta(n^2)$ dengan $c_1 = 3, c_2 = 5, n_0 = 1$.

3.3 Andaikan $2^n \leq cn^2$ untuk semua $n \geq n_0$. Maka $\frac{2^n}{n^2} \leq c$ untuk semua n semacam itu. Tetapi $\frac{2^n}{n^2} \to \infty$ menurut §3.8.5 (setiap eksponensial mengalahkan setiap polinomial), sehingga ia pada akhirnya melampaui c — sebuah kontradiksi. Konkretnya, pada $n = 30$, $2^{30} = 1073741824$ sedangkan $30^2 = 900$, jadi nisbahnya sudah melampaui 1.19×10^6. Karena itu $2^n \neq O(n^2)$. ■

3.4 Sederhanakan dahulu: $2^{\lg n} = n$, dan $4^{\lg n} = (2^2)^{\lg n} = 2^{2\lg n} = n^2$. Urutannya, yang paling lambat lebih dahulu:

$$\lg^2 n \ < \ n^{1/3} \ < \ 2^{\lg n} = n \ < \ n\lg n \ < \ 4^{\lg n} = n^2 \ < \ n^3 \ < \ 2^n \ < \ n!.$$

Pertidaksamaan pertama berlaku karena $\lg^2 n = o(n^\varepsilon)$ untuk setiap $\varepsilon > 0$, khususnya $\varepsilon = 1/3$.

3.5 (a) **Benar**: $n^2 \leq n^3$ untuk $n \geq 1$, jadi $c = 1, n_0 = 1$. (b) **Salah**: $n^3/n^2 = n \to \infty$, jadi tidak ada konstanta yang membatasinya (argumen yang sama dengan §3.3.3). (c) **Benar**: $2^{n+1} = 2 \cdot 2^n$, jadi $c = 2, n_0 = 1$. (d) **Salah**: $2^{2n} = (2^n)^2$, jadi $2^{2n}/2^n = 2^n \to \infty$. (e) **Benar**: $\lg(n^2) = 2\lg n$, dan faktor konstanta diserap oleh Θ.

3.6 *Lewat rumus:* $\sum_{i=1}^{n}(2i-1) = 2\sum_{i=1}^{n} i - \sum_{i=1}^{n} 1 = 2 \cdot \frac{n(n+1)}{2} - n = n^2 + n - n = n^2$. *Lewat induksi:* Basis $n = 1$: jumlahannya adalah $2(1) - 1 = 1 = 1^2$. ✓ Langkah: andaikan $\sum_{i=1}^{n-1}(2i - 1) = (n-1)^2$. Maka $\sum_{i=1}^{n}(2i-1) = (n-1)^2 + (2n-1) = n^2 - 2n + 1 + 2n - 1 = n^2$. ✓ ■

3.7 Geometri dengan $a = 3 \neq 1$: $\sum_{i=0}^{n} 3^i = \frac{1-3^{n+1}}{1-3} = \frac{3^{n+1}-1}{2}$. Karena $a > 1$ suku terakhirnya berkuasa, jadi ini $\Theta(3^n)$. Periksa pada $n = 3$: $1 + 3 + 9 + 27 = 40$ dan $\frac{81-1}{2} = 40$. ✓

3.8 $a = 2, b = 2, f(n) = n^3$. $\log_2 2 = 1$, jadi $n^{\log_b a} = n$. Pangkat $3 > 1$; ambil $\varepsilon = 2$, yang memberi $f(n) = \Omega(n^{1+2})$. ✓ Keteraturan: $2f(n/2) = 2(n/2)^3 = 2 \cdot \frac{n^3}{8} = \frac{n^3}{4} = \frac{1}{4}f(n)$, jadi $c = 1/4 < 1$. ✓ **Kasus 3:** $T(n) = \Theta(n^3)$**.**

3.9 $a = 1$, $b = \frac{10}{9}$, $f(n) = n$. $\log_{10/9} 1 = 0$, jadi $n^{\log_b a} = 1$. Pangkat $1 > 0$; ambil $\varepsilon = 1$. ✓ Keteraturan: $1 \cdot f(9n/10) = \frac{9n}{10} = 0.9\, f(n)$, jadi $c = 0.9 < 1$. ✓ **Kasus 3:** $T(n) = \Theta(n)$**.** (Mengikis masukannya sekadar sepersepuluh tiap kali tetap memberi total kerja yang linear — deret geometrinya $n(1 + 0.9 + 0.81 + \cdots) = 10n$.)

3.10 $a = 16$, $b = 4$, $f(n) = n$. $\log_4 16 = 2$, jadi $n^{\log_b a} = n^2$. Pangkat $1 < 2$; ambil $\varepsilon = 1$, yang memberi $f(n) = O(n^{2-1})$. ✓ **Kasus 1:** $T(n) = \Theta(n^2)$**.**

3.11 $a = 7$, $b = 3$, $f(n) = n^2$. $\log_3 7 = \frac{\ln 7}{\ln 3} = \frac{1.9459}{1.0986} \approx 1.771$. Pangkat $2 > 1.771$; ambil $\varepsilon = 0.2$, sasarannya $n^{1.971}$, dan $n^2 = \Omega(n^{1.971})$. ✓ Keteraturan: $7(n/3)^2 = \frac{7}{9}n^2 \approx 0.778\, f(n)$, jadi $c = 7/9 < 1$. ✓ **Kasus 3:** $T(n) = \Theta(n^2)$**.**

3.12 $a = 2$, $b = 4$, $f(n) = n^{0.51}$. $\log_4 2 = 0.5$, jadi $n^{\log_b a} = n^{0.5}$. Pangkat $0.51 > 0.5$; ambil $\varepsilon = 0.01$. ✓ Keteraturan: $2(n/4)^{0.51} = \frac{2}{4^{0.51}} n^{0.51}$, dan $4^{0.51} = e^{0.51 \ln 4} = e^{0.7070} \approx 2.028$, jadi $c \approx \frac{2}{2.028} \approx 0.986 < 1$. ✓ **Kasus 3:** $T(n) = \Theta(n^{0.51})$**.** Perhatikan betapa sempit marginnya — ε boleh sekecil yang Anda mau, tetapi ia harus berupa konstanta positif yang tetap.

3.13 $a = 1, b = 2, f(n) = n/\lg n$. $\log_2 1 = 0$, jadi $n^{\log_b a} = 1$. Apakah $f(n) = \Omega(n^{0+\varepsilon})$? Dengan $\varepsilon = 1/2$: $\frac{n/\lg n}{n^{0.5}} = \frac{n^{0.5}}{\lg n} \to \infty$. ✓ Keteraturan: $f(n/2) = \frac{n/2}{\lg n - 1}$, dan $\frac{f(n/2)}{f(n)} = \frac{n/2}{\lg n - 1} \cdot \frac{\lg n}{n} = \frac{\lg n}{2(\lg n - 1)} \to 1/2$; untuk $n \geq 16$ ia paling banyak $2/3$. ✓ **Kasus 3:** $T(n) = \Theta(n/\lg n)$**.** Perbandingannya dengan §3.14.16 itulah intinya: di sana $a = 2$ membuat cacah daunnya n, yang tak dapat dikalahkan $n/\lg n$; di sini $a = 1$ membuat cacah daunnya 1, yang mudah saja dikalahkannya.

3.14 Bukan bentuk master: argumennya $n - 1$, bukan n/b, jadi tidak ada konstanta $b > 1$. Selesaikan dengan teleskoping. Menjumlahkan $T(i) - T(i-1) = i$ untuk $i = 2, \ldots, n$ memberi $T(n) - T(1) = \sum_{i=2}^{n} i$, sehingga $T(n) = 1 + \left(\frac{n(n+1)}{2} - 1\right) = \frac{n(n+1)}{2} = \Theta(n^2)$.

3.15 *Pohon.* Tingkat 0 berbiaya n. Tingkat 1 punya 3 simpul berukuran $n/2$, berbiaya $3 \cdot \frac{n}{2} = \frac{3}{2}n$. Tingkat 2 punya 9 simpul berukuran $n/4$, berbiaya $9 \cdot \frac{n}{4} = \frac{9}{4}n$. Rasio antartingkatnya adalah $3/2 > 1$, jadi deretnya **menaik** dan daunnya yang berkuasa. Pohonnya punya $\lg n$ tingkat dan $3^{\lg n} = n^{\lg 3}$ daun, sehingga

$$T(n) = \Theta(n^{\lg 3}) \approx \Theta(n^{1.585}).$$

Master. $a = 3$, $b = 2$, $\log_2 3 \approx 1.585$; $f(n) = n$ berpangkat $1 < 1.585$, jadi dengan $\varepsilon = 0.5$ kita memperoleh Kasus 1 dan jawaban yang sama. ✓

3.16 Tebakan yang jelas $T(n) \leq cn$ gagal: $2 \cdot c\frac{n}{2} + 1 = cn + 1 > cn$. Perkuat menjadi $T(n) \leq cn - d$ dengan $d > 0$:

$$T(n) = 2T(n/2) + 1 \leq 2\left(c\frac{n}{2} - d\right) + 1 = cn - 2d + 1$$
$$= (cn - d) - (d - 1) \leq cn - d \quad \text{setiap kali } d \geq 1.$$

Kasus basis $n = 1$ dengan $T(1) = 1$: perlu $1 \leq c - d$, jadi $c \geq 1 + d$; dengan $d = 1$, ambil $c = 2$. Karena itu $T(n) \leq 2n - 1 = O(n)$. (Dan tepat sekali, sesungguhnya: $T(n) = 2n - 1$.)

3.17 *Terkecil:* m pembandingan. Jika setiap elemen senarai kirinya $\leq$ setiap elemen senarai kanannya — misalnya $L = [1,2,3]$, $R = [4,5,6]$ — gelungnya mengambil seluruh m elemen kiri, memakai satu pembandingan untuk masing-masingnya, lalu keluar karena L habis; sisa elemen kanannya disalin oleh gelung pembersihan tanpa pembandingan. *Terbesar:* $2m - 1$ pembandingan. Setiap iterasi gelungnya melakukan satu pembandingan dan menyalin satu elemen, dan gelung itu berhenti begitu salah satu senarai kosong. Ketika ia berhenti, senarai yang lain masih memuat sekurang-kurangnya satu elemen, sehingga gelungnya paling banyak menyalin $2m - 1$ dari $2m$ elemen dan paling banyak melakukan $2m - 1$ pembandingan. Maksimum itu tercapai ketika kedua senarai tetap tidak kosong sampai paling akhir, yang terjadi ketika elemen-elemennya berselang-seling. Sebuah masukan yang mencapainya: $L = [1,3,5]$, $R = [2,4,6]$. Pembandingannya adalah 1 lawan 2, 3 lawan 2, 3 lawan 4, 5 lawan 4, 5 lawan 6 — lima buah, yaitu $2 \cdot 3 - 1$ — lalu 6 disalin tanpa pembandingan. *Pemeriksaan kekonsistenan.* Untuk $n = 8$ seluruh pengurutannya melakukan 4 penggabungan $1 + 1$ (satu pembandingan masing-masing), 2 penggabungan $2 + 2$ (paling banyak 3 masing-masing), dan 1 penggabungan $4 + 4$ (paling banyak 7): paling banyak $4 + 6 + 7 = 17$. **Kode 3.8** melaporkan 16 untuk [1,5,2,4,6,3,2,6], satu di bawah maksimumnya — konsisten. ✓

Bacaan lanjutan

Pernyataan resmi Teorema Master, syarat keteraturannya, dan contoh penyangkal yang berayun pada §3.14.20 mengikuti [1, ch. 4]; perlakuan MERGE lewat loop invariant mengikuti [1, ch. 2]. Kesamaan-kesamaan jumlahan dan teknik pembatasan dengan integral pada §3.9.6 dibahas panjang lebar dalam [2]. Kerangka bagi dan kuasai sebagai siasat perancangan, serta penerapan pencacahan inversi pada §3.16.1, mengikuti [3, ch. 5]. Titik potong praktis antara Merge Sort dan Insertion Sort, dan argumen kestabilan pada §3.15.7, dibahas dalam [4] dan dianalisis secara tuntas dalam [5]. Sudut pandang pemrograman kompetitif tentang kapan metode $O(n^2)$ masih dapat diterima adalah [6].

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1994.

[3] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[4] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[5] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[6] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, 4th ed. Raleigh, NC, USA: Lulu Press, 2020.

Bab 4
Evaluasi dan Pembahasan I: Fondasi, Analisis, dan Problem Pertama

Bab ini menghentikan sejenak langkah maju buku ini dan memantapkan segala sesuatu yang telah dibangun oleh Bab 1 sampai Bab 3. Tidak ada teori baru di dalamnya. Sebagai gantinya, bab ini mengerjakan persoalan — empat belas jumlahnya, masing-masing diselesaikan dari nol, dengan setiap bilangan antara dituliskan. Di akhir bab Anda akan memiliki metode tujuh langkah untuk membaca persoalan algoritma apa pun, dan Anda akan melihat metode itu membawa dua persoalan *online judge* sungguhan, LDCSUMAB dan TOHU, dari halaman kosong sampai program yang diterima, lengkap dengan bukti bahwa programnya benar dan pengukuran berapa biayanya. Anda akan membuktikan sebuah *loop invariant* (invarian gelung) baris demi baris, menelusuri *insertion sort* (pengurutan penyisipan) dan pencarian biner dalam tabel, menghitung jumlah langkah yang tepat dari sebuah gelung rangkap tiga, membuktikan dan menyangkal pernyataan asimtotik dengan konstanta c dan n_0 yang eksplisit, mengurutkan tiga belas fungsi menurut laju pertumbuhannya, serta memecahkan rekurensi dengan metode substitusi, dengan pohon rekursi, dan dengan Teorema Master — termasuk satu rekurensi yang tidak mau dijawab oleh Teorema Master sehingga Anda harus menuntaskannya sendiri. Terakhir, Anda akan mengukur waktu sebuah algoritma gaya kasar (*brute force*) terhadap algoritma yang lebih baik pada mesin sungguhan, dan menyaksikan selisih antara $\Theta(n^2)$ dan $\Theta(n\lg n)$ tumbuh dari sekadar galat pembulatan menjadi faktor lebih dari seratus. Setiap program dalam bab ini lengkap, lolos kompilasi dengan `g++ -std=c++17 -O2 -Wall`, dan dikutip bersama keluaran yang benar-benar dicetaknya.

4.1 Posisi Kita Saat Ini

Tiga bab telah membekali kita dengan seperangkat alat. Ada baiknya kita mendaftar isinya sebelum mulai memakainya, sebab setiap persoalan dalam bab ini akan meraih salah satu alat tersebut, dan menyebut nama alatnya sudah separuh pekerjaan.

Dari **Bab 1** kita memperoleh gagasan bahwa algoritma adalah prosedur yang tepat dan berhingga yang mengubah masukan menjadi keluaran; dua tujuan, yaitu kebenaran dan efisiensi; lima paradigma perancangan (inkremental, *greedy*, *divide and conquer* atau bagi dan kuasai, pemrograman dinamis, dan matematis); serta kerangka lima fase *Understand, Abstract, Design, Analyze, Implement* untuk menggarap sebuah persoalan.

Dari **Bab 2** kita memperoleh model RAM, tempat setiap operasi sederhana berbiaya satu langkah; invarian gelung dengan tiga kewajibannya, yakni inisialisasi, pemeliharaan, dan terminasi; *insertion sort* beserta waktu jalannya pada kasus terbaik, kasus terburuk, dan kasus rata-rata; pencarian linear; dan pencarian biner dengan batas $O(\lg n)$-nya.

Dari **Bab 3** kita memperoleh notasi asimtotik O, Ω, Θ, o, dan ω, masing-masing didefinisikan dengan konstanta yang eksplisit; fungsi-fungsi baku (pembulatan ke bawah dan ke atas, logaritma, eksponensial, faktorial, bilangan Fibonacci); perkakas penjumlahan (deret aritmetika, geometri, dan harmonik, jumlah teleskopik, batas melalui integral); serta tiga metode untuk

rekurensi — substitusi, pohon rekursi, dan Teorema Master. Semuanya mengikuti penyajian Cormen dkk. [1], dan matematika diskret di balik penjumlahannya adalah bahan baku Rosen [2].

Bab ini memakai setiap alat itu pada persoalan nyata. Susunannya terdiri atas tiga lapis.

1. **Sebuah metode** (Subbab 4.2). Tujuh langkah yang mengubah nasihat "bacalah persoalannya dengan teliti" menjadi prosedur yang dapat Anda ikuti dengan sebatang pensil.
2. **Dua persoalan dari *online judge* SPOJ** (Subbab 4.3 dan 4.4), masing-masing dikerjakan dengan sangat rinci: pernyataan persoalannya diungkapkan ulang dengan kata-kata sederhana, ketujuh langkah diterapkan satu per satu, kasus-kasus pertamanya dihitung dengan tangan, rumusnya diturunkan, kebenaran rumusnya dibuktikan, waktu jalannya dianalisis, sebuah program C++17 yang lengkap dikompilasi dan dijalankan, sebuah penyelesaian kedua yang independen disajikan, dan kesalahan-kesalahan yang lazim dibuat orang diuraikan. Kedua pembahasan ini sengaja dibuat pelan. Jika Anda belum pernah menyelesaikan persoalan *online judge*, Anda semestinya dapat menirunya ketukan demi ketukan.
3. **Dua belas persoalan terkerjakan** (Subbab 4.5 sampai 4.9), masing-masing ringkas dan masing-masing membidik satu alat dari Bab 1 sampai Bab 3. Semuanya berbentuk sama: pernyataan persoalan, apa yang diuji, penyelesaian dengan setiap langkah hitungannya, jawaban, program bila program memang membantu, dan varian *uji diri* beserta jawabannya supaya Anda dapat memastikan bahwa Anda benar-benar paham.

Subbab 4.10 kemudian menjajarkan keempat belas persoalan itu, Subbab 4.11 memperlihatkan di mana gagasan yang sama muncul dalam sebuah produk nyata, dan bab ini ditutup dengan kesalahan yang sering terjadi, ringkasan, dua belas soal latihan, beserta penyelesaian lengkapnya.

> **CARA MEMAKAI BAB INI.** Bacalah pernyataan setiap persoalan, lalu tutuplah penyelesaiannya dengan selembar kertas dan cobalah sendiri selama sepuluh menit sebelum melanjutkan membaca. Penyelesaian yang telah Anda gumuli selama sepuluh menit akan dipahami dengan cara yang tidak pernah dicapai oleh penyelesaian yang hanya Anda baca. Varian *uji diri* disediakan supaya Anda dapat menguji pemahaman itu pada sesuatu yang belum pernah Anda lihat dikerjakan.

4.2 Cara Membaca Sebuah Persoalan Algoritma

Subbab ini adalah metode, bukan nasihat. Nasihat berbunyi "bacalah persoalannya dengan teliti". Metode memberi tahu apa yang harus Anda tuliskan di halaman, dalam urutan apa, sebelum Anda mengizinkan diri sendiri memikirkan penyelesaiannya. Kerjakan ketujuh langkah ini berurutan, setiap kali. Pada persoalan yang mudah ketujuhnya memakan empat menit. Pada persoalan yang sulit, ketujuhnya adalah beda antara menyelesaikannya dan hanya menatapnya.

Ketujuh langkah ini adalah kelima fase Bab 1 yang dibuat konkret. Langkah 1 dan 2 adalah *Understand*; Langkah 3, 4, dan 5 adalah *Abstract*; Langkah 6 adalah *Design* bersama separuh *Analyze* yang menyangkut kebenaran; Langkah 7 adalah *Implement*, didahului separuh *Analyze* yang menyangkut biaya. Subbab 4.3 dan 4.4 menerapkannya satu per satu, dengan subjudul untuk tiap langkah, supaya Anda melihat metode itu bekerja alih-alih sekadar membaca tentangnya.

4.2.1 Langkah 1 — Salin format masukan dan keluaran, apa adanya

Sebelum apa pun, salinlah ke halaman Anda, kata demi kata:

- **format masukan** — berapa baris, apa isi tiap baris, dalam urutan apa;
- **format keluaran** — berapa baris, berapa angka di belakang koma, dengan kata-kata persis apa;

- **kendala** — setiap batas pada setiap peubah;
- **contoh masukan dan contoh keluaran**, huruf demi huruf.

Salinlah sekalipun semuanya terpampang di layar di depan Anda. Tindakan menyalin itulah yang memaksa Anda membaca setiap token. Sebagian besar putusan *Wrong Answer* di sebuah *online judge* adalah salah format, bukan salah algoritma: satu baris baru yang hilang, enam angka di belakang koma padahal yang diminta sebelas, dua bilangan yang dicetak dalam urutan terbalik. Pada persoalan TOHU di Subbab 4.4, keluarannya harus memuat tepat sebelas angka di belakang koma; cetaklah sepuluh, dan algoritma yang sempurna pun dinilai salah.

4.2.2 Langkah 2 — Berilah nama pada yang dicari, dan nama pada datanya

Tulislah dua daftar pendek.

Diketahui: hal-hal yang dipasok masukan, masing-masing dengan lambangnya. *Ditanya:* hal yang dituntut keluaran, sebagai rumus atau sebagai kalimat yang mengandung lambang.

Untuk persoalan LDCSUMAB di Subbab 4.3, kedua daftar itu berbunyi:

- Diketahui: sebuah posisi k, dengan $1 \leq k \leq 10^9$, sebanyak $T \leq 10$ kali.
- Ditanya: eksponen $m < n$ sedemikian sehingga $2^m + 2^n$ adalah bilangan ke-k terkecil yang berbentuk demikian.

Sekarang Anda punya lambang untuk dihitung. Sebelum punya lambang Anda tidak bisa beraljabar, dan di aljabar itulah sebagian besar persoalan dalam bab ini terpecahkan.

4.2.3 Langkah 3 — Ubah kendala menjadi anggaran waktu

Inilah langkah yang menentukan algoritma mana yang boleh dipakai, dan ia hanya memakan sepuluh detik.

Mesin penilai modern menjalankan kira-kira 10^8 sampai 10^9 operasi sederhana per detik; patokan di bawah mengikuti praktik baku pemrograman kompetitif [3]. Batas waktu pada *judge* SPOJ [4] biasanya satu sampai tiga detik. Jadi patokannya: **program Anda boleh melakukan sekitar 10^8 operasi secara keseluruhan.** Kalikan banyaknya kasus uji dengan kerja per kasus uji, lalu bandingkan dengan 10^8.

Tabel 4.1 Anggaran kompleksitas. Bacalah batas pada n, bacalah kompleksitas yang terjangkau, lalu tolak setiap gagasan yang melampauinya sebelum Anda menulis satu baris kode pun.

Batas pada n	Yang terjangkau per kasus uji	Bentuk yang lazim
$n \leq 10$	$O(n!)$, $O(2^n \cdot n)$	gaya kasar, permutasi
$n \leq 20$	$O(2^n)$	pencacahan himpunan bagian, DP bitmask
$n \leq 500$	$O(n^3)$	Floyd–Warshall, rantai matriks
$n \leq 5000$	$O(n^2)$	DP berpasangan, tabel DP sederhana
$n \leq 10^6$	$O(n)$ atau $O(n\log n)$	satu lintasan, pengurutan, saringan, tumpukan
$n \leq 10^{12}$	$O(\sqrt{n})$	blok pembagi, pembagian coba-coba
$n \leq 10^{18}$	$O(\log n)$	pencarian biner, perpangkatan cepat
n tak terbatas	$O(1)$	sebuah bentuk tertutup

Pakailah **Tabel 4.1** tanpa ampun. Tabel itu adalah model RAM Bab 2 yang diubah menjadi alat perencanaan: hitung langkahnya, bandingkan dengan anggarannya, dan buang setiap rencana yang tidak muat. Kadang-kadang sebuah persoalan menyebutkan kompleksitas sasarannya secara terang — persoalan TOHU biasanya diajukan dengan catatan *kompleksitas waktu* $O(1)$ — dan

dengan begitu ia memberi tahu bentuk jawabannya: $O(1)$ berarti ada bentuk tertutup, dan tugas Anda adalah menemukannya.

4.2.4 Langkah 4 — Hitung lima kasus pertama dengan tangan

Bukan dengan program. Dengan tangan, di kertas, dalam sebuah tabel, dengan hitungannya diperlihatkan. Inilah langkah yang dilewati para pemula, dan inilah langkah yang memecahkan persoalan.

Untuk TOHU, tabel empat nilai pertama barisan yang tidak diketahui itu berbunyi $2/3, \frac{1}{24}, \frac{1}{60}, \frac{1}{120}$, dan siapa pun yang sudah berkenalan dengan jumlah teleskopik Bab 3 akan mengenali tiga yang terakhir sebagai $\frac{1}{2\cdot3\cdot4}, \frac{1}{3\cdot4\cdot5}, \frac{1}{4\cdot5\cdot6}$. Untuk LDCSUMAB, tabel sepuluh jawaban pertama memperlihatkan $n = 1,2,2,3,3,3,4,4,4,4$ — satu angka 1, dua angka 2, tiga angka 3, empat angka 4 — dan bilangan segitiga dari Bab 3 langsung melompat keluar dari halaman. Tidak ada wawasan yang turun dari langit. Wawasan itu keluar dari tabel berisi sepuluh bilangan.

> **WAWASAN KUNCI: TABEL ITULAH ALGORITMANYA.** Pada sebagian besar persoalan dalam bab ini, penyelesaiannya kelihatan di dalam tabel lima sampai sepuluh kasus pertama yang dihitung dengan tangan. Bangunlah tabelnya sebelum Anda berteori. Jika tabelnya memperlihatkan pola, buktikan polanya. Jika tabelnya tidak memperlihatkan pola apa pun, Anda juga sudah belajar sesuatu: persoalan ini bersifat struktural, bukan numerik, dan yang mesti Anda tengok adalah invarian, urutan, atau argumen pertukaran.

4.2.5 Langkah 5 — Sebutkan bentuknya

Ucapkan dengan lantang persoalan ini masuk keluarga yang mana. Keluarganya tidak banyak, dan persoalan-persoalan dalam bab ini hanya memakai enam di antaranya:

1. **Bentuk tertutup dari sebuah jumlah** — deret aritmetika, teleskopik, pecahan parsial. (TOHU; Kupon Laundry di Subbab 4.8.)
2. **Membalikkan rumus pencacahan** — menemukan di mana sebuah posisi jatuh bila kita tahu berapa butir yang dimuat tiap blok. (LDCSUMAB.)
3. **Gelung dengan invarian** — besaran berjalan yang maknanya sama sebelum setiap iterasi. (Aturan Horner, *insertion sort*, pencarian biner, studi kasus Profitina Laundry.)
4. **Mencacah langkah dengan penjumlahan** — gelung bersarang menjadi jumlah bersarang, dan jumlah bersarang menjadi bentuk tertutup. (Gelung rangkap tiga di Subbab 4.6.)
5. **Rekurensi** — rekurensi *divide and conquer* yang dipecahkan dengan substitusi, pohon rekursi, atau Teorema Master, serta rekurensi linear seperti Fibonacci. (Subbab 4.7 dan 4.8.)
6. **Gaya kasar, lalu yang lebih baik** — algoritma yang gamblang itu benar tetapi terlalu lambat; sebuah pengurutan atau pencarian menghapus satu faktor n. (Subbab 4.9.)

Jika dalam lima menit Anda dapat menempatkan persoalan ke dalam sebuah keluarga, Anda sudah hampir sampai, sebab keluarganya datang bersama metodenya.

4.2.6 Langkah 6 — Tuliskan invariannya, rekurensinya, atau rumusnya, lalu buktikan

Sebelum kode. Satu baris, dalam matematika, yang menyatakan apa yang benar — lalu argumen bahwa hal itu *memang* benar. Contoh dari bab ini:

- LDCSUMAB: eksponen jawaban n adalah bilangan bulat terkecil dengan $\frac{n(n+1)}{2} \geq k$.

- TOHU: $S_n = \frac{3}{4} - \frac{1}{2(n+1)(n+2)}$.
- Aturan Horner: *pada awal setiap iterasi, p memuat nilai polinomial yang dibentuk oleh koefisien-koefisien yang sudah diproses.*
- Lajur $2 \times N$: $D_N = D_{N-1} + D_{N-2}$, dengan $D_0 = D_1 = 1$.

Pola yang dibaca dari tabel adalah dugaan; langkah inilah yang mengubahnya menjadi teorema. Invarian gelung Bab 2 serta induksi dan substitusi Bab 3 adalah alatnya, dan setiap persoalan di bawah memakai salah satunya.

4.2.7 Langkah 7 — Baru sekarang, tulis programnya

Dan ketika Anda menulisnya, tulislah dengan kendala di tangan. Tiga pemeriksaan sebelum Anda mengirimkannya:

1. **Adakah yang melimpah (*overflow*)?** Sebuah `int` 32 bit menampung nilai sampai $2^{31} - 1 \approx 2.1 \times 10^9$; sebuah `long long` 64 bit menampung sampai $2^{63} - 1 \approx 9.22 \times 10^{18}$. Perkirakan nilai antara yang terbesar — bukan hanya jawabannya — sebelum memilih tipe data. Pada TOHU, $2(n+1)(n+2)$ melimpahi `int` jauh sebelum n mencapai sejuta.
2. **Apakah ketelitiannya bertahan?** Sebuah `double` membawa sekitar lima belas sampai enam belas angka berarti desimal; sebuah `float` membawa sekitar tujuh. TOHU meminta sebelas angka di belakang koma dari bilangan di bawah 1, jadi `double` selamat dan `float` tidak.
3. **Apakah pembaca masukannya sanggup mengejar?** Dengan puluhan ribu kasus uji, `cin >>` tanpa `ios::sync_with_stdio(false)` bisa beberapa kali lebih lambat daripada `scanf`. Setiap program dalam bab ini memakai `scanf` dan `printf`, yang cepat dan yang untai formatnya membuat ketelitian yang diminta menjadi eksplisit.

4.3 Persoalan 1 — LDCSUMAB: Jumlah Dua Pangkat Dua

Persoalan *judge* pertama kita adalah LDCSUMAB, dari bagian *tutorial* SPOJ [4]. Inilah yang lebih mudah di antara keduanya, dan ia dipilih karena hanya memakai tiga gagasan dari Bab 2 dan Bab 3: deret aritmetika $1 + 2 + \cdots + n$, fungsi pembulatan ke atas (*ceiling*), dan pencarian biner. **Gambar 4.1** merumuskan ulang halaman persoalannya.

LDCSUMAB — BRVT - Math Team Collection 2013 (SPOJ)

Kumpulan soal tutorial, 15861

Pernyataan

Tinjau barisan semua bilangan bulat yang dapat ditulis sebagai $2^m + 2^n$ dengan bilangan bulat $0 \leq m < n$, terurut naik. Diberikan bilangan bulat positif k, tentukan m dan n untuk suku ke-k barisan itu.

Masukan

Baris pertama berisi bilangan bulat T ($T \leq 10$), banyaknya kasus uji. Masing-masing dari T baris berikutnya berisi satu bilangan bulat k ($k \leq 10^9$).

Keluaran

Untuk setiap kasus uji, cetak kedua bilangan m dan n ($m < n$) dari suku ke-k.

Contoh

Masukan	Keluaran
3 1 3 5	0 1 1 2 1 3

Gambar 4.1 Himpunan persoalan SPOJ (*tutorial*) 15861, *LDCSUMAB — BRVT - Math Team Collection 2013*, dirumuskan ulang dari *judge*-nya: pernyataan, format masukan dan keluaran, serta uji contohnya.

4.3.1 Persoalannya dengan kata-kata kita sendiri

Ambillah setiap bilangan yang dapat ditulis sebagai jumlah dua pangkat dua yang *berbeda*, $2^m + 2^n$ dengan $0 \leq m < n$. Misalnya $2^0 + 2^1 = 3$, $2^0 + 2^2 = 5$, $2^1 + 2^2 = 6$, $2^0 + 2^3 = 9$. Tuliskan semua bilangan ini dalam urutan menaik, sehingga terbentuk daftar tak berhingga 3,5,6,9,10,12,17, ... Anda diberi sebuah posisi k di dalam daftar itu dan harus menyebutkan dua eksponen mana yang menghasilkan bilangan yang duduk di sana: cetak m, lalu n.

Masukan diawali banyaknya kasus uji T, paling banyak 10. Tiap baris dari T baris berikutnya memuat satu posisi k, paling besar 10^9. Untuk setiap posisi kita mencetak satu baris yang memuat m dan n, dipisahkan satu spasi. Dengan contoh masukan `3`, `1`, `3`, `5`, keluaran yang diharapkan adalah `0 1`, `1 2`, `1 3`: bilangan pertama dalam daftar adalah $3 = 2^0 + 2^1$, yang ketiga adalah $6 = 2^1 + 2^2$, dan yang kelima adalah $10 = 2^1 + 2^3$.

> **PERSOALAN.** Daftarkan semua bilangan bulat berbentuk $2^m + 2^n$ dengan $0 \leq m < n$ dalam urutan menaik. Diberikan sebuah posisi k ($1 \leq k \leq 10^9$), keluarkan eksponen m dan n dari bilangan ke-k dalam daftar itu. Paling banyak ada 10 pertanyaan.

4.3.2 Langkah 1 sampai 3: format, lambang, anggaran

Langkah 1 — formatnya, disalin.

- Masukan: baris 1 memuat T ($T \leq 10$). Lalu T baris, masing-masing dengan satu bilangan bulat k ($1 \leq k \leq 10^9$).
- Keluaran: untuk tiap k, satu baris `m n` — eksponen yang lebih kecil lebih dahulu.
- Contoh masukan `3 / 1 / 3 / 5`; contoh keluaran `0 1 / 1 2 / 1 3`.

Perhatikan urutan keluarannya: m dahulu, lalu n, dan $m < n$. Mencetaknya terbalik berarti *Wrong Answer* sejak baris pertama contohnya.

Langkah 2 — diketahui dan ditanya.

- Diketahui: k, sebuah posisi dalam daftar terurut, $1 \leq k \leq 10^9$.
- Ditanya: pasangan (m, n) dengan $0 \leq m < n$ sedemikian sehingga $2^m + 2^n$ adalah bilangan ke-k terkecil yang berbentuk demikian.

Langkah 3 — anggarannya. Mari kita taksir biaya gagasan-gagasan yang gamblang terhadap **Tabel 4.1**.

- *Bangun daftarnya.* Bangkitkan bilangan-bilangannya secara berurutan sampai kita punya k buah. Itu kerja $\Theta(k)$, sampai 10^9 per pertanyaan dan 10^{10} untuk sepuluh pertanyaan — seratus kali di atas anggaran 10^8, dan di samping itu membutuhkan memori bergigabita. Ditolak.
- *Susuri blok-bloknya.* Pada Langkah 4 kita akan melihat bahwa daftar ini terbagi menjadi blok-blok berukuran 1,2,3, ... Melangkah dari blok ke blok sampai mencapai posisi k memakan sekitar $\sqrt{2k}$ langkah, kira-kira 44,721 untuk $k = 10^9$, jadi sekitar 4.5×10^5 untuk sepuluh pertanyaan. Itu muat.
- *Pencarian biner atas blok-bloknya.* Sekitar $\lg 50{,}000 \approx 16$ langkah per pertanyaan. Muat dengan mudah.
- *Sebuah rumus tertutup.* $O(1)$ per pertanyaan. Yang terbaik yang mungkin.

Jadi gagasan kedua pun sudah cukup cepat; anggarannya tidak *memaksa* kita memakai rumus di sini. Kita akan tetap menurunkan rumusnya, sebab menurunkannya paling banyak mengajarkan sesuatu, lalu menyajikan pencarian biner sebagai alternatif di Subbab 4.3.8. Situasi ini lazim: anggaran memberi tahu gagasan mana yang *terlarang*, bukan gagasan mana yang terbaik.

4.3.3 Langkah 4: kasus-kasus pertama dengan tangan

Kita perlu melihat daftarnya. Satu-satunya cara jujur untuk yakin akan urutannya adalah menuliskan setiap calon lalu mengurutkannya. Dengan $n \leq 4$ ada sepuluh calon:

- $n = 1$: hanya $m = 0$, memberi $2^0 + 2^1 = 1 + 2 = 3$.
- $n = 2$: $m = 0$ memberi $1 + 4 = 5$; $m = 1$ memberi $2 + 4 = 6$.
- $n = 3$: $m = 0,1,2$ memberi $1 + 8 = 9$, $2 + 8 = 10$, $4 + 8 = 12$.
- $n = 4$: $m = 0,1,2,3$ memberi $1 + 16 = 17$, $2 + 16 = 18$, $4 + 16 = 20$, $8 + 16 = 24$.

Calon terkecil dengan $n = 5$ adalah $1 + 32 = 33$, lebih besar daripada kesepuluh bilangan di atas, sehingga kesepuluh bilangan ini tepat merupakan sepuluh bilangan pertama dalam daftar. Mengurutkannya menghasilkan **Tabel 4.2**.

Tabel 4.2 Sepuluh bilangan pertama berbentuk $2^m + 2^n$, dihitung dengan tangan. Kolom n berbunyi satu angka 1, dua angka 2, tiga angka 3, empat angka 4; di dalam tiap blok, m menghitung 0,1,2, ...

k	Nilai	m	n	Posisi di dalam blok n
1	3	0	1	ke-1 dari 1
2	5	0	2	ke-1 dari 2
3	6	1	2	ke-2 dari 2
4	9	0	3	ke-1 dari 3
5	10	1	3	ke-2 dari 3
6	12	2	3	ke-3 dari 3
7	17	0	4	ke-1 dari 4
8	18	1	4	ke-2 dari 4
9	20	2	4	ke-3 dari 4
10	24	3	4	ke-4 dari 4

Bacalah tabel itu kolom demi kolom, sebagaimana diminta Langkah 4.

- **Kolom** n berbunyi 1,2,2,3,3,3,4,4,4,4. Blok n memuat tepat n entri.
- **Kolom** m, di dalam tiap blok, berjalan $0,1,\ldots,n-1$ secara berurutan.
- **Posisi terakhir tiap blok** adalah 1,3,6,10. Inilah bilangan segitiga dari Bab 3.

Gambar 4.2 memperlihatkan pengamatan yang sama digambar sebagai tiga tabel yang bersanding: daftarnya, peta dari k ke n, dan pembalikan yang akan segera kita turunkan.

k	m	n	$2^m + 2^n$
1	0	1	3
2	0	2	5
3	1	2	6
4	0	3	9
5	1	3	10
6	2	3	12
7	0	4	17
8	1	4	18
9	2	4	20
10	3	4	24

n	k
1	**1**
2	**2**
2	**3**
3	**4**
3	**5**
3	**6**
4	**7**
4	**8**
4	**9**
4	**10**

$$k = \frac{n(n+1)}{2}$$

$$n^2 + n - 2k = 0$$

$$n = \frac{-1+\sqrt{1+8k}}{2}$$

Gambar 4.2 Sepuluh suku pertama barisan LDCSUMAB (kiri), dirangkum menjadi peta dari k ke n (tengah), dan pembalikannya $k = \frac{n(n+1)}{2}$, $n^2 + n - 2k = 0$, $n = \frac{-1+\sqrt{1+8k}}{2}$ (kanan).

4.3.4 Langkah 5 dan 6: bentuknya, dan rumusnya

Langkah 5 — bentuknya. Ini keluarga *membalikkan rumus pencacahan*. Kita tahu berapa entri yang dimuat tiap blok; kita harus menemukan blok mana yang memuat posisi k, dan di bagian mana di dalam blok itu k jatuh. Persoalan semacam ini disebut **pemeringkatan dan pembalikan peringkat** (*ranking and unranking*): *ranking* mengubah sebuah objek menjadi posisinya, *unranking* mengubah posisi kembali menjadi objeknya.

Langkah 6 — rumusnya. Tabel sepuluh baris adalah bukti petunjuk, bukan pembuktian. Sebelum membangun di atas pola itu kita harus memastikan bahwa pola itu berlanjut selamanya. Dua klaim menuntaskannya.

Klaim 1: setiap nilai dengan eksponen yang lebih besar $n+1$ melampaui setiap nilai dengan eksponen n. Nilai terbesar di blok n memakai m terbesar yang diizinkan, yaitu $n-1$:

$$2^{n-1} + 2^n = 2^{n-1} + 2 \cdot 2^{n-1} = 3 \cdot 2^{n-1}.$$

Nilai terkecil di blok $n+1$ memakai $m = 0$:

$$2^0 + 2^{n+1} = 1 + 4 \cdot 2^{n-1}.$$

Karena $3 \cdot 2^{n-1} < 4 \cdot 2^{n-1} < 1 + 4 \cdot 2^{n-1}$, setiap nilai di blok n lebih kecil daripada setiap nilai di blok $n+1$. Blok-bloknya tidak pernah berselang-seling.

Klaim 2: di dalam blok n, nilainya menaik seiring m. Untuk n yang tetap, menaikkan m sebanyak satu menaikkan 2^m dan membiarkan 2^n, sehingga $2^m + 2^n$ bertambah.

Bersama-sama, kedua klaim itu menyatakan bahwa daftarnya persis pasangan (m, n) dalam **urutan leksikografis menurut n dahulu, lalu menurut m**:

$$(0,1),\ \ (0,2), (1,2),\ \ (0,3), (1,3), (2,3),\ \ (0,4), (1,4), (2,4), (3,4),\ \ \ldots$$

Mencacah bloknya. Blok n memuat pasangan dengan $m = 0,1, \ldots, n-1$, jadi ia punya tepat n entri. Banyaknya entri di blok 1 sampai n bersama-sama karena itu adalah deret aritmetika pada Subbab 3.9.1,

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \tag{4.1}$$

yakni **bilangan segitiga** ke-n, yang akan kita tulis T_n. Ingat kembali mengapa Persamaan (4.1) berlaku: tuliskan jumlahnya maju dan mundur, $S = 1 + 2 + \cdots + n$ dan $S = n + (n-1) + \cdots + 1$, lalu jumlahkan kolom demi kolom; tiap kolom dari n kolom itu berjumlah $n + 1$, sehingga $2S = n(n+1)$. Kita juga akan membutuhkan $T_0 = 0$ dan fakta bahwa $T_n = T_{n-1} + n$.

Blok mana yang memuat posisi k? Blok 1 sampai $n - 1$ mengisi posisi 1 sampai T_{n-1}; blok n mengisi posisi $T_{n-1} + 1$ sampai T_n. Jadi posisi k terletak di blok n tepat ketika

$$T_{n-1} < k \leq T_n,$$

yang sama saja dengan mengatakan: n **adalah bilangan bulat terkecil dengan $T_n \geq k$.**

Memecahkan untuk n. Perlakukan $T_n = k$ sebagai persamaan dalam peubah real n, lalu pecahkan:

$$\frac{n(n+1)}{2} = k \ \Rightarrow\ n^2 + n = 2k \ \Rightarrow\ n^2 + n - 2k = 0.$$

Ini persamaan kuadrat $an^2 + bn + c = 0$ dengan $a = 1, b = 1, c = -2k$. Rumus kuadrat $n = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ memberi

$$n = \frac{-1 \pm \sqrt{1 - 4 \cdot 1 \cdot (-2k)}}{2} = \frac{-1 \pm \sqrt{1 + 8k}}{2}.$$

Tanda minus menghasilkan bilangan negatif, yang tak mungkin menjadi indeks blok, jadi kita memakai tanda plus. Sebutlah hasilnya $x = \frac{-1+\sqrt{1+8k}}{2}$. Fungsi $T(n) = n(n+1)/2$ menaik untuk $n \geq 0$, dan $T(x) = k$ secara tepat. Bilangan *bulat* terkecil n dengan $T_n \geq k$ karena itu adalah bilangan bulat terkecil $n \geq x$, yaitu pembulatan ke atas pada Subbab 3.8.2:

$$n = \left\lceil \frac{-1 + \sqrt{1 + 8k}}{2} \right\rceil \tag{4.2}$$

Mengapa pembulatan ke atas dan bukan ke bawah? Jika k jatuh tegas di dalam sebuah blok, x bukan bilangan bulat — untuk $k = 5$ nilainya sekitar 2.70 — sedangkan posisi 5 berada di blok 3, jadi kita harus membulatkan *ke atas*. Jika k tepat sama dengan sebuah bilangan segitiga — entri terakhir bloknya — maka x adalah bilangan bulat dan pembulatan ke atas membiarkannya apa adanya.

Di bagian mana di dalam blok? Blok n menempati posisi $T_{n-1}+1, T_{n-1}+2, \dots, T_n$, yang memuat $m = 0,1,\dots,n-1$ dalam urutan itu (Klaim 2). Posisi $T_{n-1}+1$ memuat $m=0$, posisi $T_{n-1}+2$ memuat $m=1$, dan secara umum posisi k memuat $m = k - T_{n-1} - 1$. Dengan menulis $r = T_n$ untuk posisi terakhir blok itu, sehingga $T_{n-1} = r - n$,

$$m = k - (r-n) - 1 = n + k - (r+1) \tag{4.3}$$

Mengerjakan kasus-kasusnya dengan tangan. Setiap langkah Persamaan (4.2) dan Persamaan (4.3), satu baris demi satu baris.

$k = 1.\ 1 + 8 \cdot 1 = 9.\ \sqrt{9} = 3.\ \frac{-1+3}{2} = 1.\ n = \lceil 1 \rceil = 1.\ r = \frac{1 \cdot 2}{2} = 1.\ m = 1 + 1 - (1+1) = 0.$ Jawaban `0 1` ✓

$k = 3.\ 1 + 24 = 25.\ \sqrt{25} = 5.\ \frac{-1+5}{2} = 2.\ n = 2.\ r = \frac{2 \cdot 3}{2} = 3.\ m = 2 + 3 - 4 = 1.$ Jawaban `1 2` ✓

$k = 5.\ 1 + 40 = 41.\ \sqrt{41} \approx 6.4031.\ \frac{-1+6.4031}{2} \approx 2.7016.\ n = \lceil 2.7016 \rceil = 3.\ r = \frac{3 \cdot 4}{2} = 6.\ m = 3 + 5 - 7 = 1.$ Jawaban `1 3` ✓ — dan **Tabel 4.2** menegaskan bahwa posisi 5 memuat $2^1 + 2^3 = 10$.

$k = 6$, *sebuah batas blok.* $1 + 48 = 49.\ \sqrt{49} = 7.\ \frac{-1+7}{2} = 3.\ n = 3.\ r = 6.\ m = 3 + 6 - 7 = 2.$ Jawaban `2 3`, yaitu nilai $4 + 8 = 12$ ✓.

$k = 7$, *entri pertama sebuah blok baru.* $1 + 56 = 57.\ \sqrt{57} \approx 7.5498.\ \frac{-1+7.5498}{2} \approx 3.2749.\ n = 4.\ r = \frac{4 \cdot 5}{2} = 10.\ m = 4 + 7 - 11 = 0.$ Jawaban `0 4`, yaitu nilai $1 + 16 = 17$ ✓.

$k = 10^9$, *yang terbesar yang diizinkan.* $1 + 8 \times 10^9 = 8{,}000{,}000{,}001$. Akar kuadratnya: $89{,}442^2 = 7{,}999{,}871{,}364$, menyisakan 128,637, dan $128{,}637/(2 \times 89{,}442) \approx 0.719$, jadi $\sqrt{8{,}000{,}000{,}001} \approx 89{,}442.719$. Lalu $\frac{-1+89{,}442.719}{2} \approx 44{,}720.86$, sehingga $n = 44{,}721$. Periksalah: $T_{44720} = 22{,}360 \times 44{,}721 = 999{,}961{,}560 < 10^9$ dan $T_{44721} = 44{,}721 \times 22{,}361 = 1{,}000{,}006{,}281 \geq 10^9$. ✓ Jadi $r = 1{,}000{,}006{,}281$ dan $m = 44{,}721 + 1{,}000{,}000{,}000 - 1{,}000{,}006{,}282 = 38{,}439$. Jawaban `38439 44721`.

4.3.5 Bukti kebenaran

Kita punya sebuah rumus dan enam pemeriksaan tangan. Sekarang kita buktikan bahwa rumus itu benar untuk *setiap* k, sehingga tidak ada pemeriksaan ketujuh yang mungkin gagal.

> **TEOREMA.** Untuk setiap bilangan bulat $k \geq 1$, misalkan n adalah bilangan bulat terkecil dengan $T_n \geq k$ dan misalkan $m = k - T_{n-1} - 1$. Maka $0 \leq m < n$, dan $2^m + 2^n$ adalah bilangan bulat ke-k terkecil yang berbentuk $2^a + 2^b$ dengan $0 \leq a < b$.

Bukti. Menurut Klaim 1 dan 2, daftar terurut itu adalah pasangan-pasangan dalam urutan menurut n, lalu menurut m. Pasangan dengan eksponen paling besar $n-1$ berjumlah T_{n-1} menurut Persamaan (4.1), dan pasangan-pasangan itu menempati T_{n-1} posisi pertama; pasangan dengan eksponen paling besar n menempati T_n posisi pertama. Menurut pilihan n, $T_{n-1} < k$ (sebab bila tidak, $n-1$ akan menjadi bilangan bulat yang lebih kecil dengan $T_{n-1} \geq k$) dan $k \leq T_n$. Jadi posisi k tidak termasuk T_{n-1} posisi pertama tetapi termasuk T_n posisi pertama: eksponennya tepat n. Di dalam blok n, posisi $T_{n-1}+1, \dots, T_n$ membawa $m = 0, \dots, n-1$ dalam urutan menaik, sehingga posisi k membawa $m = k - T_{n-1} - 1$. Akhirnya, $T_{n-1} < k$ memberi $m \geq 0$, dan $k \leq T_n = T_{n-1} + n$ memberi $m \leq n - 1$. ■

Program pada Subbab 4.3.7 tidak memercayai akar kuadrat titik-mengambang begitu saja; ia menghitung terkaan pertama lalu mengoreksinya dengan dua gelung kecil. Kebenaran kedua gelung itu adalah argumen invarian gelung kecil bergaya Bab 2, dan layak dituliskan karena justru itulah yang membuat programnya aman. **Pseudokode 4.1** adalah prosedurnya.

Pseudokode 4.1 Menjawab satu pertanyaan LDCSUMAB: terkaan pertama titik-mengambang untuk bloknya, dua gelung pengoreksi, dan posisi di dalam blok.

```
LDCSUMAB-QUERY(k)
1  n = floor((-1 + sqrt(1 + 8k)) / 2)     // terkaan pertama, dari rumus kuadrat
2  while n(n+1)/2 < k                      // terkaan terlalu kecil: naik
3      n = n + 1
4  while n >= 1 and (n-1)n/2 >= k         // terkaan terlalu besar: turun
5      n = n - 1
6  r = n(n+1)/2                            // posisi terakhir blok n
7  m = n + k - (r + 1)
8  return (m, n)
```

- *Sesudah baris 2–3,* $T_n \geq k$. Gelung itu hanya berhenti ketika ujinya, $T_n < k$, bernilai salah. Ia memang berhenti, karena T_n tumbuh tanpa batas seiring bertambahnya n.
- *Baris 4–5 mempertahankan invarian* $T_n \geq k$. Invarian itu berlaku saat masuk. Badan gelungnya hanya dijalankan ketika $T_{n-1} \geq k$, dan saat itu ia mengganti n dengan $n - 1$, yang bilangan segitiganya tepat $T_{n-1} \geq k$ tadi. Jadi invariannya bertahan pada setiap iterasi. Gelungnya berakhir karena n menurun dan terbatas di bawah oleh 0.
- *Saat keluar* ujinya bernilai salah: entah $n = 0$ atau $T_{n-1} < k$. Yang pertama mustahil, sebab invariannya akan memberi $T_0 = 0 \geq k \geq 1$. Jadi $T_{n-1} < k \leq T_n$, dan n adalah bilangan bulat terkecil dengan $T_n \geq k$ — tepat n pada Teorema.
- *Baris 6–7* menghitung $m = n + k - (r + 1)$, yaitu Persamaan (4.3), yang sama dengan $k - T_{n-1} - 1$, yang kebenarannya dibuktikan oleh Teorema.

4.3.6 Kompleksitas

Setiap pertanyaan berbiaya satu akar kuadrat, sejumlah konstan operasi aritmetika, dan kedua gelung pengoreksi. Berapa kali gelung-gelung itu berjalan? Jawaban jujurnya, untuk $k \leq 10^9$, adalah *tidak pernah*, dan alasannya layak diketahui.

Pada batas blok $k = T_n = \frac{n(n+1)}{2}$ kita punya $1 + 8k = 1 + 4n(n + 1) = 4n^2 + 4n + 1 = (2n + 1)^2$ — selalu kuadrat sempurna ganjil yang eksak. Sebuah `double` menyimpan setiap bilangan bulat sampai $2^{53} \approx 9 \times 10^{15}$ secara eksak, dan standar IEEE-754 mengharuskan `sqrt` *dibulatkan dengan benar*, yang untuk jawaban eksak yang terwakili secara eksak berarti tepat benar. Dengan $k \leq 10^9$ kita punya $1 + 8k \leq 8 \times 10^9$, nyaman di dalam jangkauan eksaknya, sehingga terkaan pertama pada baris 1 sudah benar dan tak satu pun badan gelung dijalankan. Untuk masukan yang jauh melampaui batas persoalan ini, terkaannya bisa meleset satu, dan saat itu tiap gelung berjalan sekali. Bagaimanapun juga kerja per pertanyaan adalah $O(1)$, dan seluruh program berwaktu $\Theta(T)$ dengan ruang tambahan $O(1)$.

Lalu mengapa gelung-gelung itu dipertahankan? Karena argumen di atas bergantung pada $k \leq 10^9$, dan kendala bisa berubah. Andai versi berikutnya persoalan ini mengizinkan $k \leq 10^{18}$, maka $1 + 8k \approx 8 \times 10^{18}$ melampaui 2^{53}, argumen keeksakannya runtuh, dan program tanpa gelung itu akan gagal pada sebagian batas blok. Dua gelung `while` yang nyaris tak pernah dijalankan tidak memakan biaya dan menyingkirkan ketergantungan pada argumen yang harus disusun ulang oleh pembacanya.

Limpahan. n terbesar yang pernah kita temui adalah 44,721 (ditambah satu selama koreksi), sehingga $n(n+1) \approx 2.0 \times 10^9$. Itu tidak nyaman dekatnya dengan batas `int` sebesar 2,147,483,647, sehingga program ini memakai `long long` di mana-mana.

4.3.7 Langkah 7: programnya

Gambar 4.3 memperlihatkan tabel pada Subbab 4.3.3 di samping sebuah penyelesaian C++ ringkas yang mengimplementasikannya secara langsung dengan `ceil`.

k	m	n	$2^m + 2^n$
1	0	1	3
2	0	2	5
3	1	2	6
4	0	3	9
5	1	3	10
6	2	3	12
7	0	4	17
8	1	4	18
9	2	4	20
10	3	4	24

```
#include <cmath>
#include <cstdio>

int main(){
    int T, k, n, r, m;
    scanf("%d",&T);
    while(T--){
        scanf("%d",&k);
        n=(int)ceil((-1.0+sqrt(1.0+8.0*(double)k))/2.0);
        r=n*(n+1)/2;
        m=n+k-(r+1);
        printf("%d %d\n",m,n);
    }
    return 0;
}
```

Gambar 4.3 Sebuah penyelesaian LDCSUMAB yang ringkas di samping tabel yang diimplementasikannya: `n = ceil((-1+sqrt(1+8k))/2)`, `r = n(n+1)/2`, `m = n + k - (r+1)`.

Kode 4.1 adalah versi tangguh dari gagasan yang sama, baris demi baris sesuai prosedur pada **Pseudokode 4.1**. Akar kuadrat titik-mengambang dipakai hanya sebagai *terkaan awal*, lalu kedua gelung koreksi memakukan bilangan bulatnya secara eksak.

Kode 4.1 LDCSUMAB — membalikkan peringkat barisan $2^m + 2^n$, dalam $O(1)$ per pertanyaan sesudah akar kuadratnya.

```
#include <cstdio>
#include <cmath>
using namespace std;
typedef long long LL;

int main() {
    int T;
    if (scanf("%d", &T) != 1) return 0;
    while (T--) {
        LL k;
        if (scanf("%lld", &k) != 1) break;
        // n terkecil dengan n(n+1)/2 >= k, dicari lewat rumus kuadrat lalu dikoreksi
        LL n = (LL)floor((-1.0 + sqrt(1.0 + 8.0 * (double)k)) / 2.0);
        while (n * (n + 1) / 2 < k) ++n;          // koreksi ke atas  (menjaga galat sqrt)
        while (n >= 1 && (n - 1) * n / 2 >= k) --n;  // koreksi ke bawah
        LL r = n * (n + 1) / 2;                   // berapa pasangan yang memakai eksponen
<= n
        LL m = n + k - (r + 1);
        printf("%lld %lld\n", m, n);
    }
    return 0;
}
```

Mengompilasi dan menjalankannya, langkah demi langkah. Jika Anda belum pernah menjalankan program *judge*, lakukanlah persis seperti ini.

1. Simpan kode itu dalam berkas bernama `ldcsumab.cpp`.

2. Kompilasi: `g++ -std=c++17 -O2 -Wall -o ldcsumab ldcsumab.cpp`. Opsi `-std=c++17` memilih versi bahasanya, `-O2` menyalakan optimasi sebagaimana dilakukan *judge*, dan `-Wall` meminta kompilator memperingatkan apa pun yang mencurigakan. Perintah itu semestinya tidak mencetak apa-apa.
3. Simpan contoh masukannya dalam berkas `sample.txt`, satu bilangan per baris: `3`, `1`, `3`, `5`.
4. Jalankan program dengan berkas itu sebagai masukan bakunya: `./ldcsumab < sample.txt` (di Windows, `ldcsumab.exe < sample.txt`).

Program itu mencetak

```
0 1
1 2
1 3
```

yang merupakan contoh keluarannya, huruf demi huruf. Dengan posisi 1 sampai 10 sebagai masukan ia mencetak

```
0 1
0 2
1 2
0 3
1 3
2 3
0 4
1 4
2 4
3 4
```

yang merupakan kolom (m, n) pada **Tabel 4.2**, baris demi baris. Dan dengan posisi $6, 7, 499500, 499501, 10^9$ — dua batas, sebuah bilangan segitiga $T_{999} = 499{,}500$, entri sesudahnya, dan k terbesar yang diizinkan — ia mencetak

```
2 3
0 4
998 999
0 1000
38439 44721
```

Setiap baris cocok dengan hitungan tangan: posisi $499{,}500 = T_{999}$ adalah entri terakhir blok 999, sehingga $m = 998$; posisi 499,501 membuka blok 1000 dengan $m = 0$; dan baris terakhir adalah jawaban yang kita hitung dengan tangan untuk $k = 10^9$ pada Subbab 4.3.4.

4.3.8 Alternatif: pencarian biner atas nomor blok

Bentuk tertutup itu membutuhkan akar kuadrat dan argumen kebenaran tentang pembulatan. Ada jalan kedua yang tidak membutuhkan keduanya, dan ia hanya memakai Bab 2: **pencarian biner atas jawabannya**. Pertanyaan "apakah $T_n \geq k$?" berjawaban *tidak* untuk n yang kecil dan *ya* untuk semua n mulai titik tertentu, dan kita menginginkan *ya* yang pertama. Itu pertanyaan ya/tidak yang monoton, yang dijawab pencarian biner dalam waktu logaritmik.

Kita memerlukan ujung atas pencarian. Karena $T_{50000} = \frac{50000 \times 50001}{2} = 1{,}250{,}025{,}000 \geq 10^9$, jawabannya tidak pernah melebihi 50,000, jadi kita mencari di $[1, 50000]$. Invariannya: *jawabannya terletak di* $[lo, hi]$. Jika $T_{mid} \geq k$, maka mid sudah cukup besar dan jawabannya paling besar mid, jadi tetapkan $hi = mid$; jika tidak, mid terlalu kecil dan jawabannya paling kecil $mid + 1$, jadi tetapkan $lo = mid + 1$. Ketika $lo = hi$, jendelanya memuat satu calon saja, dan menurut invariannya calon itulah jawabannya.

Untuk melihatnya bekerja dengan tangan, mulailah dengan jendela yang lebih kecil [1,8] dan $k = 5$:

1. $lo = 1, hi = 8, mid = 1 + \lfloor 7/2 \rfloor = 4$. $T_4 = 10 \geq 5$, jadi $hi = 4$.
2. $lo = 1, hi = 4, mid = 1 + \lfloor 3/2 \rfloor = 2$. $T_2 = 3 < 5$, jadi $lo = 3$.
3. $lo = 3, hi = 4, mid = 3 + \lfloor 1/2 \rfloor = 3$. $T_3 = 6 \geq 5$, jadi $hi = 3$.
4. $lo = hi = 3$. Bloknya $n = 3$, dan $m = 5 - T_2 - 1 = 5 - 3 - 1 = 1$. Jawaban `1 3` ✓

Kode 4.2 adalah programnya. Perhatikan titik tengah `lo + (hi - lo) / 2`, bentuk aman-limpahan dari `(lo + hi) / 2` yang dibahas pada Subbab 2.14.6.

Kode 4.2 LDCSUMAB dengan pencarian biner atas nomor blok, $\Theta(\lg 50000) \approx 16$ langkah per pertanyaan, tanpa titik-mengambang sama sekali.

```
#include <cstdio>
using namespace std;
typedef long long LL;

// n >= 1 terkecil dengan n(n+1)/2 >= k, lewat pencarian biner atas jawabannya.
// Invarian: jawabannya terletak di [lo, hi].
LL blockOf(LL k) {
    LL lo = 1, hi = 50000;                // 50000*50001/2 > 10^9, jadi hi cukup besar
    while (lo < hi) {
        LL mid = lo + (hi - lo) / 2;      // tak pernah melimpah, tidak seperti (lo + hi)
/ 2
        if (mid * (mid + 1) / 2 >= k) hi = mid;   // mid cukup besar:  jawaban <= mid
        else lo = mid + 1;                         // mid terlalu kecil: jawaban >  mid
    }
    return lo;
}

int main() {
    int T;
    if (scanf("%d", &T) != 1) return 0;
    while (T--) {
        LL k;
        if (scanf("%lld", &k) != 1) break;
        LL n = blockOf(k);
        LL m = k - (n - 1) * n / 2 - 1;    // posisi di dalam blok n, dihitung dari 0
        printf("%lld %lld\n", m, n);
    }
    return 0;
}
```

Pada masukan kasus-tepi di atas ia mencetak kelima baris yang sama, dan pada posisi 1 sampai 10 kesepuluh baris yang sama. Biayanya sekitar 16 iterasi per pertanyaan alih-alih $O(1)$ — tidak berarti apa-apa dengan sepuluh pertanyaan, dan harga yang pantas untuk program yang argumen kebenarannya cukup satu invarian.

4.3.9 Memeriksa terhadap gaya kasar

Bukti pada Subbab 4.3.5 adalah jaminan yang sesungguhnya. Tetapi sebuah bukti bisa mengandung kekhilafan, dan eksperimen yang murah menangkap sebagian besar kekhilafan. **Kode 4.3** membangun daftarnya dengan cara lambat — setiap nilai $2^m + 2^n$ dengan $n \leq 40$, lalu diurutkan — dan membandingkannya dengan rumus pada setiap posisi dari $820 = T_{40}$ posisinya. (Setiap nilai dengan $n \leq 40$ lebih kecil daripada setiap nilai dengan $n = 41$, menurut Klaim 1, sehingga 820 nilai terurut pertama tepat merupakan nilai-nilai yang dibangkitkan.)

Kode 4.3 Pemeriksaan silang gaya kasar untuk LDCSUMAB: daftarnya dibangun dan diurutkan secara eksplisit, lalu dibandingkan dengan rumus pada kedelapan ratus dua puluh posisi.

```
#include <cstdio>
#include <cmath>
#include <vector>
#include <tuple>
#include <algorithm>
using namespace std;
typedef long long LL;

LL formulaN(LL k) {                          // metode bentuk tertutup
    LL n = (LL)floor((-1.0 + sqrt(1.0 + 8.0 * (double)k)) / 2.0);
    while (n * (n + 1) / 2 < k) ++n;
    while (n >= 1 && (n - 1) * n / 2 >= k) --n;
    return n;
}

int main() {
    // Gaya kasar: bangun setiap nilai 2^m + 2^n dengan 0 <= m < n <= 40 lalu urutkan
menurut nilai.
    vector<tuple<LL, int, int>> seq;
    for (int n = 1; n <= 40; ++n)
        for (int m = 0; m < n; ++m)
            seq.emplace_back((1LL << m) + (1LL << n), m, n);
    sort(seq.begin(), seq.end());
    printf(" k  value  m  n\n");
    for (int k = 1; k <= 10; ++k)
        printf("%2d  %5lld  %d  %d\n", k, get<0>(seq[k - 1]), get<1>(seq[k - 1]),
get<2>(seq[k - 1]));
    int bad = 0;
    for (LL k = 1; k <= 820; ++k) {          // 820 = 40*41/2 suku, semuanya dengan n <= 40
        LL n = formulaN(k), m = n + k - (n * (n + 1) / 2 + 1);
        if (m != get<1>(seq[k - 1]) || n != get<2>(seq[k - 1])) ++bad;
    }
    printf("terms compared: 820, mismatches: %d\n", bad);
    return 0;
}
```

Program itu mencetak

```
 k  value  m  n
 1      3  0  1
 2      5  0  2
 3      6  1  2
 4      9  0  3
 5     10  1  3
 6     12  2  3
 7     17  0  4
 8     18  1  4
 9     20  2  4
10     24  3  4
terms compared: 820, mismatches: 0
```

Sepuluh baris pertama mereproduksi **Tabel 4.2** secara tepat — nilai, m, dan n — dan tidak ada ketidakcocokan di mana pun. Membangun pemeriksa semacam ini adalah kebiasaan yang mengubah “saya kira ini benar” menjadi “saya sudah memeriksanya”.

4.3.10 Kesalahan yang sering terjadi pada persoalan ini

Pertama: menganggap blok-bloknya tidak berselang-seling tanpa membuktikannya. Memang tidak, dan Klaim 1 adalah bukti satu paragrafnya — tetapi bukti itu harus dituliskan, sebab “jelas bahwa blok-bloknya tidak bertumpang tindih” adalah jenis kejelasan yang kadang-kadang salah.

Kedua: memakai pembulatan ke bawah di tempat yang seharusnya pembulatan ke atas, atau $>$ di tempat yang seharusnya $\geq$, ketika memburu bloknya, sehingga k mendarat satu blok terlalu awal

atau terlalu lambat. Ujilah $k = 1,3,6,10$ dan $k = 2,4,7,11$: setiap bilangan segitiga adalah batas blok, dan setiap bilangan segitiga ditambah satu membuka blok baru.

Ketiga: mencetak `n m` alih-alih `m n`. Persoalannya mengatakan $m < n$ dan contohnya menegaskan urutan itu.

Keempat: menghitung $n(n+1)$ dalam `int`. Untuk k di sekitar 10^9 hasil kalinya sekitar 2.0×10^9, setipis rambut di bawah batas `int`; kendala yang sedikit lebih besar saja akan melimpahinya tanpa suara. Pakailah `long long`.

Kelima: membangun daftarnya. Itu gagasan pertama yang terlintas pada semua orang, dan Langkah 3 menolaknya dalam sepuluh detik.

4.4 Persoalan 2 — TOHU: Membantu Tohu

Persoalan *judge* kedua, TOHU, adalah persoalan klasik SPOJ [4]. Bila LDCSUMAB melatih deret aritmetika, TOHU melatih perkakas penjumlahan agung lainnya dari Bab 3: **jumlah teleskopik** (*telescoping sum*) pada Subbab 3.9.5. Seluruh penyelesaiannya hanyalah lima baris aljabar dan program satu baris, tetapi tiap baris dari kelima baris itu menuntut kecermatan, dan persoalan ini menyembunyikan jebakan pada $k = 1$ yang menjerat orang-orang yang cermat sekalipun.

4.4.1 Persoalannya dengan kata-kata kita sendiri

Tohu ingin belajar di sebuah universitas dan harus lebih dahulu lulus tes masuk bidang matematika. Ia meminta bantuan untuk satu soal. Sebuah barisan bilangan $a_1, a_2, a_3, \ldots$ tidak diberikan kepada kita secara langsung. Sebagai gantinya, kita diberi tahu bahwa untuk setiap bilangan bulat positif k, suatu jumlah berbobot dari k suku pertamanya memiliki nilai yang diketahui:

$$a_1 + 2a_2 + 3a_3 + \cdots + k\,a_k = \frac{k+1}{k+2} \quad \text{untuk setiap } k \geq 1 \tag{4.4}$$

Kita harus menghitung jumlah biasa yang tidak berbobot dari n suku pertamanya, $S_n = a_1 + a_2 + \cdots + a_n$, untuk banyak nilai n.

Masukan diawali banyaknya kasus uji T, paling banyak 20,000; tiap baris berikutnya memuat satu n. Untuk setiap n kita mencetak S_n dengan tepat sebelas angka di belakang koma. Dengan contoh masukan `2`, `2`, `5`, keluaran yang diharapkan adalah `0.70833333333` dan `0.73809523810`.

> **PERSOALAN.** Barisan $a_1, a_2, \ldots$ memenuhi Persamaan (4.4). Untuk tiap pertanyaan n, sebanyak paling banyak 20,000 pertanyaan, cetak $S_n = a_1 + \cdots + a_n$ dengan sebelas angka di belakang koma. Persoalan ini biasanya diajukan dengan catatan *kompleksitas waktu* $O(1)$.

4.4.2 Langkah 1 sampai 3: format, lambang, anggaran

Langkah 1 — formatnya, disalin.

- Masukan: baris 1 memuat T ($T \leq 20{,}000$). Lalu T baris, masing-masing memuat satu bilangan bulat positif n.
- Keluaran: untuk tiap n, satu baris berisi S_n yang dicetak dengan tepat 11 angka di belakang koma.
- Contoh masukan `2 / 2 / 5`; contoh keluaran `0.70833333333 / 0.73809523810`.

Sebelas angka itu banyak. Ketentuan itu langsung menyingkirkan `float`, dan berarti untai formatnya harus `%.11f`.

Langkah 2 — diketahui dan ditanya.

- Diketahui: $n \geq 1$; dan, secara tersirat, barisannya melalui Persamaan (4.4).
- Ditanya: $S_n = \sum_{k=1}^{n} a_k$.

Sesungguhnya ada *dua* yang tidak diketahui di sini: pertama suku-suku a_k, yang disembunyikan persoalan di balik jumlah berbobot, lalu jumlah biasanya. Menamai keduanya itulah yang membuat rencananya terlihat: pulihkan a_k, lalu jumlahkan.

Langkah 3 — anggarannya. Pernyataan persoalan tidak menetapkan batas kecil bagi n, jadi kita harus siap menghadapi nilai yang besar. Dengan sampai 20,000 pertanyaan, menjumlahkan n suku per pertanyaan berbiaya $\Theta(Tn)$, yaitu 2×10^{13} bila n dapat mencapai 10^9 — mustahil menurut **Tabel 4.1**. Catatan $O(1)$ mengatakan hal yang sama dengan lebih lugas: ada bentuk tertutup untuk S_n, dan tugas kita adalah menemukannya.

4.4.3 Langkah 4: kasus-kasus pertama dengan tangan

Tuliskan W_k untuk jumlah berbobot di ruas kiri Persamaan (4.4), sehingga $W_k = \frac{k+1}{k+2}$. Berpindah dari W_{k-1} ke W_k menambahkan tepat satu suku baru, yaitu $k\,a_k$. Jadi

$$k\,a_k = W_k - W_{k-1}, \qquad \text{dengan } W_0 = 0 \text{ (jumlah kosong).}$$

Satu pengamatan itu memungkinkan kita menghitung barisannya suku demi suku. Setiap pengurangan di bawah dikerjakan di atas penyebut bersama.

- $k = 1$: $W_1 = \frac{2}{3}$ dan $W_0 = 0$, jadi $1 \cdot a_1 = \frac{2}{3}$ dan $a_1 = \frac{2}{3}$. Maka $S_1 = \frac{2}{3}$.
- $k = 2$: $W_2 = \frac{3}{4}$. $2a_2 = \frac{3}{4} - \frac{2}{3} = \frac{9}{12} - \frac{8}{12} = \frac{1}{12}$, jadi $a_2 = \frac{1}{24}$. $S_2 = \frac{2}{3} + \frac{1}{24} = \frac{16}{24} + \frac{1}{24} = \frac{17}{24}$.
- $k = 3$: $W_3 = \frac{4}{5}$. $3a_3 = \frac{4}{5} - \frac{3}{4} = \frac{16}{20} - \frac{15}{20} = \frac{1}{20}$, jadi $a_3 = \frac{1}{60}$. $S_3 = \frac{17}{24} + \frac{1}{60} = \frac{85}{120} + \frac{2}{120} = \frac{87}{120} = \frac{29}{40}$.
- $k = 4$: $W_4 = \frac{5}{6}$. $4a_4 = \frac{5}{6} - \frac{4}{5} = \frac{25}{30} - \frac{24}{30} = \frac{1}{30}$, jadi $a_4 = \frac{1}{120}$. $S_4 = \frac{29}{40} + \frac{1}{120} = \frac{87}{120} + \frac{1}{120} = \frac{88}{120} = \frac{11}{15}$.
- $k = 5$: $W_5 = \frac{6}{7}$. $5a_5 = \frac{6}{7} - \frac{5}{6} = \frac{36}{42} - \frac{35}{42} = \frac{1}{42}$, jadi $a_5 = \frac{1}{210}$. $S_5 = \frac{11}{15} + \frac{1}{210} = \frac{154}{210} + \frac{1}{210} = \frac{155}{210} = \frac{31}{42}$.

Tabel 4.3 menghimpun hasilnya.

Tabel 4.3 Lima suku pertama barisan TOHU, dipulihkan dengan tangan dari jumlah-jumlah berbobotnya. Mulai $k = 2$, a_k adalah kebalikan hasil kali tiga bilangan bulat berurutan; a_1 mematahkan pola itu.

k	$W_k=(k+1)/(k+2)$	$k\,a_k=W_k-W_{k-1}$	a_k	a_k Difaktorkan	S_k	S_k Sebagai desimal
1	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	—	$\frac{2}{3}$	0.66666666667
2	$\frac{3}{4}$	$\frac{1}{12}$	$\frac{1}{24}$	$\frac{1}{2 \cdot 3 \cdot 4}$	$\frac{17}{24}$	0.70833333333
3	$\frac{4}{5}$	$\frac{1}{20}$	$\frac{1}{60}$	$\frac{1}{3 \cdot 4 \cdot 5}$	$\frac{29}{40}$	0.72500000000
4	$\frac{5}{6}$	$\frac{1}{30}$	$\frac{1}{120}$	$\frac{1}{4 \cdot 5 \cdot 6}$	$\frac{11}{15}$	0.73333333333
5	$\frac{6}{7}$	$\frac{1}{42}$	$\frac{1}{210}$	$\frac{1}{5 \cdot 6 \cdot 7}$	$\frac{31}{42}$	0.73809523810

Dua hal langsung mencuat dari tabel ini. Pertama, baris S_2 dan S_5 tepat merupakan contoh keluarannya, jadi kita sudah menyelesaikan contohnya dengan tangan. Kedua, mulai $k = 2$ suku-sukunya adalah $\frac{1}{2\cdot3\cdot4}, \frac{1}{3\cdot4\cdot5}, \frac{1}{4\cdot5\cdot6}, \frac{1}{5\cdot6\cdot7}$: dugaannya adalah $a_k = \frac{1}{k(k+1)(k+2)}$. Tetapi $a_1 = \frac{2}{3}$, sedangkan $\frac{1}{1\cdot2\cdot3} = \frac{1}{6}$. Suku pertama *tidak* cocok. Tabel yang memperlihatkan pola juga memperlihatkan di mana pola itu patah, dan itu informasi yang sangat berharga.

4.4.4 Langkah 5 dan 6: memulihkan a_k dengan menyelisihkan

Langkah 5 — bentuknya. Ini *bentuk tertutup dari sebuah jumlah*, dua kali: pertama membongkar sebuah jumlah untuk memulihkan suku-sukunya, lalu menjumlahkan suku-suku itu dalam bentuk tertutup.

Langkah 6 — rumus a_k, dibuktikan. Tuliskan Persamaan (4.4) pada k dan pada $k - 1$, bertumpuk, untuk sembarang $k \geq 2$:

$$a_1 + 2a_2 + \cdots + (k-1)a_{k-1} + ka_k = \frac{k+1}{k+2}$$

$$a_1 + 2a_2 + \cdots + (k-1)a_{k-1} \qquad = \frac{k}{k+1}$$

(Baris kedua adalah Persamaan (4.4) dengan $k - 1$ menggantikan k: $\frac{(k-1)+1}{(k-1)+2} = \frac{k}{k+1}$.) Kurangkan baris kedua dari baris pertama. Setiap suku di ruas kiri saling meniadakan kecuali ka_k:

$$k\,a_k = \frac{k+1}{k+2} - \frac{k}{k+1}.$$

Taruh ruas kanannya di atas penyebut bersama $(k+1)(k+2)$:

$$k\,a_k = \frac{(k+1)(k+1) - k(k+2)}{(k+1)(k+2)} = \frac{(k^2+2k+1) - (k^2+2k)}{(k+1)(k+2)} = \frac{1}{(k+1)(k+2)}.$$

Bagi kedua ruas dengan k:

$$a_k = \frac{1}{k(k+1)(k+2)} \quad \text{untuk } k \geq 2, \qquad a_1 = \frac{2}{3} \tag{4.5}$$

Rumus untuk $k \geq 2$ kini adalah teorema, bukan terkaan: aljabarnya berlaku untuk setiap $k \geq 2$ sekaligus. Dan aljabar itu juga *menjelaskan* mengapa $k = 1$ berbeda. Pengurangan tadi memakai syarat pada $k - 1$, yang hanya ada bila $k - 1 \geq 1$. Andaikan kita tetap menerapkan rumus $W_{k-1} = \frac{k}{k+1}$ pada $k = 1$, rumus itu akan mengklaim $W_0 = \frac{1}{2}$. Padahal W_0 adalah jumlah kosong, yang bernilai 0. Rumus itu melebihkan jumlah berbobot sebelumnya sebesar $\frac{1}{2}$, sehingga pola itu mengurangkan $1 \cdot a_1$ sebesar $\frac{1}{2}$: pola memberi $\frac{1}{1\cdot2\cdot3} = \frac{1}{6}$, sedangkan yang sebenarnya adalah $\frac{1}{6} + \frac{1}{2} = \frac{1}{6} + \frac{3}{6} = \frac{4}{6} = \frac{2}{3}$. Tak ada yang misterius pada a_1; ia semata-mata satu-satunya suku yang tidak punya pendahulu bagi siasat penyelisihan itu.

4.4.5 Menjumlahkan secara teleskopik

Kita menginginkan

$$S_n = a_1 + \sum_{k=2}^{n} a_k = \frac{2}{3} + \sum_{k=2}^{n} \frac{1}{k(k+1)(k+2)}.$$

Jumlah teleskopik Bab 3 membutuhkan suku yang berupa *selisih dua nilai berurutan dari satu fungsi*. Suku $\frac{1}{k(k+1)(k+2)}$ tidak tampak seperti itu, tetapi dapat dijadikan demikian. Hitunglah selisih $\frac{1}{k(k+1)}$ dan penerusnya $\frac{1}{(k+1)(k+2)}$:

$$\frac{1}{k(k+1)} - \frac{1}{(k+1)(k+2)} = \frac{(k+2)-k}{k(k+1)(k+2)} = \frac{2}{k(k+1)(k+2)}.$$

Bagi dengan 2:

$$\frac{1}{k(k+1)(k+2)} = \frac{1}{2}\left[\frac{1}{k(k+1)} - \frac{1}{(k+1)(k+2)}\right] \tag{4.6}$$

Sekarang jumlahkan Persamaan (4.6) dari $k = 2$ sampai $k = n$ dan saksikan suku-suku tengahnya saling meniadakan. Menuliskan isi kurung siku untuk tiap k:

- $k = 2$: $\frac{1}{2\cdot 3} - \frac{1}{3\cdot 4}$
- $k = 3$: $\frac{1}{3\cdot 4} - \frac{1}{4\cdot 5}$
- $k = 4$: $\frac{1}{4\cdot 5} - \frac{1}{5\cdot 6}$
- $\vdots$
- $k = n$: $\frac{1}{n(n+1)} - \frac{1}{(n+1)(n+2)}$

Setiap suku negatif ditiadakan oleh suku positif pada baris berikutnya. Hanya suku positif pertama dan suku negatif terakhir yang bertahan:

$$\sum_{k=2}^{n} \frac{1}{k(k+1)(k+2)} = \frac{1}{2}\left[\frac{1}{2\cdot 3} - \frac{1}{(n+1)(n+2)}\right] = \frac{1}{12} - \frac{1}{2(n+1)(n+2)}.$$

Karena itu

$$\begin{aligned} S_n = \frac{2}{3} + \frac{1}{12} - \frac{1}{2(n+1)(n+2)} &= \frac{8}{12} + \frac{1}{12} - \frac{1}{2(n+1)(n+2)} \\ &= \frac{9}{12} - \frac{1}{2(n+1)(n+2)}, \end{aligned}$$

dan karena $\frac{9}{12} = \frac{3}{4}$,

$$S_n = \frac{3}{4} - \frac{1}{2(n+1)(n+2)} \tag{4.7}$$

Periksa terhadap tabel. $n = 2$: $\frac{3}{4} - \frac{1}{2\cdot3\cdot4} = \frac{3}{4} - \frac{1}{24} = \frac{18}{24} - \frac{1}{24} = \frac{17}{24}$ ✓, yaitu 0.708333 … dan tercetak sebagai `0.70833333333`. $n = 5$: $\frac{3}{4} - \frac{1}{2\cdot6\cdot7} = \frac{3}{4} - \frac{1}{84} = \frac{63}{84} - \frac{1}{84} = \frac{62}{84} = \frac{31}{42}$ ✓, yaitu 0.738095238095 … dan dibulatkan menjadi `0.73809523810`.

Jalan kedua: pecahan parsial. Cara baku lain untuk memecah $\frac{1}{k(k+1)(k+2)}$ adalah mencari konstanta A, B, C dengan

$$\frac{1}{k(k+1)(k+2)} = \frac{A}{k} + \frac{B}{k+1} + \frac{C}{k+2}.$$

Kalikan seluruhnya dengan $k(k+1)(k+2)$:

$$\begin{aligned} 1 &= A(k+1)(k+2) + Bk(k+2) + Ck(k+1) \\ &= (A+B+C)k^2 + (3A+2B+C)k + 2A. \end{aligned}$$

Dua polinomial sama untuk setiap k hanya bila koefisien-koefisiennya cocok, jadi

$$A + B + C = 0, \qquad 3A + 2B + C = 0, \qquad 2A = 1.$$

Dari persamaan ketiga, $A = 1/2$. Mengurangkan persamaan pertama dari yang kedua memberi $2A + B = 0$, sehingga $B = -1$. Lalu $C = -A - B = -1/2 + 1 = 1/2$. Jadi

$$\frac{1}{k(k+1)(k+2)} = \frac{1}{2k} - \frac{1}{k+1} + \frac{1}{2(k+2)}.$$

Bila dijumlahkan dari 2 sampai n, ketiga potongan ini juga meneleskop, meskipun peniadaannya tersebar di tiga kolom alih-alih dua dan butuh pembukuan lebih banyak untuk melihatnya. Mengelompokkannya sebagai $\frac{1}{2}\left[\frac{1}{k} - \frac{1}{k+1}\right] - \frac{1}{2}\left[\frac{1}{k+1} - \frac{1}{k+2}\right]$ mengembalikan bentuk pecahan parsial itu menjadi Persamaan (4.6). Kedua jalan tiba di Persamaan (4.7); yang pertama lebih pendek, yang kedua berlaku untuk fungsi rasional apa pun, dan mengenal keduanya sepadan dengan satu halaman tambahan.

4.4.6 Bukti kebenaran

Penurunan di atas sudah merupakan bukti, karena setiap langkahnya adalah identitas. Tetapi hasil yang dicapai lewat pengelompokan ulang yang cerdik layak ditegaskan dengan metode kedua yang sepenuhnya mekanis, dan induksi Bab 3 tepat untuk itu.

TEOREMA. Untuk setiap $n \geq 1$, $S_n = \frac{3}{4} - \frac{1}{2(n+1)(n+2)}$.

Bukti dengan induksi pada n.

Kasus dasar, $n = 1$. $S_1 = a_1 = \frac{2}{3}$. Rumusnya memberi $\frac{3}{4} - \frac{1}{2\cdot2\cdot3} = \frac{3}{4} - \frac{1}{12} = \frac{9}{12} - \frac{1}{12} = \frac{8}{12} = \frac{2}{3}$. ✓ Perhatikan bahwa rumusnya mencakup $n = 1$ walaupun a_1 adalah suku perkecualian; perkecualian itu sudah terserap ke dalam konstanta $\frac{3}{4}$.

Langkah induksi. Misalkan $n \geq 2$ dan andaikan $S_{n-1} = \frac{3}{4} - \frac{1}{2n(n+1)}$. Karena $n \geq 2$, Persamaan (4.5) memberi $a_n = \frac{1}{n(n+1)(n+2)}$. Maka

$$S_n = S_{n-1} + a_n = \frac{3}{4} - \frac{1}{2n(n+1)} + \frac{1}{n(n+1)(n+2)}.$$

Taruh dua suku terakhir di atas penyebut bersama $2n(n+1)(n+2)$:

$$-\frac{n+2}{2n(n+1)(n+2)} + \frac{2}{2n(n+1)(n+2)} = -\frac{(n+2)-2}{2n(n+1)(n+2)}$$
$$= -\frac{n}{2n(n+1)(n+2)} = -\frac{1}{2(n+1)(n+2)}.$$

Jadi $S_n = \frac{3}{4} - \frac{1}{2(n+1)(n+2)}$, yang merupakan klaim untuk n. ■

Teorema itu juga memberi tahu sesuatu yang tidak pernah ditanyakan persoalannya: seiring bertambahnya n, $\frac{1}{2(n+1)(n+2)}$ menyusut ke 0 dan S_n merambat naik menuju $\frac{3}{4}$ tanpa pernah mencapainya. Setiap jawaban yang dicetak program terletak di $\left[\frac{2}{3}, \frac{3}{4}\right)$.

4.4.7 Kompleksitas, ketelitian, dan limpahan

Waktu dan ruang. Persamaan (4.7) berbiaya dua penjumlahan, dua perkalian, satu pembagian, dan satu pengurangan: $\Theta(1)$ per pertanyaan, $\Theta(T)$ secara keseluruhan, ruang $O(1)$. Dengan $T = 20{,}000$ itu sekitar 10^5 operasi — seperseribu anggaran pada **Tabel 4.1**.

Ketelitian. Jawabannya di bawah 0.75 dan harus dicetak sampai sebelas angka di belakang koma, jadi kita membutuhkan sekitar sebelas angka berarti yang benar. Sebuah `double` membawa lima belas sampai enam belas, sehingga galat pembulatan dari segelintir operasi di atas jauh di bawah angka terakhir yang dicetak. Sebuah `float`, dengan sekitar tujuh, sudah gagal pada contohnya sendiri.

Limpahan. Penyebut $2(n+1)(n+2)$ tumbuh seperti $2n^2$. Andai `n` bertipe `int` dan kita menulis `2 * (n+1) * (n+2)` dalam aritmetika bilangan bulat, hasil kalinya akan melewati batas `int` sebesar 2,147,483,647 begitu $n = 32{,}767$, tempat nilainya menjadi $2 \times 32{,}768 \times 32{,}769 = 2{,}147{,}549{,}184$ — tanpa suara menghasilkan sampah. Program di bawah menghindari persoalan itu sepenuhnya dengan mengubah tiap faktor menjadi `double` *sebelum* mengalikannya. Sebuah `double` dapat menampung $2(n+1)(n+2)$ bahkan untuk $n = 10^9$, tempat nilainya sekitar 2×10^{18}; dan pada ukuran itu $\frac{1}{2(n+1)(n+2)} \approx 5 \times 10^{-19}$ jauh di bawah angka kesebelas di belakang koma, sehingga program dengan benar mencetak `0.75000000000`.

4.4.8 Langkah 7: programnya

Pseudokode 4.2 adalah keseluruhan algoritmanya.

Pseudokode 4.2 Algoritma TOHU: satu evaluasi bentuk tertutup Persamaan (4.7) per pertanyaan.

```
TOHU()
1  read T
2  for q = 1 to T
3      read n
4      print 3/4 - 1 / (2(n+1)(n+2)) with 11 decimal places
```

Gambar 4.4 memperlihatkan penyelesaian C++ lain yang juga benar. Program itu memperlakukan $n = 1$ sebagai kasus khusus dan langsung mencetak $\frac{2}{3}$.

```
#include <stdio.h>
int main() {
    int t, n;
    double formula;
    scanf("%d", &t);
    while (t--) {
        scanf("%d", &n);
        if (n==1) {
            printf("%.11lf\n",2.0/3.0);
        } else {
            formula = ((double) 1.0/12.0) - 1/ ((double) 2 * (n+1) * (n+2));
            formula = formula + ((double) 2.0/3.0);
            printf("%.11lf\n", formula);
        }
    }
    return 0;
}
```

Gambar 4.4 Sebuah program TOHU alternatif dengan cabang terpisah `if (n==1) printf("%.11lf\n", 2.0/3.0);` dan cabang umum yang menghitung $\frac{1}{12} - \frac{1}{2(n+1)(n+2)} + \frac{2}{3}$.

Cabang itu tidak merugikan, tetapi tidak perlu: kasus dasar induksi pada Subbab 4.4.6 sudah memperlihatkan bahwa Persamaan (4.7) memberi $\frac{2}{3}$ pada $n = 1$. Memperhatikan kasus khusus yang mubazir, dan mengatakan mengapa ia mubazir, adalah persis jenis pemeriksaan yang diminta Langkah 6. Perhatikan juga, pada **Gambar 4.4**, pemeranan (*cast*) dalam penyebut `(double) 2 * (n+1) * (n+2)`: mengubah angka `2` di depan menjadi `double` memaksa perkalian pertama masuk ke titik-mengambang, sehingga `(n+1)` dan `(n+2)` dipromosikan sebelum dikalikan dan tidak pernah ada dua `int` yang dikalikan. Itu cara yang benar tetapi rapuh untuk menghindari limpahan pada Subbab 4.4.7. **Kode 4.4** membuang cabang itu dan memeranan setiap faktor secara eksplisit.

Kode 4.4 TOHU — Help Tohu, dalam $O(1)$ per kasus uji, tanpa kasus khusus untuk $n = 1$.

```
#include <cstdio>
using namespace std;

int main() {
    int T;
    if (scanf("%d", &T) != 1) return 0;
    while (T--) {
        long long n;
        if (scanf("%lld", &n) != 1) break;
        // S(n) = 3/4 - 1 / (2 (n+1)(n+2))   -- berlaku untuk setiap n >= 1
        double denom = 2.0 * (double)(n + 1) * (double)(n + 2);
        printf("%.11f\n", 0.75 - 1.0 / denom);
    }
    return 0;
}
```

Kompilasilah persis seperti pada Subbab 4.3.7 — `g++ -std=c++17 -O2 -Wall -o tohu tohu.cpp` — lalu jalankan pada contoh masukan `2`, `2`, `5`. Program itu mencetak

```
0.70833333333
0.73809523810
```

yang merupakan contoh keluarannya. Pada enam pertanyaan $n = 1,2,3,4,10,10^9$ ia mencetak

```
0.66666666667
0.70833333333
0.72500000000
0.73333333333
0.74621212121
0.75000000000
```

Empat baris pertama adalah kolom desimal pada **Tabel 4.3**; baris kelima adalah $\frac{3}{4} - \frac{1}{2 \cdot 11 \cdot 12} = \frac{3}{4} - \frac{1}{264} = \frac{197}{264}$; baris keenam memperlihatkan limit $\frac{3}{4}$ tercapai sampai sebelas angka.

4.4.9 Alternatif: pecahan eksak, tanpa mengandaikan rumus

Bentuk tertutup itulah penyelesaiannya. Tetapi andaikan Anda tidak memercayai aljabarnya, atau belum menemukannya. Anda tetap dapat menghitung barisannya *langsung dari definisinya*, memakai aturan $k\,a_k = W_k - W_{k-1}$ pada Subbab 4.4.3, dalam aritmetika rasional eksak sehingga tidak ada pembulatan yang dapat menyembunyikan kekeliruan. **Kode 4.5** melakukan persis hal itu. Program itu menyimpan tiap pecahan sebagai pembilang dan penyebut yang disederhanakan dengan faktor persekutuan terbesarnya (`std::gcd`, tersedia sejak C++17), memulihkan $a_1, \ldots, a_{10}$ hanya dari Persamaan (4.4), menghimpun S_k, lalu mencetak pecahan eksaknya di samping nilai desimal dan bentuk tertutupnya.

Kode 4.5 TOHU dari prinsip pertama: tiap a_k dipulihkan secara eksak dari syarat jumlah berbobot, tanpa mengandaikan rumus apa pun, lalu dibandingkan dengan bentuk tertutupnya.

```
#include <cstdio>
#include <numeric>
using namespace std;
typedef long long LL;

struct Frac {                                   // pecahan eksak p/q, selalu dalam bentuk
paling sederhana
    LL p, q;
    Frac(LL a = 0, LL b = 1) { LL g = gcd(a, b); p = a / g; q = b / g; }
};
Frac operator+(Frac a, Frac b) { return Frac(a.p * b.q + b.p * a.q, a.q * b.q); }
Frac operator-(Frac a, Frac b) { return Frac(a.p * b.q - b.p * a.q, a.q * b.q); }

int main() {
    // Pulihkan a_k dari syarat pendefinisinya, secara eksak, tanpa mengandaikan rumus apa
pun:
    // k*a_k = (k+1)/(k+2) - (jumlah j*a_j untuk j < k).
    Frac weighted(0, 1), S(0, 1);
    printf(" k   a_k        S_k (exact)     S_k (decimal)   closed form\n");
    for (LL k = 1; k <= 10; ++k) {
        Frac kak = Frac(k + 1, k + 2) - weighted;        // inilah k * a_k
        Frac ak(kak.p, kak.q * k);                       // bagi dengan k
        weighted = weighted + kak;
        S = S + ak;
        double closed = 0.75 - 1.0 / (2.0 * (k + 1) * (k + 2));
        printf("%2lld  %lld/%-6lld  %6lld/%-8lld  %.11f  %.11f\n",
               k, ak.p, ak.q, S.p, S.q, (double)S.p / S.q, closed);
    }
    return 0;
}
```

Program itu mencetak

```
 k   a_k        S_k (exact)     S_k (decimal)   closed form
 1  2/3            2/3         0.66666666667  0.66666666667
 2  1/24          17/24        0.70833333333  0.70833333333
 3  1/60          29/40        0.72500000000  0.72500000000
 4  1/120         11/15        0.73333333333  0.73333333333
 5  1/210         31/42        0.73809523810  0.73809523810
 6  1/336         83/112       0.74107142857  0.74107142857
 7  1/504        107/144       0.74305555556  0.74305555556
 8  1/720         67/90        0.74444444444  0.74444444444
 9  1/990         41/55        0.74545454545  0.74545454545
10  1/1320       197/264       0.74621212121  0.74621212121
```

Setiap pecahan eksak untuk $k \leq 5$ cocok dengan **Tabel 4.3**, kolom a_k menegaskan Persamaan (4.5) sampai $k = 10$ — $a_{10} = \frac{1}{10 \cdot 11 \cdot 12} = \frac{1}{1320}$ — dan dua kolom terakhir sepakat sampai kesebelas angkanya. Sebagai *penyelesaian*, pendekatan ini tidak berguna untuk n yang besar: biayanya $\Theta(n)$ per pertanyaan, dan penyebutnya tumbuh seperti n^3 sampai melimpahi `long long`. Sebagai *pemeriksaan*, ia sangat baik, sebab ia tidak mengandaikan apa pun yang telah kita turunkan.

4.4.10 Kesalahan yang sering terjadi pada persoalan ini

Pertama: menghitung S_n dengan menggelung k dari 1 sampai n. Dengan dua puluh ribu pertanyaan dan n yang besar, itu mustahil, dan persoalannya sudah mengatakan demikian dengan menuliskan $O(1)$.

Kedua: membalik arah penyelisihan — mengurangkan persamaan k dari persamaan $k - 1$ dan berakhir dengan $-ka_k$. Tuliskan kedua persamaan itu bertumpuk, seperti pada Subbab 4.4.4, dan peniadaannya akan terlihat.

Ketiga: mengandaikan $a_k = \frac{1}{k(k+1)(k+2)}$ berlaku pula untuk $k = 1$. Hasilnya $S_n = \frac{1}{4} - \frac{1}{2(n+1)(n+2)}$, yang keliru tepat sebesar $\frac{1}{2}$. Tabel tangan pada Langkah 4 menangkap kesalahan ini dalam satu baris.

Keempat: mencetak dengan `float`, atau dengan `%.10f`, atau dengan `%lf` pada kompilator lama yang salah menanganinya. Pakailah `double` dan `%.11f`.

Kelima: mengalikan `(n+1)*(n+2)` dalam aritmetika `int`, yang melimpah mulai $n = 32{,}767$.

Keenam: menulis `3/4` dalam C++ dan mendapatkan `0`. Pembagian bilangan bulat memangkas; tulislah `0.75` atau `3.0/4.0`.

4.5 Persoalan Terkerjakan tentang Kebenaran: Invarian dan Penelusuran

Kedua persoalan *judge* tadi memperlihatkan metodenya secara utuh. Dua belas persoalan terkerjakan berikut lebih ringkas, dan masing-masing membidik satu alat dari Bab 1 sampai Bab 3. Semuanya berbentuk sama, supaya Anda selalu tahu sedang berada di mana:

- **Persoalan** — pernyataannya.
- **Yang diuji** — alatnya, dan di bagian mana buku ini alat itu diperkenalkan.
- **Penyelesaian** — langkah demi langkah, dengan setiap langkah hitungannya diperlihatkan.
- **Jawaban** — hasilnya dalam satu baris.

- **Program** — program C++17 yang lengkap bila menjalankannya mengajarkan sesuatu, bersama keluaran yang benar-benar dicetaknya.
- **Uji diri** — sebuah varian untuk Anda kerjakan sendiri, beserta jawabannya.

Subbab ini mengerjakan tiga persoalan tentang *kebenaran*: membuktikan bahwa sebuah gelung benar, dan mengikutinya dengan tepat.

4.5.1 Persoalan Terkerjakan A — invarian gelung untuk aturan Horner

Persoalan. Polinomial berderajat n berbentuk $P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$. Simpan koefisiennya dalam larik $A[0..n]$, dengan $A[k] = a_k$. Prosedur pada **Pseudokode 4.3**, yang dikenal sebagai **aturan Horner**, mengklaim dapat mengevaluasi $P(x)$ hanya dengan $n + 1$ perkalian. Nyatakan sebuah invarian gelung untuknya, buktikan invarian itu, lalu gunakan untuk membuktikan bahwa prosedur itu mengembalikan $P(x)$. Kemudian telusurilah pada $P(x) = 2x^3 - 6x^2 + 2x - 1$ di $x = 3$.

Pseudokode 4.3 Aturan Horner: mengevaluasi $a_0 + a_1x + \cdots + a_nx^n$ mulai dari koefisien tertinggi ke bawah.

```
HORNER(A, n, x)
1  p = 0
2  for i = n downto 0
3      p = A[i] + x * p
4  return p
```

Yang diuji. Bukti invarian gelung tiga bagian pada Subbab 2.9 (inisialisasi, pemeliharaan, terminasi), diterapkan pada gelung yang belum pernah Anda lihat dibuktikan, serta manipulasi indeks pada penjumlahan Bab 3.

Penyelesaian. Dari mana gagasannya datang? Faktorkan x dari segala sesuatu kecuali suku konstanta, berulang-ulang:

$$a_0 + a_1x + a_2x^2 + a_3x^3 = a_0 + x(a_1 + x(a_2 + x\,a_3)).$$

Mengevaluasinya dari dalam ke luar membutuhkan satu perkalian per tingkat. Gelung itu melakukan persis hal tersebut: pada tiap langkah ia mengambil apa yang sudah dimilikinya, mengalikannya dengan x, lalu menambahkan koefisien berikutnya ke bawah. Jadi pada setiap saat p adalah polinomial yang dibangun dari koefisien-koefisien yang *sejauh ini* sudah diproses, yaitu yang tertinggi, dengan yang terendah di antaranya sebagai suku konstanta. Kalimat itulah invariannya; sekarang tuliskan dalam lambang.

> **INVARIAN GELUNG (aturan Horner).** Pada awal setiap iterasi gelung `for` di baris 2–3, dengan indeks gelung i,

$$p = \sum_{k=0}^{n-(i+1)} A\,[k + i + 1]\,x^k \tag{4.8}$$

Dengan kata-kata: p adalah polinomial yang koefisiennya $A[i + 1], A[i + 2], \ldots, A[n]$, dengan $A[i + 1]$ sebagai suku konstanta. Sekarang ketiga kewajibannya.

Inisialisasi. Sebelum iterasi pertama, $i = n$. Jumlah pada Persamaan (4.8) berjalan dari $k = 0$ sampai $k = n - (n + 1) = -1$. Jumlah yang batas atasnya di bawah batas bawahnya adalah jumlah kosong, dan jumlah kosong bernilai 0. Baris 1 menetapkan $p = 0$. ✓

Pemeliharaan. Andaikan Persamaan (4.8) berlaku pada awal iterasi dengan indeks i. Baris 3 menghitung nilai baru

$$p' = A[i] + x \cdot \sum_{k=0}^{n-(i+1)} A\,[k+i+1]\,x^k = A[i] + \sum_{k=0}^{n-i-1} A\,[k+i+1]\,x^{k+1}.$$

Ganti nama indeks penjumlahannya: ambil $j = k + 1$, sehingga j berjalan dari 1 sampai $n - i$ dan $A[k + i + 1] = A[j + i]$:

$$p' = A[i]\,x^0 + \sum_{j=1}^{n-i} A\,[j+i]\,x^j = \sum_{j=0}^{n-i} A\,[j+i]\,x^j,$$

sebab suku tunggal $A[i]$ tepat merupakan suku $j = 0$ dari jumlah yang sama. Sekarang gelung mengurangi indeksnya: iterasi berikutnya berindeks $i' = i - 1$. Menyulih $i = i' + 1$ memberi $p' = \sum_{j=0}^{n-(i'+1)} A\,[j + i' + 1]\,x^j$, yaitu Persamaan (4.8) untuk indeks i'. ✓

Terminasi. Gelung berjalan untuk $i = n, n - 1, \ldots, 0$ dan berhenti ketika i mencapai -1, sesudah tepat $n + 1$ iterasi. Menyulih $i = -1$ ke dalam Persamaan (4.8):

$$p = \sum_{k=0}^{n-0} A\,[k+0]\,x^k = \sum_{k=0}^{n} a_k\,x^k = P(x).$$

Baris 4 mengembalikan p, yaitu $P(x)$. ✓ ■

Penelusurannya. Untuk $P(x) = 2x^3 - 6x^2 + 2x - 1$ koefisiennya adalah $A[0] = -1$, $A[1] = 2$, $A[2] = -6$, $A[3] = 2$, dengan $n = 3$ dan $x = 3$. **Tabel 4.4** mengikuti gelungnya, dan kolom terakhirnya mengevaluasi ruas kanan Persamaan (4.8) secara terpisah, sehingga invariannya terlihat berlaku pada setiap langkah.

Tabel 4.4 Aturan Horner pada $2x^3 - 6x^2 + 2x - 1$ di $x = 3$. Sesudah tiap iterasi, p sama dengan polinomial parsial yang diramalkan invariannya; sesudah iterasi terakhir, nilainya sama dengan $P(3) = 5$.

Iterasi i	$A[i]$	p Sebelum	Baris 3: $A[i]+3p$	Invarian untuk indeks berikutnya, dievaluasi langsung
3	2	0	$2 + 3 \cdot 0 = 2$	$A[3] = 2$
2	-6	2	$-6 + 3 \cdot 2 = 0$	$A[2] + 3A[3] = -6 + 6 = 0$
1	2	0	$2 + 3 \cdot 0 = 2$	$A[1] + 3A[2] + 9A[3] = 2 - 18 + 18 = 2$
0	-1	2	$-1 + 3 \cdot 2 = 5$	$A[0] + 3A[1] + 9A[2] + 27A[3] = -1 + 6 - 54 + 54 = 5$

Biaya. Baris 3 dijalankan $n + 1$ kali, masing-masing dengan satu perkalian dan satu penjumlahan: $\Theta(n)$. Metode naif menghitung setiap pangkat x^k dari nol dengan k perkalian lalu mengalikannya dengan a_k, sehingga berbiaya $\sum_{k=0}^{n}(k + 1) = \frac{(n+1)(n+2)}{2} = \Theta(n^2)$ perkalian — deret aritmetika lagi.

Jawaban. Invariannya adalah Persamaan (4.8); saat terminasi ia memberi $p = P(x)$. Untuk contohnya, $P(3) = 5$, dihitung dengan 4 perkalian.

Program. **Kode 4.6** menjalankan aturan Horner dengan penelusuran, menjalankan metode naif, dan menghitung banyaknya perkalian pada keduanya.

Kode 4.6 Aturan Horner terhadap evaluasi naif, dengan penelusuran gelung dan hitungan perkalian.

```
#include <cstdio>
#include <vector>
using namespace std;
typedef long long LL;

// HORNER(A, n, x): p = 0; for i = n downto 0: p = A[i] + x*p
LL horner(const vector<LL>& A, LL x, LL& mults) {
    LL p = 0;
    int n = (int)A.size() - 1;
    for (int i = n; i >= 0; --i) {
        // invarian: p = sum_{k=0}^{n-(i+1)} A[k+i+1] * x^k
        p = A[i] + x * p;
        ++mults;
        printf("  i = %d   p = A[%d] + %lld * (previous p) = %lld\n", i, i, x, p);
    }
    return p;
}

// Evaluasi naif: hitung setiap pangkat x^k dari nol.
LL naive(const vector<LL>& A, LL x, LL& mults) {
    LL sum = 0;
    for (int k = 0; k < (int)A.size(); ++k) {
        LL power = 1;
        for (int t = 0; t < k; ++t) { power *= x; ++mults; }   // k perkalian
        sum += A[k] * power; ++mults;                          // satu lagi untuk A[k]*x^k
    }
    return sum;
}

int main() {
    vector<LL> A = {-1, 2, -6, 2};          // P(x) = 2x^3 - 6x^2 + 2x - 1, A[k] =
koefisien x^k
    LL x = 3, mh = 0, mn = 0;
    printf("Horner on P(x) = 2x^3 - 6x^2 + 2x - 1 at x = 3\n");
    LL h = horner(A, x, mh);
    LL v = naive(A, x, mn);
    printf("Horner: P(3) = %lld using %lld multiplications\n", h, mh);
    printf("Naive : P(3) = %lld using %lld multiplications\n", v, mn);
```

Kode 4.6 (Lanjutan) *Aturan Horner terhadap evaluasi naif, dengan penelusuran gelung dan hitungan perkalian.*

```
    // Hitungan operasi untuk derajat lebih besar (semua koefisien 1, x = 1, jadi tanpa
limpahan).
    printf("\n degree n   Horner mults   naive mults\n");
    for (int n : {3, 10, 100, 1000}) {
        vector<LL> B(n + 1, 1);
        LL a = 0, b = 0;
        LL p = 0; for (int i = n; i >= 0; --i) { p = B[i] + 1 * p; ++a; }
        naive(B, 1, b);
        printf(" %8d   %12lld   %11lld\n", n, a, b);
    }
    return 0;
}
```

```
Horner on P(x) = 2x^3 - 6x^2 + 2x - 1 at x = 3
  i = 3   p = A[3] + 3 * (previous p) = 2
  i = 2   p = A[2] + 3 * (previous p) = 0
  i = 1   p = A[1] + 3 * (previous p) = 2
  i = 0   p = A[0] + 3 * (previous p) = 5
Horner: P(3) = 5 using 4 multiplications
Naive : P(3) = 5 using 10 multiplications

 degree n   Horner mults   naive mults
        3              4            10
       10             11            66
      100            101          5151
     1000           1001        501501
```

Penelusurannya mereproduksi **Tabel 4.4**, dan hitungannya adalah $n + 1$ lawan $\frac{(n+1)(n+2)}{2}$: untuk derajat 1000, 1,001 perkalian lawan 501,501.

> **UJI DIRI.** Gelung `r = 1; for i = 1 to n: r = r * x` menghitung x^n. Nyatakan invariannya dan buktikan.
>
> *Jawaban.* Pada awal iterasi berindeks i, $r = x^{i-1}$. Inisialisasi: $i = 1$, $r = 1 = x^0$. Pemeliharaan: $r \cdot x = x^{i-1} \cdot x = x^i$, yaitu invarian untuk indeks berikutnya $i + 1$. Terminasi: gelung berhenti pada $i = n + 1$, tempat invariannya memberi $r = x^n$.

4.5.2 Persoalan Terkerjakan B — menelusuri *insertion sort*

Persoalan. Telusurilah INSERTION-SORT (**Pseudokode 4.4**) pada $A = \langle 8,3,6,3,9,1 \rangle$. Sesudah tiap iterasi gelung luar, berikan lariknya, banyaknya perbandingan kunci $A[i] > key$ yang dievaluasi, dan banyaknya elemen yang digeser. Bandingkan total pergeseran dengan banyaknya *inversi* A — pasangan posisi $p < q$ dengan $A[p] > A[q]$.

Pseudokode 4.4 *Insertion sort*, seperti pada Subbab 2.8.2, dengan indeks yang berjalan dari 1.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      i = j - 1
4      while i > 0 and A[i] > key
5          A[i + 1] = A[i]
6          i = i - 1
7      A[i + 1] = key
```

Yang diuji. Mengikuti sebuah algoritma dengan tepat (Subbab 2.8.3), makna biaya t_j dalam analisis pada Subbab 2.10, dan sifat stabil yang diberikan oleh `>` yang tegas pada baris 4.

Penyelesaian. Kerjakan satu iterasi luar setiap kali. Bagian larik di sebelah kiri posisi j selalu terurut — itulah invarian pada Subbab 2.9.

- $j = 2$, $key = 3$. Bandingkan $A[1] = 8 > 3$: ya, geser 8 ke posisi 2, $i = 0$. Uji $i > 0$ gagal, jadi tak ada perbandingan lagi. Letakkan 3 di posisi 1. Larik $\langle 3,8,6,3,9,1 \rangle$: 1 perbandingan, 1 pergeseran.
- $j = 3$, $key = 6$. $8 > 6$: geser. $3 > 6$: tidak, berhenti. Letakkan 6 di posisi 2. Larik $\langle 3,6,8,3,9,1 \rangle$: 2 perbandingan, 1 pergeseran.
- $j = 4$, $key = 3$. $8 > 3$: geser. $6 > 3$: geser. $3 > 3$: **tidak** — sama tidak berarti lebih besar — berhenti. Letakkan kuncinya di posisi 2. Larik $\langle 3,3,6,8,9,1 \rangle$: 3 perbandingan, 2 pergeseran. Kunci itu berhenti *di belakang* angka 3 yang sama nilainya, sehingga kedua angka 3 mempertahankan urutan aslinya; inilah kestabilan yang dibeli oleh `>` yang tegas.

- $j = 5$, $key = 9$. $8 > 9$: tidak, langsung berhenti. Larik tidak berubah: 1 perbandingan, 0 pergeseran. Elemen yang sudah berada di tempatnya berbiaya satu perbandingan — kasus terbaik Subbab 2.11.1 dalam versi mini.
- $j = 6$, $key = 1$. 9,8,6,3,3 semuanya lebih besar daripada 1: lima pergeseran, lalu $i = 0$ mengakhiri gelung. Larik $\langle 1,3,3,6,8,9 \rangle$: 5 perbandingan, 5 pergeseran. Inilah kasus terburuk untuk satu iterasi: kuncinya menempuh seluruh awalan yang terurut.

Totalnya: $1 + 2 + 3 + 1 + 5 = 12$ perbandingan dan $1 + 1 + 2 + 0 + 5 = 9$ pergeseran.

Inversi. Hitung pasangan $p < q$ dengan $A[p] > A[q]$ pada larik *asli* $\langle 8,3,6,3,9,1 \rangle$, dari kiri ke kanan:

- 8 lebih besar daripada 3,6,3,1 yang datang kemudian: 4 inversi.
- Angka 3 pertama lebih besar daripada 1 yang datang kemudian: 1 (tidak lebih besar daripada angka 3 kedua).
- 6 lebih besar daripada 3,1 yang datang kemudian: 2.
- Angka 3 kedua lebih besar daripada 1 yang datang kemudian: 1.
- 9 lebih besar daripada 1 yang datang kemudian: 1.

Totalnya $4 + 1 + 2 + 1 + 1 = 9$ — tepat sama dengan banyaknya pergeseran. Ini bukan kebetulan: setiap pergeseran pada baris 5 memindahkan elemen yang lebih besar melewati elemen yang lebih kecil yang semula berada di kanannya, yang menghapus tepat satu inversi dan tidak menciptakan inversi baru. Larik terurut ketika tidak ada inversi yang tersisa, sehingga **banyaknya pergeseran sama dengan banyaknya inversi**, selalu. **Tabel 4.5** merangkumnya.

Tabel 4.5 *Insertion sort* pada $\langle 8,3,6,3,9,1 \rangle$. Total pergeserannya, 9, sama dengan banyaknya inversi; t_j menghitung berapa kali uji `while` pada baris 4 dijalankan dan selalu satu lebih banyak daripada pergeserannya.

j	Kunci	Larik sesudah iterasi	Perbandingan	Pergeseran	t_j
2	3	$\langle 3,8,6,3,9,1 \rangle$	1	1	2
3	6	$\langle 3,6,8,3,9,1 \rangle$	2	1	2
4	3	$\langle 3,3,6,8,9,1 \rangle$	3	2	3
5	9	$\langle 3,3,6,8,9,1 \rangle$	1	0	1
6	1	$\langle 1,3,3,6,8,9 \rangle$	5	5	6
		total	**12**	**9**	**14**

Kolom terakhir menghubungkan penelusuran ini dengan analisis pada Subbab 2.10, tempat t_j adalah berapa kali uji `while` pada baris 4 *dijalankan* dalam iterasi j. Setiap pergeseran didahului satu uji yang berhasil, dan gelungnya berakhir dengan tepat satu uji yang gagal — entah $A[i] > key$ bernilai salah, entah $i > 0$ bernilai salah. Jadi t_j selalu sama dengan banyaknya pergeseran ditambah satu: 2,2,3,1,6, berjumlah 14.

Jawaban. Larik akhir $\langle 1,3,3,6,8,9 \rangle$; 12 perbandingan, 9 pergeseran, 9 inversi, $\sum_{j=2}^{6} t_j = 14$.

Program. **Kode 4.7** melakukan penelusuran yang sama dan menghitung inversinya secara terpisah.

Kode 4.7 *Insertion sort* dengan penelusuran per iterasi, dan hitungan inversi terpisah sebagai pembanding.

```
#include <cstdio>
#include <vector>
using namespace std;

void show(const vector<int>& A) {                    // cetak A[1..n]
    printf("[");
    for (size_t t = 1; t < A.size(); ++t) printf("%s%d", t > 1 ? ", " : "", A[t]);
    printf("]");
}

int main() {
    vector<int> A = {0, 8, 3, 6, 3, 9, 1};          // A[0] tak dipakai: indeks berjalan 1..n
seperti pada pseudokode
    int n = (int)A.size() - 1;
    long long totalCmp = 0, totalShift = 0;
    printf("start       "); show(A); printf("\n");
    for (int j = 2; j <= n; ++j) {
        int key = A[j], i = j - 1, cmp = 0, shift = 0;
        while (i > 0) {
            ++cmp;                                   // satu kali uji A[i] > key
            if (A[i] <= key) break;
            A[i + 1] = A[i]; --i; ++shift;
        }
        A[i + 1] = key;
        totalCmp += cmp; totalShift += shift;
        printf("j=%d key=%d  ", j, key); show(A);
        printf("  comparisons=%d shifts=%d\n", cmp, shift);
    }
    // Hitung inversi larik ASLI secara langsung, sebagai pembanding.
    vector<int> B = {0, 8, 3, 6, 3, 9, 1};
    int inv = 0;
    for (int p = 1; p <= n; ++p)
        for (int q = p + 1; q <= n; ++q)
            if (B[p] > B[q]) ++inv;
    printf("total comparisons = %lld, total shifts = %lld, inversions = %d\n",
           totalCmp, totalShift, inv);
    return 0;
}
```

```
start       [8, 3, 6, 3, 9, 1]
j=2 key=3  [3, 8, 6, 3, 9, 1]  comparisons=1 shifts=1
j=3 key=6  [3, 6, 8, 3, 9, 1]  comparisons=2 shifts=1
j=4 key=3  [3, 3, 6, 8, 9, 1]  comparisons=3 shifts=2
j=5 key=9  [3, 3, 6, 8, 9, 1]  comparisons=1 shifts=0
j=6 key=1  [1, 3, 3, 6, 8, 9]  comparisons=5 shifts=5
total comparisons = 12, total shifts = 9, inversions = 9
```

UJI DIRI. Telusurilah *insertion sort* pada $\langle 4,1,3,9,7\rangle$. Berapa pergeserannya, dan berapa inversinya?

Jawaban. Larik sesudah $j = 2,3,4,5$ adalah $\langle 1,4,3,9,7\rangle$, $\langle 1,3,4,9,7\rangle$, $\langle 1,3,4,9,7\rangle$, $\langle 1,3,4,7,9\rangle$. Pergeseran $1 + 1 + 0 + 1 = 3$; perbandingan $1 + 2 + 1 + 2 = 6$. Inversinya adalah (4,1), (4,3), dan (9,7): tiga, sama dengan banyaknya pergeseran.

4.5.3 Persoalan Terkerjakan C — menelusuri pencarian biner

Persoalan. Larik terurut $A[1..15] = \langle 3,8,11,15,19,22,27,31,36,40,44,51,57,62,70\rangle$ dicari dengan BINARY-SEARCH (**Pseudokode 4.5**). Telusurilah pencarian untuk $q = 44$ dan untuk $q = 20$, dengan memberikan $left$, $right$, dan mid pada setiap probe. Berapa probe paling banyak yang mungkin dibutuhkan sebuah pencarian pada larik ini?

Pseudokode 4.5 Pencarian biner, seperti pada Subbab 2.14.3, yang mengembalikan sebuah indeks q atau NIL.

```
BINARY-SEARCH(A, q)
left = 1
right = A.length
while left <= right
    mid = floor((left + right) / 2)
    if A[mid] == q then return mid
    else if A[mid] > q then right = mid - 1
    else left = mid + 1
return NIL
```

Yang diuji. Prosedur pencarian biner dan invarian jendelanya (Subbab 2.14.3), serta batas $\lfloor \lg n \rfloor + 1$ pada banyaknya probe (Subbab 2.14.5).

Penyelesaian. Jendela $[left, right]$ adalah tempat q mungkin masih berada. Pada tiap probe hitung mid, bandingkan, lalu buang separuhnya.

Pencarian untuk $q = 44$.

1. $left = 1, right = 15$. $mid = \lfloor 16/2 \rfloor = 8$. $A[8] = 31 < 44$, jadi q ada di sebelah kanan: $left = 9$.
2. $left = 9, right = 15$. $mid = \lfloor 24/2 \rfloor = 12$. $A[12] = 51 > 44$: $right = 11$.
3. $left = 9, right = 11$. $mid = \lfloor 20/2 \rfloor = 10$. $A[10] = 40 < 44$: $left = 11$.
4. $left = 11, right = 11$. $mid = 11$. $A[11] = 44$. Ditemukan; kembalikan 11.

Pencarian untuk $q = 20$.

1. $left = 1, right = 15, mid = 8$. $A[8] = 31 > 20$: $right = 7$.
2. $left = 1, right = 7, mid = \lfloor 8/2 \rfloor = 4$. $A[4] = 15 < 20$: $left = 5$.
3. $left = 5, right = 7, mid = 6$. $A[6] = 22 > 20$: $right = 5$.
4. $left = 5, right = 5, mid = 5$. $A[5] = 19 < 20$: $left = 6$.
5. Kini $left = 6 > right = 5$: jendelanya kosong, jadi kembalikan NIL.

Invariannya menjelaskan NIL itu. Pada akhirnya jendela kosong, dan 20 seharusnya duduk di antara $A[5] = 19$ dan $A[6] = 22$ — tempat yang sama sekali tidak memiliki posisi. **Tabel 4.6** menjajarkan kedua pencarian itu.

Tabel 4.6 Pencarian biner pada larik 15 elemen untuk kunci yang ada (44) dan kunci yang tidak ada (20). Jendelanya menyusut $15 \to 7 \to 3 \to 1 \to 0$; empat probe cukup dalam setiap kasus.

q	Probe	*left*	*right*	*mid*	$A[mid]$	Keputusan	Ukuran jendela sesudahnya
44	1	1	15	8	31	terlalu kecil: $left = 9$	7
44	2	9	15	12	51	terlalu besar: $right = 11$	3
44	3	9	11	10	40	terlalu kecil: $left = 11$	1
44	4	11	11	11	44	ditemukan, kembalikan 11	—
20	1	1	15	8	31	terlalu besar: $right = 7$	7
20	2	1	7	4	15	terlalu kecil: $left = 5$	3
20	3	5	7	6	22	terlalu besar: $right = 5$	1
20	4	5	5	5	19	terlalu kecil: $left = 6$	0

Kasus terburuk. Lihatlah kolom terakhir. Dengan $15 = 2^4 - 1$ elemen, setiap probe yang gagal membuang mid dan menyisakan salah satu dari dua paruh yang masing-masing tepat 7; lalu 3; lalu 1; lalu 0. Secara umum jendela berukuran w menyisakan paling banyak $\lfloor w/2 \rfloor$, sehingga sesudah p probe paling banyak tersisa $\lfloor n/2^p \rfloor$, dan pencarian berakhir sesudah paling banyak $\lfloor \lg n \rfloor + 1$ probe. Untuk $n = 15$: $\lfloor \lg 15 \rfloor + 1 = 3 + 1 = 4$, karena $2^3 = 8 \leq 15 < 16 = 2^4$.

Jawaban. $q = 44$ ditemukan pada indeks 11 sesudah 4 probe; $q = 20$ dilaporkan tidak ada sesudah 4 probe; tidak ada pencarian pada larik ini yang membutuhkan lebih dari 4.

Program. **Kode 4.8** mencetak penelusuran yang sama.

Kode 4.8 Pencarian biner dengan satu baris keluaran per probe, pada larik Persoalan Terkerjakan C.

```
#include <cstdio>
#include <vector>
using namespace std;

// BINARY-SEARCH(A, q) pada A[1..n]; mencetak satu baris per probe. Mengembalikan 0 untuk
NIL.
int binarySearch(const vector<int>& A, int q) {
    int left = 1, right = (int)A.size() - 1, probe = 0;
    while (left <= right) {
        int mid = left + (right - left) / 2;      // nilainya sama dengan (left+right)/2,
tanpa limpahan
        ++probe;
        printf("  probe %d: left=%2d right=%2d mid=%2d A[mid]=%2d  ", probe, left, right,
mid, A[mid]);
        if (A[mid] == q) { printf("equal -> found at %d\n", mid); return mid; }
        if (A[mid] > q) { right = mid - 1; printf("too big   -> right=%d\n", right); }
        else            { left = mid + 1;  printf("too small -> left=%d\n", left); }
    }
    printf("  window empty (left=%d > right=%d) -> NIL\n", left, right);
    return 0;
}

int main() {
    vector<int> A = {0, 3, 8, 11, 15, 19, 22, 27, 31, 36, 40, 44, 51, 57, 62, 70};  // A[0]
tak dipakai
    for (int q : {44, 20}) {
        printf("search for %d\n", q);
        int r = binarySearch(A, q);
        printf("  result: %d\n", r);
    }
    return 0;
}
```

```
search for 44
  probe 1: left= 1 right=15 mid= 8 A[mid]=31  too small -> left=9
  probe 2: left= 9 right=15 mid=12 A[mid]=51  too big   -> right=11
  probe 3: left= 9 right=11 mid=10 A[mid]=40  too small -> left=11
  probe 4: left=11 right=11 mid=11 A[mid]=44  equal -> found at 11
  result: 11
search for 20
  probe 1: left= 1 right=15 mid= 8 A[mid]=31  too big   -> right=7
  probe 2: left= 1 right= 7 mid= 4 A[mid]=15  too small -> left=5
  probe 3: left= 5 right= 7 mid= 6 A[mid]=22  too big   -> right=5
  probe 4: left= 5 right= 5 mid= 5 A[mid]=19  too small -> left=6
  window empty (left=6 > right=5) -> NIL
  result: 0
```

UJI DIRI. Pada larik yang sama, telusurilah pencarian untuk $q = 70$ dan $q = 3$.

Jawaban. Untuk 70: $mid = 8$ (31, ke kanan), $mid = 12$ (51, ke kanan), $mid = 14$ (62, ke kanan), $mid = 15$ (70, ditemukan) — 4 probe. Untuk 3: $mid = 8$ (31, ke kiri), $mid = 4$ (15, ke kiri), $mid = 2$ (8, ke kiri), $mid = 1$ (3, ditemukan) — 4 probe. Kedua ujung larik sama mahalnya dengan posisi lain, tidak pernah lebih.

4.6 Persoalan Terkerjakan tentang Pencacahan dan Laju Pertumbuhan

Kelompok kedua persoalan terkerjakan melatih aritmetika efisiensi: menghitung dengan tepat berapa langkah yang ditempuh sebuah gelung, membuktikan pernyataan asimtotik dengan konstanta yang eksplisit, dan menempatkan fungsi-fungsi menurut urutan laju pertumbuhannya.

4.6.1 Persoalan Terkerjakan D — hitungan langkah yang tepat dari gelung rangkap tiga

Persoalan. Berapa kali baris 5 dari LOOPS (**Pseudokode 4.6**) dijalankan, sebagai fungsi eksak dari n? Berikan bentuk tertutupnya dan kelas Θ-nya.

Pseudokode 4.6 Gelung rangkap tiga yang baris terdalamnya hendak dihitung secara tepat.

```
LOOPS(n)
1  count = 0
2  for i = 1 to n
3      for j = i to n
4          for k = i to j
5              count = count + 1
6  return count
```

Yang diuji. Mengubah gelung bersarang menjadi jumlah bersarang (Subbab 2.10), dan mengevaluasi jumlah-jumlah itu dengan rumus pada Subbab 3.9.

Penyelesaian. Bekerjalah dari dalam ke luar. Setiap gelung menjadi sebuah jumlah, dan badan gelung terdalam menyumbang 1 per eksekusi.

Gelung k. Untuk i dan j yang tetap, k berjalan dari i sampai j: itu $j - i + 1$ nilai. (Gelung dari a sampai b berjalan $b - a + 1$ kali bila $b \geq a$. Periksa dengan $a = b$: sekali.)

Gelung j. Untuk i yang tetap, j berjalan dari i sampai n, dan gelung k menyumbang $j - i + 1$ setiap kali. Ketika j bergerak $i, i + 1, \ldots, n$, besaran $t = j - i + 1$ bergerak $1,2, \ldots, n - i + 1$. Jadi gelung j menyumbang

$$\sum_{j=i}^{n}(j - i + 1) = \sum_{t=1}^{n-i+1} t = \frac{(n - i + 1)(n - i + 2)}{2},$$

menurut deret aritmetika dengan $n - i + 1$ menggantikan n.

Gelung i. Jumlahkan atas i dari 1 sampai n. Sulihkan $m = n - i + 1$: ketika i berjalan $1,2, \ldots, n$, m berjalan $n, n - 1, \ldots, 1$ — nilai-nilai yang sama dalam urutan terbalik, dan urutan tidak berpengaruh pada sebuah jumlah. Jadi totalnya

$$\sum_{i=1}^{n}\frac{(n - i + 1)(n - i + 2)}{2} = \sum_{m=1}^{n}\frac{m(m + 1)}{2} = \frac{1}{2}\left(\sum_{m=1}^{n} m^2 + \sum_{m=1}^{n} m\right).$$

Dua jumlah baku. $\sum_{m=1}^{n} m = \frac{n(n+1)}{2}$ adalah Persamaan (4.1). Untuk kuadrat, kartu rujukan pada Subbab 3.9 memberi $\sum_{m=1}^{n} m^2 = \frac{n(n+1)(2n+1)}{6}$; karena bab ini bertujuan memantapkan, mari kita turunkan juga, secara teleskopik. Jabarkan $(m + 1)^3 - m^3 = 3m^2 + 3m + 1$ dan jumlahkan kedua ruasnya untuk $m = 1$ sampai n. Ruas kiri meneleskop menjadi $(n + 1)^3 - 1$. Jadi

$$(n+1)^3 - 1 = 3\sum_{m=1}^{n} m^2 + 3 \cdot \frac{n(n+1)}{2} + n.$$

Pindahkan suku-suku yang diketahui ke seberang, dengan memakai $(n+1)^3 - 1 - n = (n+1)^3 - (n+1) = (n+1)((n+1)^2 - 1) = (n+1)(n^2+2n)$:

$$3\sum_{m=1}^{n} m^2 = (n+1)(n^2+2n) - \frac{3n(n+1)}{2} = (n+1)\left(n^2 + 2n - \frac{3n}{2}\right)$$
$$= (n+1)\left(n^2 + \frac{n}{2}\right) = \frac{n(n+1)(2n+1)}{2},$$

dan membagi dengan 3 memberi $\sum m^2 = \frac{n(n+1)(2n+1)}{6}$.

Menyatukan semuanya.

$$\frac{1}{2}\left(\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}\right) = \frac{1}{2} \cdot \frac{n(n+1)}{6}((2n+1)+3)$$
$$= \frac{1}{2} \cdot \frac{n(n+1)(2n+4)}{6} = \frac{n(n+1)(n+2)}{6}.$$

$$\text{eksekusi baris 5} = \sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 = \frac{n(n+1)(n+2)}{6} \tag{4.9}$$

Periksa dengan tangan pada $n = 4$. Tabel 4.7 mendaftar setiap pasangan (i, j) dan banyaknya nilai k yang dihasilkannya.

Tabel 4.7 Menghitung eksekusi terdalam LOOPS untuk $n = 4$ dengan tangan. Subtotalnya adalah bilangan segitiga 10,6,3,1, dan jumlahnya $\frac{4 \cdot 5 \cdot 6}{6} = 20$.

i	Sumbangan *j−i+1* untuk *j=i,…,4*	Subtotal
1	$1+2+3+4$	10
2	$1+2+3$	6
3	$1+2$	3
4	1	1
	total	**20**

Rumusnya memberi $\frac{4 \cdot 5 \cdot 6}{6} = \frac{120}{6} = 20$ ✓. Tabel itu juga memperlihatkan struktur jawabannya sekilas: tiap baris adalah bilangan segitiga, dan menjumlahkan bilangan-bilangan segitiga menghasilkan apa yang disebut bilangan *tetrahedral* 1,4,10,20,35, … — banyaknya jeruk dalam sebuah limas segitiga.

Asimtotik. $\frac{n(n+1)(n+2)}{6} = \frac{n^3+3n^2+2n}{6}$. Suku utamanya $\frac{n^3}{6}$, jadi hitungannya $\Theta(n^3)$, dengan konstanta utama $\frac{1}{6}$. Dibandingkan gelung rangkap tiga biasa yang setiap indeksnya berjalan dari 1 sampai n, yang dijalankan n^3 kali, LOOPS mengerjakan sekitar seperenam pekerjaannya — orde pertumbuhan yang sama, konstanta yang jauh berbeda.

Jawaban. Baris 5 dijalankan tepat $\frac{n(n+1)(n+2)}{6}$ kali, yaitu $\Theta(n^3)$.

Program. **Kode 4.9** menjalankan gelungnya dan membandingkan hasilnya dengan Persamaan (4.9).

Kode 4.9 Menghitung eksekusi baris terdalam LOOPS dan membandingkannya dengan $\frac{n(n+1)(n+2)}{6}$.

```
#include <cstdio>
#include <initializer_list>
using namespace std;
typedef long long LL;

// Menjalankan baris 5 LOOPS(n) dan menghitung berapa kali ia dijalankan.
LL loops(LL n) {
    LL count = 0;
    for (LL i = 1; i <= n; ++i)
        for (LL j = i; j <= n; ++j)
            for (LL k = i; k <= j; ++k)
                count = count + 1;
    return count;
}

int main() {
    printf("    n        count   n(n+1)(n+2)/6\n");
    for (LL n : {1LL, 2LL, 3LL, 4LL, 5LL, 6LL, 10LL, 100LL, 500LL}) {
        LL formula = n * (n + 1) * (n + 2) / 6;
        printf("%5lld  %11lld  %14lld  %s\n", n, loops(n), formula,
               loops(n) == formula ? "equal" : "DIFFERENT");
    }
    return 0;
}
```

```
    n        count   n(n+1)(n+2)/6
    1            1              1  equal
    2            4              4  equal
    3           10             10  equal
    4           20             20  equal
    5           35             35  equal
    6           56             56  equal
   10          220            220  equal
  100       171700         171700  equal
  500     20958500       20958500  equal
```

UJI DIRI. Berapa kali badan `for i = 1 to n: for j = i + 1 to n: body` dijalankan? Dan badan `i = n; while i > 1: i = floor(i / 2)`?

Jawaban. Yang pertama: untuk tiap i gelung dalamnya berjalan $n - i$ kali, jadi totalnya $\sum_{i=1}^{n}(n-i) = (n-1) + (n-2) + \cdots + 0 = \frac{n(n-1)}{2}$, yaitu banyaknya pasangan. Yang kedua: tiap iterasi membagi dua i (dibulatkan ke bawah), dan gelungnya berhenti ketika i mencapai 1, yang memakan tepat $\lfloor \lg n \rfloor$ iterasi — untuk $n = 10$, $10 \to 5 \to 2 \to 1$, tiga iterasi, dan $\lfloor \lg 10 \rfloor = 3$.

4.6.2 Persoalan Terkerjakan E — pernyataan asimtotik dengan konstanta yang eksplisit

Persoalan. Buktikan atau sangkal setiap klaim, dengan memberikan konstanta eksplisit bila klaimnya benar dan sebuah kontradiksi bila klaimnya salah.

(a) $5n^2 - 3n + 12 = \Theta(n^2)$. (b) $2^{n+1} = O(2^n)$. (c) $2^{2n} = O(2^n)$. (d) $\lg(n!) = \Theta(n \lg n)$.

Yang diuji. Definisi-definisi Bab 3, dipakai secara harfiah. Ingat bahwa $f(n) = \Theta(g(n))$ berarti ada konstanta positif c_1, c_2, dan n_0 dengan

$$0 \le c_1\, g(n) \le f(n) \le c_2\, g(n) \quad \text{untuk semua } n \ge n_0 \tag{4.10}$$

dan $f(n) = O(g(n))$ adalah separuh kanan Persamaan (4.10) saja. Sebuah bukti harus *menyebutkan* konstantanya; sebuah sangkalan harus memperlihatkan bahwa tidak ada konstanta yang bisa berhasil.

Penyelesaian (a). *Batas atas.* Kita menginginkan $5n^2 - 3n + 12 \le c_2 n^2$. Cobalah $c_2 = 5$: ketaksamaannya menjadi $-3n + 12 \le 0$, yaitu $12 \le 3n$, yaitu $n \ge 4$. Jadi $c_2 = 5$ berhasil untuk semua $n \ge 4$.

Batas bawah. Kita menginginkan $c_1 n^2 \le 5n^2 - 3n + 12$. Cobalah $c_1 = 4$: ketaksamaannya menjadi $0 \le n^2 - 3n + 12$. Lengkapkan kuadratnya: $n^2 - 3n + 12 = \left(n - \frac{3}{2}\right)^2 - \frac{9}{4} + 12 = \left(n - \frac{3}{2}\right)^2 + \frac{39}{4}$, yang positif untuk setiap n. Jadi $c_1 = 4$ berhasil untuk semua $n \ge 1$.

Bersama-sama: $c_1 = 4$, $c_2 = 5$, $n_0 = 4$ memenuhi Persamaan (4.10). Ambang n_0 bekerja sungguhan di sini: pada $n = 3$ ketaksamaan atasnya gagal, karena $f(3) = 45 - 9 + 12 = 48 > 45 = 5 \cdot 3^2$. Klaim (a) **benar**.

Penyelesaian (b). $2^{n+1} = 2 \cdot 2^n$, jadi $2^{n+1} \le c \cdot 2^n$ berlaku dengan $c = 2$ untuk setiap $n \ge 1$. Klaim (b) **benar** (bahkan $2^{n+1} = \Theta(2^n)$, dengan $c_1 = c_2 = 2$). Pergeseran konstan pada *argumen* eksponen, $2^n \to 2^{n+1}$, hanyalah faktor konstan pada nilainya.

Penyelesaian (c). Andaikan, untuk mencapai kontradiksi, ada konstanta $c > 0$ dan n_0 dengan $2^{2n} \le c \cdot 2^n$ untuk semua $n \ge n_0$. Karena $2^{2n} = 2^n \cdot 2^n$, bagi kedua ruas dengan $2^n > 0$: $2^n \le c$ untuk semua $n \ge n_0$. Tetapi 2^n tumbuh tanpa batas. Ambil $n = \max(n_0, \lfloor \lg c \rfloor + 1)$; maka $n \ge n_0$ dan $n > \lg c$, sehingga $2^n > c$ — kontradiksi. Klaim (c) **salah**. Faktor konstan *di dalam eksponen*, $2^n \to 2^{2n} = 4^n$, bukan faktor konstan pada nilainya; ia mengubah bilangan pokoknya.

Penyelesaian (d). *Batas atas.* $n! = 1 \cdot 2 \cdot 3 \cdots n$ adalah hasil kali n faktor yang tak satu pun lebih besar daripada n, jadi $n! \le n^n$. Mengambil logaritma (yang mempertahankan urutan), $\lg(n!) \le \lg(n^n) = n \lg n$. Jadi $c_2 = 1$, untuk semua $n \ge 1$.

Batas bawah. Pertahankan hanya separuh atas faktor-faktornya. Faktor $\lceil n/2 \rceil, \ldots, n$ berjumlah $n - \lceil n/2 \rceil + 1 \ge n/2$, dan masing-masing paling kecil $n/2$; faktor yang dibuang paling kecil 1. Jadi $n! \ge (n/2)^{n/2}$, dan

$$\lg(n!) \ge \frac{n}{2} \lg \frac{n}{2} = \frac{n}{2}(\lg n - 1).$$

Kita ingin ini paling kecil $c_1\, n \lg n$ dengan $c_1 = \frac{1}{4}$:

$$\frac{n}{2}(\lg n - 1) \ge \frac{n}{4} \lg n \Leftrightarrow 2(\lg n - 1) \ge \lg n \Leftrightarrow \lg n \ge 2 \Leftrightarrow n \ge 4.$$

Jadi $c_1 = \frac{1}{4}$, $c_2 = 1$, $n_0 = 4$. Klaim (d) **benar**.

Jawaban. (a) benar, $c_1 = 4$, $c_2 = 5$, $n_0 = 4$. (b) benar, $c = 2$, $n_0 = 1$. (c) salah. (d) benar, $c_1 = \frac{1}{4}$, $c_2 = 1$, $n_0 = 4$.

Program. Sebuah program tidak dapat membuktikan pernyataan asimtotik — ia hanya dapat memeriksa berhingga banyak n — tetapi program adalah cara yang sangat baik untuk menangkap konstanta yang keliru. **Kode 4.10** menguji konstanta (a) dan (d) untuk setiap n dari 4 sampai 10^6, dan untuk (c) ia mencari, bagi tiga nilai c, n pertama tempat 4^n lolos dari $c \cdot 2^n$. Program itu menghitung $\lg(n!)$ dengan `lgamma`, logaritma fungsi gamma, karena $\Gamma(n+1) = n!$ sedangkan $n!$ sendiri sudah melimpah pada $n = 21$.

Kode 4.10 Pemeriksaan numerik saksi c_1, c_2, n_0 untuk klaim (a) dan (d), serta titik lolos pada klaim (c).

```
#include <cstdio>
#include <initializer_list>
#include <cmath>
using namespace std;
typedef long long LL;

int main() {
    // (a) 4n^2 <= 5n^2 - 3n + 12 <= 5n^2 untuk semua n >= 4
    printf("(a)  n   4n^2   f(n)=5n^2-3n+12   5n^2   upper holds?\n");
    for (LL n = 1; n <= 6; ++n) {
        LL f = 5 * n * n - 3 * n + 12;
        printf("   %2lld  %5lld  %16lld  %5lld   %s\n", n, 4 * n * n, f, 5 * n * n,
               f <= 5 * n * n ? "yes" : "no");
    }
    LL bad = 0;
    for (LL n = 4; n <= 1000000; ++n) {
        LL f = 5 * n * n - 3 * n + 12;
        if (!(4 * n * n <= f && f <= 5 * n * n)) ++bad;
    }
    printf("     violations for 4 <= n <= 10^6: %lld\n", bad);

    // (c) 4^n <= c * 2^n gagal begitu 2^n > c
    printf("(c)  c          first n with 4^n > c*2^n\n");
    for (double c : {10.0, 1000.0, 1000000.0}) {
        int n = 0;
        while (pow(4.0, n) <= c * pow(2.0, n)) ++n;
        printf("     %-9.0f  %d\n", c, n);
    }

    // (d) (1/4) n lg n <= lg(n!) <= n lg n untuk n >= 4
    printf("(d)  n        lg(n!)      (1/4)n lg n     n lg n\n");
    for (double n : {4.0, 16.0, 1024.0, 1048576.0}) {
        double lgfact = lgamma(n + 1.0) / log(2.0);    // lg(n!) tanpa menghitung n!
        printf("     %-8.0f %12.2f  %12.2f  %12.2f\n", n, lgfact, 0.25 * n * log2(n), n *
log2(n));
    }
    int badd = 0;
    for (int n = 4; n <= 1000000; ++n) {
        double lgfact = lgamma(n + 1.0) / log(2.0), nl = n * log2((double)n);
        if (!(0.25 * nl <= lgfact && lgfact <= nl)) ++badd;
    }
    printf("     violations for 4 <= n <= 10^6: %d\n", badd);
    return 0;
}
```

```
(a)  n   4n^2   f(n)=5n^2-3n+12   5n^2   upper holds?
     1      4                14      5   no
     2     16                26     20   no
     3     36                48     45   no
     4     64                80     80   yes
     5    100               122    125   yes
     6    144               174    180   yes
      violations for 4 <= n <= 10^6: 0
(c)  c          first n with 4^n > c*2^n
     10         4
     1000       10
     1000000    20
(d)  n        lg(n!)      (1/4)n lg n     n lg n
     4               4.58          2.00          8.00
     16             44.25         16.00         64.00
     1024         8769.01       2560.00      10240.00
     1048576   19458755.93    5242880.00   20971520.00
     violations for 4 <= n <= 10^6: 0
```

Blok pertama memperlihatkan batas atas (a) gagal pada $n = 1,2,3$ dan berlaku mulai $n = 4$, persis seperti kata aljabarnya. Titik lolos pada (c) adalah $\lfloor \lg c \rfloor + 1$: 4 untuk $c = 10$, 10 untuk $c = 1000$, 20 untuk $c = 10^6$ — berapa pun c yang Anda pilih, 2^{2n} akan lolos.

> **UJI DIRI.** Buktikan $(n+3)^2 = \Theta(n^2)$ dengan konstanta yang eksplisit.
>
> *Jawaban.* Bawah: $(n+3)^2 \geq n^2$ untuk semua $n \geq 1$, jadi $c_1 = 1$. Atas: untuk $n \geq 3$, $n + 3 \leq 2n$, sehingga $(n+3)^2 \leq 4n^2$, dan $c_2 = 4$. Jadi $c_1 = 1$, $c_2 = 4$, $n_0 = 3$.

4.6.3 Persoalan Terkerjakan F — mengurutkan tiga belas fungsi menurut laju pertumbuhan

Persoalan. Susunlah fungsi-fungsi berikut dalam urutan laju pertumbuhan yang menaik, dengan mengelompokkan fungsi-fungsi yang saling Θ:

$$100,\quad \lg n,\quad \lg^2 n,\quad \sqrt{n},\quad 2^{\lg n},\quad n\lg n,\quad \lg(n!),\quad n^2,\quad 4^{\lg n},\quad n^3,\quad (3/2)^n,\quad 2^n,\quad n!.$$

Yang diuji. Hierarki laju pertumbuhan pada Subbab 2.12.4, fakta-fakta tentang logaritma, eksponensial, dan faktorial pada Subbab 3.8, serta kebiasaan menyederhanakan fungsi sebelum membandingkannya.

Penyelesaian. Pertama, *sederhanakan* dua fungsi yang menyamar, dengan memakai identitas $a^{\lg n} = n^{\lg a}$ (ambil lg kedua ruas: keduanya $\lg n \cdot \lg a$):

- $2^{\lg n} = n^{\lg 2} = n^1 = n$.
- $4^{\lg n} = n^{\lg 4} = n^2$. Jadi $4^{\lg n}$ dan n^2 bukan hanya saling Θ; keduanya sama.

Selain itu, menurut Persoalan Terkerjakan E(d), $\lg(n!) = \Theta(n\lg n)$. Sekarang urutkan daftarnya, dengan membenarkan setiap langkah melalui limit rasionya: jika $f(n)/g(n) \to 0$ maka $f = o(g)$ dan f datang tegas lebih dahulu.

1. 100 adalah konstanta: $\Theta(1)$.
2. $\lg n \to \infty$, jadi konstanta datang lebih dahulu.
3. $\frac{\lg n}{\lg^2 n} = \frac{1}{\lg n} \to 0$, jadi $\lg n$ mendahului $\lg^2 n$.

4. $\lg^2 n = o(\sqrt{n})$: setiap pangkat logaritma dikalahkan oleh setiap pangkat positif n (Subbab 3.8.6). Menyulih $n = 2^m$ membuatnya gamblang: $\lg^2 n = m^2$ sedangkan $\sqrt{n} = 2^{m/2}$, polinomial melawan eksponensial dalam m.
5. $\frac{\sqrt{n}}{n} = \frac{1}{\sqrt{n}} \to 0$, jadi $\sqrt{n}$ mendahului $n = 2^{\lg n}$.
6. $\frac{n}{n\lg n} = \frac{1}{\lg n} \to 0$, jadi n mendahului $n\lg n$, yang seri dengan $\lg(n!)$.
7. $\frac{n\lg n}{n^2} = \frac{\lg n}{n} \to 0$, jadi berikutnya datang n^2 dan kembarannya $4^{\lg n}$.
8. $\frac{n^2}{n^3} = \frac{1}{n} \to 0$: lalu n^3.
9. $n^3 = o((3/2)^n)$: setiap polinomial dikalahkan oleh setiap eksponensial dengan bilangan pokok lebih dari 1 (Subbab 3.8.5).
10. $\frac{(3/2)^n}{2^n} = \left(\frac{3}{4}\right)^n \to 0$: bilangan pokok yang lebih kecil kalah.
11. $\frac{n!}{2^n} = \frac{1}{2} \cdot \frac{2}{2} \cdot \frac{3}{2} \cdots \frac{n}{2} \geq \frac{1}{2}\left(\frac{3}{2}\right)^{n-2} \to \infty$ untuk $n \geq 2$, karena dua faktor pertama berhasil kali $\frac{1}{2}$ dan masing-masing dari $n - 2$ faktor sisanya paling kecil $\frac{3}{2}$. Jadi 2^n mendahului $n!$.

Tabel 4.8 mencatat urutannya.

Tabel 4.8 Ketiga belas fungsi pada Persoalan Terkerjakan F dalam urutan laju pertumbuhan yang menaik. Fungsi pada baris yang sama saling Θ; kolom terakhir membenarkan setiap langkah naik.

Peringkat	Fungsi	Kelas yang disederhanakan	Alasan ia mengalahkan baris di atasnya
1	100	$\Theta(1)$	—
2	$\lg n$	$\Theta(\lg n)$	tumbuh tanpa batas
3	$\lg^2 n$	$\Theta(\lg^2 n)$	rasio $\lg n \to \infty$
4	$\sqrt{n}$	$\Theta(n^{1/2})$	polilog $= o(n^{\varepsilon})$
5	$2^{\lg n}$	$= n$	rasio $\sqrt{n} \to \infty$
6	$n\lg n, \lg(n!)$	$\Theta(n\lg n)$	rasio $\lg n \to \infty$
7	$n^2, 4^{\lg n}$	$= n^2$	rasio $n/\lg n \to \infty$
8	n^3	$\Theta(n^3)$	rasio $n \to \infty$
9	$(3/2)^n$	eksponensial	eksponensial mengalahkan polinomial
10	2^n	eksponensial	rasio $(4/3)^n \to \infty$
11	$n!$	$\Theta(\sqrt{n}\,(n/e)^n)$	rasio $\geq \frac{1}{2}(3/2)^{n-2} \to \infty$

Jawaban. $100 \prec \lg n \prec \lg^2 n \prec \sqrt{n} \prec 2^{\lg n} \prec n\lg n \asymp \lg(n!) \prec n^2 = 4^{\lg n} \prec n^3 \prec (3/2)^n \prec 2^n \prec n!$, dengan $\prec$ berarti "adalah *little-o* dari" dan $\asymp$ berarti "adalah Θ dari".

Program. Urutan asimtotik menggambarkan n yang *besar*. Apa yang terjadi pada ukuran-ukuran yang benar-benar Anda temui? **Kode 4.11** mencetak $\lg f(n)$ — banyaknya digit biner $f(n)$ — untuk tiap fungsi pada $n = 2^4, 2^8, 2^{16}, 2^{32}$. Mencetak logaritmanya membuat $n!$ pun tetap muat di halaman.

Kode 4.11 Logaritma biner ketiga belas fungsi pada empat ukuran, yang memperlihatkan di mana urutan asimtotik sudah dan belum berlaku.

```
#include <cstdio>
#include <cmath>
using namespace std;

// Mencetak lg f(n) -- banyaknya digit biner f(n) -- supaya n! pun muat di halaman.
int main() {
    const char* name[] = {"100", "lg n", "lg^2 n", "sqrt n", "2^(lg n)", "n lg n",
"lg(n!)",
                          "n^2", "4^(lg n)", "n^3", "(3/2)^n", "2^n", "n!"};
    double ns[] = {16.0, 256.0, 65536.0, 4294967296.0};
    printf("%-10s", "f(n)");
    for (double n : ns) printf("   n=2^%-6.0f", log2(n));
    printf("\n");
    for (int f = 0; f < 13; ++f) {
        printf("%-10s", name[f]);
        for (double n : ns) {
            double L = log2(n), v;                      // v = lg f(n)
            switch (f) {
                case 0:  v = log2(100.0); break;
                case 1:  v = log2(L); break;
                case 2:  v = 2 * log2(L); break;
                case 3:  v = L / 2; break;
                case 4:  v = L; break;                  // 2^(lg n) = n
                case 5:  v = L + log2(L); break;
                case 6:  v = log2(lgamma(n + 1) / log(2.0)); break;
                case 7:  v = 2 * L; break;
                case 8:  v = 2 * L; break;              // 4^(lg n) = n^2
                case 9:  v = 3 * L; break;
                case 10: v = n * log2(1.5); break;
                case 11: v = n; break;
                default: v = lgamma(n + 1) / log(2.0); break;
            }
            printf("   %12.4g", v);
        }
        printf("\n");
    }
    return 0;
}
```

```
f(n)          n=2^4         n=2^8         n=2^16        n=2^32
100                6.644         6.644         6.644         6.644
lg n                   2             3             4             5
lg^2 n                 4             6             8            10
sqrt n                 2             4             8            16
2^(lg n)               4             8            16            32
n lg n                 6            11            20            37
lg(n!)             5.468         10.72         19.86         36.93
n^2                    8            16            32            64
4^(lg n)               8            16            32            64
n^3                   12            24            48            96
(3/2)^n            9.359         149.8     3.834e+04     2.512e+09
2^n                   16           256     6.554e+04     4.295e+09
n!                44.25          1684      9.54e+05     1.312e+11
```

Bacalah kolom-kolomnya, bukan hanya baris-barisnya. Pada $n = 16$, fungsi yang "paling lambat", konstanta $100 \approx 2^{6.64}$, lebih besar daripada $\lg^2 n = 2^4$, lebih besar daripada $\sqrt{n} = 2^2$, bahkan lebih besar daripada $n = 2^4$ sendiri; dan $(3/2)^{16} \approx 2^{9.36}$ masih lebih kecil daripada $n^3 = 2^{12}$. Pada $n = 2^{16}$, $\lg^2 n$ dan $\sqrt{n}$ tepat sama, keduanya $2^8 = 256$ — sebelum titik itu logaritma kuadrat adalah yang lebih besar. Baru pada $n = 2^{32}$ setiap baris lebih besar daripada baris di atasnya. Urutan asimtotik adalah pernyataan tentang jangka panjang, dan jangka panjang bisa dimulai lebih lambat daripada yang Anda kira; itulah sebabnya Bab 3 bersikeras pada ambang n_0.

> **UJI DIRI.** Di mana tempat $n^{1/\lg n}$ dan $n^{\lg 7}$ dalam urutan itu?

Jawaban. $n^{1/\lg n} = (2^{\lg n})^{1/\lg n} = 2^1 = 2$, sebuah konstanta: ia bergabung dengan baris 1 bersama 100. Dan $\lg 7 \approx 2.807$, sehingga $n^{\lg 7}$ duduk tegas di antara n^2 dan n^3 — itulah waktu jalan perkalian matriks Strassen, yang di Bab 3 muncul sebagai rekurensi $T(n) = 7T(n/2) + n^2$.

4.7 Persoalan Terkerjakan tentang Rekurensi

Bab 3 memberi tiga metode untuk memecahkan rekurensi: substitusi, pohon rekursi, dan Teorema Master. Ketiga persoalan terkerjakan dalam subbab ini masing-masing memakai satu metode, lalu memeriksa silang jawabannya dengan metode kedua dan dengan program yang mengevaluasi rekurensinya secara eksak. Kebiasaan itu — pecahkan dengan satu cara, tegaskan dengan cara lain — adalah perlindungan paling andal terhadap kekhilafan kecil yang mudah terjadi pada rekurensi.

4.7.1 Persoalan Terkerjakan G — metode substitusi, dan hipotesis yang harus diperkuat

Persoalan. (a) Buktikan bahwa $T(n) = T(\lfloor n/2 \rfloor) + n$ dengan $T(1) = 1$ adalah $\Theta(n)$. (b) Buktikan bahwa

$$U(n) = 2\,U(\lfloor n/2 \rfloor) + 1, \qquad U(1) = 1 \tag{4.11}$$

adalah $O(n)$, dan jelaskan mengapa hipotesis induksi yang paling wajar gagal.

Yang diuji. Metode substitusi pada Subbab 3.11: terka bentuknya, buktikan dengan induksi, tangani kasus dasarnya, dan — ketika induksinya nyaris berhasil — perkuat hipotesisnya alih-alih melemahkan klaimnya.

Penyelesaian (a). *Batas atas: terka $T(n) \le 2n$.* Kasus dasar: $T(1) = 1 \le 2 \cdot 1$. ✓ Langkah induksi: misalkan $n \ge 2$ dan andaikan $T(m) \le 2m$ untuk setiap $m < n$. Karena $\lfloor n/2 \rfloor < n$,

$$T(n) = T(\lfloor n/2 \rfloor) + n \le 2\lfloor n/2 \rfloor + n \le 2 \cdot \frac{n}{2} + n = 2n. \ \checkmark$$

Batas bawah: $T(n) \ge n$. Setiap nilai T positif (bilangan positif ditambah n yang positif), sehingga $T(n) = T(\lfloor n/2 \rfloor) + n \ge n$ untuk $n \ge 2$, dan $T(1) = 1 \ge 1$. ✓

Jadi $n \le T(n) \le 2n$ untuk setiap $n \ge 1$: $T(n) = \Theta(n)$ dengan $c_1 = 1$, $c_2 = 2$, $n_0 = 1$. Intuisinya adalah deret geometri pada Subbab 3.9.3: $n + n/2 + n/4 + \cdots < 2n$.

Penyelesaian (b). *Terkaan yang wajar, $U(n) \le cn$, gagal.* Sulihkan terkaan itu ke dalam Persamaan (4.11). Andaikan $U(m) \le cm$ untuk setiap $m < n$. Maka

$$U(n) = 2U(\lfloor n/2 \rfloor) + 1 \le 2c\lfloor n/2 \rfloor + 1 \le cn + 1.$$

Kita membutuhkan $U(n) \le cn$, dan yang kita peroleh $cn + 1$. Tidak ada pilihan c yang dapat menolongnya: $cn + 1 \le cn$ salah untuk setiap c. Godaannya adalah mengangkat bahu — "$cn + 1$ toh masih $O(n)$" — tetapi justru itulah kekeliruan yang diperingatkan Subbab 3.11.3. Induksinya harus membuktikan *ketaksamaan yang sama dengan konstanta yang sama* dengan yang diandaikannya; jika tidak, konstantanya merayap naik satu pada setiap tingkat dan tidak membuktikan apa pun.

Perbaikannya: kurangkan sebuah suku berorde lebih rendah. Terkalah sebagai gantinya $U(n) \leq cn - d$ untuk konstanta c dan $d \geq 0$ yang akan dipilih. Andaikan itu berlaku untuk semua $m < n$:

$$U(n) \leq 2(c\lfloor n/2 \rfloor - d) + 1 \leq cn - 2d + 1.$$

Kita membutuhkan $cn - 2d + 1 \leq cn - d$, yang sederhananya $d \geq 1$. Ambil $d = 1$. Kasus dasarnya menuntut $U(1) = 1 \leq c \cdot 1 - 1$, yaitu $c \geq 2$. Ambil $c = 2$. Jadi $U(n) \leq 2n - 1$ untuk semua $n \geq 1$, dan $U(n) = O(n)$. ✓

Paradoksnya layak dinyatakan terang-terangan: klaim yang tampak lebih lemah, $U(n) \leq 2n$, itu *benar*, tetapi tidak dapat dibuktikan dengan induksi ini, sedangkan klaim yang lebih kuat, $U(n) \leq 2n - 1$, dapat. Hipotesis yang lebih kuat memberi langkah induksi lebih banyak bahan untuk bekerja. Untuk batas bawah yang berpasangan, $U(n) \geq 2\lfloor n/2 \rfloor + 1 \geq (n - 1) + 1 = n$ dengan argumen yang sama seperti pada (a), sehingga $U(n) = \Theta(n)$. (Sebenarnya $U(n) = 2^{\lfloor \lg n \rfloor + 1} - 1$ secara eksak: banyaknya simpul dalam pohon biner lengkap dengan $\lfloor \lg n \rfloor + 1$ tingkat.)

Jawaban. (a) $n \leq T(n) \leq 2n$, jadi $T(n) = \Theta(n)$. (b) $n \leq U(n) \leq 2n - 1$, jadi $U(n) = \Theta(n)$; buktinya membutuhkan hipotesis yang diperkuat $U(n) \leq cn - 1$.

Program. **Kode 4.12** mentabulasi kedua rekurensi untuk $n \leq 10$ dan memeriksa batas-batas yang telah dibuktikan untuk setiap n sampai 10^6; program itu juga memeriksa klaim naif $U(n) \leq n$, untuk memperlihatkan betapa cepat konstanta yang keliru tertangkap.

Kode 4.12 Kedua rekurensi pada Persoalan Terkerjakan G dievaluasi secara eksak, dengan batas-batas yang telah dibuktikan diperiksa untuk setiap $n \leq 10^6$.

```
#include <cstdio>
#include <vector>
using namespace std;
typedef long long LL;

int main() {
    const int N = 1000000;
    vector<LL> T(N + 1), U(N + 1);
    T[1] = 1; U[1] = 1;
    for (int n = 2; n <= N; ++n) {
        T[n] = T[n / 2] + n;          // T(n) = T(floor(n/2)) + n
        U[n] = 2 * U[n / 2] + 1;      // U(n) = 2U(floor(n/2)) + 1
    }
    printf("  n   T(n)  2n   U(n)  2n-1\n");
    for (int n = 1; n <= 10; ++n)
        printf("%3d  %5lld %3d  %5lld %5d\n", n, T[n], 2 * n, U[n], 2 * n - 1);
    int badT = 0, badU = 0, badUn = 0, firstBadUn = 0;
    for (int n = 1; n <= N; ++n) {
        if (!(n <= T[n] && T[n] <= 2LL * n)) ++badT;
        if (!(U[n] <= 2LL * n - 1)) ++badU;
        if (U[n] > n) { ++badUn; if (!firstBadUn) firstBadUn = n; }
    }
    printf("n <= T(n) <= 2n     fails for %d values of n up to 10^6\n", badT);
    printf("U(n) <= 2n - 1      fails for %d values of n up to 10^6\n", badU);
    printf("U(n) <= n (c = 1)   fails for %d values, the first at n = %d\n", badUn,
firstBadUn);
    return 0;
}
```

```
  n   T(n)  2n   U(n)  2n-1
  1      1   2      1     1
  2      3   4      3     3
  3      4   6      3     5
  4      7   8      7     7
  5      8  10      7     9
  6     10  12      7    11
  7     11  14      7    13
  8     15  16     15    15
  9     16  18     15    17
 10     18  20     15    19
n <= T(n) <= 2n     fails for 0 values of n up to 10^6
U(n) <= 2n - 1      fails for 0 values of n up to 10^6
U(n) <= n (c = 1)   fails for 999981 values, the first at n = 2
```

Kolom U berbunyi 1,3,3,7,7,7,7,15, ...: ia menyentuh $2n-1$ tepat pada pangkat-pangkat dua, sehingga konstanta $c = 2$ tidak dapat diperbaiki. Dan klaim dengan $c = 1$ langsung gagal, pada $n = 2$, tempat $U(2) = 3 > 2$.

> **UJI DIRI.** Buktikan bahwa $T(n) = T(\lfloor n/2 \rfloor) + 1$ dengan $T(1) = 1$ memenuhi $T(n) \le \lg n + 1$.
>
> *Jawaban.* Dasar: $T(1) = 1 = \lg 1 + 1$. Langkah, untuk $n \ge 2$: $T(n) \le \lg\lfloor n/2 \rfloor + 1 + 1 \le \lg(n/2) + 2 = \lg n - 1 + 2 = \lg n + 1$. Inilah rekurensi pencarian biner, dan batasnya adalah $\lfloor \lg n \rfloor + 1$ probe pada Persoalan Terkerjakan C.

4.7.2 Persoalan Terkerjakan H — pohon rekursi dengan tabel tingkat

Persoalan. Pecahkan

$$T(n) = 3\,T(n/4) + n^2, \qquad T(1) = 1 \tag{4.12}$$

untuk n berupa pangkat 4, dengan menggambar pohon rekursinya. Temukan nilai eksak $T(n)$ dan kelas Θ-nya, lalu tegaskan batasnya dengan substitusi dan dengan Teorema Master.

Yang diuji. Metode pohon rekursi pada Subbab 3.12: hitung simpul dan biaya pada tiap tingkat, jumlahkan tingkat-tingkatnya (biasanya deret geometri), lalu tambahkan daun-daunnya.

Penyelesaian. Setiap pemanggilan Persamaan (4.12) mengerjakan n^2 kerja sendiri dan melakukan tiga pemanggilan pada masukan seperempat ukurannya. Tuliskan $n = 4^k$, sehingga pohonnya bertingkat $0, 1, \ldots, k$, dengan $k = \log_4 n$. Akarnya berbiaya n^2 dan memiliki tiga anak, masing-masing persoalan berukuran $n/4$. Masing-masing anak itu berbiaya $(n/4)^2$ dan memiliki tiga anak berukuran $n/16$, dan seterusnya. Gambar di bawah memperlihatkan bagian atas pohonnya.

```
tingkat 0                        n^2                                total  n^2
                       /          |          \
tingkat 1        (n/4)^2      (n/4)^2      (n/4)^2                   total  (3/16) n^2
                 /  |  \      /  |  \      /  |  \
tingkat 2   sembilan simpul, masing-masing (n/16)^2                 total  (3/16)^2 n^2
              ...
tingkat i   3^i simpul, masing-masing (n/4^i)^2                     total  (3/16)^i n^2
              ...
tingkat k   3^k = n^(log_4 3) daun, masing-masing T(1) = 1          total  n^(log_4 3)
```

Biaya sebuah tingkat adalah banyaknya simpul dikalikan biaya per simpul. Pada tingkat $i < k$ ada 3^i simpul berukuran $n/4^i$, masing-masing berbiaya $(n/4^i)^2 = n^2/16^i$, sehingga tingkat itu berbiaya

$$3^i \cdot \frac{n^2}{16^i} = \left(\frac{3}{16}\right)^i n^2.$$

Pada tingkat k ada 3^k daun, masing-masing berbiaya $T(1) = 1$. Karena $3^k = 3^{\log_4 n} = n^{\log_4 3} \approx n^{0.792}$, daun-daunnya berbiaya total $n^{\log_4 3}$. **Tabel 4.9** adalah tabel tingkatnya.

Tabel 4.9 Pohon rekursi $T(n) = 3T(n/4) + n^2$, tingkat demi tingkat. Tiap tingkat berbiaya $\frac{3}{16}$ dari tingkat di atasnya, sehingga akarnya mendominasi.

Tingkat i	Simpul	Ukuran tiap simpul	Biaya tiap simpul	Biaya tingkat
0	1	n	n^2	n^2
1	3	$n/4$	$n^2/16$	$\frac{3}{16}n^2$
2	9	$n/16$	$n^2/256$	$\left(\frac{3}{16}\right)^2 n^2$
i	3^i	$n/4^i$	$n^2/16^i$	$\left(\frac{3}{16}\right)^i n^2$
$k = \log_4 n$	$3^k = n^{\log_4 3}$	1	1	$n^{\log_4 3}$

Menjumlahkan tingkat-tingkatnya. Tingkat-tingkat internal membentuk deret geometri dengan rasio $r = \frac{3}{16}$, yang dijumlahkan Subbab 3.9.3 sebagai $\sum_{i=0}^{k-1} r^i = \frac{1-r^k}{1-r}$. Di sini $1 - r = \frac{13}{16}$, dan $r^k = \frac{3^k}{16^k} = \frac{3^k}{n^2}$, karena $16^k = (4^k)^2 = n^2$. Jadi tingkat-tingkat internal berbiaya

$$n^2 \cdot \frac{1 - 3^k/n^2}{13/16} = \frac{16}{13}(n^2 - 3^k),$$

dan dengan menambahkan daun-daunnya,

$$T(n) = \frac{16}{13}n^2 - \frac{16}{13}3^k + 3^k = \frac{16}{13}n^2 - \frac{3}{13}3^k.$$

$$T(n) = \frac{16\,n^2 - 3^{k+1}}{13}, \qquad n = 4^k \tag{4.13}$$

Memeriksa Persamaan (4.13) *dengan tangan.* $n = 4$, $k = 1$: rekurensinya memberi $T(4) = 3T(1) + 16 = 3 + 16 = 19$; rumusnya memberi $\frac{16 \cdot 16 - 9}{13} = \frac{256-9}{13} = \frac{247}{13} = 19$ ✓. $n = 16$, $k = 2$: $T(16) = 3 \cdot 19 + 256 = 313$; rumusnya memberi $\frac{16 \cdot 256 - 27}{13} = \frac{4096-27}{13} = \frac{4069}{13} = 313$ ✓.

Kelas Θ-nya. Akarnya saja sudah berbiaya n^2, jadi $T(n) \geq n^2$; dan Persamaan (4.13) memberi $T(n) < \frac{16}{13}n^2$. Karena itu $T(n) = \Theta(n^2)$, dan rasio $T(n)/n^2$ harus mendekati $\frac{16}{13} \approx 1.230769$.

Penegasan dengan substitusi. Pohonnya mengusulkan batas $T(n) \leq \frac{16}{13}n^2$; metode substitusi membuktikannya tanpa penjumlahan apa pun. Dasar: $T(1) = 1 \leq \frac{16}{13}$. ✓ Langkah: dengan mengandaikan batas itu pada $n/4$,

$$T(n) = 3T(n/4) + n^2 \leq 3 \cdot \frac{16}{13} \cdot \frac{n^2}{16} + n^2 = \frac{3}{13}n^2 + \frac{13}{13}n^2 = \frac{16}{13}n^2. \checkmark$$

Inilah alur kerja yang dianjurkan Bab 3: pohon rekursi *menemukan* jawabannya, metode substitusi *membuktikannya*.

Penegasan dengan Teorema Master. $a = 3$, $b = 4$, $n^{\log_b a} = n^{\log_4 3} \approx n^{0.792}$, dan $f(n) = n^2 = \Omega(n^{0.792+\varepsilon})$ dengan $\varepsilon \approx 1.2$. Keteraturan: $af(n/b) = 3(n/4)^2 = \frac{3}{16}n^2$, jadi $c = \frac{3}{16} < 1$. Kasus 3 berlaku: $T(n) = \Theta(n^2)$. ✓ Konstanta keteraturan $\frac{3}{16}$ tepat merupakan rasio antara tingkat-tingkat berurutan pada pohonnya — dan memang itulah makna syarat keteraturan.

Jawaban. $T(n) = \frac{16n^2 - 3^{k+1}}{13}$ untuk $n = 4^k$, jadi $T(n) = \Theta(n^2)$ dan $T(n)/n^2 \to \frac{16}{13}$.

Program. **Kode 4.13** mencetak tabel tingkat untuk $n = 256$ lalu rasio $T(n)/n^2$ untuk $n = 4, 16, \ldots, 4^{12}$.

Kode 4.13 Pohon rekursi $T(n) = 3T(n/4) + n^2$ pada $n = 256$, tingkat demi tingkat, dan rasio $T(n)/n^2$ seiring bertambahnya n.

```
#include <cstdio>
#include <cmath>
using namespace std;
typedef long long LL;

// T(n) = 3T(n/4) + n^2 dengan T(1) = 1, untuk n berupa pangkat 4.
LL T(LL n) { return n == 1 ? 1 : 3 * T(n / 4) + n * n; }

int main() {
    LL n = 256;
    printf("Recursion tree for n = %lld\n", n);
    printf("level  nodes  size  cost/node  level total  total/n^2\n");
    LL nodes = 1, size = n, sum = 0;
    for (int i = 0; size >= 1; ++i) {
        LL per = (size == 1) ? 1 : size * size;   // sebuah daun berbiaya T(1) = 1
        LL tot = nodes * per;
        sum += tot;
        printf("%5d  %5lld  %4lld  %9lld  %11lld  %9.6f\n", i, nodes, size, per, tot,
(double)tot / (n * n));
        if (size == 1) break;
        nodes *= 3; size /= 4;
    }
    printf("sum of the levels = %lld, T(256) computed directly = %lld\n\n", sum, T(n));
    printf("        n            T(n)   T(n)/n^2\n");
    for (LL m = 4, k = 1; k <= 12; m *= 4, ++k)
        printf("%9lld  %14lld   %.6f\n", m, T(m), (double)T(m) / ((double)m * m));
    printf("16/13 = %.6f\n", 16.0 / 13.0);
    return 0;
}
```

```
Recursion tree for n = 256
level  nodes  size  cost/node  level total  total/n^2
    0      1   256      65536        65536   1.000000
    1      3    64       4096        12288   0.187500
    2      9    16        256         2304   0.035156
    3     27     4         16          432   0.006592
    4     81     1          1           81   0.001236
sum of the levels = 80641, T(256) computed directly = 80641

        n            T(n)   T(n)/n^2
        4              19   1.187500
       16             313   1.222656
       64            5035   1.229248
      256           80641   1.230484
     1024         1290499   1.230716
     4096        20648713   1.230759
    16384       330381595   1.230767
    65536      5286112081   1.230769
   262144     84577812979   1.230769
  1048576   1353245066713   1.230769
  4194304  21651921244555   1.230769
 16777216  346430740444321   1.230769
16/13 = 1.230769
```

Total tingkat 65536,12288,2304,432 menyusut dengan faktor $\frac{3}{16} = 0.1875$ setiap kali, daun-daunnya menambahkan $81 = 3^4$, dan jumlahnya, 80,641, tepat sama dengan $\frac{16 \cdot 65536 - 3^5}{13} = \frac{1{,}048{,}576 - 243}{13} = \frac{1{,}048{,}333}{13}$. Kolom rasionya naik ke $1.230769 = \frac{16}{13}$ dan menetap di sana.

> **UJI DIRI.** Gambarlah pohon rekursi $T(n) = 4T(n/2) + n$ dan temukan kelas Θ-nya.
>
> *Jawaban.* Tingkat i memiliki 4^i simpul berukuran $n/2^i$, berbiaya $4^i \cdot n/2^i = 2^i n$: tingkat-tingkatnya *bertambah* dengan faktor 2. Daun-daunnya, pada tingkat $\lg n$, berjumlah $4^{\lg n} = n^2$ dan mendominasi; tingkat-tingkat internalnya berjumlah $n(2^{\lg n} - 1) = n^2 - n$. Jadi $T(n) = \Theta(n^2)$ — Kasus 1 Teorema Master, dengan $n^{\log_2 4} = n^2$.

4.7.3 Persoalan Terkerjakan I — empat rekurensi dan Teorema Master, termasuk sebuah celah

Persoalan. Gunakan Teorema Master, bila berlaku, untuk memecahkan

(a) $T(n) = 8T(n/4) + n$; (b) $T(n) = 27T(n/3) + n^3$; (c) $T(n) = 5T(n/4) + n^2$; (d) $T(n) = T(n/2) + \lg n$,

masing-masing dengan $T(1) = 1$. Bila teorema itu tidak berlaku, katakan dengan tepat mengapa, lalu pecahkan rekurensinya dengan cara lain.

Yang diuji. Pernyataan Teorema Master pada Subbab 3.13.1, perhitungan $\log_b a$, pemilihan ε, syarat keteraturan, dan celah-celah pada Subbab 3.13.6.

Penyelesaian. Untuk setiap rekurensi, ikuti strategi lima langkah pada Subbab 3.13.7: baca a, b, dan $f(n)$; hitung eksponen kritis $\log_b a$; bandingkan $f(n)$ dengan $n^{\log_b a}$; periksa syarat sampingnya; simpulkan.

(a) $a = 8$, $b = 4$, $f(n) = n$. $\log_4 8 = \frac{\lg 8}{\lg 4} = \frac{3}{2}$, jadi $n^{\log_b a} = n^{1.5}$. Eksponen f adalah $1 < 1.5$, sehingga $f(n) = O(n^{1.5-\varepsilon})$ dengan $\varepsilon = 0.5$. **Kasus 1**: $T(n) = \Theta(n^{1.5})$. Daun-daunnya mendominasi.

(b) $a = 27$, $b = 3$, $f(n) = n^3$. $\log_3 27 = 3$, jadi $n^{\log_b a} = n^3$, dan $f(n) = \Theta(n^3)$ secara tepat. **Kasus 2**: $T(n) = \Theta(n^3 \lg n)$. Setiap tingkat berbiaya n^3 yang sama, dan ada $\log_3 n + 1$ tingkat.

(c) $a = 5$, $b = 4$, $f(n) = n^2$. $\log_4 5 = \frac{\lg 5}{\lg 4} \approx \frac{2.3219}{2} \approx 1.161$. Eksponen f adalah $2 > 1.161$, sehingga $f(n) = \Omega(n^{1.161+\varepsilon})$ dengan $\varepsilon \approx 0.839$. Keteraturan: $af(n/b) = 5(n/4)^2 = \frac{5}{16}n^2$, jadi $c = \frac{5}{16} < 1$. ✓ **Kasus 3**: $T(n) = \Theta(n^2)$. Akarnya mendominasi.

(d) $a = 1$, $b = 2$, $f(n) = \lg n$. $\log_2 1 = 0$, jadi $n^{\log_b a} = n^0 = 1$. Sekarang ujilah ketiga kasusnya.

- Kasus 1 membutuhkan $\lg n = O(n^{-\varepsilon})$ untuk suatu $\varepsilon > 0$. Tetapi $n^{-\varepsilon} \to 0$ sedangkan $\lg n \to \infty$. Tidak.
- Kasus 2 membutuhkan $\lg n = \Theta(1)$. Tetapi $\lg n$ tidak terbatas. Tidak.
- Kasus 3 membutuhkan $\lg n = \Omega(n^{\varepsilon})$ untuk suatu $\varepsilon > 0$. Tetapi $\lg n = o(n^{\varepsilon})$ untuk setiap $\varepsilon > 0$ (Subbab 3.8.6). Tidak.

$f(n)$ lebih besar daripada n^0, tetapi tidak *secara polinomial* lebih besar. Rekurensinya jatuh ke dalam celah antara Kasus 2 dan 3, dan **Teorema Master tidak berlaku**. Kita memecahkannya dengan membuka rekurensinya. Dengan $n = 2^k$, $\lg n = k$, dan setiap langkah pembukaan membagi dua n serta menurunkan logaritmanya sebesar satu:

$$\begin{aligned} T(2^k) = k + T(2^{k-1}) = k + (k-1) + T(2^{k-2}) &= \cdots \\ &= k + (k-1) + \cdots + 1 + T(1). \end{aligned}$$

Itu lagi-lagi deret aritmetika:

$$T(n) = \frac{k(k+1)}{2} + 1 = \frac{\lg n\,(\lg n + 1)}{2} + 1 = \Theta(\lg^2 n).$$

Periksa pada $n = 16$, $k = 4$: $T(16) = 4 + 3 + 2 + 1 + 1 = 11$, dan $\frac{4 \cdot 5}{2} + 1 = 11$ ✓.

Tabel 4.10 menghimpun keempat hasilnya, bersama nilai eksak pada pangkat-pangkat b yang ditegaskan oleh hitungan singkat (atau oleh **Kode 4.14**).

Tabel 4.10 Empat rekurensi melalui Teorema Master. Yang terakhir terletak di celah antara Kasus 2 dan 3 dan dipecahkan dengan membuka rekurensinya.

Rekurensi	$\log_b a$	Bandingkan $f(n)$	Kasus	Penyelesaian	Eksak pada pangkat b
$8T(n/4) + n$	1.5	$n = O(n^{1.5-0.5})$	1	$\Theta(n^{1.5})$	$2n^{1.5} - n$
$27T(n/3) + n^3$	3	$n^3 = \Theta(n^3)$	2	$\Theta(n^3 \lg n)$	$n^3(\log_3 n + 1)$
$5T(n/4) + n^2$	≈ 1.161	$n^2 = \Omega(n^{1.161+0.839})$, $c = \frac{5}{16}$	3	$\Theta(n^2)$	$\frac{16n^2 - 5n^{\log_4 5}}{11}$
$T(n/2) + \lg n$	0	$\lg n$: tidak secara polinomial di atas 1	celah	$\Theta(\lg^2 n)$ dengan membuka rekurensi	$\frac{\lg n(\lg n + 1)}{2} + 1$

Cara lain teorema itu bisa tidak berlaku. Celah bukan satu-satunya. Teorema itu juga tidak mengatakan apa-apa bila rekurensinya tidak berbentuk $aT(n/b) + f(n)$ dengan konstanta $a \geq 1$, $b > 1$: misalnya $T(n) = 2T(n-1) + 1$ (argumennya $n - 1$, bukan n/b; membukanya memberi $\Theta(2^n)$), $T(n) = \frac{1}{2}T(n/2) + n$ (kurang dari satu subpersoalan, $a < 1$), atau $T(n) = 2^n T(n/2) + n$ (banyaknya subpersoalan bergantung pada n). Sebelum meraih teorema itu, periksalah bahwa ia memang boleh dipakai.

Jawaban. (a) $\Theta(n^{1.5})$; (b) $\Theta(n^3 \lg n)$; (c) $\Theta(n^2)$; (d) Teorema Master tidak berlaku, dan $T(n) = \frac{\lg n(\lg n+1)}{2} + 1 = \Theta(\lg^2 n)$.

Program. **Kode 4.14** mengevaluasi tiap rekurensi secara eksak pada pangkat-pangkat b dalam `long double` lalu membaginya dengan batas yang diramalkan. Rasio yang mengendap ke sebuah konstanta menegaskan kelas Θ-nya.

Kode 4.14 Keempat rekurensi pada Persoalan Terkerjakan I dievaluasi secara eksak lalu dibagi dengan batas yang diramalkan.

```
#include <cstdio>
#include <cmath>
using namespace std;
typedef long double LD;

// Tiap rekurensi dievaluasi secara eksak pada pangkat-pangkat b, dengan T(1) = 1,
// lalu dibagi dengan batas yang diramalkan analisisnya. Rasio yang mengendap
// ke sebuah konstanta menegaskan batas Theta-nya.
LD Ta(LD n) { return n <= 1 ? 1 : 8 * Ta(n / 4) + n; }          // Theta(n^1.5)
LD Tb(LD n) { return n <= 1 ? 1 : 27 * Tb(n / 3) + n * n * n; } // Theta(n^3 lg n)
LD Tc(LD n) { return n <= 1 ? 1 : 5 * Tc(n / 4) + n * n; }      // Theta(n^2)
LD Td(LD n) { return n <= 1 ? 1 : Td(n / 2) + log2l(n); }       // Theta(lg^2 n)

int main() {
    printf("(a) T(n)=8T(n/4)+n      T(n)/n^1.5\n");
    for (int k = 2; k <= 14; k += 4) { LD n = powl(4, k); printf("    n=4^%-2d  %.6Lf\n",
k, Ta(n) / powl(n, 1.5L)); }
    printf("(b) T(n)=27T(n/3)+n^3   T(n)/(n^3 log3 n)\n");
    for (int k = 2; k <= 14; k += 4) { LD n = powl(3, k); printf("    n=3^%-2d  %.6Lf\n",
k, Tb(n) / (n * n * n * k)); }
    printf("(c) T(n)=5T(n/4)+n^2    T(n)/n^2\n");
    for (int k = 2; k <= 14; k += 4) { LD n = powl(4, k); printf("    n=4^%-2d  %.6Lf\n",
k, Tc(n) / (n * n)); }
    printf("(d) T(n)=T(n/2)+lg n    T(n)/lg^2 n\n");
    for (int k = 4; k <= 64; k *= 2) { LD n = powl(2, k); printf("    n=2^%-2d  %.6Lf\n",
k, Td(n) / ((LD)k * k)); }
    return 0;
}
```

```
(a) T(n)=8T(n/4)+n      T(n)/n^1.5
    n=4^2   1.750000
    n=4^6   1.984375
    n=4^10  1.999023
    n=4^14  1.999939
(b) T(n)=27T(n/3)+n^3   T(n)/(n^3 log3 n)
    n=3^2   1.500000
    n=3^6   1.166667
    n=3^10  1.100000
    n=3^14  1.071429
(c) T(n)=5T(n/4)+n^2    T(n)/n^2
    n=4^2   1.410156
    n=4^6   1.454122
    n=4^10  1.454541
    n=4^14  1.454545
(d) T(n)=T(n/2)+lg n    T(n)/lg^2 n
    n=2^4   0.687500
    n=2^8   0.578125
    n=2^16  0.535156
    n=2^32  0.516602
    n=2^64  0.508057
```

Setiap rasio mengendap di tempat yang dikatakan rumus eksak pada **Tabel 4.10**: (a) menuju 2; (b) menuju 1, perlahan, sebagai $1 + \frac{1}{\log_3 n}$; (c) menuju $\frac{16}{11} \approx 1.454545$; (d) menuju $\frac{1}{2}$, sebagai $\frac{k(k+1)/2+1}{k^2}$. Perhatikan betapa lambatnya (b) dan (d) konvergen. Rasio yang bergeser seperti $1 +$

$1/k$ adalah tanda bahwa Anda sedang melihat faktor logaritmik, dan kolom semacam itu mudah disalahbaca sebagai "tidak mengendap" bila Anda tidak tahu apa yang harus diharapkan.

> **UJI DIRI.** Apakah Teorema Master berlaku untuk $T(n) = 9T(n/3) + n^2 \lg n$? Jika tidak, pecahkanlah.
>
> *Jawaban.* $n^{\log_3 9} = n^2$, dan $f(n) = n^2 \lg n$ lebih besar daripada n^2 tetapi hanya dengan faktor logaritmik, bukan dengan n^ε: lagi-lagi celah. Pada pohon rekursinya, tingkat i memiliki 9^i simpul berukuran $n/3^i$, berbiaya $9^i(n/3^i)^2 \lg(n/3^i) = n^2(\lg n - i\lg 3)$. Menjumlahkan atas $\log_3 n$ tingkat memberi n^2 dikalikan deret aritmetika dengan sekitar $\log_3 n$ suku yang rata-ratanya $\frac{1}{2}\lg n$, sehingga $T(n) = \Theta(n^2 \lg^2 n)$.

4.8 Persoalan Terkerjakan tentang Bentuk Tertutup dan Rekurensi Tersembunyi

Kedua persoalan dalam subbab ini ditulis dengan gaya persoalan *online judge*, berlatar studi kasus Profitina Laundry yang menyertai seluruh buku ini. Yang pertama adalah kerabat TOHU: sebuah jumlah yang meneleskop. Yang kedua menyembunyikan bilangan Fibonacci dari Subbab 3.8.9 di dalam pertanyaan pencacahan klasik tentang penubinan domino [5], lalu memintanya pada indeks yang terlalu besar untuk tabel mana pun.

4.8.1 Persoalan Terkerjakan J — Kupon Laundry

Persoalan.

> **LAUNDRT — Kupon Laundry.** Profitina Laundry membagikan satu kupon pada hari ke-1, dua pada hari ke-2, tiga pada hari ke-3, dan seterusnya, sehingga pada akhir hari ke-k ada $T_k = 1 + 2 + \cdots + k$ kupon yang beredar. Setiap hari laundry itu juga membayarkan bonus kesetiaan kepada seluruh pemegang kupon, dan bonus yang dibayarkan pada hari ke-k adalah $1/T_k$: makin banyak kupon, makin kecil bonus hari itu. Diberikan n, hitunglah total bonus yang dibayarkan pada hari 1 sampai n, yaitu

$$V_n = \sum_{k=1}^{n} \frac{1}{T_k}.$$

Masukan: $T \le 20000$ kasus uji, masing-masing satu bilangan bulat n, $1 \le n \le 10^9$. Keluaran: V_n dengan 11 angka di belakang koma. Kompleksitas sasaran: $O(1)$.

> **Yang diuji.** Bilangan segitiga pada Persamaan (4.1), jumlah teleskopik (Subbab 3.9.5), dan jebakan aritmetika ketika mencetak sebuah rasio.
>
> **Penyelesaian.** *Langkah 1–3.* Diketahui: n sampai 10^9, sampai 20,000 kali. Ditanya: $\sum_{k=1}^{n} 1/T_k$, sampai sebelas angka di belakang koma. Anggarannya, seperti catatan $O(1)$, melarang kita menjumlahkan 10^9 suku sebanyak dua puluh ribu kali: carilah bentuk tertutupnya.
>
> *Langkah 4 — kasus-kasus pertama dengan tangan.*

- $T_1 = 1$, jadi suku pertamanya 1. $V_1 = 1$.
- $T_2 = 3$, sukunya $1/3$. $V_2 = 1 + 1/3 = 4/3$.
- $T_3 = 6$, sukunya $1/6$. $V_3 = 4/3 + 1/6 = \frac{8}{6} + \frac{1}{6} = 9/6 = 3/2$.
- $T_4 = 10$, sukunya $1/10$. $V_4 = 3/2 + 1/10 = \frac{15}{10} + \frac{1}{10} = \frac{16}{10} = 8/5$.

- $T_5 = 15$, sukunya $1/15$. $V_5 = 8/5 + 1/15 = \frac{24}{15} + \frac{1}{15} = \frac{25}{15} = 5/3$.

> Jumlah parsialnya adalah $1/1, 4/3, 3/2, 8/5, 5/3$. Tuliskan ulang masing-masing dengan penyebut $n+1$: $\frac{2}{2}, \frac{4}{3}, \frac{6}{4}, \frac{8}{5}, \frac{10}{6}$. Polanya tak bisa dikelirukan: $V_n = \frac{2n}{n+1}$.
>
> *Langkah 5–6 — buktikan.* Bentuknya adalah *bentuk tertutup dari sebuah jumlah*. Karena $T_k = \frac{k(k+1)}{2}$,

$$\frac{1}{T_k} = \frac{2}{k(k+1)} = 2\left(\frac{1}{k} - \frac{1}{k+1}\right),$$

> sebab $\frac{1}{k} - \frac{1}{k+1} = \frac{(k+1)-k}{k(k+1)} = \frac{1}{k(k+1)}$. Bila dijumlahkan, setiap suku dalam saling meniadakan dengan tetangganya:

$$V_n = 2\sum_{k=1}^{n}\left(\frac{1}{k} - \frac{1}{k+1}\right) = 2\left(1 - \frac{1}{n+1}\right) = \frac{2n}{n+1}$$

> (4.14)

Langkah 7 — programnya. **Pseudokode 4.7** adalah keseluruhan algoritmanya.

Pseudokode 4.7 Algoritma Kupon Laundry: satu evaluasi Persamaan (4.14) per pertanyaan.

```
LAUNDRT()
read T
for q = 1 to T
    read n
    print 2n / (n + 1) with 11 decimal places
```

Biaya: satu perkalian dan satu pembagian per kasus uji, waktu $\Theta(1)$ dan ruang $O(1)$, $\Theta(T)$ secara keseluruhan. Kebenarannya mengikut dari identitas teleskopiknya, yang eksak. Ketelitian: $V_n < 2$ selalu, dan sebelas angka di belakang koma dari bilangan di bawah 2 membutuhkan dua belas angka berarti; `double` memasok sekitar enam belas. Bentuk tertutupnya juga memberi tahu bagian keuangan laundry itu sesuatu yang berguna: $V_n \to 2$ saat $n \to \infty$, sehingga total bonusnya terbatas, seberapa lama pun promosinya berjalan.

Jawaban. $V_n = \frac{2n}{n+1}$.

Program. **Kode 4.15** adalah programnya.

Kode 4.15 LAUNDRT — Kupon Laundry, diselesaikan dalam $O(1)$ secara teleskopik.

```
#include <cstdio>
using namespace std;

int main() {
    int T;
    if (scanf("%d", &T) != 1) return 0;
    while (T--) {
        long long n;
        if (scanf("%lld", &n) != 1) break;
        // sum_{k=1..n} 1/T_k  dengan T_k = k(k+1)/2   sama dengan   2n/(n+1)
        printf("%.11f\n", 2.0 * (double)n / ((double)n + 1.0));
    }
    return 0;
}
```

Untuk $n = 1, 2, 3, 10, 10^9$ ia mencetak

```
1.00000000000
1.33333333333
1.50000000000
1.81818181818
1.99999999800
```

dan sebuah program yang menjumlahkan sukunya satu demi satu sepakat sampai kesebelas angkanya untuk $n = 1$ sampai 12. Jebakannya di sini adalah menulis `2 * n / (n+1)` dalam aritmetika bilangan bulat, yang terpangkas menjadi 1 untuk setiap $n \geq 1$; pengubahan ke `double` harus terjadi *sebelum* pembagiannya.

> **UJI DIRI.** Temukan bentuk tertutup untuk $\sum_{k=1}^{n} \frac{1}{k(k+2)}$.
>
> *Jawaban.* $\frac{1}{k(k+2)} = \frac{1}{2}\left(\frac{1}{k} - \frac{1}{k+2}\right)$. Teleskopnya kini meniadakan suku-suku yang berjarak dua, sehingga dua suku bertahan di tiap ujung: $\frac{1}{2}\left(1 + \frac{1}{2} - \frac{1}{n+1} - \frac{1}{n+2}\right) = \frac{3}{4} - \frac{2n+3}{2(n+1)(n+2)}$. Periksa $n = 1$: $\frac{3}{4} - \frac{5}{12} = \frac{9-5}{12} = \frac{1}{3} = \frac{1}{1 \cdot 3}$ ✓.

4.8.2 Persoalan Terkerjakan K — menubin lajur $2 \times N$

Persoalan.

> **STRIPTILE — Menubin Lajur $2 \times N$.** Sebuah lorong di laundry itu selebar 2 ubin dan sepanjang N ubin. Lorong itu harus tertutup seluruhnya oleh ubin 1×2, masing-masing dipasang mendatar atau tegak, tanpa tumpang tindih dan tanpa celah. Ada berapa cara melakukannya? Laporkan jawabannya modulo $10^9 + 7$. Masukan: $T \leq 10^5$ kasus uji, masing-masing satu bilangan bulat N, $0 \leq N \leq 10^{18}$. Keluaran: satu bilangan per kasus uji. Kompleksitas sasaran: $O(\log N)$.

Yang diuji. Menurunkan rekurensi dari sebuah argumen pencacahan, mengenali bilangan Fibonacci (Subbab 3.8.9), dan mengevaluasi rekurensi linear pada indeks yang luar biasa besar dengan *divide and conquer*.

Penyelesaian. *Langkah 1–3.* Diketahui: N sampai 10^{18}, sampai 10^5 kali. Ditanya: banyaknya penubinan lengkap sebuah persegi panjang $2 \times N$ oleh domino, modulo $10^9 + 7$. Tabel berisi 10^{18} entri tidak mungkin; sasaran $O(\log N)$ berkata: sebuah rekurensi ditambah siasat membagi dua.

Langkah 4 — cacah kasus kecil dengan tangan. Misalkan D_N adalah jawabannya. Tuliskan V untuk sebuah ubin tegak dan HH untuk sepasang ubin mendatar yang bertumpuk.

- $N = 0$: lorong kosong tertubin dengan tepat satu cara — dengan tidak memasang apa pun. $D_0 = 1$.
- $N = 1$: lajur 2×1 menerima tepat satu ubin tegak. $D_1 = 1$.
- $N = 2$: VV atau HH. $D_2 = 2$.
- $N = 3$: VVV, VHH, HHV. $D_3 = 3$.
- $N = 4$: VVVV, VVHH, VHHV, HHVV, HHHH. $D_4 = 5$.

Langkah 5–6 — rekurensinya, diargumentasikan bukan diterka. Lihatlah kolom paling kiri. Tepat dua hal bisa terjadi.

- Kolom itu ditutup oleh satu ubin **tegak**. Yang tersisa adalah lajur $2 \times (N - 1)$, yang tertubin dalam D_{N-1} cara.
- Kolom itu ditutup oleh separuh kiri dua ubin **mendatar**, satu di tiap baris. (Satu ubin mendatar tidak dapat menutupnya sendirian, sebab kotak di bawah atau di atasnya kemudian harus ditutup oleh ubin tegak yang mencuat keluar dari kolom itu.) Kedua ubin itu juga menempati

kolom 2, sehingga kolom 2 kini penuh, dan yang tersisa adalah lajur $2 \times (N-2)$, yang tertubin dalam D_{N-2} cara.

Tidak ada kasus lain, dan kedua kasusnya saling lepas, sehingga

$$D_N = D_{N-1} + D_{N-2}, \qquad D_0 = D_1 = 1 \tag{4.15}$$

Itulah rekurensi Fibonacci dengan indeks yang tergeser: $D_N = F_{N+1}$, dengan $F_0 = 0$, $F_1 = 1$. Periksa: $F_1, \ldots, F_5 = 1,1,2,3,5$ ✓.

Merancang evaluasi $O(\log N)$. Dua jalan menuju F_{N+1} mod p dalam $\Theta(\log N)$ langkah.

Persamaan karakteristik $r^2 - r - 1 = 0$ memiliki akar $\varphi = \frac{1+\sqrt{5}}{2}$ dan $\psi = \frac{1-\sqrt{5}}{2}$, yang memberi rumus Binet $F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}$. Indah, tetapi dalam modulo p ia membutuhkan akar kuadrat dari 5, yang hanya ada karena 5 kebetulan residu kuadratik modulo $10^9 + 7$; mesin tambahannya tidak sepadan di sini.

Pelipatduaan cepat (*fast doubling*) hanya membutuhkan aritmetika bilangan bulat. Mulailah dari identitas matriks

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \qquad (n \geq 1),$$

yang berlaku menurut induksi pada n: untuk $n = 1$ kedua ruasnya adalah $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, dan mengalikan ruas kanan dengan $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ memberi $\begin{pmatrix} F_{n+1} + F_n & F_{n+1} \\ F_n + F_{n-1} & F_n \end{pmatrix} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix}$. Sekarang kuadratkan matriks untuk $n = k$ guna memperoleh matriks untuk $n = 2k$. Entri kiri atas kuadratnya adalah $F_{k+1}^2 + F_k^2$, dan entri itu harus sama dengan F_{2k+1}. Entri kanan atasnya adalah $F_{k+1}F_k + F_kF_{k-1} = F_k(F_{k+1} + F_{k-1})$, dan entri itu harus sama dengan F_{2k}; memakai $F_{k-1} = F_{k+1} - F_k$ untuk menyingkirkan F_{k-1} memberi

$$F_{2k} = F_k(2F_{k+1} - F_k), \qquad F_{2k+1} = F_k^2 + F_{k+1}^2 \tag{4.16}$$

Jadi dari pasangan (F_k, F_{k+1}) kita memperoleh (F_{2k}, F_{2k+1}) dengan sejumlah konstan perkalian, dan satu penjumlahan lagi memberi F_{2k+2} bila kita membutuhkan langkah ganjil. Rekursi yang membagi dua n pada setiap pemanggilan dan mengembalikan pasangan (F_n, F_{n+1}) karena itu berbiaya $\Theta(\log N)$ — rekurensi $T(n) = T(n/2) + \Theta(1)$ milik pencarian biner. **Pseudokode 4.8** adalah prosedurnya.

Pseudokode 4.8 Pelipatduaan cepat: pasangan Fibonacci (F_n, F_{n+1}) modulo p dalam $\Theta(\log n)$ perkalian.

```
FIB(n)                          // mengembalikan pasangan (F(n), F(n+1)) modulo p
1  if n == 0
2      return (0, 1)
3  (a, b) = FIB(floor(n / 2))    // a = F(k), b = F(k+1), dengan k = floor(n/2)
4  c = a * (2b - a) mod p        // F(2k)
5  d = (a*a + b*b) mod p         // F(2k+1)
6  if n is odd
7      return (d, c + d mod p)   // (F(2k+1), F(2k+2))
8  else return (c, d)            // (F(2k), F(2k+1))
```

Penelusuran dengan tangan. Jawaban untuk $N = 5$ adalah $D_5 = F_6$, jadi kita memanggil FIB(6). **Tabel 4.11** mengikuti rekursinya dari bawah ke atas.

Tabel 4.11 Pelipatduaan cepat menghitung $(F_6, F_7) = (8,13)$ dalam empat pemanggilan. Tiap baris memeriksa satu kejadian identitas pelipatduaan.

Pemanggilan	$k=\lfloor n/2 \rfloor$	$(a,b)=(F_k,F_{k+1})$	$c=a(2b-a)$	$d=a^2+b^2$	Mengembalikan
FIB(0)	—	—	—	—	(0,1)
FIB(1)	0	(0,1)	$0 \cdot (2-0) = 0$	$0+1=1$	ganjil: $(1,0+1)=(1,1)$
FIB(3)	1	(1,1)	$1 \cdot (2-1) = 1$	$1+1=2$	ganjil: $(2,1+2)=(2,3)$
FIB(6)	3	(2,3)	$2 \cdot (6-2) = 8$	$4+9 = 13$	genap: (8,13)

Jadi $D_5 = F_6 = 8$, cocok dengan nilai $D_5 = D_4 + D_3 = 5 + 3 = 8$ dari Persamaan (4.15).

Analisis. Kedalaman rekursinya adalah $\lfloor \log_2 N \rfloor + 2 \leq 61$ untuk $N \leq 10^{18}$, dengan kerja konstan per tingkat, sehingga tiap pertanyaan berbiaya waktu $\Theta(\log N)$ dan ruang tumpukan $\Theta(\log N)$. Totalnya $\Theta(T \log N) \approx 10^5 \times 61 \approx 6 \times 10^6$ operasi — nyaman di dalam anggaran pada **Tabel 4.1**. Kebenaran rekurensinya adalah argumen dua kasus di atas; kebenaran pelipatduaan cepatnya mengikut dari Persamaan (4.16) lewat induksi pada n.

Jawaban. $D_N = F_{N+1}$, dihitung modulo $10^9 + 7$ dengan pelipatduaan cepat dalam $\Theta(\log N)$.

Program. **Kode 4.16** adalah programnya. Dua rincian patut diperhatikan. Pertama, setiap hasil kali langsung direduksi modulo p: kedua faktornya di bawah $p \approx 10^9$, sehingga tiap hasil kali di bawah 10^{18} dan muat dalam `long long`, tetapi hasil kali tiga bilangan semacam itu tidak akan muat. Kedua, pengurangan pada $2F_{k+1} - F_k$ bisa menjadi negatif sesudah direduksi, dan dalam C++ operator `%` mempertahankan tanda bilangan yang dibagi, sehingga kodenya menambahkan `MOD` sebelum mengambil sisanya: `(2 * b % MOD - a % MOD + MOD) % MOD`.

Kode 4.16 STRIPTILE — penubinan lajur $2 \times N$ dengan pelipatduaan cepat, $\Theta(\log N)$ per pertanyaan.

```
#include <cstdio>
#include <utility>
using namespace std;
typedef long long LL;
const LL MOD = 1000000007LL;

// pelipatduaan cepat: mengembalikan (F(n) mod p, F(n+1) mod p) dengan F(0)=0, F(1)=1
pair<LL, LL> fib(LL n) {
    if (n == 0) return {0, 1};
    auto p = fib(n >> 1);
    LL a = p.first, b = p.second;             // a = F(k), b = F(k+1), k = n/2
    LL c = a * ((2 * b % MOD - a % MOD + MOD) % MOD) % MOD;   // F(2k)   = F(k)(2F(k+1)-
F(k))
    LL d = (a * a % MOD + b * b % MOD) % MOD;                  // F(2k+1) = F(k)^2 +
F(k+1)^2
    if (n & 1) return {d, (c + d) % MOD};
    return {c, d};
}

int main() {
    int T;
    if (scanf("%d", &T) != 1) return 0;
    while (T--) {
        LL n;
        if (scanf("%lld", &n) != 1) break;
        printf("%lld\n", fib(n + 1).first);    // T(n) = F(n+1)
    }
    return 0;
}
```

Untuk $N = 0,1,2,3,4,5,30, 10^{18}$ ia mencetak

```
1
1
2
3
5
8
1346269
680057396
```

dan tabel lugas yang dibangun dari Persamaan (4.15) menegaskan $D_0, \ldots, D_6 = 1,1,2,3,5,8,13$ dan $D_{30} = 1{,}346{,}269$. Jebakan-jebakannya: menjawab F_N alih-alih F_{N+1} (periksalah $N = 2$, yang harus memberi 2); melupakan bahwa $D_0 = 1$, sebab penubinan kosong tetap sebuah penubinan; nilai antara yang negatif pada $2F_{k+1} - F_k$; dan mencoba membangun tabel berukuran N — 10^{18} entri bukanlah tabel, melainkan angan-angan.

UJI DIRI. Andaikan ubin persegi 2×2 juga diizinkan. Temukan rekurensi barunya, lima nilai pertamanya, dan bentuk tertutupnya.

Jawaban. Kolom paling kiri kini ditutup oleh sebuah domino tegak (menyisakan $N - 1$), dua domino mendatar (menyisakan $N - 2$), atau satu persegi 2×2 (juga menyisakan $N - 2$). Jadi $D_N = D_{N-1} + 2D_{N-2}$ dengan $D_0 = D_1 = 1$, yang memberi 1,1,3,5,11. Persamaan karakteristik $r^2 - r - 2 = 0$ memiliki akar 2 dan -1, dan mencocokkan kedua nilai awalnya memberi $D_N = \frac{2^{N+1}+(-1)^N}{3}$. Periksa $N = 4$: $\frac{32+1}{3} = 11$ ✓.

4.9 Dari Gaya Kasar ke yang Lebih Baik: Perbandingan Berwaktu

Bab 1 menjanjikan bahwa efisiensi bukan soal selera, dan tabel laju pertumbuhan Bab 2 memperlihatkan angka-angkanya di atas kertas. Persoalan terkerjakan terakhir ini memperlihatkannya pada mesin sungguhan.

4.9.1 Persoalan Terkerjakan L — adakah pasangan yang berjumlah x?

Persoalan. Diberikan n bilangan bulat $a_0, \ldots, a_{n-1}$ dan sebuah sasaran x, putuskan apakah ada dua posisi berbeda $i \neq j$ dengan $a_i + a_j = x$. Tulislah sebuah algoritma gaya kasar dan satu yang lebih baik, buktikan kebenaran keduanya, analisis keduanya, dan ukur waktunya pada masukan berukuran 2,000 sampai 64,000.

Yang diuji. Menaksir biaya algoritma gaya kasar (Subbab 1.4.1), memperbaikinya dengan pengurutan (*merge sort* Bab 3, atau pengurutan pustaka) dan pencarian biner (Subbab 2.14), serta membaca waktu jalan terukur terhadap ramalan analisis asimtotik.

Penyelesaian. *Gaya kasar.* Cobalah setiap pasangan $i < j$. Ada $\binom{n}{2} = \frac{n(n-1)}{2}$ pasangan, sehingga kasus terburuknya — ketika tidak ada pasangan yang cocok dan setiap pasangan harus dicoba — berbiaya $\Theta(n^2)$. Kebenarannya langsung terlihat: ia memeriksa setiap pasangan.

Algoritma yang lebih baik. Urutkan lariknya secara menaik. Lalu, untuk setiap posisi i, pasangan yang kita butuhkan adalah $x - a_i$, dan karena lariknya terurut kita dapat mencarinya dengan pencarian biner di bagian sebelah kanan i. **Pseudokode 4.9** adalah prosedurnya; pencariannya adalah varian yang menemukan posisi *pertama* yang nilainya paling kecil sama dengan nilai yang dicari.

Pseudokode 4.9 Algoritma urutkan-lalu-cari untuk persoalan jumlah pasangan, dalam $\Theta(n\lg n)$.

```
PAIR-SUM(a[0..n-1], x)
1   sort a into increasing order
2   for i = 0 to n - 1
3       want = x - a[i]
4       lo = i + 1, hi = n            // cari di a[i+1 .. n-1]
5       while lo < hi                 // invarian: a[lo-1] < want <= a[hi], bila keduanya
ada
6           mid = lo + floor((hi - lo) / 2)
7           if a[mid] < want then lo = mid + 1
8           else hi = mid
9       if lo < n and a[lo] == want
10          return TRUE
11  return FALSE
```

Kebenaran. Pengurutan hanya menyusun ulang nilai-nilainya, jadi pertanyaannya berjawaban sama sebelum dan sesudah pengurutan. Jika prosedur mengembalikan TRUE pada suatu i, maka $a[lo] = want = x - a[i]$ dengan $lo > i$: dua posisi berbeda yang berjumlah x. Sebaliknya, andaikan ada sebuah pasangan; sesudah pengurutan kedua nilainya duduk pada posisi $p < q$ dengan $a[p] + a[q] = x$. Ketika gelung mencapai $i = p$, nilai $want = a[q]$ muncul dalam rentang yang dicari $a[p+1..n-1]$. Menurut invarian pada baris 5, ketika pencarian berakhir lo adalah posisi pertama dalam rentang itu yang nilainya paling kecil $want$; karena $a[q] = want$ ada dalam rentang itu, posisi pertama tersebut bernilai tepat $want$, dan baris 9 mengembalikan TRUE. Jadi prosedur itu mengembalikan TRUE tepat ketika sebuah pasangan ada.

Biaya. Baris 1 berbiaya $\Theta(n\lg n)$. Masing-masing dari n pencarian biner berbiaya $O(\lg n)$. Totalnya $\Theta(n\lg n)$.

Pemeriksaan tangan. Ambil $a = \langle 8,3,6,3,9,1 \rangle$, larik Persoalan Terkerjakan B. Terurut: $\langle 1,3,3,6,8,9 \rangle$. Untuk $x = 12$: $i = 0$ mencari 11, tidak ada; $i = 1$ mencari $12 - 3 = 9$, yang ditemukan pencariannya di posisi 5. TRUE $(3 + 9)$. Untuk $x = 16$: pasangan yang dicari untuk $i = 0, \ldots, 5$ adalah 15,13,13,10,8,7. Tak satu pun ditemukan di sebelah kanan i-nya. Satu-satunya yang nyaris adalah $i = 4$, yang mencari $16 - 8 = 8$; tetapi rentang pencariannya $a[5..5] = \langle 9 \rangle$, dan angka 8 di posisi 4 tidak boleh dipasangkan dengan dirinya sendiri. FALSE.

Program. **Kode 4.17** mengimplementasikan kedua algoritma dan mengukur waktunya dengan `std::chrono::steady_clock`. Datanya adalah n bilangan ganjil yang berbeda dalam urutan teracak, dan sasarannya $x = 1$. Jumlah dua bilangan ganjil adalah genap, sehingga tidak ada pasangan yang dapat mencapai 1: kedua algoritma dipaksa mengerjakan seluruh kerja kasus terburuknya, persis yang ingin kita ukur. Pembangkit bilangan acaknya memakai benih tetap, sehingga setiap eksekusi memakai data yang sama.

Kode 4.17 Gaya kasar melawan urutkan-dan-cari untuk persoalan jumlah pasangan, diukur waktunya pada masukan yang ukurannya berlipat dua.

```
#include <cstdio>
#include <vector>
#include <random>
#include <algorithm>
#include <chrono>
using namespace std;
typedef long long LL;

// Gaya kasar: coba setiap pasangan i < j.  Theta(n^2) perbandingan pada kasus terburuk.
bool pairBrute(const vector<LL>& a, LL x) {
    size_t n = a.size();
    for (size_t i = 0; i < n; ++i)
        for (size_t j = i + 1; j < n; ++j)
            if (a[i] + a[j] == x) return true;
    return false;
}

// Lebih baik: urutkan sekali, lalu cari biner pasangan x - a[i] bagi setiap elemen.
// Theta(n lg n) untuk pengurutan ditambah n pencarian masing-masing Theta(lg n).
bool pairSorted(vector<LL> a, LL x) {             // menerima salinan: larik milik
pemanggil tak tersentuh
    sort(a.begin(), a.end());
    size_t n = a.size();
    for (size_t i = 0; i < n; ++i) {
        LL want = x - a[i];
        size_t lo = i + 1, hi = n;                // cari pasangannya di a[i+1 .. n-1]
        while (lo < hi) {
            size_t mid = lo + (hi - lo) / 2;
            if (a[mid] < want) lo = mid + 1; else hi = mid;
        }
        if (lo < n && a[lo] == want) return true;
    }
    return false;
}

double millis(chrono::steady_clock::time_point s) {
    return chrono::duration<double, milli>(chrono::steady_clock::now() - s).count();
}
```

Kode 4.17 (Lanjutan) *Gaya kasar melawan urutkan-dan-cari untuk persoalan jumlah pasangan, diukur waktunya pada masukan yang ukurannya berlipat dua.*

```
int main() {
    mt19937_64 rng(2026);                          // benih tetap: data yang sama pada
setiap eksekusi
    printf("     n   brute (ms)   sorted (ms)   same answer?\n");
    for (int n = 2000; n <= 64000; n *= 2) {
        vector<LL> a(n);
        for (int i = 0; i < n; ++i) a[i] = 2LL * i + 1;    // n bilangan ganjil yang
berbeda
        shuffle(a.begin(), a.end(), rng);
        LL x = 1;                                  // ganjil + ganjil adalah genap, jadi
tak ada pasangan berjumlah 1:
                                                   // kedua metode harus bekerja penuh
        auto s1 = chrono::steady_clock::now(); bool r1 = pairBrute(a, x);  double t1 =
millis(s1);
        auto s2 = chrono::steady_clock::now(); bool r2 = pairSorted(a, x); double t2 =
millis(s2);
        printf("%6d   %10.2f   %11.3f   %s\n", n, t1, t2, r1 == r2 ? "yes" : "NO");
    }
    vector<LL> small = {8, 3, 6, 3, 9, 1};
    printf("small example {8,3,6,3,9,1}: x=12 -> %d %d, x=16 -> %d %d\n",
           pairBrute(small, 12), pairSorted(small, 12), pairBrute(small, 16),
pairSorted(small, 16));
    return 0;
}
```

Dikompilasi dengan `g++ -std=c++17 -O2 -Wall` dan dijalankan sekali, program itu mencetak

```
     n   brute (ms)   sorted (ms)   same answer?
  2000         0.63         0.125   yes
  4000         2.46         0.230   yes
  8000         9.87         0.480   yes
 16000        41.79         0.994   yes
 32000       173.16         2.452   yes
 64000       670.66         4.631   yes
small example {8,3,6,3,9,1}: x=12 -> 1 1, x=16 -> 0 0
```

Waktu Anda akan berbeda — waktu bergantung pada prosesor, kompilator, dan apa saja yang sedang dikerjakan mesin — tetapi *rasio-rasionya* tidak akan banyak berbeda, dan rasio itulah yang diramalkan analisis. **Tabel 4.12** menghitungnya dari eksekusi ini.

Tabel 4.12 Waktu terukur dari **Kode 4.17**. Melipatduakan n mengalikan waktu gaya kasar dengan sekitar 4, sebagaimana diramalkan $\Theta(n^2)$, dan waktu urutkan-dan-cari dengan sekitar 2, sebagaimana diramalkan $\Theta(n\lg n)$; keunggulannya sendiri berlipat dua pada setiap baris.

n	Gaya kasar (ms)	Rasio terhadap baris sebelumnya	Urutkan dan cari (ms)	Rasio terhadap baris sebelumnya	Kasar / lebih baik
2,000	0.63	—	0.125	—	5.0
4,000	2.46	3.90	0.230	1.84	10.7
8,000	9.87	4.01	0.480	2.09	20.6
16,000	41.79	4.23	0.994	2.07	42.0
32,000	173.16	4.14	2.452	2.47	70.6
64,000	670.66	3.87	4.631	1.89	144.8

Membaca tabelnya. Untuk algoritma $\Theta(n^2)$, melipatduakan n semestinya mengalikan waktunya dengan $2^2 = 4$; rasio terukurnya 3.87 sampai 4.23. Untuk algoritma $\Theta(n\lg n)$, melipatduakan n mengalikan waktunya dengan $2 \cdot \frac{\lg 2n}{\lg n}$, sedikit di atas 2 (untuk $n = 32{,}000$ nilainya $2 \times \frac{15.97}{14.97} \approx 2.13$); rasio terukurnya tersebar di sekitar 2, seperti lazimnya pengukuran waktu yang kecil.

Kolom terakhir adalah pelajarannya: keunggulan algoritma yang lebih baik bukan persentase yang tetap, melainkan *bertambah seiring* n — lima kali lebih cepat pada 2,000 elemen, seratus empat puluh lima kali lebih cepat pada 64,000.

Kita juga dapat mengekstrapolasi. Pada $n = 10^6$ gaya kasar semestinya memakan sekitar $670.66 \times \left(\frac{10^6}{64{,}000}\right)^2 \approx 670.66 \times 244 \approx 164{,}000$ ms — hampir tiga menit — sedangkan algoritma yang lebih baik semestinya memakan sekitar $4.631 \times \frac{10^6}{64{,}000} \times \frac{\lg 10^6}{\lg 64{,}000} \approx 4.631 \times 15.6 \times 1.25 \approx 90$ ms. Anggaran pada Subbab 4.2.3 sudah mengatakan hal yang sama tanpa stopwatch: $\frac{n^2}{2} = 5 \times 10^{11}$ pemeriksaan pasangan jauh di atas 10^8, sedangkan $n\lg n \approx 2 \times 10^7$ nyaman di bawahnya.

Mengapa keunggulan terukur pada $n = 64{,}000$ "hanya" 145, padahal hitungan operasinya berbeda sekitar $\frac{n^2/2}{2n\lg n} = \frac{n}{4\lg n} \approx 1{,}000$ kali? Karena operasi dasar kedua algoritma berbeda biayanya. Gelung dalam gaya kasar hanyalah dua pembacaan larik, satu penjumlahan, dan satu perbandingan atas memori yang berurutan — hal tercepat yang dapat dikerjakan prosesor, dan yang dapat divektorkan oleh kompilator. Pengurutan dan pencarian biner melompat-lompat di dalam memori dan bercabang secara tak terduga. Notasi asimtotik menyembunyikan persis konstanta-konstanta ini, dan itulah sebabnya Bab 3 bersikeras bahwa konstanta dibuang dari *perbandingan laju pertumbuhan* tetapi tidak pernah dari *ramalan detik*.

Jawaban. Gaya kasar: benar, $\Theta(n^2)$. Urutkan dan cari: benar, $\Theta(n\lg n)$. Terukur: sekitar $4\times$ per pelipatduaan lawan sekitar $2\times$ per pelipatduaan, dan percepatan yang tumbuh dari 5 sampai 145 di sepanjang rentang yang diukur.

> **UJI DIRI.** Sesudah pengurutan, gelung pencarian dapat diganti dengan dua penunjuk: $i = 0$, $j = n - 1$; selama $i < j$, jika $a_i + a_j = x$ jawab TRUE, jika jumlahnya terlalu kecil naikkan i, jika tidak turunkan j. Berapa biayanya, dan mengapa ia benar?
>
> *Jawaban.* Tiap langkah menggeser satu penunjuk ke dalam, sehingga paling banyak ada $n - 1$ langkah: $\Theta(n)$ sesudah pengurutan $\Theta(n\lg n)$. Ia benar karena sebuah invarian: *tidak ada pasangan yang memakai posisi di luar* $[i, j]$ *yang berjumlah* x. Jika $a_i + a_j < x$, maka a_i ditambah elemen mana pun sampai a_j juga terlalu kecil, sehingga posisi i dapat dibuang; secara simetris untuk j bila jumlahnya terlalu besar.

4.10 Persoalan-Persoalan Bab Ini Dibandingkan

Sesudah mengerjakan empat belas persoalan, mundurlah selangkah dan tanyakan apa yang dipakai masing-masing. **Tabel 4.13** menjajarkannya.

Tabel 4.13 Keempat belas persoalan bab ini sekilas: keluarga tempat masing-masing bernaung, alat dari Bab 1 sampai Bab 3 yang menyelesaikannya, dan biaya penyelesaiannya.

Persoalan	Subbab	Keluarga (Subbab 4.2.5)	Alat utama, dan di mana diajarkan	Biaya penyelesaiannya
LDCSUMAB	4.3	membalikkan rumus pencacahan	deret aritmetika, pembulatan ke atas (§3.9.1, §3.8.2); pencarian biner (§2.14)	$O(1)$ per pertanyaan
TOHU	4.4	bentuk tertutup dari sebuah jumlah	penyelisihan, teleskopik, induksi (§3.9.5)	$O(1)$ per pertanyaan
A — aturan Horner	4.5.1	gelung dengan invarian	bukti invarian gelung (§2.9)	$\Theta(n)$
B — penelusuran *insertion sort*	4.5.2	gelung dengan invarian	penelusuran tepat; t_j (§2.8, §2.10)	$\Theta(n + \text{inversi})$
C — penelusuran pencarian biner	4.5.3	gelung dengan invarian	invarian jendela; $\lfloor \lg n \rfloor + 1$ probe (§2.14)	$\Theta(\lg n)$
D — gelung rangkap tiga	4.6.1	mencacah langkah dengan penjumlahan	jumlah bersarang, $\sum m^2$ (§3.9)	$\frac{n(n+1)(n+2)}{6}$ langkah
E — konstanta eksplisit	4.6.2	pembuktian asimtotik	definisi O, Θ (§3.3, §3.5)	—
F — mengurutkan fungsi	4.6.3	pembuktian asimtotik	limit rasio (§3.8)	—
G — substitusi	4.7.1	rekurensi	hipotesis yang diperkuat (§3.11)	$\Theta(n)$
H — pohon rekursi	4.7.2	rekurensi	jumlah tingkat geometris (§3.12)	$\Theta(n^2)$
I — Teorema Master	4.7.3	rekurensi	tiga kasus dan celahnya (§3.13)	beragam
J — Kupon Laundry	4.8.1	bentuk tertutup dari sebuah jumlah	teleskopik (§3.9.5)	$O(1)$ per pertanyaan
K — lajur $2 \times N$	4.8.2	rekurensi	Fibonacci, pelipatduaan cepat (§3.8.9)	$\Theta(\log N)$ per pertanyaan
L — jumlah pasangan	4.9	gaya kasar, lalu yang lebih baik	pengurutan dan pencarian biner (§2.14, §3.15)	$\Theta(n \lg n)$

Apa yang sama pada semuanya.

1. **Setiap persoalan memiliki metode gamblang yang entah salah entah terlalu lambat.** Membangun daftar LDCSUMAB, menggelung atas k pada TOHU, mengevaluasi setiap pangkat secara terpisah pada persoalan Horner, memercayai pola untuk a_1, menerka $U(n) \leq cn$, menerapkan Teorema Master di dalam celahnya, mencoba setiap pasangan. Sebagian besar nilai tiap penyelesaian terletak pada *menjelaskan mengapa* metode yang gamblang itu gagal.
2. **Setiap persoalan tunduk pada tabel kasus-kasus kecil.** Bahkan pembuktiannya pun berawal dari tabel: kolom n pada **Tabel 4.2**, kolom a_k pada **Tabel 4.3**, total tingkat pada **Tabel 4.9**.
3. **Setiap jawaban diperiksa dua kali.** Sebuah bukti dan sebuah program; sebuah rumus dan sebuah penelusuran tangan; satu metode pemecahan rekurensi dan metode lainnya.

POLA DI BALIK POLA. Hampir setiap persoalan dalam bab ini diselesaikan dengan dua gerakan yang sama, dalam urutan yang sama. **Gerakan 1: hitung kasus-kasus kecil dengan tangan dan tabulasikan. Gerakan 2: susun ulang perhitungannya sehingga kerja per jawaban menurun** — dengan meneleskopkan sebuah jumlah, membalikkan rumus pencacahan, membawa besaran berjalan melalui sebuah gelung, membagi dua persoalan pada setiap langkah, atau mengurutkan sehingga pencarian menggantikan pemindaian. Lalu buktikan bahwa perhitungan yang telah disusun ulang itu tetap memberi jawaban yang benar.

> Jika Anda hanya melatih gerakan-gerakan itu, Anda akan mengenali sebagian besar persoalan yang Anda jumpai di sisa buku ini.

4.11 Profitina dalam Praktik

Profitina adalah platform visibilitas AI — GEO, *Generative Engine Optimization* — yang berjalan di lingkungan produksi pada `app.profitina.com`. Platform ini memindai situs web sebuah merek, menilai seberapa siap situs itu *dikutip oleh mesin jawaban AI* berdasarkan sinyal nyata seperti keterbacaan konten, data terstruktur, dan kredibilitas teknis situs itu, lalu membantu memperbaiki apa yang ditemukannya. Dasbor analitik historisnya menyimpan tren skor mingguan per merek, dan penjadwalnya memindai ulang setiap situs secara otomatis pada irama yang bergantung pada tingkatan langganan pelanggannya. Gagasan-gagasan dalam bab ini tampak jelas di dalamnya.

Anggaran kompleksitas adalah aturan perancangan, bukan siasat lomba. Sebuah pemindaian adalah rangkaian pengukuran, dan setiap tahapnya punya anggaran dalam arti yang persis sama dengan **Tabel 4.1**: banyaknya halaman sebuah situs, banyaknya sinyal per halaman, dan banyaknya situs yang harus dipindai ulang dalam satu jam ke depan saling dikalikan, dan hasil kalinya harus muat dalam waktu yang tersedia. Pertanyaan pertama yang diajukan kepada setiap sinyal baru adalah pertanyaan Langkah 3: *berapa biayanya per halaman, dan ada berapa halaman?*

Proyeksi berasal dari bentuk tertutup. Dasbornya memperlihatkan, untuk tiap merek, berapa banyak pemindaian ulang otomatis yang akan dijalankan tingkatan langganannya selama cakrawala waktu yang dipilih. Mensimulasikan kalender hari demi hari berbiaya $\Theta(\text{hari})$ per merek dan harus diulang untuk setiap merek pada setiap pemuatan dasbor; sebuah bentuk tertutup berbiaya $\Theta(1)$. Itulah gerakan pada TOHU dan Kupon Laundry — keluarkan tenaga aljabarnya sekali saja supaya mesinnya tidak pernah perlu menggelung.

Tren mingguan adalah tabel yang terus tumbuh, dan bentangan terbaiknya adalah pemindaian satu lintasan. Pemilik merek ingin tahu *kapan* kerja visibilitasnya membuahkan hasil: "skor Anda naik dari 41 ke 68 sepanjang kuartal ini" kurang berguna dibandingkan "perbaikan berkelanjutan terbesar berlangsung dari pekan ke-6 sampai pekan ke-13". Menemukan bentangan itu atas perubahan antarpekan adalah persoalan jendela berjumlah maksimum, yang diselesaikan studi kasus di bawah dengan jumlah awalan dan sebuah invarian gelung. Tabelnya bertambah satu baris setiap pekan dan tidak pernah menyusut, sehingga selisih antara pencarian $\Theta(w^2)$ atas semua jendela dan pemindaian $\Theta(w)$ bertambah persis seperti waktu-waktu pada Persoalan Terkerjakan L.

Halaman yang nyaris kembar adalah persoalan pasangan. Situs sering menyajikan konten yang hampir sama di beberapa alamat, dan skor yang adil tidak boleh menghitungnya dua kali. Membandingkan setiap halaman dengan setiap halaman lain adalah gaya kasar Persoalan Terkerjakan L, $\Theta(n^2)$ perbandingan untuk n halaman; mereduksi setiap halaman menjadi sebuah kunci kanonis lebih dahulu lalu mengurutkan atau meng-*hash* kunci-kuncinya menurunkan kerjanya menjadi sekitar $\Theta(n\lg n)$ — penyusunan ulang yang sama, diterapkan pada dokumen alih-alih bilangan bulat.

Memeriksa dua kali adalah gerbang rilis. Setiap penyelesaian dalam bab ini ditegaskan dengan cara kedua: pemeriksa gaya kasar untuk LDCSUMAB, pecahan eksak untuk TOHU, sebuah program untuk setiap rekurensi. Profitina menerapkan disiplin yang sama dalam skala besar: seluruh rangkaian uji otomatisnya, jauh lebih dari seribu uji, harus lulus sebelum rilis apa pun,

dan sebuah rilis baru melaporkan keberhasilan sesudah versi yang dipasang menjawab pemeriksaan kesehatannya.

Dan satu aturan mengatasi semua algoritma. Platform itu *memberi saran*; pelanggan yang *memutuskan*. Sebuah skor adalah taksiran dari sinyal-sinyal yang diukur, tidak pernah jaminan akan dikutip, dan produk itu menyajikannya dengan cara demikian.

4.11.1 Studi kasus terkerjakan: Profitina Laundry

Profitina Laundry adalah studi kasus pengajaran dalam buku ini — sebuah laundry kecil di Keputih, Surabaya, bukan produk yang sesungguhnya. Pemiliknya punya catatan laba bersih harian dalam rupiah selama dua pekan dan ingin tahu bentangan hari berurutan mana yang layak diulang sebagai promosi. Aturannya: bentangan *berurutan* yang paling menguntungkan; jika dua bentangan seri, utamakan yang lebih panjang, sebab promosi yang lebih panjang lebih mudah diiklankan; jika masih seri juga, utamakan yang lebih dini.

Gaya kasar. Sebuah bentangan adalah pasangan hari $i \le j$, dan dengan $n = 14$ hari ada $\frac{14 \cdot 15}{2} = 105$ bentangan. Menjumlahkan masing-masing dari nol berbiaya total $\Theta(n^3)$; mempertahankan jumlah berjalan seiring bertambahnya j berbiaya $\Theta(n^2)$. Untuk dua pekan itu seketika. Untuk data harian setahun dari sederet cabang, tidak lagi.

Algoritma yang lebih baik, dari penjumlahan Bab 3. Definisikan jumlah awalan $P[0] = 0$ dan $P[j] = p_1 + \cdots + p_j$, yaitu total berjalan sesudah hari ke-j. Laba bentangan $i..j$ kemudian cukup satu pengurangan, $P[j] - P[i-1]$, sebagaimana jumlah teleskopik runtuh menjadi kedua ujungnya. Untuk hari terakhir j yang tetap, bentangan terbaik yang berakhir di sana dimulai tepat sesudah hari ketika total berjalannya *terkecil* sebelum j. Jadi satu lintasan dari kiri ke kanan sudah cukup, dengan membawa satu peubah tambahan yang memiliki invarian gelung: *pada awal iterasi untuk hari j, `argmin` adalah indeks paling dini di $0..j-1$ yang nilai P-nya terkecil.* Memperbarui `argmin` hanya pada `<` yang tegas itulah yang menjaganya tetap *paling dini*, yang menegakkan aturan "utamakan yang lebih panjang"; membandingkan panjang ketika seri menegakkan aturan itu bagi jendela yang dilaporkan. Inilah pemindaian subarray maksimum klasik yang dipopulerkan Bentley [6] dan dianalisis dalam Cormen dkk. [1]. **Kode 4.18** mengimplementasikannya dan juga menjalankan satu himpunan data yang sengaja dibuat seri, supaya aturan pemutus serinya terlihat bekerja.

Kode 4.18 Profitina Laundry — bentangan hari berurutan yang paling menguntungkan, dengan metode minimum awalan, beserta aturan pemutus seri yang diperagakan pada himpunan data kedua.

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

struct Window { long long total; int first, last; };

// Jendela berjumlah maksimum atas p[0..n-1]; seri diputus oleh jendela TERPANJANG, lalu
// oleh awal PALING DINI. Dilaporkan dalam nomor hari berbasis 1. Berjalan dalam Theta(n).
Window bestWindow(const vector<long long>& p) {
    int n = (int)p.size();
    vector<long long> P(n + 1, 0);
    for (int i = 0; i < n; ++i) P[i + 1] = P[i] + p[i];   // P[j] = laba hari 1..j
    Window w{0, 0, -1};                      // jendela kosong: total 0, tanpa hari sama
sekali
    int argmin = 0;                          // i < j paling dini yang P[i]-nya terkecil
    for (int j = 1; j <= n; ++j) {
        long long val = P[j] - P[argmin];   // jendela terbaik yang berakhir di hari j
        int len  = j - argmin;
        int wlen = (w.last < 0) ? 0 : (w.last - w.first + 1);
        if (val > w.total || (val == w.total && len > wlen)) {
            w.total = val; w.first = argmin + 1; w.last = j;
        }
        if (P[j] < P[argmin]) argmin = j;   // '<' yang tegas menjaga minimum PALING DINI
    }
    return w;
}

string rupiah(long long v) {
    bool neg = v < 0; if (neg) v = -v;
    string s = to_string(v), out;
    int c = 0;
    for (int i = (int)s.size() - 1; i >= 0; --i) {
        out += s[i];
        if (++c % 3 == 0 && i > 0) out += '.';
    }
    if (neg) out += '-';
    return "Rp " + string(out.rbegin(), out.rend());
}
```

Kode 4.18 (Lanjutan) *Profitina Laundry — bentangan hari berurutan yang paling menguntungkan, dengan metode minimum awalan, beserta aturan pemutus seri yang diperagakan pada himpunan data kedua.*

```
void report(const string& title, const vector<string>& day, const vector<long long>& p) {
    cout << title << "\n" << string(58, '-') << "\n";
    long long run = 0;
    for (size_t i = 0; i < p.size(); ++i) {
        run += p[i];
        cout << left << setw(9) << day[i] << right << setw(15) << rupiah(p[i])
             << "   running " << setw(15) << rupiah(run) << "\n";
    }
    cout << string(58, '-') << "\n";
    Window w = bestWindow(p);
    if (w.last < 0) { cout << "No stretch of days is profitable.\n\n"; return; }
    cout << "Best stretch : day " << w.first << " to day " << w.last
         << "  (" << day[w.first - 1] << " .. " << day[w.last - 1] << ")\n";
    cout << "Net over it  : " << rupiah(w.total) << "\n";
    cout << "Length       : " << (w.last - w.first + 1) << " days\n\n";
}

int main() {
    vector<string> day = {"Mon 1","Tue 2","Wed 3","Thu 4","Fri 5","Sat 6","Sun 7",
                          "Mon 8","Tue 9","Wed 10","Thu 11","Fri 12","Sat 13","Sun 14"};
    vector<long long> profit = { -90000, 120000, -45000,  80000, -30000,  95000, 210000,
                                 -400000,  55000, -70000,  60000,  40000, 150000,-200000 };
    report("Profitina Laundry - Keputih branch, 14-day net profit", day, profit);

    // Kasus yang sengaja dibuat seri, untuk memperlihatkan aturan pemutus seri bekerja.
    vector<string> d3 = {"Mon 1","Tue 2","Wed 3"};
    vector<long long> t3 = { 30000, -30000, 30000 };
    report("Tie-break demonstration: three windows all worth Rp 30.000", d3, t3);
    return 0;
}
```

Setelah dikompilasi dan dijalankan, program itu mencetak:

```
Profitina Laundry - Keputih branch, 14-day net profit
-------------------------------------------------------
Mon 1         Rp -90.000   running      Rp -90.000
Tue 2         Rp 120.000   running       Rp 30.000
Wed 3         Rp -45.000   running      Rp -15.000
Thu 4          Rp 80.000   running       Rp 65.000
Fri 5         Rp -30.000   running       Rp 35.000
Sat 6          Rp 95.000   running      Rp 130.000
Sun 7         Rp 210.000   running      Rp 340.000
Mon 8        Rp -400.000   running      Rp -60.000
Tue 9          Rp 55.000   running       Rp -5.000
Wed 10        Rp -70.000   running      Rp -75.000
Thu 11         Rp 60.000   running      Rp -15.000
Fri 12         Rp 40.000   running       Rp 25.000
Sat 13        Rp 150.000   running      Rp 175.000
Sun 14       Rp -200.000   running      Rp -25.000
-------------------------------------------------------
Best stretch : day 2 to day 7  (Tue 2 .. Sun 7)
Net over it  : Rp 430.000
Length       : 6 days

Tie-break demonstration: three windows all worth Rp 30.000
-------------------------------------------------------
Mon 1          Rp 30.000   running       Rp 30.000
Tue 2         Rp -30.000   running            Rp 0
Wed 3          Rp 30.000   running       Rp 30.000
-------------------------------------------------------
Best stretch : day 1 to day 3  (Mon 1 .. Wed 3)
Net over it  : Rp 30.000
Length       : 3 days
```

Jawaban pertama, hari 2 sampai 7 senilai Rp 430.000, mudah dipastikan dari kolom *running*, yang tidak lain adalah jumlah awalan $P[j]$: total berjalan terendah sebelum puncaknya adalah $-90{,}000$ pada hari 1, puncaknya 340,000 pada hari 7, dan $340{,}000 - (-90{,}000) = 430{,}000$. Keruntuhan Rp 400.000 pada Senin ke-8 — sebuah mesin pengering yang rusak — membelah dua pekan itu menjadi dua, dan justru itulah sebabnya bentangan terbaik berhenti pada hari 7 alih-alih terus berlanjut. Jawaban kedua memperlihatkan aturannya bekerja: tiga jendela yang berbeda masing-masing bernilai Rp 30.000 (hari 1 saja, hari 3 saja, dan hari 1 sampai 3), dan program mengembalikan yang terpanjang. Seluruh perhitungannya berwaktu $\Theta(n)$ dan berruang $\Theta(n)$ untuk larik awalannya, dibandingkan $\Theta(n^2)$ untuk gaya kasar dengan jumlah berjalan. Seperti biasa, program memberi saran; pemiliknyalah yang memutuskan promosi mana yang dijalankan.

4.12 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. Menulis kode lebih dahulu. Sebagian besar program yang salah sudah salah sebelum baris pertamanya diketik, karena rumus atau invarian di baliknya tidak pernah dituliskan. *Perbaikan:* Subbab 4.2, Langkah 1 sampai 6, sebelum satu baris pun.

2. Melewatkan kasus-kasus kecil. Tabel lima nilai pertama menyelesaikan LDCSUMAB, TOHU, Kupon Laundry, dan lajur $2 \times N$, serta menyingkap a_1 yang merupakan perkecualian pada TOHU. *Perbaikan:* bangunlah tabelnya, selalu, bahkan ketika Anda merasa sudah melihat jawabannya.

3. Memercayai pola tanpa membuktikannya. Lima baris tabel adalah petunjuk, bukan bukti. $a_k = \frac{1}{k(k+1)(k+2)}$ pada TOHU berlaku mulai $k = 2$ tetapi tidak pada $k = 1$. *Perbaikan:* buktikan polanya — dengan penyelisihan, teleskopik, induksi, atau invarian — dan biarkan buktinya memberi tahu di mana pola itu berlaku.

4. Mengabaikan anggaran kompleksitas. Penyelesaian $\Theta(n^2)$ yang benar pada $n = 10^5$ terlalu lambat, dan daftar $\Theta(k)$ yang benar pada $k = 10^9$ tidak muat di memori. *Perbaikan:* bacalah kendalanya, rujuklah **Tabel 4.1**, dan tuliskan hitungan anggarannya.

5. Meleset satu pada sebuah batas. Pembulatan ke atas lawan ke bawah pada LDCSUMAB; `right = mid` lawan `right = mid - 1` pada pencarian biner; gelung dari i sampai j yang berjalan $j - i + 1$ kali, bukan $j - i$; F_N lawan F_{N+1} pada lajur ubin. *Perbaikan:* ujilah kasus-kasus batas secara eksplisit — setiap bilangan segitiga dan penerusnya, jendela satu elemen, $N = 0$ dan $N = 2$.

6. Invarian tanpa ketiga bagiannya. Membuktikan inisialisasi dan pemeliharaan tetapi melupakan terminasi tidak membuktikan apa pun tentang keluarannya. *Perbaikan:* tuliskan ketiga subjudulnya — *Inisialisasi, Pemeliharaan, Terminasi* — lalu isilah masing-masing, seperti pada Persoalan Terkerjakan A.

7. Membiarkan konstanta merayap dalam bukti substitusi. Menurunkan $U(n) \leq cn + 1$ dari hipotesis $U(n) \leq cn$ lalu menyimpulkan "jadi $U(n) = O(n)$" tidak sah. *Perbaikan:* induksinya harus menghasilkan kembali hipotesisnya secara tepat; jika tidak, perkuat hipotesisnya dengan mengurangkan suku berorde lebih rendah (Persoalan Terkerjakan G).

8. Konstanta yang bergantung pada n. Dalam bukti asimtotik, c dan n_0 harus bilangan tetap. "Ambil $c = n$" tidak membuktikan apa-apa. *Perbaikan:* sebutkan konstantanya secara eksplisit, seperti pada Persoalan Terkerjakan E, dan periksa secara numerik.

9. Menerapkan Teorema Master di tempat yang tidak semestinya. Faktor logaritmik bukan faktor polinomial: $T(n) = T(n/2) + \lg n$ berada di dalam celah, dan $T(n) = 2T(n-1) + 1$ sama sekali tidak berbentuk master. *Perbaikan:* hitunglah ε secara eksplisit; bila tidak ada ε positif, bukalah rekurensinya atau gambarlah pohonnya.

10. Aritmetika bilangan bulat padahal yang dimaksud aritmetika real. `2 * n / (n + 1)` bernilai 1 untuk setiap $n \geq 1$; `3/4` bernilai 0. *Perbaikan:* ubahlah ke `double` sebelum membagi, dan tulislah konstanta real dengan titik desimal.

11. Melimpah tanpa suara. $2(n+1)(n+2)$ melimpahi `int` mulai $n = 32{,}767$; $n(n+1)$ setipis rambut dari batas `int` pada LDCSUMAB; hasil kali tiga residu modulo $10^9 + 7$ melimpahi `long long`. *Perbaikan:* perkirakan nilai antara terbesar sebelum memilih tipe data, dan reduksi modulo p sesudah setiap perkalian.

12. Memercayai titik-mengambang untuk pekerjaan bilangan bulat. `sqrt` atas kuadrat sempurna bisa mendarat setipis rambut di sisi mana pun dari bilangan bulatnya bila argumennya melampaui 2^{53}. *Perbaikan:* pakailah akar titik-mengambang hanya sebagai terkaan pertama lalu koreksi dengan uji bilangan bulat, seperti pada **Kode 4.1**, atau hindari sama sekali dengan pencarian biner, seperti pada **Kode 4.2**.

13. Memakai `float` untuk sebelas angka desimal. Tujuh angka berarti tidak dapat mencetak sebelas. *Perbaikan:* `double` untuk segala hal numerik dalam bab ini.

14. Membaca pengukuran waktu tanpa rasio. Satu pengukuran waktu saja hanya sedikit artinya; "gaya kasar memakan 670 ms" bergantung pada mesinnya. *Perbaikan:* ukurlah deretan ukuran yang berlipat dua dan bacalah *rasionya*, seperti pada **Tabel 4.12**; rasio itulah yang diramalkan analisis.

4.13 Ringkasan Bab

Bab ini memantapkan Bab 1 sampai Bab 3 dengan mengerjakan persoalan, bukan dengan menyajikan teori baru.

Metode tujuh langkah pada Subbab 4.2 adalah bagian yang dapat dibawa ke mana-mana: salin formatnya; namai yang diketahui dan yang ditanya; ubah kendala menjadi anggaran waktu; hitung kasus-kasus pertama dengan tangan; sebutkan keluarganya; tuliskan invarian, rekurensi, atau rumusnya, lalu buktikan; baru sesudah itu tulis programnya — sambil memeriksa limpahan, ketelitian, dan kecepatan masukan.

Kedua **persoalan *judge*** memperlihatkan metode itu secara utuh. **LDCSUMAB** mengubah sebuah posisi dalam daftar terurut yang tak berhingga menjadi sepasang eksponen: dua klaim memastikan urutan daftarnya, deret aritmetika mencacah blok-bloknya, rumus kuadrat dan pembulatan ke atas membalikkan hitungan itu, sebuah invarian gelung membuat terkaan titik-mengambangnya aman, dan pencarian biner memberi penyelesaian kedua yang sama sekali tanpa titik-mengambang. **TOHU** memulihkan barisan tersembunyi dengan menyelisihkan jumlah berbobot, mendapati bahwa suku pertamanya adalah perkecualian beserta alasannya, menjumlahkan sisanya secara teleskopik menjadi $S_n = \frac{3}{4} - \frac{1}{2(n+1)(n+2)}$, dan menegaskan hasilnya dengan induksi dan dengan aritmetika rasional eksak.

Kedua belas **persoalan terkerjakan** masing-masing melatih satu alat: invarian gelung untuk aturan Horner; penelusuran tepat *insertion sort* (tempat banyaknya pergeseran sama dengan banyaknya inversi) dan pencarian biner (tempat $\lfloor \lg n \rfloor + 1$ probe sudah cukup); hitungan tepat $\frac{n(n+1)(n+2)}{6}$ dari gelung rangkap tiga; pernyataan asimtotik yang dibuktikan dan disangkal dengan c dan n_0 yang eksplisit; tiga belas fungsi yang diurutkan menurut laju pertumbuhan, dengan peringatan bahwa n yang kecil dapat menyesatkan; bukti substitusi yang membutuhkan hipotesis yang diperkuat; pohon rekursi yang dijumlahkan secara eksak menjadi $\frac{16n^2 - 3^{k+1}}{13}$; Teorema Master dalam ketiga kasusnya dan di dalam celahnya; bentuk tertutup teleskopik; bilangan Fibonacci yang tersembunyi dalam persoalan penubinan dan dihitung dengan pelipatduaan cepat; serta perlombaan berwaktu tempat algoritma $\Theta(n \lg n)$ menjauhi algoritma $\Theta(n^2)$ dengan faktor yang berlipat dua pada setiap pelipatduaan n.

Dua gerakan mengerjakan sebagian besar pekerjaannya: **tabulasikan kasus-kasus kecil**, lalu **susun ulang perhitungannya** sehingga biaya per jawaban menurun — dan buktikan bahwa perhitungan yang telah disusun ulang itu tetap benar.

Menatap ke depan. Bab 5 kembali ke alur utama buku ini dengan *heapsort* dan *quicksort* — dua algoritma pengurutan yang sama-sama berjalan dalam $\Theta(n \lg n)$ pada masukan yang lazim karena alasan yang sama sekali berbeda, dan yang analisisnya memakai invarian gelung, penjumlahan, dan rekurensi yang telah dilatih di sini.

4.14 Soal Latihan

4.1 Buktikan, dengan sebuah invarian gelung, bahwa prosedur berikut mengembalikan elemen terbesar dari $A[1..n]$, $n \geq 1$: `m = A[1]; for i = 2 to n: if A[i] > m then m = A[i]; return m`.

4.2 Temukan banyaknya tepat eksekusi badan gelung, beserta kelas Θ-nya: (a) `for i = 1 to n: for j = 1 to 2i: body`; (b) `for i = 1 to n: { j = 1; while j < i: { j = 2j; body } }`. Untuk (b), berikan juga hitungan tepatnya untuk $n = 8$.

4.3 Telusurilah *insertion sort* pada larik yang terurut terbalik $\langle 6,5,4,3,2,1 \rangle$. Berikan lariknya sesudah tiap iterasi luar, total pergeseran, dan $\sum_j t_j$. Hubungkan banyaknya pergeseran dengan banyaknya inversi.

4.4 Pada larik $A[i] = i$ untuk $i = 1, \ldots, 1000$, telusurilah pencarian biner untuk $q = 1000$ dengan mendaftar nilai-nilai *mid* yang berturut-turut. Berapa probe paling banyak yang mungkin dibutuhkan sebuah pencarian pada larik ini?

4.5 Buktikan $3n^3 - 100n^2 = \Theta(n^3)$ dengan konstanta eksplisit c_1, c_2, n_0.

4.6 Apakah $2^n = \Theta(2^{n/2})$? Apakah $2^{n/2} = O(2^n)$? Buktikan jawaban Anda.

4.7 Pecahkan $T(n) = 2T(n/2) + n^2$, $T(1) = 1$, untuk n berupa pangkat 2, dengan pohon rekursi. Berikan nilai eksaknya dan kelas Θ-nya, lalu tegaskan dengan Teorema Master.

4.8 Terapkan Teorema Master, atau jelaskan mengapa ia tidak berlaku lalu pecahkan dengan cara lain: (a) $T(n) = 2T(n/4) + 1$; (b) $T(n) = 2T(n/4) + n$; (c) $T(n) = 4T(n/2) + n^2 \lg n$; (d) $T(n) = 6T(n/3) + n^2 \lg n$.

4.9 LDCSUMAB meminta *pembalikan peringkat*. Tuliskan kebalikannya, *pemeringkatan*: diberikan $0 \le m < n$, hitunglah posisi k dari $2^m + 2^n$ dalam daftar terurut. Periksalah pada (1,3) dan pada (38439,44721).

4.10 Ubahlah syarat TOHU menjadi $a_1 + 2a_2 + \cdots + ka_k = \frac{k}{k+2}$. Temukan a_k dan S_n dalam bentuk tertutup, dan jelaskan mengapa kali ini suku pertamanya bukan perkecualian.

4.11 Dengan memakai **Tabel 4.12**, ramalkan waktu jalan kedua algoritma pada Persoalan Terkerjakan L untuk $n = 128{,}000$. Lalu jelaskan mengapa, untuk n yang sangat kecil, gaya kasar bisa menjadi yang lebih cepat di antara keduanya.

4.12 Susunlah sebuah persoalan dengan gaya Subbab 4.8 yang jawabannya berupa bentuk tertutup yang ditemukan secara teleskopik, lalu selesaikan dengan ketujuh langkah pada Subbab 4.2.

4.15 Penyelesaian Soal Latihan

4.1 *Invarian:* pada awal iterasi berindeks i, $m = \max(A[1..i-1])$. *Inisialisasi:* $i = 2$ dan $m = A[1] = \max(A[1..1])$. ✓ *Pemeliharaan:* jika $A[i] > m$ badan gelung menetapkan $m = A[i]$, jika tidak m dibiarkan; bagaimanapun juga m menjadi $\max(m, A[i]) = \max(A[1..i])$, yaitu invarian untuk indeks berikutnya $i + 1$. ✓ *Terminasi:* gelung berakhir dengan $i = n + 1$, tempat invariannya mengatakan $m = \max(A[1..n])$. ✓ Untuk $n = 1$ gelungnya tidak berjalan dan $m = A[1]$ sudah benar.

4.2 (a) Untuk tiap i gelung dalamnya berjalan $2i$ kali, jadi totalnya $\sum_{i=1}^{n} 2\,i = 2 \cdot \frac{n(n+1)}{2} = n(n+1) = \Theta(n^2)$. (b) Untuk suatu i, j mengambil nilai 1,2,4, … dan badan gelungnya berjalan sekali per pelipatduaan sampai $j \ge i$, yaitu $\lceil \lg i \rceil$ kali (nol kali untuk $i = 1$). Totalnya $\sum_{i=1}^{n} \lceil \lg i \rceil$. Untuk $n = 8$: $0 + 1 + 2 + 2 + 3 + 3 + 3 + 3 = 17$. Karena $\lg i \le \lceil \lg i \rceil < \lg i + 1$, totalnya terletak di antara $\lg(n!)$ dan $\lg(n!) + n$, sehingga menurut Persoalan Terkerjakan E(d) totalnya $\Theta(n \lg n)$.

4.3 Sesudah $j = 2,3,4,5,6$ lariknya adalah $\langle 5,6,4,3,2,1\rangle$, $\langle 4,5,6,3,2,1\rangle$, $\langle 3,4,5,6,2,1\rangle$, $\langle 2,3,4,5,6,1\rangle$, $\langle 1,2,3,4,5,6\rangle$. Iterasi j menggeser seluruh $j-1$ elemen sebelumnya, sehingga pergeserannya $1+2+3+4+5 = 15 = \frac{6\cdot 5}{2}$, dan $t_j = j$, yang memberi $\sum_{j=2}^{6} t_j = 2+3+4+5+6 = 20$. Setiap pasangan dari $\binom{6}{2} = 15$ pasangan adalah inversi dalam larik yang terbalik, sehingga banyaknya pergeseran sama dengan banyaknya inversi, sebagaimana dibuktikan secara umum pada Persoalan Terkerjakan B. Inilah kasus terburuk pada Subbab 2.11.2: $\Theta(n^2)$.

4.4 Titik-titik tengahnya adalah 500,750,875,938,969,985,993,997,999,1000: sepuluh probe, yang terakhir berhasil. Probe paling banyak yang mungkin dibutuhkan sebuah pencarian adalah $\lfloor \lg 1000 \rfloor + 1 = 9 + 1 = 10$, karena $2^9 = 512 \le 1000 < 1024 = 2^{10}$; jadi $q = 1000$ adalah salah satu kasus terburuk.

4.5 Atas: $3n^3 - 100n^2 \le 3n^3$ untuk semua $n \ge 1$, jadi $c_2 = 3$. Bawah: $2n^3 \le 3n^3 - 100n^2 \Leftrightarrow 100n^2 \le n^3 \Leftrightarrow n \ge 100$, jadi $c_1 = 2$ dengan $n_0 = 100$. Bersama-sama: $c_1 = 2$, $c_2 = 3$, $n_0 = 100$. (Untuk $n < 100$ fungsinya bahkan tidak positif pada, misalnya, $n = 10$, tempat nilainya $3000 - 10000 = -7000$; ambang itu mutlak diperlukan.)

4.6 $2^n = \Theta(2^{n/2})$ salah: jika $2^n \le c\,2^{n/2}$ untuk semua $n \ge n_0$, maka membagi dengan $2^{n/2}$ memberi $2^{n/2} \le c$ untuk semua $n \ge n_0$, yang gagal begitu $n > 2\lg c$. Sebaliknya $2^{n/2} \le 1 \cdot 2^n$ untuk semua $n \ge 0$, jadi $2^{n/2} = O(2^n)$ dengan $c = 1$, $n_0 = 0$. Seperti pada Persoalan Terkerjakan E(c), faktor konstan di dalam eksponen mengubah bilangan pokoknya: $2^{n/2} = \left(\sqrt{2}\right)^n$.

4.7 Tingkat i memiliki 2^i simpul berukuran $n/2^i$, masing-masing berbiaya $(n/2^i)^2$, sehingga tingkat itu berbiaya $2^i \cdot n^2/4^i = n^2/2^i$. Dengan $n = 2^k$, tingkat-tingkat internal $i = 0, \dots, k-1$ berjumlah $n^2\left(2 - \frac{2}{2^k}\right) = 2n^2 - 2n$, dan $2^k = n$ daunnya menambahkan n. Jadi $T(n) = 2n^2 - n$ secara eksak. Periksa: $T(2) = 2 \cdot 1 + 4 = 6 = 8 - 2$ ✓; $T(4) = 2 \cdot 6 + 16 = 28 = 32 - 4$ ✓. $\Theta(n^2)$. Teorema Master: $n^{\log_2 2} = n$, $f(n) = n^2 = \Omega(n^{1+1})$, keteraturan $2(n/2)^2 = \frac{1}{2}n^2$ dengan $c = \frac{1}{2}$: Kasus 3, $\Theta(n^2)$ ✓.

4.8 (a) $\log_4 2 = \frac{1}{2}$; $f(n) = 1 = O(n^{1/2-1/2})$: Kasus 1, $\Theta(\sqrt{n})$. (b) $f(n) = n = \Omega(n^{1/2+1/2})$; keteraturan $2 \cdot \frac{n}{4} = \frac{1}{2}n$, $c = \frac{1}{2}$: Kasus 3, $\Theta(n)$. (c) $\log_2 4 = 2$ dan $f(n) = n^2 \lg n$ melampaui n^2 hanya dengan faktor logaritmik: celah, dan teoremanya tidak berlaku. Menurut pohonnya, tingkat i berbiaya $4^i(n/2^i)^2 \lg(n/2^i) = n^2(\lg n - i)$; menjumlahkan untuk $i = 0, \dots, \lg n - 1$ memberi $n^2 \cdot \frac{\lg n(\lg n+1)}{2}$, ditambah n^2 daun: $\Theta(n^2 \lg^2 n)$. (d) $\log_3 6 \approx 1.631$ dan $f(n) = n^2 \lg n = \Omega(n^{1.631+0.3})$; keteraturan: $6f(n/3) = 6 \cdot \frac{n^2}{9} \lg \frac{n}{3} \le \frac{2}{3} n^2 \lg n$, jadi $c = \frac{2}{3}$: Kasus 3, $\Theta(n^2 \lg n)$.

4.9 Blok $1, \dots, n-1$ mengisi $T_{n-1} = \frac{(n-1)n}{2}$ posisi pertama, dan di dalam blok n pasangan dengan eksponen lebih kecil m adalah yang ke-$(m+1)$. Jadi $k = \frac{n(n-1)}{2} + m + 1$. Periksa (1,3): $3 + 1 + 1 = 5$ ✓. Periksa (38439,44721): $\frac{44721 \cdot 44720}{2} + 38440 = 999{,}961{,}560 + 38{,}440 = 1{,}000{,}000{,}000$ ✓ — pertanyaan pada Subbab 4.3.4, dibalik.

4.10 Menyelisihkan seperti pada Subbab 4.4.4: $ka_k = \frac{k}{k+2} - \frac{k-1}{k+1} = \frac{k(k+1)-(k-1)(k+2)}{(k+1)(k+2)} = \frac{(k^2+k)-(k^2+k-2)}{(k+1)(k+2)} = \frac{2}{(k+1)(k+2)}$, jadi $a_k = \frac{2}{k(k+1)(k+2)}$. Pada $k = 1$ syaratnya memberi $a_1 = \frac{1}{3}$, dan rumusnya pun memberi $\frac{2}{6} = \frac{1}{3}$: tidak ada perkecualian, karena rumus jumlah berbobot

sebelumnya, $\frac{k-1}{k+1}$, bernilai 0 pada $k = 1$, yang tepat sama dengan jumlah kosong. Menurut Persamaan (4.6), $a_k = \frac{1}{k(k+1)} - \frac{1}{(k+1)(k+2)}$, sehingga $S_n = \frac{1}{1\cdot 2} - \frac{1}{(n+1)(n+2)} = \frac{1}{2} - \frac{1}{(n+1)(n+2)}$. Periksa $n = 2$: $a_1 + a_2 = \frac{1}{3} + \frac{1}{12} = \frac{5}{12}$, dan $\frac{1}{2} - \frac{1}{12} = \frac{5}{12}$ ✓.

4.11 Melipatduakan n dari 64,000 mengalikan waktu gaya kasar dengan sekitar 4 dan waktu urutkan-dan-cari dengan sedikit di atas 2: kira-kira $4 \times 670.66 \approx 2{,}700$ ms lawan $2.1 \times 4.631 \approx 9.7$ ms, rasio hampir 280. Untuk n yang sangat kecil konstantanya yang mendominasi: dengan $n = 10$ gaya kasar mengerjakan 45 penjumlahan dan perbandingan dalam gelung yang ketat, sedangkan algoritma yang lebih baik harus menyalin lariknya, mengurutkannya, dan menjalankan sepuluh pencarian biner, masing-masing dengan ongkos tambahannya sendiri. Keunggulan asimtotiknya baru tampak setelah n cukup besar sehingga n^2 menenggelamkan konstanta-konstanta itu — itulah sebabnya **Tabel 4.12** hanya memperlihatkan faktor 5 pada $n = 2{,}000$.

4.12 Salah satu jawaban yang dapat diterima. *Persoalan SQPYRAMID.* Sebuah piramida keranjang laundry memiliki 1 keranjang di lapisan teratas, 4 di lapisan kedua, 9 di lapisan ketiga, dan k^2 di lapisan ke-k. Diberikan $n \le 10^6$, sampai 10^5 kali, berapa banyak keranjang dalam n lapisan pertama? *Langkah 1:* satu bilangan bulat per baris masuk, satu bilangan bulat per baris keluar. *Langkah 2:* diketahui n; ditanya $P_n = \sum_{k=1}^{n} k^2$. *Langkah 3:* 10^5 pertanyaan kali 10^6 suku adalah 10^{11}, di atas anggaran; carilah bentuk tertutupnya. *Langkah 4:* $P_1, \ldots, P_5 = 1,5,14,30,55$. *Langkah 5:* bentuk tertutup dari sebuah jumlah. *Langkah 6:* teleskopkan $(k+1)^3 - k^3 = 3k^2 + 3k + 1$, persis seperti pada Persoalan Terkerjakan D, untuk memperoleh $P_n = \frac{n(n+1)(2n+1)}{6}$; periksa $n = 3$: $\frac{3\cdot 4\cdot 7}{6} = 14$ ✓. *Langkah 7:* $O(1)$ per pertanyaan; jawaban terbesar, pada $n = 10^6$, sekitar 3.3×10^{17}, yang muat dalam `long long`, tetapi hasil kali $n(n+1)(2n+1) \approx 2 \times 10^{18}$ harus dibentuk sebelum dibagi 6, dan itu pun muat (di bawah 9.22×10^{18}) — jadi hitunglah dalam `long long` dan bagilah paling akhir, serta catatlah bahwa kendalanya tidak bisa tumbuh jauh lagi tanpa melimpah.

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] K. H. Rosen, *Discrete Mathematics and Its Applications*, 7th ed. New York, NY, USA: McGraw-Hill, 2012.

[3] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4*, 4th ed. Singapore: Lulu, 2020.

[4] Sphere Research Labs, "SPOJ — Sphere Online Judge." [Online]. Available: https://www.spoj.com/

[5] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1994.

[6] J. Bentley, *Programming Pearls*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2000.

Invarian gelung, analisis *insertion sort* dan pencarian biner, notasi asimtotik, metode-metode rekurensi, dan Teorema Master yang dipakai di sepanjang bab ini mengikuti Cormen dkk. [1, bab 2–4]; aturan Horner muncul di sana sebagai soal di akhir Bab 2. Penjumlahan, bilangan segitiga pada LDCSUMAB, serta argumen penyelisihan dan induksi pada TOHU adalah matematika diskret baku [2]. Anggaran kompleksitas pada **Tabel 4.1** mengikuti kebiasaan praktik pemrograman kompetitif sebagaimana diuraikan dalam [3], dan kedua persoalan *judge* tersedia di SPOJ [4]. Mengadu algoritma satu sama lain dengan stopwatch, beserta pemindaian jendela berjumlah

maksimum pada studi kasus Profitina Laundry, sejalan dengan semangat [6]. Jumlah teleskopik, penubinan domino pada lajur $2 \times N$, dan identitas pelipatduaan cepat untuk bilangan Fibonacci dibahas panjang lebar dalam [5, bab 2, 6, dan 7].

Bab 5
Pengurutan dalam Praktik: Heapsort dan Quicksort

Di akhir bab ini Anda akan sanggup membangun sebuah *binary heap* dari larik yang tidak berurut dalam waktu linear dan membuktikan bahwa batas linear itu memang benar; mengurutkan larik apa pun dalam waktu $\Theta(n\lg n)$ yang terjamin, tanpa memakai memori tambahan sama sekali; mengimplementasikan antrean prioritas yang menyisipkan, mengintip, mengeluarkan, dan mengubah prioritas dalam $O(\lg n)$; menulis Quicksort dari ingatan, memartisi sebuah larik dengan benar pada percobaan pertama, dan membuktikan invarian gelung empat wilayah yang membuat pemartisian itu bekerja; menjelaskan dengan persis mengapa Quicksort adalah $\Theta(n^2)$ pada kasus terburuk dan $\Theta(n\lg n)$ pada kasus rata-rata, serta mengapa yang rata-rata itulah yang sungguh Anda dapatkan; mengacak *pivot*-nya dan menyebutkan dengan tepat apa yang Anda beli dengan pengacakan itu; dan memilih, untuk persoalan pengurutan konkret apa pun yang disodorkan kepada Anda, mana di antara empat algoritma yang kini Anda kenal yang merupakan pilihan yang tepat — lalu membela pilihan itu dengan angka, bukan dengan pendapat. Anda juga akan melihat mengapa tidak ada pengurutan berbasis pembandingan yang dapat mengalahkan $\Omega(n\lg n)$, yang memberitahu Anda bahwa kedua algoritma dalam bab ini, dalam arti yang sesungguhnya, adalah ujung jalan.

5.1 Dari Mana Kita Melanjutkan

Bab 2 dan 3 memberi kita empat algoritma pengurutan beserta matematika untuk menimbangnya. Bab 2 memperkenalkan model RAM dan Insertion Sort, lalu membuktikan Insertion Sort benar dengan sebuah invarian gelung. Bab 3 memberi kita O, Ω, dan Θ; memberi kita rekurensi dan tiga cara menyelesaikannya — substitusi, pohon rekursi, dan Teorema Master; dan berakhir dengan menurunkan Merge Sort dari nol sama sekali serta membuktikan bahwa $T(n) = 2T(n/2) + \Theta(n)$ mempunyai penyelesaian $\Theta(n\lg n)$.

Sebelum kita menambahkan sesuatu yang baru, mari kita jajarkan keempat algoritma itu berdampingan dan melihat dengan jujur apa yang sudah kita punya.

5.1.1 Perkakas sejauh ini

Tabel 5.1 adalah daftar inventarisnya, dan tabel itu layak dibaca pelan-pelan.

Tabel 5.1 Empat algoritma pengurutan dari Bab 2 dan 3. Perhatikan bahwa yang cepat justru satu-satunya yang tidak bekerja di tempat.

Algoritma	Kasus terburuk	Kasus terbaik	Di tempat?	Ruang tambahan	Sifat utama
Insertion Sort	$\Theta(n^2)$	$\Theta(n)$	Ya	$O(1)$	Bagus sekali untuk data yang hampir terurut
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	Ya	$O(1)$	Selalu tepat $n-1$ penukaran
Bubble Sort	$\Theta(n^2)$	$\Theta(n)$	Ya	$O(1)$	Sederhana tetapi lambat
Merge Sort	$\Theta(n\lg n)$	$\Theta(n\lg n)$	Tidak	$O(n)$	Cepat, tetapi butuh larik kedua

Dua di antara kolom-kolom itu perlu dibongkar dulu sebelum kita melanjutkan, sebab bab ini berporos pada keduanya.

Di tempat (*in-place*) berarti algoritmanya menata ulang elemen-elemen di dalam larik yang diberikan kepadanya, dengan hanya memakai sejumlah tetap peubah tambahan — sebuah pencacah gelung, sebuah tempat singgah untuk penukaran, sebuah nilai *pivot* — berapa pun besarnya n. Insertion Sort bekerja di tempat: ia menggeser elemen ke kanan di dalam A lalu menjatuhkan kuncinya ke celah yang terbuka. Merge Sort tidak: `MERGE` tidak dapat menggabungkan dua paruh terurut tanpa suatu tempat untuk menaruh hasil gabungannya, dan tempat itu adalah sebuah larik berisi n elemen.

Ruang tambahan memberi angka pada gagasan yang sama. $O(1)$ berarti "sejumlah tetap, tidak bergantung pada n". $O(n)$ berarti "sejumlah yang tumbuh sebanding dengan n" — untuk larik berisi sepuluh juta bilangan bulat berukuran 4 bita, itu berarti empat puluh megabita memori lagi yang harus tersedia, dan empat puluh megabita data lagi yang harus didorong prosesor melalui bus memorinya.

5.1.2 Celah yang menjengkelkan

Sekarang bacalah **Tabel 5.1** sekali lagi dengan hanya melihat dua kolom: **kasus terburuk** dan **di tempat**.

Setiap algoritma yang mengurutkan di tempat adalah $\Theta(n^2)$. Satu-satunya algoritma yang mencapai $\Theta(n\lg n)$ tidak bekerja di tempat. Ada lubang di tabel itu tepat di tempat yang paling kita inginkan terisi:

> **IMPIAN ITU.** Sebuah algoritma pengurutan yang berjalan dalam waktu $O(n\lg n)$ **dan** mengurutkan di tempat, dengan hanya memakai $O(1)$ memori tambahan. Tidak satu pun yang sudah kita temui melakukan keduanya sekaligus. Bab ini menyerahkan dua algoritma yang mengisi lubang itu: **Heapsort**, yang $\Theta(n\lg n)$ pada kasus terburuk dan memakai $O(1)$ ruang tambahan, dan **Quicksort**, yang $\Theta(n\lg n)$ pada kasus rata-rata, memakai $O(\lg n)$ ruang tambahan, dan merupakan pengurutan serbaguna tercepat dalam praktik.

5.1.3 Mengapa $\Theta(n\lg n)$ adalah sasarannya, dan berapa biaya pengurutan dalam sistem sungguhan

Ada baiknya kita konkret tentang mengapa jurang antara n^2 dan $n\lg n$ bukan sekadar kehalusan akademis.

Ambil $n = 10{,}000{,}000$ — sepuluh juta rekaman, yang menurut ukuran basis data produksi mana pun adalah tabel kecil. Maka

$$n^2 = 10^{14} = 100{,}000{,}000{,}000{,}000$$

dan, karena $\lg(10^7) \approx 23.25$,

$$n\lg n \approx 10^7 \times 23.25 = 232{,}500{,}000 \approx 2.3 \times 10^8.$$

Nisbahnya adalah

$$\frac{10^{14}}{2.3 \times 10^8} \approx 430{,}000.$$

Sebuah komputer yang melakukan 10^9 operasi dasar per detik — angka yang wajar bagi satu inti yang mengerjakan pekerjaan larik sederhana — akan menyelesaikan algoritma $n\lg n$ dalam kira-kira seperempat detik. Algoritma n^2 pada mesin yang sama akan memerlukan $10^{14}/10^9 = 10^5$ detik, yaitu **dua puluh tujuh jam**. Ini bukan "agak lebih lambat". Ini adalah beda antara halaman yang termuat dan halaman yang tidak pernah termuat.

Lalu mengapa $n\lg n$ yang menjadi *sasaran*, dan bukan, katakanlah, n? Karena, sebagaimana akan kita buktikan di Subbab 5.11 pada akhir bab ini, **tidak ada algoritma yang mengurutkan dengan membandingkan pasangan elemen yang dapat berbuat lebih baik daripada $\Omega(n\lg n)$ pada kasus terburuk.** Batas itu bukan batas kecerdikan kita; ia adalah sifat persoalannya sendiri. Begitu sebuah algoritma mencapai $\Theta(n\lg n)$, ia telah sampai ke lantai, dan satu-satunya persaingan yang tersisa adalah tentang faktor konstanta, lalu lintas memori, dan jaminan. Persaingan itulah yang justru menjadi isi Subbab 5.9 sampai 5.12.

Terakhir, sepatah kata tentang di mana pengurutan sesungguhnya muncul, sebab pemula sering membayangkannya sebagai sesuatu yang Anda kerjakan sekali terhadap sebuah daftar nama. Dalam sistem sungguhan, pengurutan ada di mana-mana, biasanya tersembunyi:

- Sebuah basis data yang menjalankan `ORDER BY` mengurutkan. Begitu pula basis data yang membangun indeks, dan begitu pula basis data yang melakukan *sort-merge join* atas dua tabel.
- Mesin pencari yang memeringkat hasil mengurutkan. Umpan rekomendasi mengurutkan. Papan analitik yang menggambar bagan "20 teratas" mengurutkan.
- Menghitung **median**, **persentil**, atau **cacah duplikat** paling sederhana dikerjakan dengan mengurutkan lebih dulu.
- Algoritma graf yang akan Anda temui di Bab 11 dan 13 — pohon merentang minimum Kruskal, lintasan terpendek Dijkstra — entah mengurutkan sisinya secara langsung atau memakai antrean prioritas dari Subbab 5.7, yang merupakan struktur data yang sama.
- Setiap `GROUP BY` yang tidak dikerjakan dengan tabel hash dikerjakan dengan pengurutan.

Sebuah layanan produksi yang mengurutkan pada setiap permintaan tidak sanggup menanggung $\Theta(n^2)$, dan ia juga tidak sanggup dikejutkan oleh sebuah kasus terburuk. Kedua paruh kalimat itu sama-sama penting, dan masing-masing dari dua algoritma bab ini menjawab salah satunya.

5.2 Wawasan dari Selection Sort

Jalan menuju Heapsort bermula dari sepotong kritik-diri yang dialamatkan kepada Selection Sort.

Ingat kembali apa yang dikerjakan Selection Sort. Ia bekerja dari ujung kanan larik menuju ke kiri. Pada setiap langkah ia sudah menempatkan elemen-elemen terbesar di ekor $A[i+1 \ldots n]$, dan kini ia harus mencari yang terbesar di antara sisanya lalu menaruhnya di posisi i. Jadi setiap iterasi terdiri atas dua langkah:

- **Langkah A — cari yang terbesar.** Pindai seluruh $A[1 \ldots i]$ dan ingat nilai terbesar yang terlihat. Ini memeriksa i elemen, jadi ongkosnya $\Theta(i)$, yang berarti $\Theta(n)$ pada iterasi-iterasi awal.
- **Langkah B — tempatkan.** Tukarkan elemen itu ke posisi i. Satu penukaran. $\Theta(1)$.

Ada $n-1$ iterasi, jadi totalnya adalah

$$\underbrace{\Theta(n)}_{\text{Langkah A}} \times \underbrace{(n-1)}_{\text{iterasi}} = \Theta(n^2).$$

Lihatlah di mana ongkosnya berada. Langkah B gratis. **Seluruh** ongkosnya ada di Langkah A: mencari maksimum, berulang-ulang, dari nol, setiap kali. Setiap pemindaian membuang semua yang sudah dipelajarinya. Pada lintasan kedua, algoritma itu membandingkan pasangan yang sama dengan yang dibandingkannya pada lintasan pertama, setelah melupakan jawabannya.

Maka ajukanlah pertanyaan yang gamblang: *bagaimana jika mencari maksimum tidak berongkos* $\Theta(n)$*?* Kalau kita dapat menemukan dan membuang maksimum dalam $O(\lg n)$, totalnya akan menjadi

$$O(\lg n) \times (n-1) = O(n \lg n),$$

dan impian pada Subbab 5.1.2 tercapai tanpa mengubah *bentuk* algoritmanya sedikit pun.

> **WAWASAN KUNCI:** Kadang-kadang cara mempercepat sebuah algoritma bukanlah mengubah algoritmanya, melainkan mengubah **struktur data** tempat ia menyimpan datanya. Selection Sort lambat karena larik biasa adalah tempat yang buruk sekali untuk mencari maksimum berulang kali. Simpanlah bilangan-bilangan yang sama itu di dalam sebuah **binary heap**, dan maksimumnya menjadi gratis, sementara memulihkan keteraturan setelah maksimum itu diambil hanya berongkos $O(\lg n)$. Algoritma yang sama, wadah yang berbeda, kuadratik berubah menjadi $n \lg n$.

Itulah seluruh rencana untuk paruh pertama bab ini. Bangun wadahnya (Subbab 5.3), bangun satu operasi yang menjaganya tetap jujur (Subbab 5.4), bangun wadah itu dari nol dengan murah (Subbab 5.5), lalu jalankan Selection Sort di atasnya (Subbab 5.6).

5.3 Binary Heap: Sebuah Pohon yang Tersembunyi di Dalam Larik

Sepatah kata tentang istilah sebelum kita mulai. Dalam buku ini kata **tumpukan** sudah dipakai untuk *stack* — tumpukan panggilan, yang akan kita perlukan lagi di Subbab 5.10.4. Karena itu struktur data bab ini kita sebut dengan nama Inggrisnya, ***heap***, dan bentuk maksimumnya ***max-heap*** (tumpukan-maks biner). Dua kata itu tidak boleh tertukar: yang satu adalah tempat bingkai panggilan bertumpuk, yang lain adalah pohon yang sebentar lagi kita bangun.

5.3.1 Pohon biner hampir lengkap

Kita mulai dari sebuah bentuk.

Sebuah **pohon biner** (*binary tree*) adalah kumpulan simpul yang salah satunya adalah *akar*, dan setiap simpul mempunyai paling banyak dua anak, yang dibedakan sebagai anak *kiri* dan anak *kanan*. *Kedalaman* sebuah simpul adalah banyaknya sisi pada lintasan dari akar turun ke simpul itu, jadi akar berkedalaman 0. Sebuah *tingkat* (*level*) adalah himpunan semua simpul yang berkedalaman sama. *Tinggi* sebuah simpul adalah banyaknya sisi pada lintasan menurun terpanjang dari simpul itu ke sebuah daun; tinggi pohonnya adalah tinggi akarnya. Simpul tanpa anak disebut **daun**.

Sebuah pohon biner disebut **hampir lengkap** (*nearly complete*) bila ia memenuhi dua syarat:

1. Setiap tingkat terisi penuh, kecuali mungkin tingkat yang terakhir.
2. Tingkat terakhir terisi dari kiri, tanpa celah.

> **CONTOH:** Dengan sepuluh simpul, bentuknya sudah terpaksa: tingkat 0 memuat 1 simpul, tingkat 1 memuat 2, tingkat 2 memuat 4, dan sisa $10 - 7 = 3$ simpul duduk di ujung paling kiri

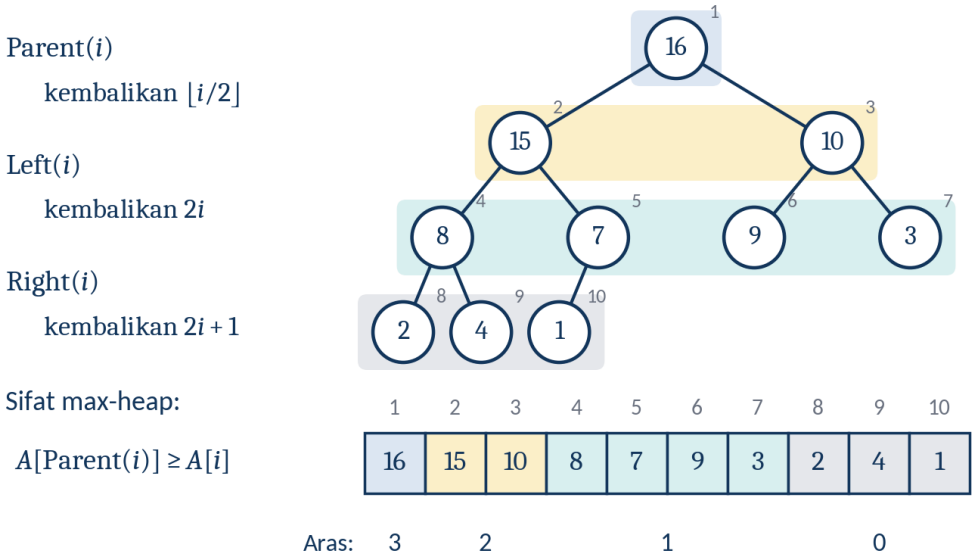
> tingkat 3. Hanya ada tepat satu pohon biner hampir lengkap atas sepuluh simpul, dan tingginya 3.

Kekakuan itulah intinya. Karena bentuknya terpaksa, kita tidak pernah perlu menyimpannya. Sebuah pohon biner hampir lengkap dengan n simpul terperikan seluruhnya oleh satu bilangan n saja — dan itulah yang memungkinkan kita menyembunyikan pohon itu di dalam sebuah larik tanpa satu pun penunjuk.

5.3.2 Aritmetika indeksnya, diturunkan

Nomori simpul-simpulnya 1,2,3, … , n dalam **urutan tingkat**: akarnya 1; lalu dua simpul tingkat 1, dari kiri ke kanan, adalah 2 dan 3; lalu empat simpul tingkat 2 adalah 4,5,6,7; lalu 8,9,10, … dan seterusnya. Simpan nilai simpul i di slot larik $A[i]$. Slot $A[0]$ dibiarkan tidak terpakai supaya aritmetikanya keluar bersih.

Kini tiga rumus navigasinya, yang akan kita *turunkan* alih-alih kita hafalkan. **Gambar 5.1** menampilkan ketiganya, tertulis berdampingan dengan pohon dan larik yang mereka perikan.



Gambar 5.1 Tiga rumus navigasi, pohon yang mereka perikan, dan larik yang memuatnya. Pita-pita warna mengelompokkan simpul menurut kedalamannya dan menandai potongan larik yang bersesuaian di bawahnya; baris berlabel "Level" dihitung dari bawah ke atas, sehingga pita akar bernomor 3 dan baris terbawah bernomor 0.

Inilah asal-usul rumus-rumus itu. Tingkat d memuat tepat 2^d simpul (tingkat 0 memuat $2^0 = 1$, tingkat 1 memuat $2^1 = 2$, dan setiap simpul di satu tingkat memperanakkan dua simpul di tingkat berikutnya). Jadi banyaknya simpul yang benar-benar berada di atas tingkat d adalah

$$1 + 2 + 4 + \cdots + 2^{d-1} = 2^d - 1,$$

menurut jumlahan geometri dari Bab 3. Karena itu simpul pertama tingkat d membawa indeks 2^d, dan simpul-simpul tingkat d berjalan dari 2^d sampai $2^{d+1} - 1$.

Ambil sebuah simpul berindeks i di tingkat d. Ia adalah simpul ke-k pada tingkatnya, dihitung mulai dari nol, dengan $i = 2^d + k$. Kedua anaknya berada di tingkat $d + 1$. Simpul-simpul tingkat $d + 1$ adalah anak-anak dari simpul-simpul tingkat d secara berurutan, dua-dua, sehingga anak-anak dari simpul ke-k pada tingkat d adalah simpul ke-$2k$ dan ke-$(2k + 1)$ pada tingkat $d + 1$. Indeks mereka karena itu adalah

$$2^{d+1} + 2k \quad \text{dan} \quad 2^{d+1} + 2k + 1.$$

Tetapi $2^{d+1} + 2k = 2(2^d + k) = 2i$, dan $2^{d+1} + 2k + 1 = 2i + 1$. Itulah penurunannya:

$$\text{Left}(i) = 2i, \qquad \text{Right}(i) = 2i + 1. \tag{5.1}$$

Rumus induknya menyusul dengan membalik. Jika j adalah anak dari i maka j adalah $2i$ atau $2i + 1$. Pada kasus pertama $j/2 = i$ persis; pada kasus kedua $j/2 = i + \frac{1}{2}$, yang lantainya adalah i. Bagaimanapun juga

$$\text{Parent}(j) = \lfloor j/2 \rfloor. \tag{5.2}$$

Bersama-sama, Persamaan (5.1) dan Persamaan (5.2) adalah seluruh navigasinya: setiap gerakan di dalam sebuah *heap* adalah salah satu dari ketiga rumus itu. **Tabel 5.2** mencatat ketiganya, lengkap dengan satu contoh yang dikerjakan dan sebuah catatan tentang mengapa ketiganya begitu murah dihitung.

Tabel 5.2 Berkeliling di pohon yang tersembunyi itu dengan aritmetika belaka. Setiap satu di antaranya adalah sebuah instruksi mesin tunggal.

Hubungan	Rumus	Contoh, i=3	Trik perangkat keras
Induk simpul i	$\lfloor i/2 \rfloor$	$\text{Parent}(3) = 1$	Geser kanan satu bit
Anak kiri simpul i	$2i$	$\text{Left}(3) = 6$	Geser kiri satu bit
Anak kanan simpul i	$2i + 1$	$\text{Right}(3) = 7$	Geser kiri, lalu nyalakan bit terendah

Kolom "trik perangkat keras" itu pantas mendapat satu kalimat. Dalam biner, mengalikan dengan dua berarti menambahkan satu bit nol di belakang, dan membagi dengan dua secara terpancung berarti membuang bit terakhir. Prosesor mengerjakan keduanya dalam satu siklus dengan instruksi geser. Bandingkan dengan pohon yang dibangun dari simpul dan penunjuk, tempat mencari anak berarti *mendereferensi sebuah penunjuk*: mengambil sebuah alamat dari memori, lalu mengambil simpul yang ditunjuknya, dari mana pun di memori simpul itu kebetulan berada. Sebuah *heap* tidak punya penunjuk untuk dikejar dan tidak punya simpul yang berserak di ruang alamat. Ia adalah satu larik datar yang bersambungan. Itulah sebabnya *heap* cepat dalam praktik, bukan hanya dalam teori.

5.3.3 Sifat heap

Bentuknya baru separuh definisi. Separuh lainnya adalah sebuah kendala keterurutan.

> **DEFINISI.** Sebuah **binary max-heap** adalah larik $A[1 \ldots n]$, yang dipandang sebagai pohon biner hampir lengkap melalui aritmetika indeks di atas, yang di dalamnya setiap simpul sekurang-kurangnya sebesar masing-masing anaknya:

$$A[\text{Parent}(i)] \geq A[i] \qquad \text{untuk setiap } i = 2,3, \ldots, n. \tag{5.3}$$

> Ketaksamaan ini disebut **sifat max-heap**. Sebuah **min-heap** membalikkannya: $A[\text{Parent}(i)] \leq A[i]$ untuk setiap $i > 1$.

Kita akan bekerja dengan *max-heap* sepanjang bab ini, sebab *max-heap*-lah yang dibutuhkan Heapsort. *Min-heap* adalah yang dibutuhkan algoritma graf: algoritma lintasan terpendek Dijkstra (Bab 13) dan pohon merentang minimum Prim (Bab 11) sama-sama berulang kali meminta kunci

terkecil yang tersisa. Segala isi bab ini berpindah begitu saja dengan membalik setiap $\geq$ menjadi $\leq$ dan setiap "terbesar" menjadi "terkecil".

Perhatikan baik-baik apa yang **tidak** dikatakan sifat *heap* pada Persamaan (5.3). Ia tidak mengatakan apa-apa tentang hubungan antara dua saudara, tidak apa-apa tentang hubungan antara sebuah simpul dan pamannya, tidak apa-apa tentang dua simpul mana pun yang bukan pasangan induk-anak. Ia semata-mata kendala *lokal*, dan kelemahan itulah yang justru membuat *heap* murah diperbaiki.

5.3.4 Sebuah heap yang konkret

Ambil

$$A = [16,\ 14,\ 10,\ 8,\ 7,\ 9,\ 3,\ 2,\ 4,\ 1].$$

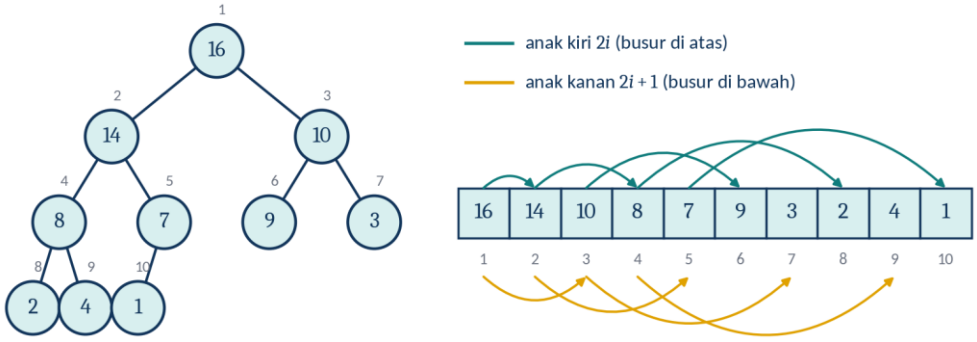
Digambar sebagai pohon, ia berupa

```
              16  (indeks 1)
            /    \
        14        10          (indeks 2, 3)
       /  \      /  \
      8    7    9    3        (indeks 4, 5, 6, 7)
     / \  /
    2  4 1                    (indeks 8, 9, 10)
```

Sekarang periksa sifat *heap*-nya, satu induk demi satu induk — setiap langkah dituliskan, sebab inilah jenis pemeriksaan yang harus sanggup Anda kerjakan di kepala pada akhir bab ini:

- Simpul 1 memuat 16; anak-anaknya $A[2] = 14$ dan $A[3] = 10$. $16 \geq 14$ ✓, $16 \geq 10$ ✓
- Simpul 2 memuat 14; anak-anaknya $A[4] = 8$ dan $A[5] = 7$. $14 \geq 8$ ✓, $14 \geq 7$ ✓
- Simpul 3 memuat 10; anak-anaknya $A[6] = 9$ dan $A[7] = 3$. $10 \geq 9$ ✓, $10 \geq 3$ ✓
- Simpul 4 memuat 8; anak-anaknya $A[8] = 2$ dan $A[9] = 4$. $8 \geq 2$ ✓, $8 \geq 4$ ✓
- Simpul 5 memuat 7; anaknya $A[10] = 1$, dan tidak ada anak kanan karena $2 \times 5 + 1 = 11 > 10$. $7 \geq 1$ ✓
- Simpul 6 sampai 10 berindeks $> \lfloor 10/2 \rfloor = 5$, jadi mereka sama sekali tidak punya anak dan tidak ada yang perlu diperiksa.

Setiap pengujian lulus, jadi A adalah sebuah *max-heap*. **Gambar 5.2** menggambar larik yang sama dengan dua cara sekaligus — sebagai pohon, dan sebagai larik datar dengan busur yang menghubungkan setiap induk dengan kedua anaknya — supaya Anda dapat melihat kesesuaiannya, bukan sekadar memercayainya.



Gambar 5.2 Sepuluh bilangan yang sama dilihat dua kali. Di kiri, pohon biner hampir lengkapnya, dengan indeks larik tiap simpul tercetak di atasnya. Di kanan, larik $A = [16,14,10,8,7,9,3,2,4,1]$, dengan sebuah busur digambar dari tiap simpul ke masing-masing anaknya — ke anak kiri di atas larik, ke anak kanan di bawahnya. Busur-busur itulah yang dihitung oleh aritmetika indeks; tidak satu pun busur semacam itu pernah disimpan.

Tengok kembali **Gambar 5.1**: gambar itu melukiskan larik yang sedikit sekali berbeda, yaitu $A = [16,15,10,8,7,9,3,2,4,1]$. Periksalah sendiri: $16 \geq 15$, $15 \geq 8$, $15 \geq 7$ — semuanya beres. Kedua larik itu adalah *max-heap*, dan hal itu menegaskan satu pokok yang layak dikemukakan sejak awal: **sehimpunan n bilangan mempunyai banyak heap yang sah**, bukan satu. Sifat *heap* adalah kendala yang longgar, bukan sebuah urutan total. Kita memakai versi dengan 14 mulai dari sini sebab itulah larik yang menjadi dasar gambar-gambar penelusuran di Subbab 5.5 dan 5.6.

5.3.5 Heap itu tidak terurut

Inilah salah paham tunggal yang paling sering terjadi tentang *heap*, jadi akan kita nyatakan keras-keras.

> **JEBAKAN YANG SERING TERJADI.** Sebuah *heap* **bukan** larik terurut. Dalam $A = [16,14,10,8,7,9,3,2,4,1]$ nilai 9 duduk di indeks 6 dan nilai 8 duduk di indeks 4, sehingga sebuah bilangan yang lebih besar muncul *sesudah* bilangan yang lebih kecil. Itu sepenuhnya sah: 9 dan 8 bukan pasangan induk-anak, jadi sifat *heap* tidak punya urusan apa pun dengan keduanya. *Satu-satunya* jaminan global yang diberikan sebuah *max-heap* adalah bahwa $A[1]$ **merupakan maksimum dari seluruh larik**. Selebihnya bersifat lokal.

Mengapa $A[1]$ maksimum? Karena sifat *heap* bersifat transitif di sepanjang lintasan menurun mana pun. Ambil sembarang simpul i. Maka $A[\text{Parent}(i)] \geq A[i]$, dan $A[\text{Parent}(\text{Parent}(i))] \geq A[\text{Parent}(i)]$, dan seterusnya. Menyusuri induk ke atas dari i pasti sampai ke akar setelah berhingga langkah, sebab setiap langkah memperkecil indeksnya secara tegas dan indeks adalah bilangan bulat positif. Merangkai ketaksamaan-ketaksamaan itu memberikan $A[1] \geq A[i]$. Karena i sembarang, $A[1]$ mendominasi segalanya. ■

Pembuktian satu baris itulah seluruh alasan keberadaan *heap*. Maksimumnya tidak berongkos sepeser pun untuk dicari: ia sudah duduk di slot 1.

5.3.6 Tinggi sebuah heap adalah $\lfloor \lg n \rfloor$

> **TEOREMA.** Sebuah *heap* dengan n elemen, karena merupakan pohon biner hampir lengkap, mempunyai tinggi $h = \lfloor \lg n \rfloor$.

Bukti. Misalkan tingginya h. Kita tunjukkan $2^h \leq n \leq 2^{h+1} - 1$, lalu kita ambil logaritmanya.

Batas bawahnya. Sebuah pohon bertinggi h mempunyai $h + 1$ tingkat, bernomor 0 sampai h, dan tingkat h memuat sekurang-kurangnya satu simpul — kalau tidak, tingginya akan kurang dari h.

Tingkat 0 sampai $h-1$ terisi penuh, menurut definisi "hampir lengkap", dan memuat $2^0 + 2^1 + \cdots + 2^{h-1} = 2^h - 1$ simpul. Menambahkan sekurang-kurangnya satu simpul di tingkat h memberikan

$$n \geq (2^h - 1) + 1 = 2^h.$$

Batas atasnya. Tingkat h dapat memuat paling banyak 2^h simpul. Jadi

$$n \leq (2^h - 1) + 2^h = 2^{h+1} - 1 < 2^{h+1}.$$

Menyatukan keduanya,

$$2^h \leq n < 2^{h+1}. \tag{5.4}$$

Rantai ketaksamaan Persamaan (5.4) ini menjepit n di antara dua pangkat dua yang berurutan. Mengambil lg dari ketiga bagiannya — sah, sebab lg naik secara tegas —

$$h \leq \lg n < h + 1,$$

yang persis mengatakan bahwa h adalah bagian bulat dari $\lg n$; yaitu $h = \lfloor \lg n \rfloor$. ■

CONTOH: Untuk $n = 10$: $\lg 10 \approx 3.32$, jadi $h = \lfloor 3.32 \rfloor = 3$. Periksa terhadap gambarnya: akarnya berkedalaman 0, dan simpul-simpul terdalam 8,9,10 berkedalaman 3. ✓ Untuk $n = 1{,}000{,}000$: $\lg 10^6 \approx 19.93$, jadi $h = 19$. Sebuah *heap* berisi sejuta elemen dalamnya **sembilan belas** tingkat. Operasi apa pun yang menyusuri satu lintasan akar-ke-daun menyentuh paling banyak dua puluh simpul. Itulah sebabnya operasi *heap* cepat.

5.3.7 Di mana daun-daunnya berada

TEOREMA. Dalam sebuah *heap* berisi n elemen, daun-daunnya tepat adalah simpul-simpul pada indeks $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$. Setara dengan itu, simpul-simpul *dalam* — yaitu yang mempunyai sekurang-kurangnya satu anak — tepat adalah $1, 2, \ldots, \lfloor n/2 \rfloor$.

Bukti. Simpul i mempunyai anak kiri jika dan hanya jika $\text{Left}(i) = 2i \leq n$, yaitu jika dan hanya jika $i \leq n/2$. Karena i bilangan bulat, syarat itu adalah $i \leq \lfloor n/2 \rfloor$. Sebuah simpul tanpa anak kiri juga tidak dapat punya anak kanan, sebab $2i + 1 > 2i > n$. Jadi simpul i adalah simpul dalam tepat ketika $i \leq \lfloor n/2 \rfloor$, dan merupakan daun tepat ketika $i > \lfloor n/2 \rfloor$. ■

Dua akibat yang akan terus-menerus kita pakai:

1. **Sedikit lebih dari separuh sebuah heap adalah daun.** Banyaknya $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$.
2. **Setiap daun sudah merupakan heap yang sah dengan sendirinya.** Pohon beranggotakan satu elemen tidak punya pasangan induk-anak yang dapat melanggar apa pun. Inilah yang membuat `BUILD-MAX-HEAP` sanggup melewatkan seluruh paruh kedua larik, dan inilah separuh alasan mengapa prosedur itu berjalan dalam $O(n)$.

5.3.8 Berapa banyak simpul yang duduk di tiap tinggi

Untuk analisis $O(n)$ di Subbab 5.5 kita memerlukan satu cacahan lagi. Bukan banyaknya simpul pada tiap *kedalaman* (yang adalah 2^d, mudah), melainkan banyaknya pada tiap *tinggi* — dengan tinggi sebuah simpul berarti tinggi subpohon yang berakar padanya, sehingga daun bertinggi 0.

LEMA. Dalam sebuah *heap* berisi n elemen terdapat paling banyak $\lceil n/2^{h+1} \rceil$ simpul bertinggi h.

Gagasannya lugas. Simpul-simpul bertinggi 0 adalah daun-daunnya, dan baru saja kita mencacahnya: $\lceil n/2 \rceil = \lceil n/2^{0+1} \rceil$ ✓. Sekarang copotlah daun-daunnya. Yang tersisa lagi-lagi sebuah pohon biner hampir lengkap, dengan paling banyak $\lfloor n/2 \rfloor$ simpul, dan simpul-simpul bertinggi 1 pada pohon semula justru adalah daun-daun pohon yang lebih kecil ini — yang banyaknya paling banyak $\lceil \lfloor n/2 \rfloor / 2 \rceil \leq \lceil n/4 \rceil = \lceil n/2^{1+1} \rceil$ ✓. Mengulangi hujah copot-daun itu memberikan batasnya pada setiap tinggi, dengan induksi atas h.

Bentuk hasil itulah yang penting, maka mari kita katakan dengan kata-kata:

DENGAN BAHASA SEHARI-HARI. Sebuah *heap* berat sekali di bawah. Kira-kira separuh dari semua simpulnya adalah daun, seperempatnya bertinggi 1, seperdelapannya bertinggi 2, dan seterusnya. Hanya *satu* simpul — akarnya — yang mempunyai tinggi penuh $\lfloor \lg n \rfloor$. Jadi sebuah operasi yang ongkosnya sebanding dengan tinggi sebuah simpul akan murah pada hampir setiap simpul, dan mahal pada hampir tidak ada simpul. Peganglah kalimat ini; ia adalah seluruh isi Subbab 5.5.6.

5.3.9 Semuanya, di dalam sebuah program

Kode 5.1 menaruh setiap klaim subbab ini ke dalam kode yang dapat Anda jalankan: tiga rumus navigasinya, tingginya yang dihitung tanpa menyentuh satu pun bilangan titik-mengambang, rentang daunnya, cacah simpul pada tiap tinggi yang diperiksa terhadap batas dari lemanya, dan akhirnya sebuah verifikasi sifat *heap*. Program itu memakai larik [16,15,10,8,7,9,3,2,4,1] yang digambar di **Gambar 5.1**.

Kode 5.1 Setiap fakta struktural tentang *heap* dari Subbab 5.3, dihitung dan diperiksa.

```
// Aritmetika indeks heap, tinggi, daun, dan cacah simpul pada tiap tinggi.
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
using namespace std;

int n;                                          // ukuran heap, terlihat oleh fungsi
pembantunya

int parentOf(int i) { return i / 2; }           // pembagian bilangan bulat MEMANG lantai
untuk i > 0
int leftOf  (int i) { return 2 * i; }
int rightOf (int i) { return 2 * i + 1; }

int heightOfHeap(int m) {                       // floor(lg m), tanpa titik-mengambang sama
sekali
    int h = 0;
    while ((1 << (h + 1)) <= m) ++h;            // h terbesar dengan 2^(h+1) <= m
    return h;
}

int subtreeHeight(int i) {                      // tinggi subpohon yang berakar di simpul i
    if (leftOf(i) > n) return 0;                // daun bertinggi 0
    int best = subtreeHeight(leftOf(i));
    if (rightOf(i) <= n) best = max(best, subtreeHeight(rightOf(i)));
    return best + 1;
}

int main() {
    vector<int> A = {0, 16, 15, 10, 8, 7, 9, 3, 2, 4, 1};   // A[0] tak terpakai:
pengindeksan berbasis 1
    n = 10;

    cout << "Array (1-based): ";
    for (int i = 1; i <= n; ++i) cout << A[i] << " ";
    cout << "\n\nindex  value  parent  left  right\n";
    for (int i = 1; i <= n; ++i) {
        cout << setw(5) << i << setw(7) << A[i];
        cout << setw(8); if (i > 1)            cout << parentOf(i); else cout << "-";
        cout << setw(6); if (leftOf(i)  <= n)  cout << leftOf(i);   else cout << "-";
        cout << setw(7); if (rightOf(i) <= n)  cout << rightOf(i);  else cout << "-";
        cout << "\n";
    }

    cout << "\nheight of the heap = floor(lg " << n << ") = " << heightOfHeap(n) << "\n";
    cout << "leaves occupy positions " << n / 2 + 1 << " .. " << n << "\n";
```

Kode 5.1 (Lanjutan) *Setiap fakta struktural tentang heap dari Subbab 5.3, dihitung dan diperiksa.*

```
    cout << "\nNodes at each height h, against the bound ceil(n/2^(h+1)):\n";
    for (int h = 0; h <= heightOfHeap(n); ++h) {
        int actual = 0;
        for (int i = 1; i <= n; ++i) if (subtreeHeight(i) == h) ++actual;
        int bound = (n + (1 << (h + 1)) - 1) / (1 << (h + 1));  // ceil(n / 2^(h+1))
        cout << "  h = " << h << " : actual = " << actual
             << ", bound = " << bound << "\n";
    }

    bool ok = true;                             // verifikasi sifat max-heap di mana-mana
    for (int i = 2; i <= n; ++i) if (A[parentOf(i)] < A[i]) ok = false;
    cout << "\nmax-heap property holds: " << (ok ? "yes" : "no") << "\n";
    return 0;
}
```

Dikompilasi dengan `g++ -std=c++17 -O2` lalu dijalankan, program itu mencetak:

```
Array (1-based): 16 15 10 8 7 9 3 2 4 1

index  value  parent  left  right
    1     16       -     2      3
    2     15       1     4      5
    3     10       1     6      7
    4      8       2     8      9
    5      7       2    10      -
    6      9       3     -      -
    7      3       3     -      -
    8      2       4     -      -
    9      4       4     -      -
   10      1       5     -      -

height of the heap = floor(lg 10) = 3
leaves occupy positions 6 .. 10

Nodes at each height h, against the bound ceil(n/2^(h+1)):
  h = 0 : actual = 5, bound = 5
  h = 1 : actual = 3, bound = 3
  h = 2 : actual = 1, bound = 2
  h = 3 : actual = 1, bound = 1

max-heap property holds: yes
```

Bacalah keluaran itu berdampingan dengan teorinya. Tingginya 3, seperti yang dibuktikan. Daun-daunnya berjalan dari 6 sampai 10, seperti yang dibuktikan — dan $\lfloor 10/2 \rfloor + 1 = 6$ ✓. Simpul 5 mempunyai anak kiri di indeks 10 tetapi tidak punya anak kanan, sebab $11 > 10$; programnya mencetak tanda hubung alih-alih mengarang satu, dan membereskan batas itu dengan benar di kode Anda sendiri adalah beda antara *heap* yang bekerja dan sebuah kerusakan. Lalu cacah simpul menurut tinggi, 5,3,1,1, tidak pernah melampaui batas $\lceil n/2^{h+1} \rceil = 5,3,2,1$ — batas itu ketat pada tinggi 0,1,3 dan longgar pada tinggi 2, yang persis sebagaimana diizinkan oleh kata "paling banyak".

5.4 `MAX-HEAPIFY`: Mesin yang Menjaga Heap Tetap Jujur

5.4.1 Persoalan yang dipecahkannya

Segala yang dikerjakan sebuah *heap* — mengurutkan, mengeluarkan, menyisipkan — mengerucut pada satu operasi perbaikan. Inilah keadaan yang menjadi alasan operasi itu dibangun.

Bayangkan sebuah simpul yang kedua subpohonnya sudah merupakan *heap* yang sempurna, tetapi nilainya sendiri terlalu kecil, sehingga ia dikalahkan oleh salah satu atau kedua anaknya. Sifat *heap* gagal tepat di satu tempat: simpul ini. Di semua tempat lain sifat itu berlaku.

```
          4        <-- terlalu kecil: sifat heap dilanggar di sini
        /   \
      14     10    <-- tetapi KEDUA subpohon ini sudah merupakan heap yang sah
     /  \   /  \
    8    7 9    3
```

Nilai 4 lebih kecil daripada kedua anaknya, 14 dan 10. Tetapi lihatlah ke bawah: subpohon yang berakar di 14 adalah *heap* yang sah ($14 \geq 8$, $14 \geq 7$), dan begitu pula subpohon yang berakar di 10 ($10 \geq 9$, $10 \geq 3$). Kerusakannya terkurung pada satu simpul.

> **DEFINISI.** `MAX-HEAPIFY`(A, i) (ditulis juga `HEAPIFY`). *Prasyarat:* pohon-pohon biner yang berakar di Left(i) dan Right(i) sudah merupakan *max-heap*, tetapi $A[i]$ boleh jadi lebih kecil daripada salah satu atau kedua anaknya. *Pascasyarat:* subpohon yang berakar di i adalah *max-*

> *heap*, yang memuat tepat multihimpunan nilai yang sama dengan yang dimuatnya sebelumnya. Prosedur itu mencapainya dengan membiarkan nilai di $A[i]$ **tenggelam** — berulang kali menukarnya dengan anaknya yang lebih besar — sampai ia mencapai posisi yang membuatnya mendominasi kedua anaknya, atau sampai ia menjadi daun.

Kata yang perlu dipatrikan dalam ingatan adalah *tenggelam*. Nilai yang mengganggu itu tenggelam menembus pohon seperti batu menembus air, dan segala yang dilewatinya terangkat naik satu tingkat.

5.4.2 Algoritmanya

Pseudokode 5.1 memberikan prosedurnya. Anda juga akan menjumpainya dengan nama yang lebih pendek, `HEAPIFY`, dan dengan nama itulah baris 10 merujuk kepadanya:

Pseudokode 5.1 `MAX-HEAPIFY`, yang juga disebut `HEAPIFY`: biarkan nilai di simpul i tenggelam sampai ia mendominasi kedua anaknya.

```
HEAPIFY(A, i):
 1.  l ← LEFT(i)                          // l = 2i
 2.  r ← RIGHT(i)                         // r = 2i + 1
 3.  largest ← i
 4.  if l ≤ heap-size and A[l] > A[i]
 5.      then largest ← l
 6.  if r ≤ heap-size and A[r] > A[largest]
 7.      then largest ← r
 8.  if largest ≠ i
 9.      swap A[i] ↔ A[largest]
10.      HEAPIFY(A, largest)             // turun secara rekursif
```

Dengan bahasa sehari-hari: lihatlah tiga nilai — simpulnya dan kedua anaknya. Tentukan mana di antara ketiganya yang terbesar. Jika simpulnya sendiri yang terbesar, segalanya sudah teratur dan tidak ada yang perlu dikerjakan. Kalau tidak, tukarkan simpul itu dengan anaknya yang terbesar, lalu ulangi seluruh prosedurnya pada posisi yang baru saja ditempati simpul itu.

Tiga perincian dalam sepuluh baris itu mudah sekali salah, jadi kita sebutkan namanya.

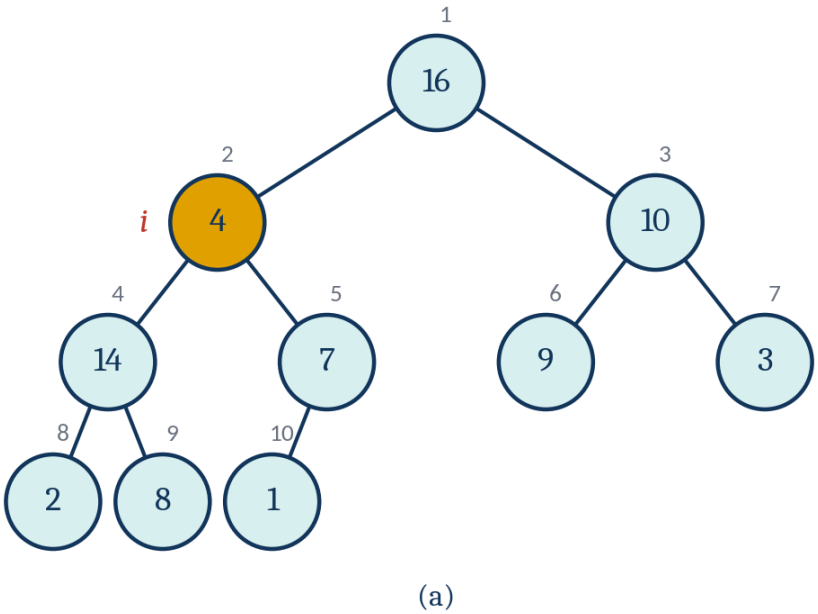
- **Penjaga `l ≤ heap-size`.** Sebuah simpul di dekat dasar boleh jadi punya satu anak atau tidak punya sama sekali. Membaca $A[2i]$ tanpa memeriksa berarti membaca melewati ujung *heap*-nya, yang dalam C++ merupakan perilaku tak terdefinisi dan dalam praktik merupakan kutu yang hanya muncul pada panjang larik tertentu.
- **`heap-size`, bukan panjang lariknya.** Keduanya adalah dua bilangan yang berbeda. Lariknya boleh panjang; *heap*-nya boleh jadi hanya menempati awalannya saja. Heapsort di Subbab 5.6 sepenuhnya bergantung pada pembedaan itu, sebab ia menyusutkan *heap*-nya sambil membiarkan lariknya utuh.
- **Baris 6 membandingkan terhadap `A[largest]`, bukan `A[i]`.** Sesudah baris 5 mungkin menyetel `largest ← l`, yang kita inginkan adalah yang terbesar di antara *tiga* nilai. Membandingkan anak kanan terhadap $A[i]$ alih-alih terhadap juara saat ini adalah kutu klasiknya: ia dapat memilih anak kanan bahkan ketika anak kiri lebih besar, dan sifat *heap* lalu gagal satu tingkat di bawahnya.

5.4.3 Penelusuran lengkap

Ambil sebuah contoh yang konkret. Lariknya adalah

$$A = [16,\ 4,\ 10,\ 14,\ 7,\ 9,\ 3,\ 2,\ 8,\ 1], \qquad \text{heap-size} = 10,$$

dan $A[2] = 4$ adalah biang keladinya. Kita memanggil `MAX-HEAPIFY(A, 2)`. **Gambar 5.3** adalah gambar keadaan awalnya.



Gambar 5.3 `MAX-HEAPIFY(A, 2)`, langkah (a). Pemanggilannya dimulai. Simpul $i = 2$, yang diarsir, memuat nilai 4. Anak-anaknya adalah $A[4] = 14$ dan $A[5] = 7$. Kedua subpohon di bawahnya sudah merupakan *max-heap* yang sah; hanya simpul 2 yang salah tempat.

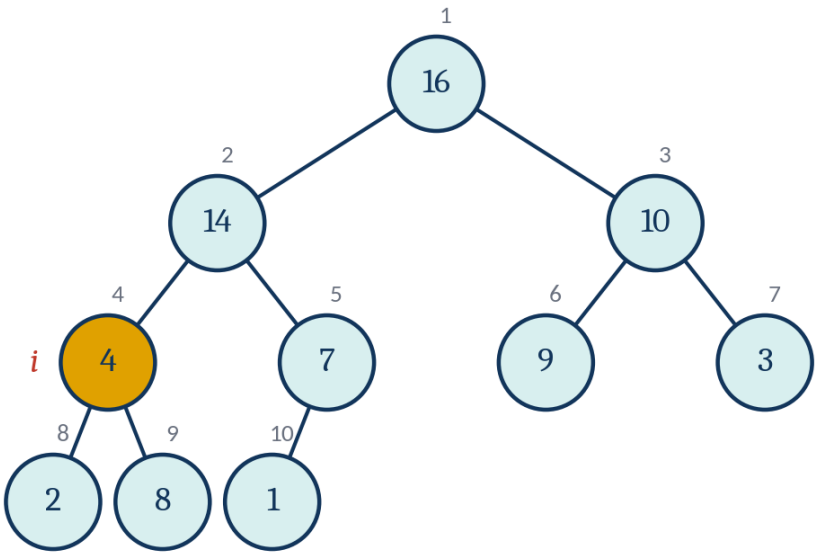
Langkah 1 — pemanggilan `MAX-HEAPIFY(A, 2)`. Kerjakan pseudokodenya satu baris demi satu baris.

- Baris 1: $l \leftarrow \text{Left}(2) = 2 \times 2 = 4$.
- Baris 2: $r \leftarrow \text{Right}(2) = 2 \times 2 + 1 = 5$.
- Baris 3: largest $\leftarrow 2$.
- Baris 4: apakah $l = 4 \leq 10$? Ya. Apakah $A[4] = 14 > A[2] = 4$? Ya.
- Baris 5: maka largest $\leftarrow 4$.
- Baris 6: apakah $r = 5 \leq 10$? Ya. Apakah $A[5] = 7 > A[\text{largest}] = A[4] = 14$? Tidak.
- Baris 7 dilewati; largest tetap 4.
- Baris 8: apakah largest $= 4 \neq i = 2$? Ya.
- Baris 9: tukarkan $A[2] \leftrightarrow A[4]$. Lariknya menjadi

$$A = [16,\ \mathbf{14},\ 10,\ \mathbf{4},\ 7,\ 9,\ 3,\ 2,\ 8,\ 1].$$

- Baris 10: rekursi pada `MAX-HEAPIFY(A, 4)`.

Gambar 5.4 menunjukkan keadaannya sekarang. Angka 4 sudah tenggelam satu tingkat, ke indeks 4, dan angka 14 sudah naik ke indeks 2.



(b)

Gambar 5.4 Langkah (b). Sesudah penukaran pertama. Nilai 14 sudah mengambil tempat 4 di indeks 2, dan si pengganggu 4 kini berada di indeks $i = 4$, yang diarsir. Anak-anaknya adalah $A[8] = 2$ dan $A[9] = 8$ — dan $8 > 4$, jadi penenggelamannya belum selesai.

Langkah 2 — pemanggilan `MAX-HEAPIFY(A, 4)`.

- $l \leftarrow 2 \times 4 = 8, r \leftarrow 2 \times 4 + 1 = 9$, largest $\leftarrow 4$.
- Apakah $8 \leq 10$? Ya. Apakah $A[8] = 2 > A[4] = 4$? Tidak. largest tetap 4.
- Apakah $9 \leq 10$? Ya. Apakah $A[9] = 8 > A[\text{largest}] = A[4] = 4$? Ya, jadi largest $\leftarrow 9$.
- largest $= 9 \neq 4$, jadi tukarkan $A[4] \leftrightarrow A[9]$:

$$A = [16,\ 14,\ 10,\ \mathbf{8},\ 7,\ 9,\ 3,\ 2,\ \mathbf{4},\ 1].$$

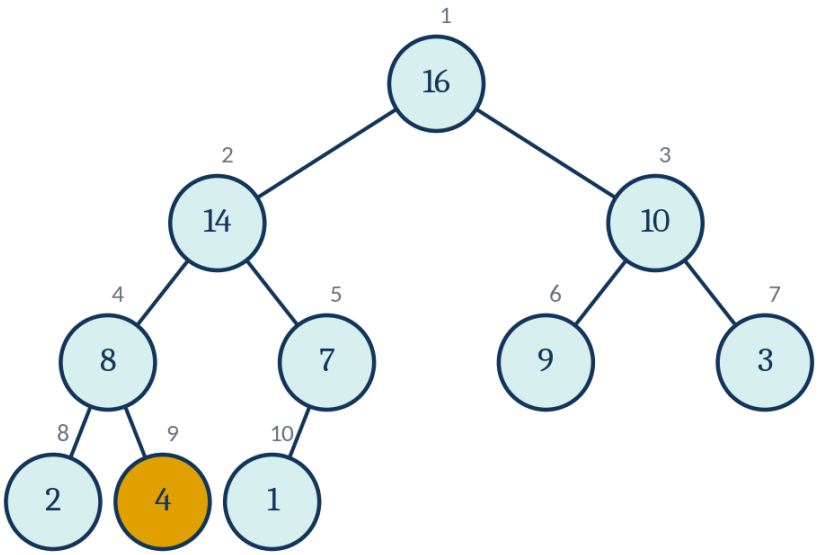
- Rekursi pada `MAX-HEAPIFY(A, 9)`.

Perhatikan bahwa langkah inilah tempat aturan "bandingkan terhadap `A[largest]`" membuktikan gunanya. Seandainya baris 6 membandingkan $A[9] = 8$ terhadap $A[i] = 4$, kita akan memperoleh jawaban yang sama di sini karena kebetulan — tetapi seandainya anak-anaknya adalah $A[8] = 6$ dan $A[9] = 5$, versi yang berkutu itu akan menaikkan 5 alih-alih 6, sehingga meninggalkan $6 > 5$ satu tingkat di bawah dan sebuah *heap* yang rusak.

Langkah 3 — pemanggilan `MAX-HEAPIFY(A, 9)`.

- $l \leftarrow 2 \times 9 = 18, r \leftarrow 19$, largest $\leftarrow 9$.
- Apakah $18 \leq 10$? **Tidak.** Pengujian pada baris 4 gagal pada syarat pertamanya dan syarat keduanya tidak pernah dievaluasi. Lewati.
- Apakah $19 \leq 10$? Tidak. Lewati.
- largest $= 9 = i$, jadi baris 8 gagal, tidak ada penukaran yang terjadi, dan prosedurnya kembali.

Simpul 9 adalah daun, dan daun secara trivial merupakan *heap*. Rekursinya sudah menyentuh dasar. **Gambar 5.5** adalah keadaan akhirnya.



(c)

Gambar 5.5 Langkah (c). Perbaikannya rampung. Nilai 4 sudah berhenti di indeks 9, sebuah daun, dan seluruh pohonnya sekali lagi menjadi *max-heap*. Nilai itu menempuh lintasan $2 \to 4 \to 9$, yang merupakan satu lintasan akar-ke-daun dan tidak lebih.

Lariknya yang terakhir adalah

$$A = [16,\ 14,\ 10,\ 8,\ 7,\ 9,\ 3,\ 2,\ 4,\ 1],$$

yaitu *heap* yang kita periksa dengan tangan di Subbab 5.3.4. Nilai 4 tenggelam dari posisi 2 ke posisi 4 lalu ke posisi 9 — **tiga posisi di sepanjang satu lintasan menurun tunggal**, dan tidak ada apa pun di tempat lain dalam pohon itu yang tersentuh.

5.4.4 Mengapa ia benar

Prosedurnya rekursif, jadi pembuktiannya dengan induksi atas tinggi simpul i.

Kasus basis. Jika i adalah daun, ia tidak punya anak, kedua penjaganya gagal, largest $= i$, dan prosedurnya langsung kembali dengan meninggalkan A tanpa perubahan. Simpul tunggal adalah *max-heap*. ✓

Langkah induktif. Andaikan prosedurnya benar pada setiap simpul bertinggi kurang dari h, dan misalkan i bertinggi $h \geq 1$. Menurut prasyaratnya, subpohon di Left(i) dan Right(i) adalah *max-heap*. Baris 1–7 menghitung largest sedemikian rupa sehingga A[largest] adalah yang terbesar di antara $A[i]$, A[Left(i)], dan A[Right(i)] — dengan anak yang tidak ada memang tidak ikut dipertimbangkan. Dua kasus:

- Jika largest $= i$, maka $A[i]$ sudah mendominasi kedua anaknya. Karena kedua subpohonnya adalah *max-heap*, setiap simpul di dalamnya didominasi oleh induknya sendiri, dan akar-akar subpohon itu didominasi oleh $A[i]$. Jadi seluruh subpohon di i adalah *max-heap* dan kita selesai, dengan benar, tanpa mengerjakan apa-apa.
- Jika largest $\neq i$ — katakanlah largest $=$ Left(i) — maka sesudah penukarannya, $A[i]$ memuat yang terbesar di antara ketiga nilainya, sehingga ia mendominasi A[Right(i)] (tak berubah, masih akar sebuah *heap*) dan A[Left(i)] (yang kini memuat $A[i]$ yang lama). Subpohon kanannya tak tersentuh dan masih *max-heap*. Subpohon kirinya kini punya akar yang mungkin terlalu kecil tetapi dua anak yang merupakan *heap* yang sah — yang justru merupakan

prasyarat `MAX-HEAPIFY` pada Left(i). Simpul itu bertinggi paling banyak $h-1$, jadi menurut hipotesis induktifnya pemanggilan rekursif itu memperbaikinya. Seluruh subpohon di i lalu menjadi *max-heap*. ✓

Kasus simetrisnya, largest = Right(i), identik dengan kiri dan kanan bertukar. Dan tidak ada elemen yang pernah diciptakan atau dimusnahkan — setiap baris entah membaca sebuah nilai atau menukarkan dua di antaranya — sehingga multihimpunan nilainya terpelihara. ■

5.4.5 Waktu jalannya

Setiap pemanggilan mengerjakan sejumlah tetap pekerjaan: dua penghitungan indeks, paling banyak dua pembandingan, paling banyak satu penukaran. Itu $\Theta(1)$. Lalu ia mungkin berekursi pada satu anak.

Jadi biaya totalnya adalah $\Theta(1)$ dikalikan banyaknya tingkat yang dituruni nilai itu — dan karena setiap pemanggilan menuruni tepat satu tingkat, itu paling banyak sebesar tinggi simpul i. Dengan tinggi seluruh *heap*-nya adalah $\lfloor \lg n \rfloor$ (Subbab 5.3.6), kita langsung memperoleh:

$$T(n) = O(\lg n).$$

Itulah hujah satu barisnya yang jujur, dan hujah itu sudah sepenuhnya memadai. Tetapi versi rekurensinya pun layak dikerjakan dengan semestinya, sebab ia merupakan latihan yang bagus untuk Teorema Master dari Bab 3.

Menuliskan rekurensinya. Ketika `MAX-HEAPIFY` berekursi, ia berekursi pada salah satu dari dua subpohon simpul i. Seberapa besar subpohon itu bisa jadi, sebagai pecahan dari subpohon di i?

Andaikan subpohon di i mempunyai m simpul. Kalau pohon di bawah i lengkap sempurna, setiap subpohon anak akan memuat $(m-1)/2 < m/2$ simpul dan jawabannya adalah separuh yang rapi. Tetapi pohonnya tidak harus lengkap: tingkat *terakhir*-nya boleh terisi sebagian, dan ia terisi **dari kiri**. Kasus terburuk untuk maksud kita karena itu adalah subpohon yang tingkat dasarnya terisi tepat setengah — sebab saat itulah subpohon anak kiri sebesar-besarnya relatif terhadap subpohon anak kanan.

Kerjakanlah. Katakanlah subpohon di i bertinggi h, tingkat 0 sampai $h-1$-nya (dihitung dari simpul i) penuh, dan tingkat dasarnya, tingkat h, terisi tepat setengah. Cacahlah simpulnya dalam tiga langkah.

- *Seluruh subpohonnya.* Tingkat 0 sampai $h-1$ memuat $1+2+\cdots+2^{h-1} = 2^h - 1$ simpul. Tingkat dasar yang penuh akan memuat 2^h simpul; setengahnya memuat 2^{h-1}. Jadi $m = (2^h - 1) + 2^{h-1}$.
- *Subpohon anak kirinya.* Setiap tingkat 1 sampai $h-1$ memberikan tepat separuh simpulnya kepada sisi kiri, yaitu $1+2+\cdots+2^{h-2} = 2^{h-1}-1$ simpul. Dan karena tingkat dasarnya terisi dari kiri, **semua** 2^{h-1} simpulnya juga menjadi milik sisi kiri. Totalnya: $(2^{h-1}-1)+2^{h-1} = 2^h - 1$.
- *Subpohon anak kanannya.* Ia hanya mendapat separuh tingkat-tingkat atasnya, $2^{h-1}-1$ simpul, dan tidak mendapat apa pun dari tingkat dasarnya. (Periksa: $1+(2^h-1)+(2^{h-1}-1) = 2^h + 2^{h-1} - 1 = m$ ✓.)

Bagian sisi kiri dari seluruh subpohonnya karena itu adalah

$$\frac{2^h - 1}{2^h - 1 + 2^{h-1}} = \frac{2 \cdot 2^{h-1} - 1}{3 \cdot 2^{h-1} - 1} < \frac{2}{3}.$$

Mengapa langkah terakhir itu benar? Tuliskan $x = 2^{h-1}$. Klaim $\frac{2x-1}{3x-1} < \frac{2}{3}$, sesudah kedua ruasnya dikalikan dengan bilangan positif 3 dan $3x - 1$, menjadi klaim $3(2x - 1) < 2(3x - 1)$, yaitu $6x - 3 < 6x - 2$, yang selalu benar. Ketika h membesar, bagian itu merayap mendekati 2/3 tetapi tidak pernah mencapainya. Cara pengisian tingkat dasar yang lain memberi sisi kiri bagian yang lebih kecil: tingkat dasar yang terisi kurang dari setengah memberi sisi kiri lebih sedikit simpul, sedangkan tingkat dasar yang terisi lebih dari setengah memberikan simpul tambahannya kepada sisi kanan.

Jadi subpohon seorang anak tidak pernah memuat lebih dari 2/3 simpul subpohon induknya, dan rekurensinya adalah

$$T(n) \leq T(2n/3) + \Theta(1). \tag{5.5}$$

Menyelesaikannya dengan Teorema Master. Pada Persamaan (5.5) kita punya $a = 1$, $b = 3/2$, dan $f(n) = \Theta(1) = \Theta(n^0)$. Hitung eksponen pembatasnya:

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1,$$

sebab $\log_b 1 = 0$ untuk basis b mana pun. Jadi $f(n) = \Theta(1) = \Theta(n^{\log_b a} \lg^0 n)$, yang merupakan Kasus 2 Teorema Master dengan $k = 0$. Kasus 2 memberikan

$$T(n) = \Theta(n^{\log_b a} \lg^{k+1} n) = \Theta(n^0 \lg^1 n) = \Theta(\lg n).$$

Karena rekurensinya berupa ketaksamaan ($\leq$), kesimpulannya adalah sebuah batas atas:

$$T(n) = O(\lg n). \qquad \blacksquare$$

> **DENGAN BAHASA SEHARI-HARI.** `MAX-HEAPIFY` berjalan menuruni satu lintasan dari simpul yang diberikan kepadanya sampai, pada kasus terburuk, sebuah daun. Sebuah *heap* berisi n elemen hanya setinggi $\lfloor \lg n \rfloor$ tingkat, jadi perjalanan itu pendek. Untuk sejuta elemen perjalanannya paling banyak sembilan belas langkah. Untuk semiliar elemen paling banyak dua puluh sembilan. Itulah seluruh klaim $O(\lg n)$; rekurensi di atas semata-mata fakta yang sama yang dituliskan dalam bahasa Bab 3.

5.4.6 `MAX-HEAPIFY` dalam C++

Kode 5.2 mengimplementasikan prosedurnya dan mencetak setiap keputusan yang diambilnya, supaya Anda dapat membandingkan penelusuran mesinnya dengan penelusuran tangan di Subbab 5.4.3 baris demi baris.

Kode 5.2 `MAX-HEAPIFY` dalam C++, yang menarasikan setiap keputusannya sendiri.

```
// MAX-HEAPIFY, dengan setiap langkah penenggelamannya dicetak.
#include <iostream>
#include <vector>
#include <string>
using namespace std;

vector<int> A;          // A[0] tak terpakai supaya aritmetika indeksnya sama dengan di
buku teks
int heapSize;

void show(const string& label) {
    cout << label;
    for (int i = 1; i <= heapSize; ++i) cout << A[i] << (i < heapSize ? " " : "");
    cout << "]\n";
}

void maxHeapify(int i) {
    int l = 2 * i, r = 2 * i + 1, largest = i;
    if (l <= heapSize && A[l] > A[largest]) largest = l;
    if (r <= heapSize && A[r] > A[largest]) largest = r;

    cout << "  MAX-HEAPIFY(A, " << i << "): A[" << i << "]=" << A[i];
    if (l <= heapSize) cout << ", left A[" << l << "]=" << A[l]; else cout << ", no left
child";
    if (r <= heapSize) cout << ", right A[" << r << "]=" << A[r]; else cout << ", no right
child";
    cout << " -> largest is index " << largest << "\n";

    if (largest != i) {
        swap(A[i], A[largest]);
        show("    swap, giving A = [");
        maxHeapify(largest);                    // nilainya terus tenggelam
    } else {
        cout << "    node " << i << " already dominates its children: stop\n";
    }
}

int main() {
    A = {0, 16, 4, 10, 14, 7, 9, 3, 2, 8, 1};   // A[2] = 4 merusak sifat heap-nya
    heapSize = 10;
    show("start:  A = [");
    cout << "\ncalling MAX-HEAPIFY(A, 2)\n";
    maxHeapify(2);
    show("\nresult: A = [");
```

Kode 5.2 (Lanjutan) *MAX-HEAPIFY dalam C++, yang menarasikan setiap keputusannya sendiri.*

```
    bool ok = true;
    for (int i = 2; i <= heapSize; ++i) if (A[i / 2] < A[i]) ok = false;
    cout << "valid max-heap: " << (ok ? "yes" : "no") << "\n";
    return 0;
}
```

Programnya mencetak:

```
start:  A = [16 4 10 14 7 9 3 2 8 1]

calling MAX-HEAPIFY(A, 2)
  MAX-HEAPIFY(A, 2): A[2]=4, left A[4]=14, right A[5]=7 -> largest is index 4
    swap, giving A = [16 14 10 4 7 9 3 2 8 1]
  MAX-HEAPIFY(A, 4): A[4]=4, left A[8]=2, right A[9]=8 -> largest is index 9
    swap, giving A = [16 14 10 8 7 9 3 2 4 1]
  MAX-HEAPIFY(A, 9): A[9]=4, no left child, no right child -> largest is index 9
    node 9 already dominates its children: stop

result: A = [16 14 10 8 7 9 3 2 4 1]
valid max-heap: yes
```

Tiga pemanggilan, tiga tingkat, persis seperti yang diramalkan penelusuran tangan tadi — dan lariknya yang terakhir cocok dengan **Gambar 5.5** elemen demi elemen. Perhatikan hubung-singkat pada `l <= heapSize && A[l] > A[largest]`: ketika `l` berada di luar rentang, C++ tidak pernah mengevaluasi `A[l]`, dan itulah sebabnya pemanggilan ketiga mencetak "no left child" alih-alih rusak.

> **JEBAKAN.** Versi mahasiswa yang lazim menulis `if (A[l] > A[largest] && l <= heapSize)`. Ini kutu yang sungguhan. C++ mengevaluasi operan kirinya lebih dulu, sehingga `A[l]` dibaca *sebelum* pengujian batasnya, dan pembacaan itu berada di luar rentang. Sering kali ia akan tampak bekerja — memori tepat di belakang `vector` biasanya memuat sesuatu yang tak berbahaya — lalu gagal pada masukan lain atau pada kompilator lain. Urutan itu penting.

5.5 `BUILD-MAX-HEAP`: Pembangunan Waktu Linear yang Mengejutkan

5.5.1 Algoritmanya

Kita sudah bisa memperbaiki sebuah *heap*. Kini kita perlu *membuat* satu, dari larik yang urutannya sembarang. **Pseudokode 5.2** mengerjakannya.

Pseudokode 5.2 `BUILD-MAX-HEAP`: perbaiki setiap simpul dalam, dari yang terakhir kembali naik sampai ke akar.

```
BUILD-MAX-HEAP(A):
 1.  heap-size ← n
 2.  for i ← ⌊n/2⌋ downto 1
 3.      MAX-HEAPIFY(A, i)
```

Tiga baris. Panggil `MAX-HEAPIFY` pada setiap simpul dalam, dari yang terakhir sampai yang pertama.

Dua pilihan dalam tiga baris itu memikul seluruh beban prosedurnya, dan masing-masing perlu penjelasan.

Mengapa mulai dari $\lfloor n/2 \rfloor$? Sebab menurut teorema di Subbab 5.3.7, simpul $\lfloor n/2 \rfloor + 1$ sampai n justru adalah daun-daunnya, dan daun sudah merupakan *heap* satu elemen yang sah. Tidak ada yang perlu diperbaiki. Memanggil `MAX-HEAPIFY` pada mereka akan benar tetapi sia-sia — dan, seperti akan ditunjukkan Subbab 5.5.6, melewatkannya bukanlah penghematan kecil: itu *separuh lariknya*.

Mengapa mencacah menurun? Sebab `MAX-HEAPIFY`(A, i) punya sebuah prasyarat: kedua subpohon i harus sudah merupakan *heap*. Anak-anak simpul i duduk di indeks $2i$ dan $2i + 1$, keduanya benar-benar lebih besar daripada i. Jadi kalau kita memproses indeks dalam urutan menurun, setiap anak selesai sebelum induknya dimulai, dan prasyaratnya terpenuhi dengan

sendirinya pada setiap pemanggilan. Mencacah *menaik* akan memanggil `MAX-HEAPIFY` pada sebuah simpul yang subpohonnya masih berantakan, prasyaratnya dilanggar, dan hasilnya sekadar salah.

5.5.2 Kebenarannya, dengan invarian gelung

> **INVARIAN GELUNG.** Pada awal setiap iterasi gelung `for` pada baris 2, setiap simpul $i+1, i+2, \ldots, n$ adalah akar sebuah *max-heap*.

Inisialisasi. Sebelum iterasi pertama, $i = \lfloor n/2 \rfloor$. Simpul $\lfloor n/2 \rfloor + 1, \ldots, n$ adalah daun, dan setiap daun merupakan *max-heap* satu elemen. Invariannya berlaku. ✓

Pemeliharaan. Andaikan invariannya berlaku pada awal sebuah iterasi dengan pencacah i. Anak-anak simpul i berada di $2i$ dan $2i+1$, dan kedua indeks itu lebih besar daripada i, sehingga menurut invariannya keduanya adalah akar *max-heap*. Itu persis prasyarat `MAX-HEAPIFY`, jadi pemanggilan pada baris 3 sah dan — menurut pembuktian kebenaran di Subbab 5.4.4 — pemanggilan itu menjadikan simpul i akar sebuah *max-heap*. Ia tidak mengusik apa pun di luar subpohon di i, sehingga simpul $i+1, \ldots, n$ pun tetap menjadi akar *heap*. Gelungnya lalu mengurangi i, dan pada awal iterasi berikutnya simpul $i, i+1, \ldots, n$ semuanya adalah akar *heap*, yang merupakan invariannya dengan i diganti oleh $i-1$. ✓

Terminasi. Gelungnya berakhir ketika $i = 0$. Substitusikan ke dalam invariannya: setiap simpul $1,2, \ldots, n$ adalah akar sebuah *max-heap*. Khususnya simpul 1, dan subpohon yang berakar di simpul 1 adalah seluruh lariknya. Jadi $A[1 \ldots n]$ adalah sebuah *max-heap*. ✓ ■

5.5.3 Penelusuran lengkap

Ambil larik masukan ini,

$$A = [4,\ 1,\ 3,\ 2,\ 16,\ 9,\ 10,\ 14,\ 8,\ 7],$$

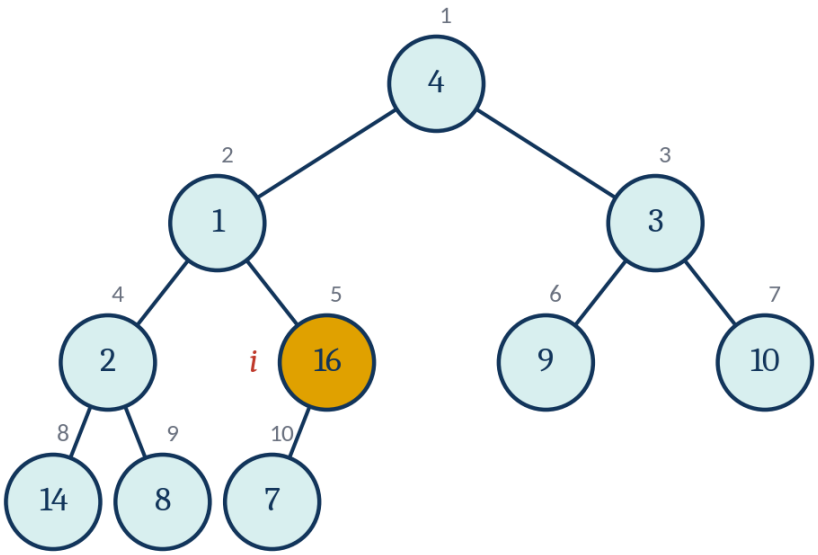
yang ditampilkan di **Gambar 5.6**. Di dalamnya tidak ada keteraturan sama sekali.

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

Gambar 5.6 Masukan bagi `BUILD-MAX-HEAP`: larik $A = [4,1,3,2,16,9,10,14,8,7]$, tanpa urutan tertentu.

Di sini $n = 10$, jadi $\lfloor n/2 \rfloor = 5$ dan gelungnya berjalan untuk $i = 5,4,3,2,1$. Simpul 6 sampai 10 — yang memuat 9,10,14,8,7 — adalah daun dan dilewati seluruhnya.

Iterasi $i = 5$. Gambar 5.7 adalah keadaannya.



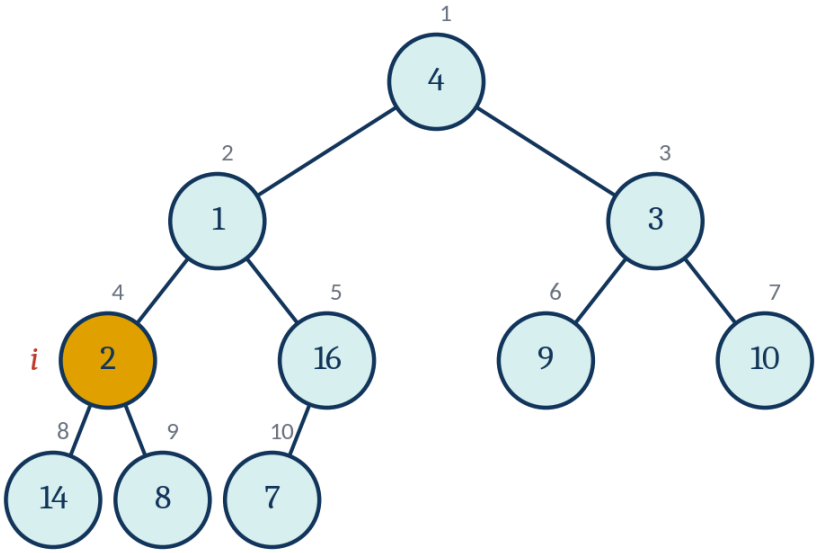
(a)

Gambar 5.7 `BUILD-MAX-HEAP`**, langkah (a):** $i = 5$**.** Lariknya digambar sebagai pohon hampir lengkap. Simpul 5, yang diarsir, memuat 16. Satu-satunya anaknya adalah $A[10] = 7$. Gelungnya mulai di sini karena simpul 5 adalah simpul terakhir yang masih punya anak sama sekali.

- $A[5] = 16$. Anak kiri $A[10] = 7$. Anak kanannya akan berupa $A[11]$, yang tidak ada sebab $11 > 10$.
- Apakah $16 > 7$? Ya, jadi simpul 5 sudah mendominasi satu-satunya anaknya.
- largest $= 5$: **tanpa penukaran**. Lariknya tak berubah:

$$A = [4,\ 1,\ 3,\ 2,\ 16,\ 9,\ 10,\ 14,\ 8,\ 7].$$

Iterasi $i = 4$. Gambar 5.8.



(b)

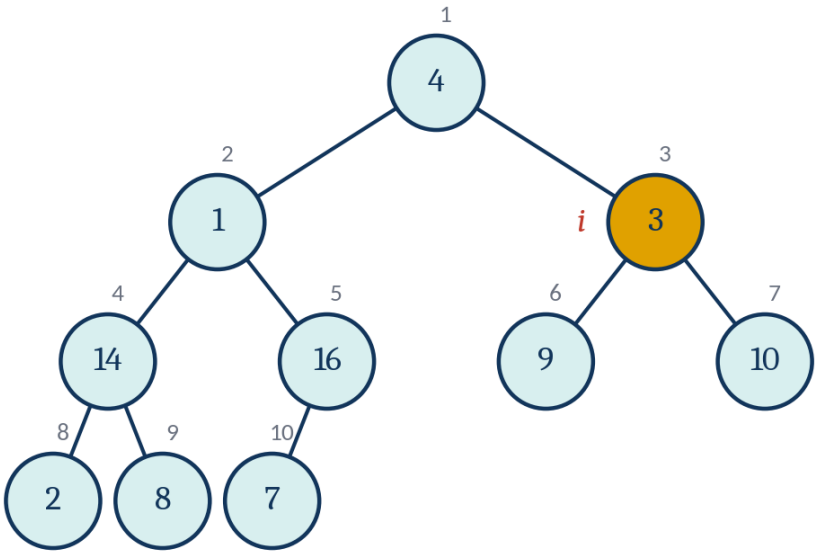
Gambar 5.8 Langkah (b): $i = 4$**.** Simpul 4, yang diarsir, memuat 2, dan kedua anaknya $A[8] = 14$ dan $A[9] = 8$ sama-sama lebih besar. Nilai 2 harus tenggelam.

- $A[4] = 2$. Anak-anaknya $A[8] = 14$, $A[9] = 8$.
- Terbesar di antara $\{2,14,8\}$ adalah 14, di indeks 8. Tukarkan $A[4] \leftrightarrow A[8]$:

$$A = [4,\ 1,\ 3,\ \mathbf{14},\ 16,\ 9,\ 10,\ \mathbf{2},\ 8,\ 7].$$

- Rekursi di indeks 8: $2 \times 8 = 16 > 10$, jadi simpul 8 adalah daun. Berhenti.

Iterasi $i = 3$. Gambar 5.9.



(c)

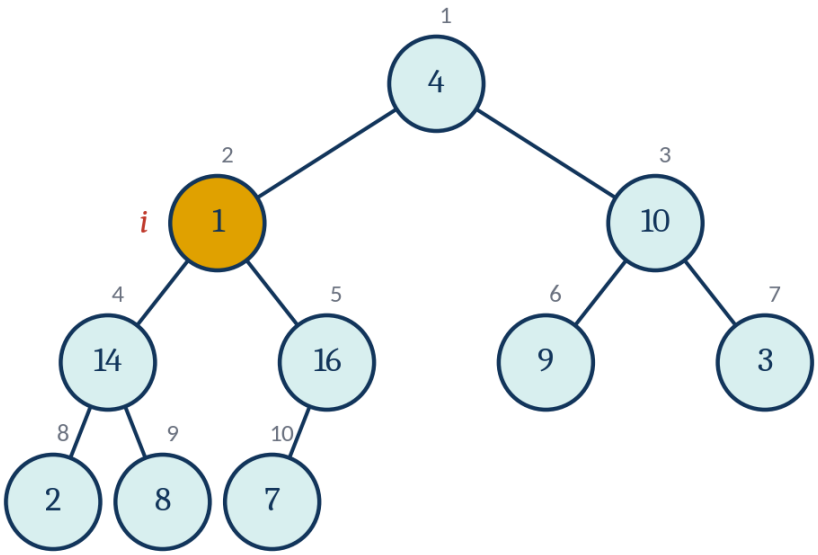
Gambar 5.9 Langkah (c): $i = 3$. Simpul 3, yang diarsir, memuat 3; anak-anaknya adalah $A[6] = 9$ dan $A[7] = 10$. Anak kanannya yang lebih besar, jadi penukarannya ke kanan.

- $A[3] = 3$. Anak-anaknya $A[6] = 9$, $A[7] = 10$.
- Terbesar di antara $\{3,9,10\}$ adalah 10, di indeks 7. Tukarkan $A[3] \leftrightarrow A[7]$:

$$A = [4,\ 1,\ \mathbf{10},\ 14,\ 16,\ 9,\ \mathbf{3},\ 2,\ 8,\ 7].$$

- Rekursi di indeks 7: $2 \times 7 = 14 > 10$, sebuah daun. Berhenti.

Iterasi $i = 2$. Gambar 5.10.



(d)

Gambar 5.10 Langkah (d): $i = 2$. Simpul 2, yang diarsir, memuat 1, nilai terkecil dalam lariknya. Ia harus tenggelam jauh: dua tingkat, ternyata.

- $A[2] = 1$. Anak-anaknya $A[4] = 14$, $A[5] = 16$.
- Terbesar di antara $\{1,14,16\}$ adalah 16, di indeks 5. Tukarkan $A[2] \leftrightarrow A[5]$:

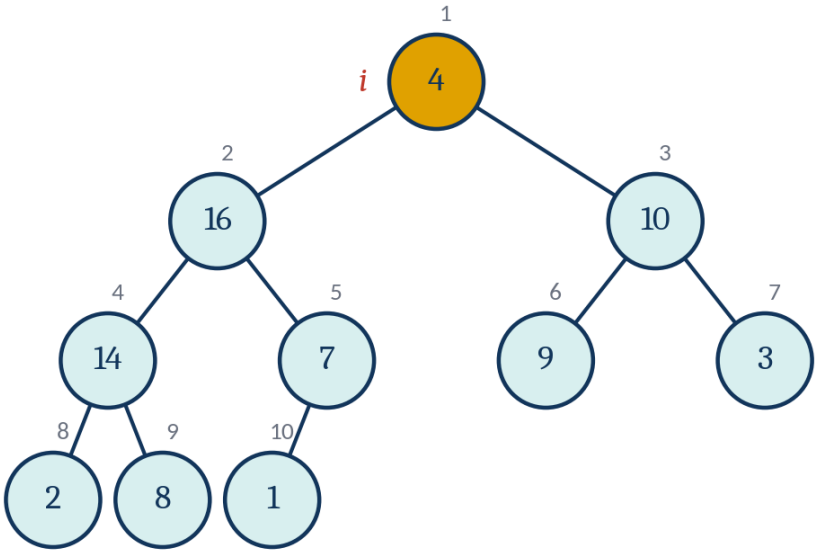
$$A = [4,\ \mathbf{16},\ 10,\ 14,\ \mathbf{1},\ 9,\ 3,\ 2,\ 8,\ 7].$$

- Rekursi di indeks 5. $A[5] = 1$, anak kiri $A[10] = 7$, tanpa anak kanan $(11 > 10)$.
- Terbesar di antara $\{1,7\}$ adalah 7. Tukarkan $A[5] \leftrightarrow A[10]$:

$$A = [4,\ 16,\ 10,\ 14,\ \mathbf{7},\ 9,\ 3,\ 2,\ 8,\ \mathbf{1}].$$

- Rekursi di indeks 10: sebuah daun. Berhenti.

Iterasi $i = 1$. Gambar 5.11.



(e)

Gambar 5.11 Langkah (e): $i = 1$. Iterasi terakhir, di akarnya. Simpul 1, yang diarsir, memuat 4. Segala yang di bawahnya sudah merupakan *heap* yang sah, jadi satu pemanggilan `MAX-HEAPIFY` ini merampungkan pekerjaannya.

- $A[1] = 4$. Anak-anaknya $A[2] = 16$, $A[3] = 10$.
- Terbesar di antara $\{4,16,10\}$ adalah 16, di indeks 2. Tukarkan $A[1] \leftrightarrow A[2]$:

$$A = [\mathbf{16},\ \mathbf{4},\ 10,\ 14,\ 7,\ 9,\ 3,\ 2,\ 8,\ 1].$$

- Rekursi di indeks 2. $A[2] = 4$, anak-anaknya $A[4] = 14$, $A[5] = 7$. Terbesarnya 14 di indeks 4. Tukarkan:

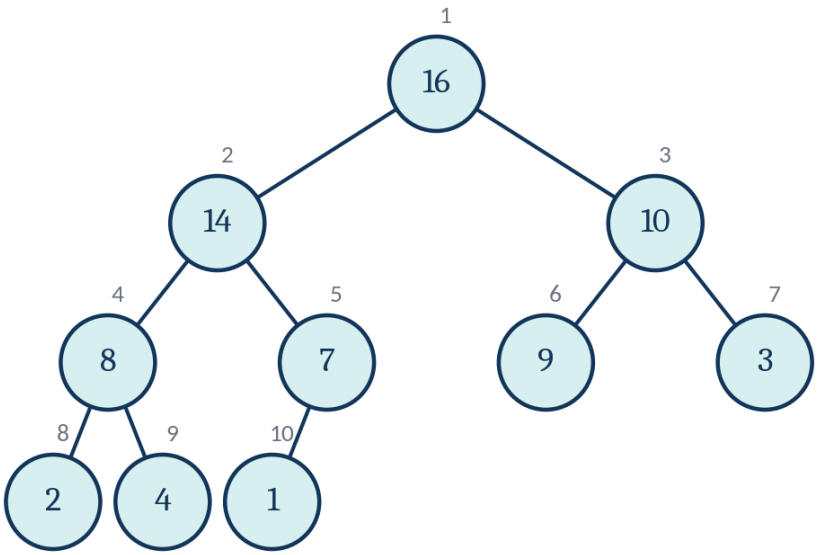
$$A = [16,\ \mathbf{14},\ 10,\ \mathbf{4},\ 7,\ 9,\ 3,\ 2,\ 8,\ 1].$$

- Rekursi di indeks 4. $A[4] = 4$, anak-anaknya $A[8] = 2$, $A[9] = 8$. Terbesarnya 8 di indeks 9. Tukarkan:

$$A = [16,\ 14,\ 10,\ \mathbf{8},\ 7,\ 9,\ 3,\ 2,\ \mathbf{4},\ 1].$$

- Rekursi di indeks 9: sebuah daun. Berhenti.

Gelungnya selesai. **Gambar 5.12** adalah hasilnya.



(f)

Gambar 5.12 Langkah (f): max-heap yang sudah jadi $A = [16,14,10,8,7,9,3,2,4,1]$. Setiap induk mendominasi kedua anaknya, dan maksimumnya, 16, sudah tiba di akar.

Amatilah bahwa iterasi terakhir saja menyebabkan tiga penukaran — nilai 4 jatuh dari akar sampai ke sebuah daun — sedangkan dua iterasi pertama menyebabkan paling banyak satu penukaran masing-masing. Ketaksetangkupan itulah benih analisis yang akan menyusul.

5.5.4 Batas yang longgar: $O(n\lg n)$

Inilah analisis yang akan dihasilkan seorang pembaca yang lalai.

Gelungnya berjalan $\lfloor n/2 \rfloor$ kali. Setiap iterasi memanggil `MAX-HEAPIFY`, yang berongkos $O(\lg n)$. Karena itu `BUILD-MAX-HEAP` berongkos $\lfloor n/2 \rfloor \times O(\lg n) = O(n\lg n)$.

Setiap langkah dalam kutipan itu **benar**. Batasnya benar. Ia semata-mata tidak *ketat*: biaya sesungguhnya jauh lebih kecil, dan $O(n\lg n)$ adalah batas atas yang jujur sebagaimana "ruangan ini memuat kurang dari sejuta orang" adalah batas atas yang jujur bagi sebuah ruang kelas.

Dari mana kelonggarannya datang? Dari frasa "setiap iterasi berongkos $O(\lg n)$". `MAX-HEAPIFY`(A, i) tidak berongkos $\lg n$; ia berongkos sebesar *tinggi simpul* i. Bagi akarnya, itu memang $\lg n$. Bagi sebuah simpul tepat di atas daun, itu 1. Dan, sebagaimana diberitahukan Subbab 5.3.8 kepada kita, sebagian besar simpul justru berada tepat di atas daun.

5.5.5 Batas yang ketat: $O(n)$, diturunkan selangkah demi selangkah

Kini kita mengerjakan analisisnya dengan semestinya. Tidak ada satu pun di sini yang lebih sukar daripada jumlahan-jumlahan Bab 3; ia hanya harus dikerjakan dengan cermat.

Langkah 1 — biaya satu pemanggilan, dinyatakan dalam tinggi. Sebuah pemanggilan `MAX-HEAPIFY` pada simpul bertinggi h menuruni paling banyak h tingkat, dengan mengerjakan $\Theta(1)$ pekerjaan per tingkat. Jadi ongkosnya $O(h)$.

Langkah 2 — berapa banyak simpul mempunyai tiap tinggi. Menurut lema di Subbab 5.3.8, banyaknya simpul bertinggi h paling banyak $\left\lceil \frac{n}{2^{h+1}} \right\rceil$.

Langkah 3 — tinggi-tinggi yang muncul. Tinggi berjalan dari 0 (daun-daunnya) sampai $\lfloor \lg n \rfloor$ (akarnya).

Langkah 4 — satukan semuanya. Menjumlahkan ongkosnya atas semua simpul, dikelompokkan menurut tinggi:

$$T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} \underbrace{\left\lceil \frac{n}{2^{h+1}} \right\rceil}_{\text{berapa simpul bertinggi } h} \times \underbrace{O(h)}_{\text{biaya masing-masing}}. \tag{5.6}$$

Langkah 5 — singkirkan langit-langitnya pada Persamaan (5.6). Kita memakai satu fakta kecil: untuk sembarang real $x \geq \frac{1}{2}$, berlaku $\lceil x \rceil \leq 2x$. (Jika $1/2 \leq x \leq 1$, maka $\lceil x \rceil = 1 \leq 2x$; jika $x > 1$, maka $\lceil x \rceil < x + 1 < 2x$.) Setiap tinggi h yang muncul memenuhi $h \leq \lg n$, yaitu $2^h \leq n$, sehingga $x = n/2^{h+1} \geq \frac{1}{2}$ dan karena itu $\lceil n/2^{h+1} \rceil \leq 2 \cdot n/2^{h+1} = n/2^h$. Jadi, dengan menarik keluar konstantanya dari O:

$$T(n) = O\left(\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} \cdot h\right) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right).$$

Langkah 6 — rentangkan jumlahannya sampai tak hingga. Setiap suku $h/2^h$ tak negatif, sehingga menambahkan lebih banyak suku hanya memperbesar jumlahannya. Mengganti batas atas berhingganya dengan ∞ karena itu mempertahankan ketaksamaannya — dan membelikan kita sebuah jumlahan yang punya bentuk tertutup:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h}.$$

Langkah 7 — hitung jumlahan tak hingganya. Kita buktikan di Subbab 5.5.6 bahwa nilainya tepat 2; hasilnya adalah Persamaan (5.8).

Langkah 8 — simpulkan.

$$T(n) = O(n \cdot 2) = O(n). \qquad \blacksquare$$

Dan karena `BUILD-MAX-HEAP` sekurang-kurangnya harus melihat kesemua n elemennya, $T(n) = \Omega(n)$ juga, sehingga sesungguhnya $T(n) = \Theta(n)$.

> **MENGAPA INI MENGEJUTKAN, DAN MENGAPA IA BENAR.** Membangun sebuah *heap* berongkos *lebih murah* daripada mengurutkannya — bahkan lebih murah daripada yang disiratkan oleh satu lintasan `MAX-HEAPIFY` per elemen. Alasannya adalah sebuah fakta pencacahan yang sederhana: **pemanggilan yang mahal itu langka dan pemanggilan yang murah itu banyak.** Tepat satu simpul bertinggi $\lg n$. Dua bertinggi $\lg n - 1$. Separuh dari semua simpul bertinggi 0 dan sama sekali tidak berongkos. Kalikan "sedikit dan mahal" dengan "banyak dan murah", dan jumlahannya konvergen alih-alih membesar. Inilah contoh paling jernih di seluruh buku ini tentang mengapa analisis yang *ketat* itu penting: batas longgar $O(n \lg n)$ akan membuat kita menolak sebuah algoritma $O(n)$ karena dikira terlalu lambat.

5.5.6 Identitas $\sum_{h=0}^{\infty} h/2^h = 2$, dibuktikan

Kita berutang pembuktian bagi Langkah 7. Pembuktiannya datang dari menurunkan deret geometri — teknik yang layak disimpan, sebab ia muncul lagi dalam peluang dan dalam analisis amortisasi (Bab 14).

Mulailah dari deret geometrinya. Untuk sembarang real x dengan $|x| < 1$,

$$\sum_{i=0}^{\infty} x^i = 1 + x + x^2 + x^3 + \cdots = \frac{1}{1-x}.$$

Inilah rumus dari Bab 3; ia konvergen justru karena $|x| < 1$.

Turunkan kedua ruasnya terhadap x. Di dalam jari-jari kekonvergenannya, sebuah deret pangkat boleh diturunkan suku demi suku. Di ruas kiri,

$$\frac{d}{dx}\sum_{i=0}^{\infty} x^i = \sum_{i=0}^{\infty} \frac{d}{dx} x^i = \sum_{i=1}^{\infty} i\, x^{i-1}$$

— suku $i = 0$ adalah konstanta 1, yang turunannya 0, dan itulah sebabnya indeksnya mulai dari 1. Di ruas kanan, dengan aturan hasil bagi atau aturan rantai,

$$\frac{d}{dx}\left(\frac{1}{1-x}\right) = \frac{d}{dx}(1-x)^{-1} = -1 \cdot (1-x)^{-2} \cdot (-1) = \frac{1}{(1-x)^2}.$$

Jadi

$$\sum_{i=1}^{\infty} i\, x^{i-1} = \frac{1}{(1-x)^2}.$$

Kalikan kedua ruasnya dengan x. Ini menggeser eksponennya naik satu, yang memang kita inginkan:

$$\sum_{i=1}^{\infty} i\, x^i = \frac{x}{(1-x)^2}. \tag{5.7}$$

Substitusikan $x = \frac{1}{2}$ ke dalam Persamaan (5.7)**.** Ini sah, sebab $|1/2| < 1$:

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = \frac{\frac{1}{2}}{\left(\frac{1}{2}\right)^2} = \frac{\frac{1}{2}}{\frac{1}{4}} = 2.$$

Akhirnya, suku pada $i = 0$ adalah $0/2^0 = 0$, sehingga menyertakannya tidak mengubah apa pun:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2. \qquad \blacksquare \tag{5.8}$$

Kalau Anda lebih suka melihat angkanya daripada aljabarnya, jumlahkan beberapa suku pertamanya dengan tangan:

$$\begin{aligned}&\frac{0}{1}+\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\frac{5}{32}+\frac{6}{64}+\frac{7}{128}\\&=0+0.5+0.5+0.375+0.25+0.15625+0.09375+0.0546875\\&=1.9296875,\end{aligned}$$

sudah berada dalam jarak empat persen dari 2 sesudah delapan suku, dan masih merangkak mendekatinya.

5.5.7 Melihat biaya linear itu pada mesin sungguhan

Teori satu hal; **Kode 5.3** adalah pemeriksaannya. Program itu mula-mula menelusuri `BUILD-MAX-HEAP` pada larik dari Subbab 5.5.3, dengan mencetak setiap penukaran, lalu menjalankan prosedurnya pada larik acak yang ukurannya membesar dan melaporkan banyaknya penukaran **dibagi** n. Kalau ongkosnya $\Theta(n\lg n)$, nisbah itu akan merangkak naik seperti $\lg n$. Kalau ongkosnya $\Theta(n)$, nisbah itu akan duduk pada sebuah konstanta.

Kode 5.3 `BUILD-MAX-HEAP`, ditelusuri pada larik dari Subbab 5.5.3 lalu diukur pada larik acak sampai sepuluh juta elemen.

```
// BUILD-MAX-HEAP: penelusuran pada larik dari Subbab 5.5.3, lalu bukti bahwa ongkosnya
O(n).
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <random>
using namespace std;

vector<int> A;
int heapSize;
long long swaps;                                // berapa banyak penukaran yang dilakukan
MAX-HEAPIFY

void show(const string& label) {
    cout << label;
    for (int i = 1; i <= heapSize; ++i) cout << A[i] << (i < heapSize ? " " : "");
    cout << "]\n";
}

void maxHeapify(int i, bool verbose) {
    int l = 2 * i, r = 2 * i + 1, largest = i;
    if (l <= heapSize && A[l] > A[largest]) largest = l;
    if (r <= heapSize && A[r] > A[largest]) largest = r;
    if (largest != i) {
        swap(A[i], A[largest]);
        ++swaps;
        if (verbose) { cout << "      swap A[" << i << "] with A[" << largest << "] -> A =
[";
                       for (int k = 1; k <= heapSize; ++k) cout << A[k] << (k < heapSize ?
" " : "");
                       cout << "]\n"; }
        maxHeapify(largest, verbose);
    }
}

void buildMaxHeap(bool verbose) {
    for (int i = heapSize / 2; i >= 1; --i) {
        if (verbose) cout << "  i = " << i << ", A[" << i << "] = " << A[i] << "\n";
        maxHeapify(i, verbose);
        if (verbose) { cout << "      after this call: A = [";
                       for (int k = 1; k <= heapSize; ++k) cout << A[k] << (k < heapSize ?
" " : "");
                       cout << "]\n"; }
    }
}
```

Kode 5.3 (Lanjutan) *BUILD-MAX-HEAP, ditelusuri pada larik dari Subbab 5.5.3 lalu diukur pada larik acak sampai sepuluh juta elemen.*

```
int main() {
    A = {0, 4, 1, 3, 2, 16, 9, 10, 14, 8, 7};    // larik masukan dari Subbab 5.5.3
    heapSize = 10; swaps = 0;
    show("input:  A = [");
    cout << "floor(n/2) = " << heapSize / 2 << ", so the loop runs i = 5, 4, 3, 2, 1\n\n";
    buildMaxHeap(true);
    show("\nresult: A = [");
    cout << "exchanges performed: " << swaps << "\n";

    // Kini pemeriksaan empiris bahwa biaya totalnya tumbuh seperti n, bukan seperti n lg
n.
    cout << "\n         n     exchanges   exchanges/n   (n lg n)/n = lg n\n";
    mt19937 rng(12345);
    for (int n : {1000, 10000, 100000, 1000000, 10000000}) {
        A.assign(n + 1, 0);
        for (int i = 1; i <= n; ++i) A[i] = (int)(rng() % 1000000000u);
        heapSize = n; swaps = 0;
        buildMaxHeap(false);
        double lg = 0; { int m = n; while (m > 1) { m >>= 1; lg += 1; } }
        cout << setw(10) << n << setw(14) << swaps
             << setw(14) << fixed << setprecision(3) << (double)swaps / n
             << setw(18) << lg << "\n";
    }
    return 0;
}
```

Keluarannya:

```
input:  A = [4 1 3 2 16 9 10 14 8 7]
floor(n/2) = 5, so the loop runs i = 5, 4, 3, 2, 1

  i = 5, A[5] = 16
      after this call: A = [4 1 3 2 16 9 10 14 8 7]
  i = 4, A[4] = 2
      swap A[4] with A[8] -> A = [4 1 3 14 16 9 10 2 8 7]
      after this call: A = [4 1 3 14 16 9 10 2 8 7]
  i = 3, A[3] = 3
      swap A[3] with A[7] -> A = [4 1 10 14 16 9 3 2 8 7]
      after this call: A = [4 1 10 14 16 9 3 2 8 7]
  i = 2, A[2] = 1
      swap A[2] with A[5] -> A = [4 16 10 14 1 9 3 2 8 7]
      swap A[5] with A[10] -> A = [4 16 10 14 7 9 3 2 8 1]
      after this call: A = [4 16 10 14 7 9 3 2 8 1]
  i = 1, A[1] = 4
      swap A[1] with A[2] -> A = [16 4 10 14 7 9 3 2 8 1]
      swap A[2] with A[4] -> A = [16 14 10 4 7 9 3 2 8 1]
      swap A[4] with A[9] -> A = [16 14 10 8 7 9 3 2 4 1]
      after this call: A = [16 14 10 8 7 9 3 2 4 1]

result: A = [16 14 10 8 7 9 3 2 4 1]
exchanges performed: 7

         n     exchanges   exchanges/n   (n lg n)/n = lg n
      1000           742         0.742             9.000
     10000          7469         0.747            13.000
    100000         74242         0.742            16.000
   1000000        743123         0.743            19.000
  10000000       7440513         0.744            23.000
```

Setiap baris penelusuran tangan di Subbab 5.5.3 direproduksi persis, sampai ke lariknya yang antara sesudah tiap penukaran.

Sekarang lihatlah tabel di bagian bawah, sebab itulah bukti paling meyakinkan dalam bab ini. Ketika n tumbuh sepuluh ribu kali lipat — dari seribu ke sepuluh juta — besaran $\lg n$ merangkak dari 9 ke 23, kira-kira dua setengah kali lipat. Seandainya `BUILD-MAX-HEAP` adalah $\Theta(n \lg n)$, angka penukaran-per-elemennya harus merangkak naik dengan faktor yang serupa. **Ia tidak bergerak.** Ia duduk di 0.74, lalu 0.75, lalu 0.74, lalu 0.74, lalu 0.74. Penukaran per elemen adalah sebuah konstanta. Konstanta itu dikali n adalah $\Theta(n)$, persis sebagaimana diramalkan jumlahannya.

> **PEMERIKSAAN AKAL SEHAT YANG DAPAT ANDA KERJAKAN SENDIRI.** Mengapa kira-kira 0.74 penukaran per elemen, dan bukan 1 atau 2? Sebab lebih dari separuh simpulnya adalah daun dan bahkan tidak pernah dikunjungi, dan di antara simpul dalam yang memang dikunjungi, sebagian besarnya hanya tenggelam satu atau dua tingkat. Soal 5.20 menaksir konstanta itu dari tinggi simpul-simpulnya dan mendarat di dekat $\frac{3}{4}$, persis seperti yang ditunjukkan pengukurannya.

5.6 Heapsort

5.6.1 Algoritmanya

Heapsort diterbitkan oleh J. W. J. Williams pada 1964 [1], dan Robert Floyd memberikan pembangunan waktu-linear pada Subbab 5.5 di tahun yang sama [2]. Kini kita punya semuanya. Ingat kembali rencana dari Subbab 5.2: jalankan Selection Sort, tetapi buatlah "cari maksimum" menjadi murah dengan menyimpan datanya di dalam sebuah *heap*. **Pseudokode 5.3** adalah hasilnya.

Pseudokode 5.3 `HEAPSORT`: bangun sebuah *max-heap*, lalu berulang kali pindahkan akarnya ke ujung *heap* yang menyusut dan perbaiki akarnya.

```
HEAPSORT(A):
 1.  BUILD-MAX-HEAP(A)                    // O(n)
 2.  for i ← n downto 2                   // n − 1 kali
 3.      swap A[1] ↔ A[i]                 // O(1)
 4.      heap-size ← heap-size − 1        // O(1)
 5.      MAX-HEAPIFY(A, 1)                // O(lg n)
```

Cara kerjanya, dengan kata-kata. Sesudah baris 1, $A[1]$ memuat elemen terbesar dari seluruh lariknya. Dalam larik yang terurut, elemen terbesar seharusnya berada di posisi paling belakang, $A[n]$ — jadi baris 3 menaruhnya di sana, dengan menukarnya dengan apa pun yang tadinya duduk di $A[n]$. Elemen itu kini berada di tempatnya yang final, benar, dan permanen; ia tidak akan pernah disentuh lagi. Baris 4 mencatat hal itu dengan menyusutkan *heap*-nya sehingga posisi n tidak lagi menjadi bagiannya. Baris 5 memperbaiki satu-satunya pelanggaran yang ditimbulkan penukaran itu — akarnya kini memuat sebuah nilai kecil dari dasar *heap* — dan sesudah perbaikannya $A[1]$ memuat elemen terbesar kedua. Ulangi dengan $i = n - 1$, lalu $n - 2$, dan seterusnya.

Lariknya karena itu terbelah menjadi dua wilayah yang saling berpapasan:

- $A[1 \ldots \text{heap-size}]$ — masih sebuah *max-heap*, memuat elemen-elemen yang belum ditempatkan;
- $A[\text{heap-size} + 1 \ldots n]$ — sudah terurut, memuat elemen-elemen terbesar pada posisi finalnya.

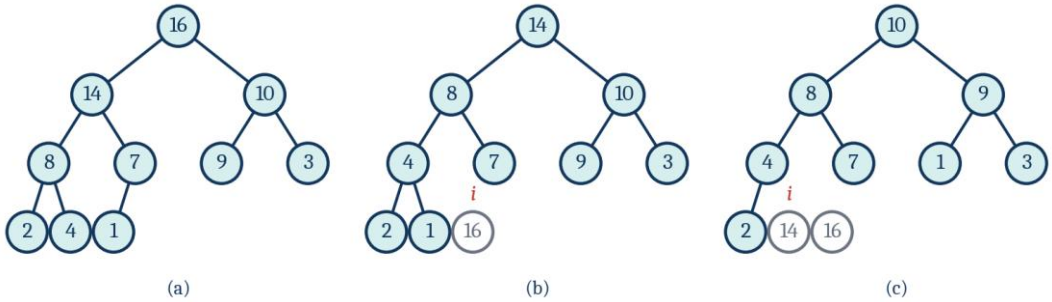
Setiap iterasi menggeser batasnya satu langkah ke kiri.

Perhatikan kehematan penukaran pada baris 3. Selection Sort harus *mencari* maksimumnya sebelum dapat menempatkannya; Heapsort sudah tahu di mana maksimumnya berada. Dan

elemen yang digesernya tidak memerlukan rumah — ia pergi ke akar, yang justru merupakan tempat `MAX-HEAPIFY` mengharapkan sebuah nilai yang salah tempat.

5.6.2 Penelusuran lengkap

Kita lanjutkan dengan larik dari Subbab 5.5, supaya baris 1 menghasilkan *heap* yang sudah kita bangun dengan tangan: $A = [16,14,10,8,7,9,3,2,4,1]$, yang digambar di **Gambar 5.13** sebagai panel (a).



Gambar 5.13 Heapsort, langkah (a)–(c). Simpul yang diarsir masih berada di dalam *heap*; yang tidak diarsir sudah ditempatkan pada posisi terurutnya yang final dan tidak lagi menjadi bagiannya. **(a)** *heap* sesudah `BUILD-MAX-HEAP`. **(b)** sesudah menukarkan 16 ke posisi 10 dan memperbaiki akarnya. **(c)** sesudah menukarkan 14 ke posisi 9 dan memperbaikinya.

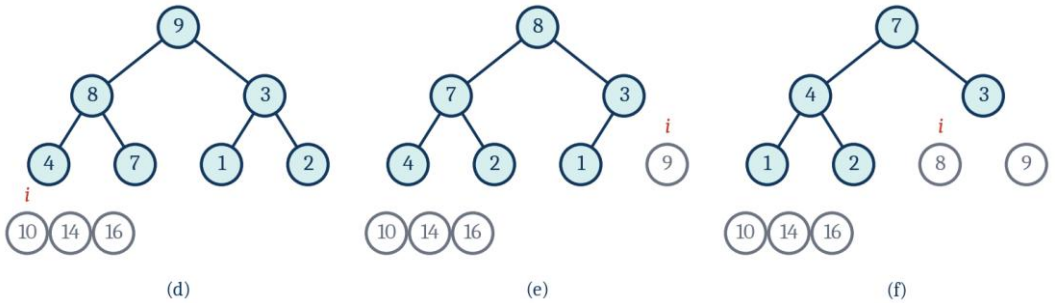
Iterasi $i = 10$.

- Tukarkan $A[1] = 16$ dengan $A[10] = 1$. Lariknya menjadi $[1,14,10,8,7,9,3,2,4 \mid 16]$, dengan palang menandai ekor yang sudah terurut.
- Susutkan: heap-size = 9. Angka 16 kini berada di luar *heap*-nya, untuk selamanya.
- `MAX-HEAPIFY(A, 1)`: akar 1, anak-anaknya 14 dan 10; terbesarnya 14 di indeks 2; tukar. Sekarang di indeks 2: nilai 1, anak-anaknya $A[4] = 8$ dan $A[5] = 7$; terbesarnya 8; tukar. Sekarang di indeks 4: nilai 1, anak-anaknya $A[8] = 2$ dan $A[9] = 4$; terbesarnya 4 di indeks 9; tukar. Sekarang di indeks 9: $2 \times 9 = 18 > 9$, sebuah daun. Berhenti.
- Hasilnya: $[14,8,10,4,7,9,3,2,1 \mid 16]$ — panel (b).

Iterasi $i = 9$.

- Tukarkan $A[1] = 14$ dengan $A[9] = 1$: $[1,8,10,4,7,9,3,2 \mid 14,16]$.
- Susutkan menjadi heap-size = 8.
- `MAX-HEAPIFY(A, 1)`: akar 1 melawan 8 dan 10 → tukar dengan 10 di indeks 3. Di indeks 3: nilai 1, anak-anaknya $A[6] = 9$ dan $A[7] = 3$ → tukar dengan 9 di indeks 6. Di indeks 6: $12 > 8$, sebuah daun. Berhenti.
- Hasilnya: $[10,8,9,4,7,1,3,2 \mid 14,16]$ — panel (c).

Tiga iterasi berikutnya ditampilkan di **Gambar 5.14**.



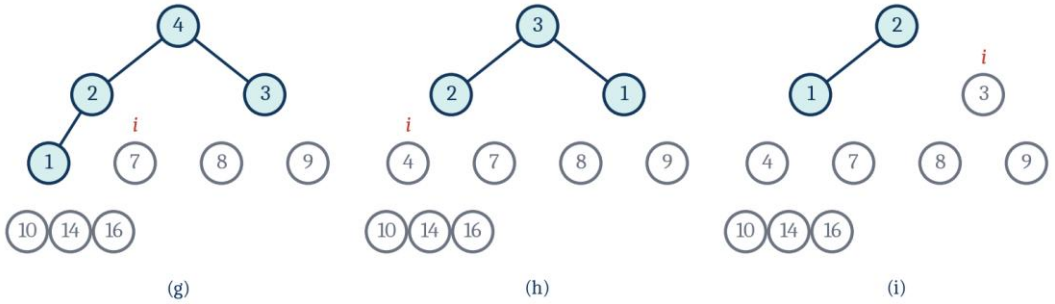
Gambar 5.14 Heapsort, langkah (d)–(f). *Heap*-nya menyusut satu simpul setiap kali dan ekor terurutnya, yang digambar tanpa arsiran dan dijajarkan di sepanjang bagian bawah, tumbuh menjemputnya. **(d)** 10 ditempatkan. **(e)** 9 ditempatkan. **(f)** 8 ditempatkan.

Iterasi $i = 8$**.** Tukarkan 10 dengan $A[8] = 2$; susutkan menjadi 7; *heapify*. Hasilnya [9,8,3,4,7,1,2 | 10,14,16] — panel (d).

Iterasi $i = 7$**.** Tukarkan 9 dengan $A[7] = 2$; susutkan menjadi 6; *heapify*. Hasilnya [8,7,3,4,2,1 | 9,10,14,16] — panel (e).

Iterasi $i = 6$**.** Tukarkan 8 dengan $A[6] = 1$; susutkan menjadi 5; *heapify*. Hasilnya [7,4,3,1,2 | 8,9,10,14,16] — panel (f).

Tiga lagi, di **Gambar 5.15**.



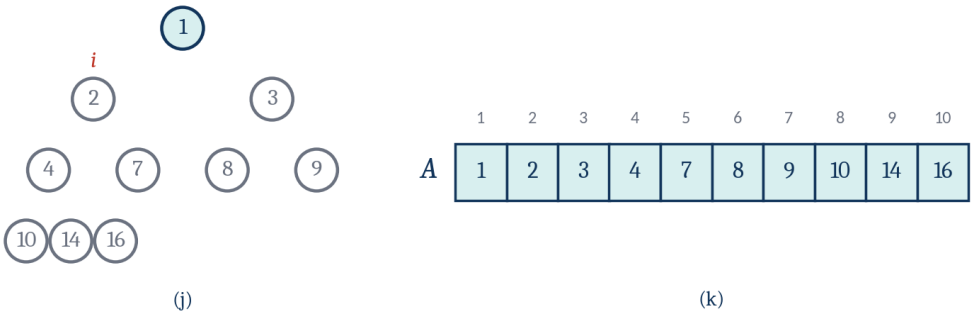
Gambar 5.15 Heapsort, langkah (g)–(i). Dengan *heap* yang tinggal empat, tiga, lalu dua simpul, pekerjaan per iterasinya sudah menjadi sepele: tingginya 2, lalu 1, lalu 1.

Iterasi $i = 5$**.** Tukarkan 7 dengan $A[5] = 2$; susutkan menjadi 4; *heapify*. Hasilnya [4,2,3,1 | 7,8,9,10,14,16] — panel (g).

Iterasi $i = 4$**.** Tukarkan 4 dengan $A[4] = 1$; susutkan menjadi 3; *heapify*. Hasilnya [3,2,1 | 4,7,8,9,10,14,16] — panel (h).

Iterasi $i = 3$**.** Tukarkan 3 dengan $A[3] = 1$; susutkan menjadi 2; *heapify*. Hasilnya [2,1 | 3,4,7,8,9,10,14,16] — panel (i).

Dan yang terakhir, di **Gambar 5.16**.



Gambar 5.16 Heapsort, langkah (j) dan (k). (j) sesudah penukaran terakhir, *heap*-nya memuat satu elemen. **(k)** lariknya yang final $A = [1,2,3,4,7,8,9,10,14,16]$, dalam urutan tidak menurun.

Iterasi $i = 2$. Tukarkan 2 dengan $A[2] = 1$; susutkan menjadi 1; pemanggilan `MAX-HEAPIFY(A, 1)` menemukan sebuah *heap* satu elemen dan langsung kembali. Hasilnya $[1 \mid 2,3,4,7,8,9,10,14,16]$ — panel (j).

Gelungnya berhenti di $i = 2$; tidak ada iterasi untuk $i = 1$, dan memang tidak perlu ada, sebab *heap* satu elemen adalah larik terurut satu elemen. Lariknya yang final adalah

$$A = [1,\ 2,\ 3,\ 4,\ 7,\ 8,\ 9,\ 10,\ 14,\ 16]$$

— panel (k). ✓

5.6.3 Kebenarannya, dengan invarian gelung

> **INVARIAN GELUNG.** Pada awal setiap iterasi gelung `for` pada baris 2, subarray $A[1 \ldots i]$ adalah sebuah *max-heap* yang memuat i elemen terkecil dari larik semula, dan subarray $A[i+1 \ldots n]$ memuat $n - i$ elemen terbesar dari larik semula, dalam urutan terurut.

Inisialisasi. Sebelum iterasi pertama, $i = n$. Baris 1 baru saja menjadikan $A[1 \ldots n]$ sebuah *max-heap*, dan secara trivial larik itu memuat n elemen terkecil dari lariknya — yaitu semuanya. Subarray $A[n+1 \ldots n]$ kosong, dan larik kosong sudah terurut. ✓

Pemeliharaan. Andaikan invariannya berlaku pada awal sebuah iterasi dengan pencacah i. Karena $A[1 \ldots i]$ adalah *max-heap*, $A[1]$ adalah yang terbesar di antara i elemen itu — dan karena mereka adalah i elemen terkecil dari larik semula, $A[1]$ adalah elemen terbesar ke-$(n - i + 1)$ secara keseluruhan, yang posisi terurutnya yang benar tepat indeks i. Baris 3 menaruhnya di sana. Baris 4 mengeluarkan posisi i dari *heap*-nya. Ekor $A[i \ldots n]$ kini memuat $n - i + 1$ elemen terbesar dalam urutan terurut, sebab elemen yang baru saja ditempatkan tidak lebih besar daripada setiap elemen yang sudah berada di $A[i+1 \ldots n]$ dan tidak lebih kecil daripada segala yang tersisa di dalam *heap*-nya. Sementara itu $A[1 \ldots i-1]$ memuat $i - 1$ elemen terkecil, dan ia memenuhi sifat *heap* di mana-mana kecuali mungkin di akarnya, tempat penukaran tadi menitipkan bekas daun; anak-anak akarnya tak berubah dan masih merupakan akar *heap*, sehingga `MAX-HEAPIFY(A, 1)` pada baris 5 memulihkan sifat *heap*-nya. Invariannya berlaku dengan i diganti oleh $i - 1$. ✓

Terminasi. Gelungnya berakhir dengan $i = 1$. Invariannya mengatakan $A[2 \ldots n]$ memuat $n - 1$ elemen terbesar dalam urutan terurut, dan $A[1 \ldots 1]$ memuat satu elemen terkecilnya. Jadi $A[1 \ldots n]$ terurut dalam urutan tidak menurun. ✓ ■

5.6.4 Waktu jalannya

Tabel 5.3 memperhitungkan setiap barisnya.

Tabel 5.3 Biaya Heapsort, baris demi baris. Suku yang mendominasi adalah $n-1$ pemanggilan `MAX-HEAPIFY`.

Langkah	Biaya tiap eksekusi	Banyaknya eksekusi	Total
`BUILD-MAX-HEAP` (baris 1)	$O(n)$	1	$O(n)$
tukar + susut (baris 3–4)	$\Theta(1)$	$n-1$	$\Theta(n)$
`MAX-HEAPIFY` (baris 5)	$O(\lg n)$	$n-1$	$O(n\lg n)$
Total			$\mathbf{O(n\lg n)}$

Menjumlahkannya: $O(n)+\Theta(n)+O(n\lg n)=O(n\lg n)$, sebab $n\lg n$ mendominasi n.

Apakah batasnya ketat — apakah Heapsort $\Theta(n\lg n)$ dan bukan sekadar $O(n\lg n)$? Untuk **kasus terburuk**, pembuktian yang paling bersih baru kita capai di akhir bab: Subbab 5.11 menunjukkan bahwa *setiap* pengurutan berbasis pembandingan memerlukan $\Omega(n\lg n)$ pembandingan pada suatu masukan, dan Heapsort adalah pengurutan berbasis pembandingan. Jadi kasus terburuknya tepat $\Theta(n\lg n)$.

Kasus terbaik memerlukan kecermatan lebih. Tinjau $n/2$ iterasi pertama gelungnya. Selama masing-masing iterasi itu, *heap*-nya masih punya sekurang-kurangnya $n/2$ elemen, sehingga tingginya sekurang-kurangnya $\lg(n/2)=\lg n-1$. Elemen yang ditukarkan ke akar berasal dari dasar *heap*, jadi ia termasuk elemen yang lebih kecil di antara yang ada, dan biasanya ia tenggelam hampir sepenuhnya kembali ke bawah. Kalau masing-masing dari $n/2$ iterasi itu berongkos $\Omega(\lg n)$, totalnya $\Omega\left(\frac{n}{2}(\lg n-1)\right)=\Omega(n\lg n)$. Mengubah kata "biasanya" menjadi sebuah pembuktian itu sukar, tetapi dapat dilakukan: ketika semua kuncinya **berbeda**, Heapsort melakukan $\Omega(n\lg n)$ pembandingan pada *setiap* masukan (sebuah soal latihan bertanda bintang dalam CLRS [3]). Jadi, untuk kunci-kunci yang berbeda,

$$T_{\text{Heapsort}}(n) = \Theta(n\lg n) \quad \text{pada kasus terburuk, kasus terbaik, dan setiap kasus di antara}$$

Ada satu pengecualian yang jujur. Kalau kesemua n kuncinya **sama**, setiap pemanggilan `MAX-HEAPIFY` mendapati bahwa akarnya sudah mendominasi anak-anaknya (ujinya adalah $>$ yang tegas) dan kembali sesudah pekerjaan $\Theta(1)$, sehingga seluruh pengurutannya hanya memakan waktu $\Theta(n)$.

Keseragaman itulah keutamaan Heapsort yang membedakannya. Di luar masukan serba-sama yang merosot itu, tidak ada masukan yang membuatnya lambat, dan tidak ada masukan yang membuatnya cepat.

> **HEAPSORT, DIRINGKAS.** Kasus terburuk $\Theta(n\lg n)$. Kasus terbaik $\Theta(n\lg n)$ bila kuncinya berbeda-beda. Kasus rata-rata $\Theta(n\lg n)$. **Di tempat**: ia memakai $O(1)$ ruang tambahan — segenggam peubah indeks dan satu tempat singgah untuk penukarannya. **Tidak stabil.** Ia mencapai impian yang dinyatakan di Subbab 5.1.2.

5.6.5 Mengapa Heapsort tidak stabil

Sebuah pengurutan disebut **stabil** jika elemen-elemen dengan kunci yang sama mempertahankan urutan relatifnya yang semula. Heapsort tidak stabil, dan mudah sekali melihat sebabnya: hal pertama yang dikerjakannya adalah melemparkan $A[1]$ ke posisi n melintasi seluruh lariknya, sama sekali tanpa peduli pada apa yang terletak di antaranya. Dua kunci yang sama dapat tertukar urutannya oleh satu penukaran semacam itu saja.

CONTOH PENYANGKAL DUA ELEMEN. Urutkan pasangan $\langle 1, a\rangle, \langle 1, b\rangle$ menurut komponen pertamanya. BUILD-MAX-HEAP membiarkan lariknya, sebab $1 \geq 1$. Lalu $i = 2$: tukarkan $A[1] \leftrightarrow A[2]$, menghasilkan $\langle 1, b\rangle, \langle 1, a\rangle$. Si b sudah mendahului si a. Kestabilan hilang pada larik sepanjang dua.

Kita kembali pada kestabilan, dan pada mengapa ia penting dalam perangkat lunak sungguhan, di Subbab 5.12.2.

5.6.6 Heapsort dalam C++

Kode 5.4 adalah Heapsort lengkap yang mencetak lariknya sesudah setiap pengeluaran, dengan sebuah palang yang menunjukkan batas antara *heap* yang menyusut dan ekor terurut yang tumbuh.

Kode 5.4 Heapsort lengkap yang dapat dijalankan, yang menarasikan setiap pengeluarannya.

```
// HEAPSORT, mencetak lariknya sesudah tiap pengeluaran, dengan ekor terurutnya ditandai |.
#include <iostream>
#include <vector>
#include <string>
using namespace std;

vector<int> A;
int heapSize;                                   // seberapa banyak bagian A yang masih
berupa heap
int n;                                          // seberapa banyak bagian A yang ada sama
sekali

void printState(const string& label) {
    cout << label << "[";
    for (int i = 1; i <= n; ++i) {
        if (i == heapSize + 1 && heapSize < n) cout << "| ";
        cout << A[i] << (i < n ? " " : "");
    }
    cout << "]\n";
}

void maxHeapify(int i) {
    int l = 2 * i, r = 2 * i + 1, largest = i;
    if (l <= heapSize && A[l] > A[largest]) largest = l;
    if (r <= heapSize && A[r] > A[largest]) largest = r;
    if (largest != i) { swap(A[i], A[largest]); maxHeapify(largest); }
}

void buildMaxHeap() { for (int i = heapSize / 2; i >= 1; --i) maxHeapify(i); }

void heapsort() {
    heapSize = n;
    buildMaxHeap();
    printState("after BUILD-MAX-HEAP:  ");
    for (int i = n; i >= 2; --i) {
        cout << "  swap A[1]=" << A[1] << " with A[" << i << "]=" << A[i];
        swap(A[1], A[i]);
        --heapSize;                             // elemen yang baru diparkir kini
dibekukan
        maxHeapify(1);
        cout << ", shrink to " << heapSize << ", heapify -> ";
        printState("");
    }
}
```

Kode 5.4 (Lanjutan) *Heapsort lengkap yang dapat dijalankan, yang menarasikan setiap pengeluarannya.*

```
int main() {
    A = {0, 4, 1, 3, 2, 16, 9, 10, 14, 8, 7};
    n = 10; heapSize = n;
    printState("input:                 ");
    heapsort();
    heapSize = n;                                // seluruh lariknya sudah terurut; buang
pemisahnya
    printState("\nsorted:                ");

    bool ok = true;                              // pemeriksaan hasilnya secara mandiri
    for (int i = 2; i <= n; ++i) if (A[i - 1] > A[i]) ok = false;
    cout << "non-decreasing: " << (ok ? "yes" : "no") << "\n";
    return 0;
}
```

Keluarannya:

```
input:                [4 1 3 2 16 9 10 14 8 7]
after BUILD-MAX-HEAP:  [16 14 10 8 7 9 3 2 4 1]
  swap A[1]=16 with A[10]=1, shrink to 9, heapify -> [14 8 10 4 7 9 3 2 1 | 16]
  swap A[1]=14 with A[9]=1, shrink to 8, heapify -> [10 8 9 4 7 1 3 2 | 14 16]
  swap A[1]=10 with A[8]=2, shrink to 7, heapify -> [9 8 3 4 7 1 2 | 10 14 16]
  swap A[1]=9 with A[7]=2, shrink to 6, heapify -> [8 7 3 4 2 1 | 9 10 14 16]
  swap A[1]=8 with A[6]=1, shrink to 5, heapify -> [7 4 3 1 2 | 8 9 10 14 16]
  swap A[1]=7 with A[5]=2, shrink to 4, heapify -> [4 2 3 1 | 7 8 9 10 14 16]
  swap A[1]=4 with A[4]=1, shrink to 3, heapify -> [3 2 1 | 4 7 8 9 10 14 16]
  swap A[1]=3 with A[3]=1, shrink to 2, heapify -> [2 1 | 3 4 7 8 9 10 14 16]
  swap A[1]=2 with A[2]=1, shrink to 1, heapify -> [1 | 2 3 4 7 8 9 10 14 16]

sorted:               [1 2 3 4 7 8 9 10 14 16]
non-decreasing: yes
```

Bandingkan baris demi baris dengan panel (b) sampai (j) pada **Gambar 5.13**, **Gambar 5.14**, **Gambar 5.15**, dan **Gambar 5.16**. Setiap larik antaranya cocok dengan gambar-gambarnya secara persis. Seperti itulah rupa implementasi yang benar: bukan sekadar jawaban terurut di ujungnya, melainkan keadaan-keadaan antara yang sama dengan yang diramalkan teorinya.

5.7 Antrean Prioritas

5.7.1 Apa itu antrean prioritas

Heapsort adalah *latihan* paling termasyhur bagi *heap*. Antrean prioritas adalah *pemakaian* paling termasyhur bagi *heap*, dan dalam praktik antrean prioritaslah yang lebih penting.

> **DEFINISI.** Sebuah **antrean prioritas** (*priority queue*) adalah tipe data abstrak yang memelihara sebuah himpunan S yang berubah-ubah, yang tiap elemennya membawa sebuah **kunci** yang menyatakan prioritasnya. Sebuah **antrean prioritas-maks** mendukung empat operasi: $\text{Insert}(S, x)$ — tambahkan elemen x ke himpunannya; $\text{Maximum}(S)$ — kembalikan elemen dengan kunci terbesar, tanpa membuangnya; $\text{Extract}-\text{Max}(S)$ — buang dan kembalikan elemen dengan kunci terbesar; $\text{Increase}-\text{Key}(S, x, k)$ — naikkan kunci elemen x ke nilai baru k, yang harus sekurang-kurangnya sebesar kunci yang sekarang. Sebuah **antrean prioritas-min** adalah bayangan cerminnya, dengan Minimum, Extract − Min, dan Decrease − Key.

Frasa "tipe data abstrak" berarti kita menetapkan *apa* yang dikerjakan operasi-operasinya, bukan *bagaimana*. Pemisahan itu penting: empat operasi yang sama dapat diimplementasikan dengan beberapa cara, dengan biaya yang sangat berbeda, dan *heap* semata-mata jawaban umum yang terbaik.

Bandingkan dengan antrean biasa. Antrean FIFO melayani siapa pun yang datang lebih dulu; antrean prioritas melayani siapa pun yang paling penting, tak peduli urutan kedatangannya. Kalau tiga pelanggan masuk ke sebuah binatu — satu dengan cucian biasa yang jatuh tempo besok, satu dengan pesanan kilat yang jatuh tempo dua jam lagi, satu dengan cucian yang jatuh tempo sore ini — antrean FIFO melayani mereka menurut urutan mereka melewati pintu. Antrean prioritas melayani pesanan kilatnya lebih dulu. Dalam sebuah sistem komputer, "pelanggannya" adalah pekerjaan, paket, kejadian, atau halaman web yang menunggu dipindai.

5.7.2 Di mana antrean prioritas dipakai

- **Penjadwalan pekerjaan sistem operasi.** Proses membawa prioritas; penjadwalnya menjalankan proses siap-jalan yang prioritasnya tertinggi. Ketika sebuah proses selesai atau terhalang, penjadwalnya melakukan `EXTRACT-MAX`. Ketika sebuah proses baru diciptakan, ia melakukan `INSERT`. Ketika sebuah proses sudah menunggu terlalu lama dan harus dinaikkan supaya tidak kelaparan, itu adalah `INCREASE-KEY`.
- **Simulasi kejadian diskret.** Antreannya memuat kejadian-kejadian tertunda yang berkunci waktu jatuh temponya, dan simulatornya berulang kali mengeluarkan yang *paling awal* — jadi yang ini adalah antrean prioritas-min. Menyimulasikan sebuah pabrik, sebuah jaringan, atau sebuah antrean pelanggan semuanya bekerja seperti ini.
- **Algoritma graf.** Algoritma lintasan terpendek Dijkstra (Bab 13) menyimpan simpul-simpul yang belum dikunjungi di dalam antrean prioritas-min berkunci jarak tentatifnya yang sekarang; setiap pelonggaran sebuah sisi adalah sebuah `DECREASE-KEY`. Algoritma pohon merentang minimum Prim (Bab 11) melakukan hal yang sama dengan bobot sisi. Pencarian A* menambahkan sebuah heuristik pada kuncinya dan selebihnya identik.
- **Pengodean Huffman.** Membangun kode awalan yang optimal (Bab 9) berarti berulang kali mengeluarkan dua simbol yang frekuensinya terkecil lalu menyisipkan simpul gabungannya — dua `EXTRACT-MIN` dan satu `INSERT` per langkah.
- **Penjadwalan perayapan dan pemindaian.** Halaman mana yang kita ambil berikutnya? Yang skor kegentingannya tertinggi. Kita kembali ke sini di Subbab 5.13.

5.7.3 `MAXIMUM`: yang mudah

Pseudokode 5.4 hanya memerlukan satu baris pekerjaan yang sungguhan.

Pseudokode 5.4 `HEAP-MAXIMUM`: kunci terbesar sebuah *max-heap* selalu berada di akarnya.

```
HEAP-MAXIMUM(A):
 1.  if heap-size < 1 then error "heap underflow"
 2.  return A[1]
```

Satu pembacaan larik. $O(1)$. Sama sekali tanpa pembandingan. Inilah seluruh alasan kita memilih *heap*: jawaban atas "mana yang terbesar?" sudah diparkir permanen di slot 1 oleh sifat *heap*, sebagaimana dibuktikan di Subbab 5.3.5.

5.7.4 `EXTRACT-MAX`

Membuang maksimumnya adalah operasi yang menarik, dan **Pseudokode 5.5** memberikan pseudokodenya.

Pseudokode 5.5 `HEAP-EXTRACT-MAX` dalam pseudokode. Simpan akarnya; pindahkan elemen terakhir ke akar; susutkan *heap*-nya satu; panggil `HEAPIFY` pada akarnya untuk memulihkan keteraturan; kembalikan nilai yang tadi disimpan.

```
HEAP-EXTRACT-MAX(A):                    // keluarkan dan kembalikan elemen terbesar A
 1.  max ← A[1]
 2.  A[1] ← A[heap-size]
 3.  heap-size ← heap-size - 1
 4.  HEAPIFY(A, 1)                       // bentuk ulang heap-nya
 5.  return max
```

Triknya sama dengan yang dipakai Heapsort pada baris 3. Kita tidak dapat begitu saja menghapus $A[1]$ dan meninggalkan lubang — lariknya tidak akan lagi berupa pohon hampir lengkap. Maka kita ambil elemen dari posisi *terakhir*, yang merupakan satu-satunya posisi yang pembuangannya menjaga bentuknya tetap sah, lalu kita jatuhkan ke akarnya. Bentuknya lalu benar, sifat *heap*-nya rusak tepat di satu tempat, dan `MAX-HEAPIFY` adalah alat untuk persis keadaan itu.

> **CONTOH YANG DIKERJAKAN.** Keluarkan dari $A = [16,14,10,8,7,9,3,2,4,1]$, heap-size $= 10$. Langkah 1: simpan max $= A[1] = 16$. Langkah 2: $A[1] \leftarrow A[10] = 1$. Lariknya kini $[1,14,10,8,7,9,3,2,4,1]$. Langkah 3: heap-size $\leftarrow 9$, sehingga salinan 1 di belakang berada di luar *heap*-nya dan tak relevan. Langkah 4: `MAX-HEAPIFY(A, 1)`. Akar 1 melawan anak-anaknya 14 dan 10: tukar dengan 14. Di indeks 2: nilai 1 melawan 8 dan 7: tukar dengan 8. Di indeks 4: nilai 1 melawan $A[8] = 2$ dan $A[9] = 4$: tukar dengan 4. Di indeks 9: daun, berhenti. Langkah 5: kembalikan 16. *Heap*-nya adalah $[14,8,10,4,7,9,3,2,1]$. Ongkosnya: satu perjalanan akar-ke-daun, jadi $O(\lg n)$.

5.7.5 `INCREASE-KEY`

Menaikkan prioritas sebuah elemen adalah bayangan cermin `MAX-HEAPIFY`: alih-alih sebuah nilai yang terlalu kecil lalu tenggelam, kini sebuah nilai terlalu besar dan harus **mengapung naik**. **Pseudokode 5.6** membiarkannya naik.

Pseudokode 5.6 `HEAP-INCREASE-KEY`: naikkan kunci di posisi i, lalu apungkan ia melewati setiap induk yang lebih kecil.

```
HEAP-INCREASE-KEY(A, i, key):
 1.  if key < A[i] then error "new key is smaller than current key"
 2.  A[i] ← key
 3.  while i > 1 and A[PARENT(i)] < A[i]
 4.      swap A[i] ↔ A[PARENT(i)]
 5.      i ← PARENT(i)
```

Gelung `while` itu membandingkan elemennya hanya dengan induknya, dan berhenti begitu induknya menang. Tidak perlu melihat saudara-saudaranya: kalau nilai barunya lebih kecil daripada induknya, sifat *heap*-nya berlaku secara lokal, dan segala yang di atas induknya sudah benar sejak tadi.

> **CONTOH YANG DIKERJAKAN.** Ambil *heap* $[10,8,9,4,7,3,2,1]$ dengan heap-size $= 8$, dan naikkan kunci di posisi 8 dari 1 menjadi 20. Langkah 1: $20 \geq 1$, jadi pemanggilannya sah. Setel $A[8] \leftarrow 20$. Langkah 2: $i = 8$. Parent$(8) = 4$. Apakah $A[4] = 4 < 20$? Ya — tukar. Kini $i = 4$. Langkah 3: Parent$(4) = 2$. Apakah $A[2] = 8 < 20$? Ya — tukar. Kini $i = 2$. Langkah 4: Parent$(2) = 1$. Apakah $A[1] = 10 < 20$? Ya — tukar. Kini $i = 1$. Langkah 5: $i = 1$, sehingga syarat gelung $i > 1$ gagal. Berhenti. Hasilnya: $[20,10,9,8,7,3,2,4]$. Angka 20 naik tiga tingkat; segala yang dilewatinya turun satu. Ongkosnya: paling banyak satu perjalanan daun-ke-akar, jadi $O(\lg n)$.

5.7.6 `INSERT`

Dan sekarang bagian yang cantik. Penyisipan sama sekali tidak memerlukan kode baru: ia adalah `INCREASE-KEY` yang sedang menyamar. **Pseudokode 5.7** menampilkannya dalam pseudokode.

Pseudokode 5.7 `HEAP-INSERT` dalam pseudokode. Tumbuhkan *heap*-nya satu slot di ujung, lalu apungkan kunci barunya ke atas, dengan memindahkan setiap induk yang kalah turun ke tempat anaknya, sampai sebuah induk menang atau akarnya tercapai.

```
HEAP-INSERT(A, key):
 1.  heap-size ← heap-size + 1
 2.  i ← heap-size
 3.  while i > 1 and A[PARENT(i)] < key
 4.      A[i] ← A[PARENT(i)]              // geser induk yang kalah ke bawah
 5.      i ← PARENT(i)
 6.  A[i] ← key                           // tulis kuncinya sekali saja, di akhir
```

Gagasannya: tumbuhkan *heap*-nya satu posisi di ujung — satu-satunya posisi yang pertumbuhannya menjaga bentuknya tetap sah — taruh nilai $-\infty$ di sana, yang mustahil melanggar sifat *heap* sebab tidak ada yang lebih kecil daripada $-\infty$, lalu panggil `INCREASE-KEY` untuk menaikkannya ke kunci yang sesungguhnya. Gelung apung-naiknya lalu membawanya ke tempat yang semestinya. **Pseudokode 5.8** menuliskannya.

Pseudokode 5.8 `MAX-HEAP-INSERT`: tambahkan satu slot berisi $-\infty$ di ujung, lalu naikkan ke kunci barunya.

```
MAX-HEAP-INSERT(A, key):
 1.  heap-size ← heap-size + 1
 2.  A[heap-size] ← -∞
 3.  HEAP-INCREASE-KEY(A, heap-size, key)
```

Perhatikan optimasi kecil dalam **Pseudokode 5.7**: alih-alih menukarkan dua nilai pada setiap langkah, ia *menggeser induknya turun* ke slot anaknya dan hanya menuliskan kuncinya sekali, di ujung. Hasilnya sama, dengan separuh operasi tulis ke memori.

5.7.7 Berapa biaya keempat operasinya

Tabel 5.4 Biaya keempat operasi antrean prioritas di bawah tiga implementasi. *Heap* adalah satu-satunya yang tidak punya entri $O(n)$.

Operasi	Antrean berbasis heap	Larik tak terurut	Larik terurut
`INSERT`	$O(\lg n)$	$O(1)$	$O(n)$
`MAXIMUM`	$O(1)$	$O(n)$	$O(1)$
`EXTRACT-MAX`	$O(\lg n)$	$O(n)$	$O(1)$
`INCREASE-KEY`	$O(\lg n)$	$O(1)$	$O(n)$

Tabel 5.4 adalah hujah bagi *heap* dalam satu gambar tunggal. Larik tak terurut membuat penyisipannya gratis tetapi pengeluarannya linear. Larik terurut membuat pengeluarannya gratis tetapi penyisipannya linear. Keduanya baik-baik saja kalau Anda mengerjakan hampir seluruhnya yang satu dan hampir tidak mengerjakan yang lain — tetapi antrean prioritas sungguhan mengerjakan campuran keduanya, dan saat itu operasi yang linear mendominasi segalanya. *Heap* tidak punya operasi yang murah maupun yang mahal; ia punya empat operasi yang logaritmik-atau-lebih-baik, dan $\lg n$ demikian nyaris konstan dalam praktik sehingga inilah tawar-menawar yang tepat.

> **ARITMETIKA TAWAR-MENAWARNYA.** Andaikan sebuah penjadwal melakukan sejuta penyisipan dan sejuta pengeluaran pada antrean yang memuat kira-kira sejuta pekerjaan. Dengan larik tak terurut: $10^6 O(1) + 10^6 O(10^6) = $ kira-kira 10^{12} operasi — sekitar

> tujuh belas menit pada semiliar operasi per detik. Dengan *heap*: $2 ^6 O(20) = $ kira-kira 4×10^7 operasi — sekitar empat perseratus detik.

5.7.8 Antrean prioritas lengkap dalam C++

Kode 5.5 mengimplementasikan keempat operasinya sebagai sebuah kelas, dan melatih setiap satu di antaranya.

Kode 5.5 Antrean prioritas-maks yang lengkap, dengan keempat operasinya dilatih.

```
// Antrean prioritas-maks di atas binary heap: MAXIMUM, EXTRACT-MAX, INCREASE-KEY, INSERT.
#include <iostream>
#include <vector>
#include <string>
#include <climits>
#include <stdexcept>
using namespace std;

class MaxPriorityQueue {
    vector<int> A;                              // A[0] tak terpakai; heap-nya tinggal di
A[1..heapSize]
    int heapSize = 0;

    void maxHeapify(int i) {                    // tenggelamkan A[i] ke kedalaman yang
semestinya
        int l = 2 * i, r = 2 * i + 1, largest = i;
        if (l <= heapSize && A[l] > A[largest]) largest = l;
        if (r <= heapSize && A[r] > A[largest]) largest = r;
        if (largest != i) { swap(A[i], A[largest]); maxHeapify(largest); }
    }

public:
    MaxPriorityQueue() : A(1, 0) {}             // A[0] hanya pengisi tempat, tak pernah
dibaca

    int size() const { return heapSize; }

    string toString() const {
        string s = "[";
        for (int i = 1; i <= heapSize; ++i) { s += to_string(A[i]); if (i < heapSize) s +=
" "; }
        return s + "]";
    }

    int maximum() const {                       // O(1): jawabannya semata-mata akarnya
        if (heapSize < 1) throw runtime_error("heap underflow");
        return A[1];
    }

    int extractMax() {                          // O(lg n)
        if (heapSize < 1) throw runtime_error("heap underflow");
        int best = A[1];
        A[1] = A[heapSize];                     // daun terakhirnya pindah ke akar ...
        --heapSize;                             // ... heap-nya menyusut ...
        maxHeapify(1);                          // ... dan akar barunya tenggelam ke
tempatnya
        return best;
    }
```

Kode 5.5 (Lanjutan) *Antrean prioritas-maks yang lengkap, dengan keempat operasinya dilatih.*

```
    void increaseKey(int i, int key, bool verbose = false) {   // O(lg n): apungkan A[i] ke
atas
        if (i < 1 || i > heapSize) throw runtime_error("index outside the heap");
        if (key < A[i]) throw runtime_error("new key is smaller than current key");
        A[i] = key;
        while (i > 1 && A[i / 2] < A[i]) {       // selama induknya dikalahkan anaknya
            if (verbose) cout << "      parent A[" << i / 2 << "]=" << A[i / 2]
                              << " < A[" << i << "]=" << A[i] << ", so swap them\n";
            swap(A[i], A[i / 2]);
            i = i / 2;
        }
        if (verbose) cout << "      the key has stopped at position " << i << "\n";
    }

    void insert(int key) {                       // O(lg n): tumbuh satu, lalu apungkan ke
atas
        ++heapSize;
        if ((int)A.size() <= heapSize) A.push_back(INT_MIN); else A[heapSize] = INT_MIN;
        increaseKey(heapSize, key);              // -takhingga hanya bisa naik
    }
};

int main() {
    MaxPriorityQueue q;
    cout << "INSERT in the order 4, 1, 3, 2, 16, 9, 10, 14, 8, 7\n";
    for (int k : {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}) {
        q.insert(k);
        cout << "  insert " << k << "\t-> " << q.toString() << "\n";
    }
    cout << "\nMAXIMUM = " << q.maximum() << "  (one array read, no comparisons)\n";

    cout << "\nINSERT(15) into the heap above\n";
    q.insert(15);
    cout << "  -> " << q.toString() << "\n";

    cout << "\nEXTRACT-MAX three times\n";
    for (int t = 0; t < 3; ++t)
        cout << "  extracted " << q.extractMax() << "\t-> " << q.toString() << "\n";

    cout << "\nINCREASE-KEY(A, 8, 20): raise the key at position 8 to 20\n";
    q.increaseKey(8, 20, true);
    cout << "  -> " << q.toString() << "\n";
```

Kode 5.5 (Lanjutan) *Antrean prioritas-maks yang lengkap, dengan keempat operasinya dilatih.*

```
    cout << "\nDrain the queue: ";
    while (q.size() > 0) cout << q.extractMax() << " ";
    cout << "\n";
    return 0;
}
```

Keluarannya:

```
INSERT in the order 4, 1, 3, 2, 16, 9, 10, 14, 8, 7
  insert 4  -> [4]
  insert 1  -> [4 1]
  insert 3  -> [4 1 3]
  insert 2  -> [4 2 3 1]
  insert 16 -> [16 4 3 1 2]
  insert 9  -> [16 4 9 1 2 3]
  insert 10 -> [16 4 10 1 2 3 9]
  insert 14 -> [16 14 10 4 2 3 9 1]
  insert 8  -> [16 14 10 8 2 3 9 1 4]
  insert 7  -> [16 14 10 8 7 3 9 1 4 2]

MAXIMUM = 16  (one array read, no comparisons)

INSERT(15) into the heap above
  -> [16 15 10 8 14 3 9 1 4 2 7]

EXTRACT-MAX three times
  extracted 16  -> [15 14 10 8 7 3 9 1 4 2]
  extracted 15  -> [14 8 10 4 7 3 9 1 2]
  extracted 14  -> [10 8 9 4 7 3 2 1]

INCREASE-KEY(A, 8, 20): raise the key at position 8 to 20
      parent A[4]=4 < A[8]=20, so swap them
      parent A[2]=8 < A[4]=20, so swap them
      parent A[1]=10 < A[2]=20, so swap them
      the key has stopped at position 1
  -> [20 10 9 8 7 3 2 4]

Drain the queue: 20 10 9 8 7 4 3 2
```

Dua hal dalam keluaran itu layak diperhatikan dengan saksama.

Pertama, lihatlah lariknya sesudah menyisipkan sepuluh bilangan itu: [16,14,10,8,7,3,9,1,4,2]. Bandingkan dengan larik yang dihasilkan `BUILD-MAX-HEAP` dari *sepuluh bilangan yang sama* di Subbab 5.5.3: [16,14,10,8,7,9,3,2,4,1]. **Keduanya adalah heap yang berbeda.** Keduanya sah — periksalah ketaksamaan induk-anak pada masing-masing dan Anda tidak akan menemukan pelanggaran — yang merupakan pokok yang sudah dikemukakan di Subbab 5.3.4: sehimpunan bilangan punya banyak *heap* yang sah. Perhatikan juga beda ongkosnya. Sepuluh `INSERT` berongkos $O(n\lg n)$; satu `BUILD-MAX-HEAP` berongkos $O(n)$. Kalau seluruh datanya sudah ada di depan Anda, bangunlah; kalau datanya datang satu per satu, sisipkanlah.

Kedua, penelusuran `INCREASE-KEY`-nya mereproduksi penelusuran tangan di Subbab 5.7.5 secara persis: tiga penukaran, tiga tingkat, berakhir di akarnya.

5.7.9 Memakai pustaka baku saja

Dalam produksi — dan dalam lomba pemrograman, tempat waktu ikut dinilai [4] — Anda tidak akan menulis kelas di atas. C++ sudah menyertakannya, dan standarnya menjamin kompleksitasnya [5]. **Kode 5.6** menunjukkan empat cara yang sungguh akan Anda pakai untuk memakai *heap* dalam C++.

Kode 5.6 *Heap* dari pustaka baku C++: `std::priority_queue`, pembanding buatan sendiri, dan algoritma mentah `make_heap` / `sort_heap`.

```
// Pekerjaan yang sama dikerjakan dengan pustaka baku: std::priority_queue dan algoritma
heap-nya.
#include <iostream>
#include <queue>
#include <vector>
#include <string>
#include <algorithm>
#include <functional>
using namespace std;

struct ScanJob {                                // sebuah pekerjaan perayapan yang
menunggu gilirannya
    int  urgency;                               // makin besar makin genting
    string url;
};
struct ByUrgency {                              // "kurang genting" datang lebih dulu
dalam urutannya,
    bool operator()(const ScanJob& a, const ScanJob& b) const {
        return a.urgency < b.urgency;           // sehingga PUNCAK antreannya adalah yang
paling genting
    }
};

int main() {
    // (1) Max-heap berisi bilangan bulat biasa: inilah bawaannya.
    priority_queue<int> maxQ;
    for (int k : {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}) maxQ.push(k);
    cout << "max-heap drain: ";
    while (!maxQ.empty()) { cout << maxQ.top() << " "; maxQ.pop(); }
    cout << "\n";

    // (2) Min-heap: wadah yang sama, pembandingnya dibalik.
    priority_queue<int, vector<int>, greater<int>> minQ;
    for (int k : {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}) minQ.push(k);
    cout << "min-heap drain: ";
    while (!minQ.empty()) { cout << minQ.top() << " "; minQ.pop(); }
    cout << "\n";

    // (3) Heap berisi rekaman, diurutkan menurut satu medan.
    priority_queue<ScanJob, vector<ScanJob>, ByUrgency> jobs;
    jobs.push({3, "/pricing"});
    jobs.push({9, "/"});
    jobs.push({5, "/blog/geo-basics"});
    jobs.push({7, "/products"});
    cout << "scan order: ";
    while (!jobs.empty()) { cout << jobs.top().url << "(" << jobs.top().urgency << ") ";
jobs.pop(); }
    cout << "\n";
```

Kode 5.6 (Lanjutan) *Heap dari pustaka baku C++: std::priority_queue, pembanding buatan sendiri, dan algoritma mentah make_heap / sort_heap.*

```
    // (4) Algoritma heap mentah pada vector milik Anda. Ingat make_heap adalah pembangunan
O(n).
    vector<int> v = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7};
    make_heap(v.begin(), v.end());              // BUILD-MAX-HEAP, dalam O(n)
    cout << "make_heap:      ";
    for (int x : v) cout << x << " ";
    cout << "\n";
    sort_heap(v.begin(), v.end());              // gelung pengeluaran milik HEAPSORT
    cout << "sort_heap:      ";
    for (int x : v) cout << x << " ";
    cout << "\n";
    return 0;
}
```

Keluarannya:

```
max-heap drain: 16 14 10 9 8 7 4 3 2 1
min-heap drain: 1 2 3 4 7 8 9 10 14 16
scan order: /(9) /products(7) /blog/geo-basics(5) /pricing(3)
make_heap:      16 14 10 8 7 9 3 2 4 1
sort_heap:      1 2 3 4 7 8 9 10 14 16
```

Lihatlah baris keempat. `make_heap` menghasilkan [16,14,10,8,7,9,3,2,4,1] — **aksara demi aksara sama dengan larik yang dihasilkan `BUILD-MAX-HEAP` buatan kita sendiri** di Subbab 5.5.7, dan sama dengan larik di **Gambar 5.12**. Pustaka bakunya mengimplementasikan persis algoritma bab ini.

> **JEBAKAN ARAH.** `std::priority_queue<int>` adalah *max*-heap: `top()` memberikan yang terbesar. Fungsi-fungsi `heapq` milik Python secara bawaan membangun *min*-heap — Python 3.14 menambahkan fungsi terpisah `heapify_max`, `heappush_max`, dan `heappop_max` untuk *max*-heap [6] — dan `PriorityQueue` milik Java pun *min*-heap. Berpindah antarbahasa dengan sembrono, dan Dijkstra Anda akan menemukan lintasan terpanjang. Untuk membalik antrean C++ menjadi *min*-heap, berikan `greater<int>` sebagai argumen templat ketiganya, seperti pada kasus (2) di atas — atau negatifkan setiap kunci saat masuk lalu negatifkan lagi saat keluar, yang bekerja tetapi mudah salah dengan kunci tak bertanda atau kunci titik-mengambang.

5.8 Quicksort

5.8.1 Gagasannya

Quicksort ditemukan oleh Tony Hoare pada 1959 dan diterbitkan pada 1962 [7], sewaktu ia menjadi mahasiswa tamu di Moskwa yang mengerjakan proyek penerjemahan mesin yang perlu mengurutkan kata-kata Rusia sebelum mencarinya di kamus. Enam puluh lima tahun kemudian ia masih merupakan pengurutan yang pertama kali diraih oleh sebagian besar pustaka baku. Ia mempunyai kasus terburuk $\Theta(n^2)$, yang kedengarannya sudah menggugurkan — dan toh ia mengalahkan Heapsort, yang sama sekali tidak punya kasus buruk, pada hampir setiap masukan sungguhan. Memahami *mengapa* adalah hal paling mencerahkan dalam bab ini.

> **DEFINISI. Quicksort** adalah algoritma pengurutan *divide and conquer* (bagi dan kuasai). Diberi subarray $A[p \ldots r]$, ia memilih sebuah elemen yang disebut **pivot**, menata ulang subarray itu sehingga setiap elemen yang tidak lebih besar daripada *pivot* mendahului setiap elemen yang tidak lebih kecil daripadanya, lalu mengurutkan kedua potongan yang dihasilkan secara rekursif. Tidak ada langkah pemaduan: begitu kedua potongan terurut, seluruh subarray itu terurut, di tempat.

5.8.2 Bagi dan kuasai, dengan pekerjaannya dipindahkan

Bab 3 memberi kita cetak biru tiga langkah bagi dan kuasai. Quicksort mengisinya seperti ini:

1. **Bagi.** Partisikan $A[p \ldots r]$ menjadi dua subarray $A[p \ldots q-1]$ dan $A[q+1 \ldots r]$ sedemikian rupa sehingga setiap elemen yang pertama $\leq A[q]$ dan setiap elemen yang kedua $\geq A[q]$. Menghitung indeks q dan melakukan penataan ulangnya adalah seluruh pekerjaannya.
2. **Kuasai.** Urutkan kedua subarray itu dengan pemanggilan rekursif Quicksort.
3. **Padu.** Tidak ada. Subarray-nya terurut di tempat, dan *pivot*-nya sudah duduk di antara keduanya, sehingga $A[p \ldots r]$ terurut pada saat kedua pemanggilan rekursifnya kembali.

Bandingkan dengan Merge Sort dari Bab 3:

Tabel 5.5 Merge Sort dan Quicksort adalah bayangan cermin satu sama lain. Masing-masing membelanjakan $\Theta(n)$ per tingkat; keduanya berbeda dalam *di mana* mereka membelanjakannya, dan dalam apakah belahannya terjamin.

	Merge Sort	Quicksort
Bagi	sepele — hitung titik tengahnya, $\Theta(1)$	mahal — pemartisian, $\Theta(n)$
Kuasai	dua pemanggilan rekursif	dua pemanggilan rekursif
Padu	mahal — pemaduan, $\Theta(n)$	sepele — tidak ada, $\Theta(1)$
Ukuran belahan	selalu tepat separuh-separuh	apa pun yang diberikan datanya
Memori tambahan	$\Theta(n)$ untuk penyangga pemaduannya	tidak ada

Tabel 5.5 menyebutkan tawar-menawar yang sesungguhnya. Merge Sort *membeli* belahannya yang selalu rata dengan membayar sebuah penyangga pemaduan. Quicksort menolak membayar, dan sebagai gantinya menerima bahwa belahannya adalah apa pun yang kebetulan diberikan datanya. Subbab 5.9 dan 5.10 membahas seberapa buruk keadaan itu bisa jadi dan bagaimana mencegahnya menjadi buruk.

5.8.3 `PARTITION` versi Lomuto, dan invarian empat wilayah

Kita sajikan dua skema pemartisian. Yang pertama, versi Lomuto, adalah yang ada di edisi mutakhir CLRS [3] dan yang sebaiknya Anda tulis kalau Anda sedang menulis Quicksort pertama Anda: ia punya satu gelung, satu arah, dan sebuah invarian yang dapat Anda lihat. **Pseudokode 5.9** memberikannya secara lengkap.

Pseudokode 5.9 `PARTITION` versi Lomuto: satu sapuan dari kiri ke kanan yang menumbuhkan wilayah elemen yang tidak lebih besar daripada *pivot*-nya.

```
PARTITION(A, p, r):
 1.  x ← A[r]                        // pivotnya adalah elemen terakhir
 2.  i ← p − 1                       // i menandai ujung wilayah "≤ x"
 3.  for j ← p to r − 1
 4.      if A[j] ≤ x
 5.          i ← i + 1
 6.          swap A[i] ↔ A[j]
 7.  swap A[i+1] ↔ A[r]              // jatuhkan pivotnya ke celahnya
 8.  return i + 1
```

Peubah gelung j menyapu dari kiri ke kanan, satu kali. Pada setiap saat, subarray-nya terbagi menjadi **empat wilayah**:

```
p                 i        j                 r
┌──────────────────┬────────┬──────────────────┬──┐
│     ≤ x          │  > x   │  belum diperiksa │ x │
└──────────────────┴────────┴──────────────────┴──┘
 A[p..i]           A[i+1..j-1]  A[j..r-1]     A[r]
```

> **INVARIAN GELUNG (empat wilayah).** Pada awal setiap iterasi gelung `for` pada baris 3, untuk sembarang indeks larik k: 1. jika $p \le k \le i$ maka $A[k] \le x$; 2. jika $i + 1 \le k \le j - 1$ maka $A[k] > x$; 3. jika $k = r$ maka $A[k] = x$. Indeks j sampai $r - 1$ adalah wilayah yang belum diperiksa, yang tentangnya tidak ada yang diklaim.

Inisialisasi. Sebelum iterasi pertama, $i = p - 1$ dan $j = p$. Wilayah 1 adalah $A[p \dots p - 1]$ — kosong, sehingga syarat 1 berlaku secara hampa. Wilayah 2 adalah $A[p \dots p - 1]$ — juga kosong, sehingga syarat 2 berlaku secara hampa. Syarat 3 berlaku sebab baris 1 menyetel $x = A[r]$ dan belum ada yang dipindahkan. ✓

Pemeliharaan. Tinjau sebuah iterasi dengan invariannya berlaku. Dua kasus:

- $A[j] > x$**.** Baris 5–6 dilewati dan hanya j yang bertambah. Elemen di indeks j, yang $>x$, kini jatuh ke dalam wilayah 2, yang sifat pendefinisinya persis "$>x$". Syarat 2 tetap berlaku. Syarat 1 dan 3 tak tersentuh. ✓
- $A[j] \leq x$**.** Baris 5 menambah i, sehingga indeks i kini menunjuk pada elemen *pertama* wilayah 2 yang lama — sebuah nilai $>x$. Baris 6 menukarkan nilai itu dengan $A[j]$, yang $\leq x$. Sesudahnya: $A[i] \leq x$, sehingga ia dengan benar bergabung ke wilayah 1; dan nilai $>x$ yang tadinya berada di $A[i]$ sudah pindah ke indeks j, yang menjadi slot terakhir wilayah 2 ketika j bertambah. Syarat 1 dan 2 berlaku kembali. ✓

Terminasi. Gelungnya berakhir ketika $j = r$. Wilayah yang belum diperiksa, $A[j \ldots r-1] = A[r \ldots r-1]$, kosong, sehingga setiap elemen $A[p \ldots r-1]$ terletak di wilayah 1 atau wilayah 2, yaitu entah $\leq x$ atau $>x$. Baris 7 lalu menukarkan $A[i+1]$ — elemen pertama wilayah 2, jadi sebuah nilai $>x$ — dengan *pivot* di $A[r]$. *Pivot*-nya mendarat di indeks $i+1$, dengan segala yang $\leq x$ di kirinya dan segala yang $>x$ di kanannya. Baris 8 mengembalikan $q = i+1$, dan pascasyaratnya Persamaan (5.9)

$$A[k] \leq A[q] \text{ untuk } p \leq k < q, \qquad A[k] > A[q] \text{ untuk } q < k \leq r \tag{5.9}$$

tegak. ✓ ■

Waktu jalannya. Gelung `for`-nya berjalan $r - p$ kali, dan setiap iterasinya mengerjakan $O(1)$ pekerjaan. Dengan $n = r - p + 1$, itu adalah $\Theta(n)$.

5.8.4 Satu pemartisian Lomuto, ditelusuri dengan tangan

Partisikan $A = [2,8,7,1,3,5,6,4]$, indeks 1 sampai 8. *Pivot*-nya adalah $x = A[8] = 4$. Kita mulai dengan $i = 0$.

Tabel 5.6 `PARTITION` versi Lomuto pada [2,8,7,1,3,5,6,4] dengan *pivot* 4. Awalan yang digarisbawahi adalah wilayah 1 — elemen-elemen yang sudah diketahui ≤ 4.

j	$A[j]$	$A[j] \leq 4$?	Tindakan	Larik sesudahnya
1	2	ya	$i \to 1$; tukar $A[1] \leftrightarrow A[1]$	[2, 8,7,1,3,5,6,4]
2	8	tidak	tidak ada	[2, 8,7,1,3,5,6,4]
3	7	tidak	tidak ada	[2, 8,7,1,3,5,6,4]
4	1	ya	$i \to 2$; tukar $A[2] \leftrightarrow A[4]$	[2,1, 7,8,3,5,6,4]
5	3	ya	$i \to 3$; tukar $A[3] \leftrightarrow A[5]$	[2,1,3, 8,7,5,6,4]
6	5	tidak	tidak ada	[2,1,3, 8,7,5,6,4]
7	6	tidak	tidak ada	[2,1,3, 8,7,5,6,4]

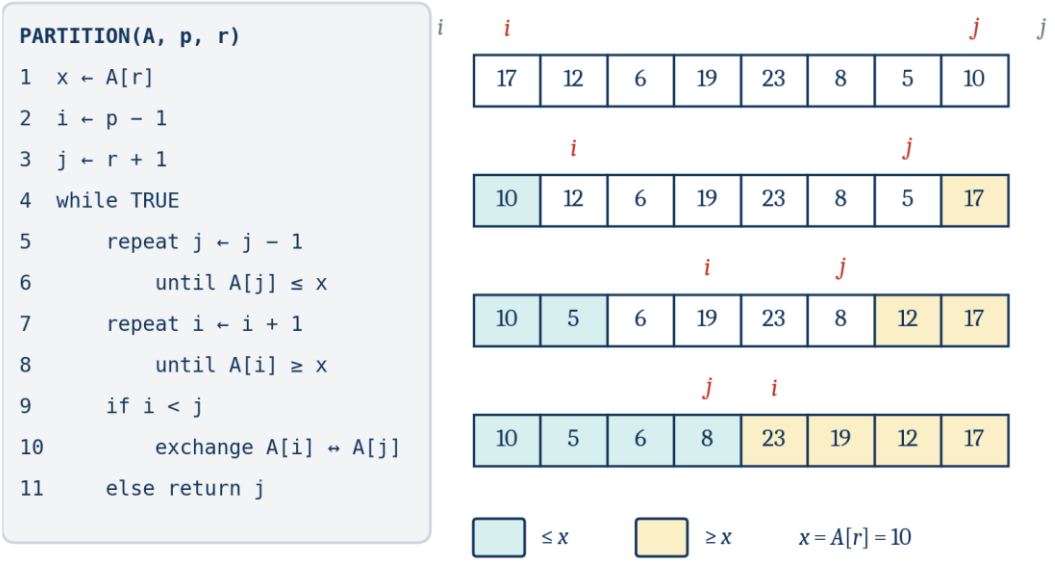
Tabel 5.6 membawa gelungnya. Sesudah itu, $i = 3$, sehingga baris 7 menukarkan $A[4] \leftrightarrow A[8]$, memberikan

$$A = [2,\ 1,\ 3,\ \mathbf{4},\ 7,\ 5,\ 6,\ 8],$$

dan baris 8 mengembalikan $q = 4$. Periksa pascasyarat Persamaan (5.9): segala yang di kiri indeks 4 adalah {2,1,3}, semuanya ≤ 4 ✓; segala yang di kanannya adalah {7,5,6,8}, semuanya >4 ✓. *Pivot* 4 kini berada di posisi terurutnya yang final dan tidak akan pernah bergerak lagi.

5.8.5 `PARTITION` versi Hoare, skema aslinya

Skema yang kedua adalah skema asli Hoare [7], yang di dalamnya dua penunjuk berangkat dari ujung yang berlawanan dan berjalan saling mendekat. **Gambar 5.17** menampilkannya: pseudokodenya di kiri, dan penelusuran lengkapnya di kanan. **Pseudokode 5.10** mengulang pseudokode itu dalam bentuk teks, supaya setiap barisnya dapat diberi komentar.



Gambar 5.17 `PARTITION` versi Hoare, dengan penelusuran lengkap pada [17,12,6,19,23,8,5,10]. Hijau toska menandai elemen yang sudah diketahui $\leq x$, kuning tua yang sudah diketahui $\geq x$. Penunjuk i dan j berangkat dari tepat di luar lariknya lalu merapat; ketika keduanya bersilangan, pemartisiannya selesai dan j dikembalikan.

Pseudokode 5.10 `PARTITION` versi Hoare: dua penunjuk merapat dari kedua ujung dan menukarkan pasangan yang salah tempat sampai keduanya bersilangan.

```
PARTITION(A, p, r):                    // Hoare, 1961
 1.  x ← A[r]                          // NILAI pivotnya (slot pivotnya tidak berpindah)
 2.  i ← p − 1
 3.  j ← r + 1
 4.  while TRUE
 5.      repeat j ← j − 1 until A[j] ≤ x
 6.      repeat i ← i + 1 until A[i] ≥ x
 7.      if i < j then swap A[i] ↔ A[j]
 8.      else return j
```

Dengan kata-kata. Penunjuk j berjalan ke kiri mencari sesuatu yang semestinya berada di kiri (sebuah nilai $\leq x$). Penunjuk i berjalan ke kanan mencari sesuatu yang semestinya berada di kanan (sebuah nilai $\geq x$). Ketika keduanya sudah menemukan elemen yang salah tempat, tukarkan keduanya — masing-masing pergi ke sisi tempatnya semestinya. Ketika penunjuknya bertemu atau bersilangan, segalanya sudah diperiksa dan j menandai batasnya.

Bedanya dari Lomuto layak dinyatakan dengan tepat, sebab beda itu sering menjebak orang:

- Lomuto mengembalikan **posisi final pivotnya**, dan *pivot*-nya dikecualikan dari kedua pemanggilan rekursifnya: `QUICKSORT(A, p, q-1)` dan `QUICKSORT(A, q+1, r)`.
- Hoare mengembalikan **sebuah indeks batas**, bukan posisi *pivot*-nya. *Pivot*-nya boleh berada di mana saja. Jadi pemanggilan rekursifnya harus berupa `QUICKSORT(A, p, q)` dan `QUICKSORT(A, q+1, r)` — perhatikan q-nya, bukan $q - 1$. Menulis $q - 1$ di sini kehilangan satu elemen;

menulis $q + 1$ untuk pemanggilan kirinya dan q untuk pemanggilan kanannya menyebabkan rekursi tak berkesudahan.

Versi Hoare melakukan kira-kira tiga kali lebih sedikit penukaran daripada versi Lomuto pada data acak, dan ia berperilaku jauh lebih baik ketika lariknya banyak memuat kunci yang sama — itulah sebabnya pustaka sungguhan memakai varian-variannya. Versi Lomuto lebih mudah dibuktikan dan lebih mudah diingat. Pelajari Lomuto lebih dulu, lalu naik kelas.

> **JEBAKAN PIVOT DALAM SKEMA HOARE.** Baris 1 **Pseudokode 5.10**, seperti **Gambar 5.17**, mengambil nilai *pivot*-nya dari slot **terakhir**, $x = A[r]$. Itu sepenuhnya benar untuk satu pemartisian, dan penelusuran pertama di bawah memakainya. Tetapi pilihan itu tidak boleh digabungkan dengan rekursi `QUICKSORT(A, p, q)` dan `QUICKSORT(A, q+1, r)`. Andaikan $A[r]$ kebetulan merupakan maksimum tegas dari $A[p \dots r]$. Penunjuk j langsung berhenti di r, sebab $A[r] \leq x$. Penunjuk i memanjat melewati setiap elemen, yang semuanya lebih kecil daripada x, dan baru berhenti di r juga. Kini $i = j = r$, sehingga prosedurnya mengembalikan $q = r$, dan pemanggilan kiri `QUICKSORT(A, p, r)` menerima subarray yang persis sama: rekursinya tidak pernah berakhir. Obatnya adalah mengambil nilai *pivot* dari slot **pertama**, $x = A[p]$, sebagaimana dilakukan CLRS [3] ketika membahas skema Hoare. Dengan begitu q yang dikembalikan selalu memenuhi $p \leq q < r$, kedua potongannya benar-benar lebih pendek daripada $A[p \dots r]$, dan rekursinya aman. Penelusuran kedua di bawah memakai pilihan itu.

5.8.6 Pemartisian Hoare, ditelusuri dua kali

Ambil larik $A = [17,12,6,19,23,8,5,10]$, indeks $1 \dots 8$.

Penelusuran pertama: pivotnya adalah elemen terakhir, $x = A[8] = 10$**.** Inilah yang sungguh dikatakan baris 1 pseudokodenya, dan inilah yang digambar **Gambar 5.17**.

- **Sapuan 1.** j mulai di 9 lalu melangkah turun: $A[8] = 10 \leq 10$ ✓ — berhenti di $j = 8$. i mulai di 0 lalu melangkah naik: $A[1] = 17 \geq 10$ ✓ — berhenti di $i = 1$. Karena $1 < 8$, tukarkan $A[1] \leftrightarrow A[8]$: [10,12,6,19,23,8,5,17].
- **Sapuan 2.** j melangkah turun dari 8: $A[7] = 5 \leq 10$ ✓ — berhenti di $j = 7$. i melangkah naik dari 1: $A[2] = 12 \geq 10$ ✓ — berhenti di $i = 2$. Karena $2 < 7$, tukar: [10,5,6,19,23,8,12,17].
- **Sapuan 3.** j melangkah turun dari 7: $A[6] = 8 \leq 10$ ✓ — berhenti di $j = 6$. i melangkah naik dari 2: $A[3] = 6 \geq 10$? Tidak. $A[4] = 19 \geq 10$ ✓ — berhenti di $i = 4$. Karena $4 < 6$, tukar: [10,5,6,8,23,19,12,17].
- **Sapuan 4.** j melangkah turun dari 6: $A[5] = 23 \leq 10$? Tidak. $A[4] = 8 \leq 10$ ✓ — berhenti di $j = 4$. i melangkah naik dari 4: $A[5] = 23 \geq 10$ ✓ — berhenti di $i = 5$. Kini $i = 5 \not< j = 4$: penunjuknya sudah bersilangan. **Kembalikan** $j = 4$**.**

Bagian kiri $A[1 \dots 4] = [10,5,6,8]$, setiap elemennya ≤ 10 ✓. Bagian kanan $A[5 \dots 8] = [23,19,12,17]$, setiap elemennya ≥ 10 ✓.

Penelusuran kedua: nilai pivotnya adalah 17**.** Sekarang ambil *pivot*-nya dari slot pertama, $x = A[1] = 17$ — pilihan yang aman untuk dipakai di dalam Quicksort, seperti dijelaskan peringatan di atas. Mencerahkan pula melihat bahwa prosedur yang sama bekerja untuk nilai *pivot* mana pun yang diambil dari lariknya.

- **Sapuan 1.** j: $A[8] = 10 \leq 17$ ✓, $j = 8$. i: $A[1] = 17 \geq 17$ ✓, $i = 1$. Tukar: [10,12,6,19,23,8,5,17].
- **Sapuan 2.** j: $A[7] = 5 \leq 17$ ✓, $j = 7$. i: $A[2] = 12 \geq 17$? Tidak. $A[3] = 6$? Tidak. $A[4] = 19 \geq 17$ ✓, $i = 4$. Tukar: [10,12,6,5,23,8,19,17].

- **Sapuan 3.** j: $A[6] = 8 \leq 17$ ✓, $j = 6$. i: $A[5] = 23 \geq 17$ ✓, $i = 5$. Tukar: [10,12,6,5,8,23,19,17].
- **Sapuan 4.** j: $A[5] = 8 \leq 17$ ✓, $j = 5$. i: $A[6] = 23 \geq 17$ ✓, $i = 6$. Bersilangan. **Kembalikan** $j = 5$.

Bagian kiri [10,12,6,5,8], semuanya ≤ 17 ✓. Bagian kanan [23,19,17], semuanya ≥ 17 ✓.

Kode 5.7 menjalankan ketiga penelusuran itu dan mencetaknya, sehingga Anda dapat memeriksa pekerjaan tangan Anda terhadap mesinnya.

Kode 5.7 Skema pemartisian Lomuto dan Hoare, masing-masing menarasikan langkahnya sendiri. Indeks di sini berbasis 0, jadi kurangilah satu dari setiap indeks pada penelusuran tangan di atas.

```
// Dua skema pemartisian, masing-masing ditelusuri selangkah demi selangkah: Lomuto (CLRS
ed. 4) dan Hoare (1961).
#include <iostream>
#include <vector>
#include <string>
using namespace std;

void dump(const vector<int>& A, int p, int r, const string& note) {
    cout << "    [";
    for (int k = p; k <= r; ++k) cout << A[k] << (k < r ? " " : "");
    cout << "]  " << note << "\n";
}

// ---------- Lomuto: satu sapuan maju, empat wilayah ----------
int lomutoPartition(vector<int>& A, int p, int r) {
    int x = A[r];                                   // pivotnya adalah elemen terakhir
    cout << "  pivot x = A[" << r << "] = " << x << "\n";
    int i = p - 1;                                  // A[p..i] memuat nilai-nilai <= x
    for (int j = p; j <= r - 1; ++j) {
        cout << "  j = " << j << ", A[j] = " << A[j];
        if (A[j] <= x) {
            ++i;
            swap(A[i], A[j]);
            cout << " <= " << x << ": grow the small region to i = " << i
                 << " and swap A[" << i << "] with A[" << j << "]\n";
        } else {
            cout << " > " << x << ": leave it in the large region, i stays " << i << "\n";
        }
        dump(A, p, r, "");
    }
    swap(A[i + 1], A[r]);                           // jatuhkan pivotnya di antara kedua
wilayahnya
    cout << "  put the pivot in its place: swap A[" << i + 1 << "] with A[" << r << "]\n";
    dump(A, p, r, "");
    return i + 1;
}

// ---------- Hoare: dua sapuan yang berjalan saling mendekat ----------
int hoarePartition(vector<int>& A, int p, int r, int pivotValue) {
    int x = pivotValue;
    cout << "  pivot value x = " << x << "\n";
    int i = p - 1, j = r + 1;
    int scan = 0;
    while (true) {
        ++scan;
        do { --j; } while (A[j] > x);               // berjalan ke kiri sampai menemukan nilai
<= x
        do { ++i; } while (A[i] < x);               // berjalan ke kanan sampai menemukan
nilai >= x
        cout << "  scan " << scan << ": j stops at index " << j << " (A[j]=" << A[j]
```

Kode 5.7 (Lanjutan) *Skema pemartisian Lomuto dan Hoare, masing-masing menarasikan langkahnya sendiri. Indeks di sini berbasis 0, jadi kurangilah satu dari setiap indeks pada penelusuran tangan di atas.*

```
            << "), i stops at index " << i << " (A[i]=" << A[i] << ")";
        if (i < j) {
            cout << "; i < j so swap\n";
            swap(A[i], A[j]);
            dump(A, p, r, "");
        } else {
            cout << "; i >= j so stop and return j = " << j << "\n";
            return j;
        }
    }
}

int main() {
    {
        cout << "LOMUTO PARTITION on [2 8 7 1 3 5 6 4]\n";
        vector<int> A = {2, 8, 7, 1, 3, 5, 6, 4};
        int q = lomutoPartition(A, 0, 7);
        cout << "  returns q = " << q << " (0-based), pivot sits at A[" << q << "] = " <<
A[q] << "\n\n";
    }
    {
        cout << "HOARE PARTITION on [17 12 6 19 23 8 5 10], pivot = last element = 10\n";
        vector<int> A = {17, 12, 6, 19, 23, 8, 5, 10};
        int q = hoarePartition(A, 0, 7, A[7]);
        cout << "  final array:";
        dump(A, 0, 7, "");
        cout << "  left part A[0.." << q << "] = ";
        for (int k = 0; k <= q; ++k) cout << A[k] << " ";
        cout << "\n  right part A[" << q + 1 << "..7] = ";
        for (int k = q + 1; k <= 7; ++k) cout << A[k] << " ";
        cout << "\n\n";
    }
    {
        cout << "HOARE PARTITION on [17 12 6 19 23 8 5 10], pivot = first element = 17\n";
        vector<int> A = {17, 12, 6, 19, 23, 8, 5, 10};
        int q = hoarePartition(A, 0, 7, A[0]);
        cout << "  final array:";
        dump(A, 0, 7, "");
        cout << "  left part A[0.." << q << "] = ";
        for (int k = 0; k <= q; ++k) cout << A[k] << " ";
        cout << "\n  right part A[" << q + 1 << "..7] = ";
        for (int k = q + 1; k <= 7; ++k) cout << A[k] << " ";
        cout << "\n";
    }
    return 0;
}
```

Keluarannya — perhatikan bahwa larik C++ mulai dari 0, sehingga $q = 3$ yang dikembalikan bersesuaian dengan $q = 4$ pada penelusuran tangan berbasis 1, dan $q = 4$ bersesuaian dengan $q = 5$ pada penelusuran tangannya:

```
LOMUTO PARTITION on [2 8 7 1 3 5 6 4]
  pivot x = A[7] = 4
  j = 0, A[j] = 2 <= 4: grow the small region to i = 0 and swap A[0] with A[0]
    [2 8 7 1 3 5 6 4]
  j = 1, A[j] = 8 > 4: leave it in the large region, i stays 0
    [2 8 7 1 3 5 6 4]
  j = 2, A[j] = 7 > 4: leave it in the large region, i stays 0
    [2 8 7 1 3 5 6 4]
  j = 3, A[j] = 1 <= 4: grow the small region to i = 1 and swap A[1] with A[3]
    [2 1 7 8 3 5 6 4]
  j = 4, A[j] = 3 <= 4: grow the small region to i = 2 and swap A[2] with A[4]
    [2 1 3 8 7 5 6 4]
  j = 5, A[j] = 5 > 4: leave it in the large region, i stays 2
    [2 1 3 8 7 5 6 4]
  j = 6, A[j] = 6 > 4: leave it in the large region, i stays 2
    [2 1 3 8 7 5 6 4]
  put the pivot in its place: swap A[3] with A[7]
    [2 1 3 4 7 5 6 8]
  returns q = 3 (0-based), pivot sits at A[3] = 4

HOARE PARTITION on [17 12 6 19 23 8 5 10], pivot = last element = 10
  pivot value x = 10
  scan 1: j stops at index 7 (A[j]=10), i stops at index 0 (A[i]=17); i < j so swap
    [10 12 6 19 23 8 5 17]
  scan 2: j stops at index 6 (A[j]=5), i stops at index 1 (A[i]=12); i < j so swap
    [10 5 6 19 23 8 12 17]
  scan 3: j stops at index 5 (A[j]=8), i stops at index 3 (A[i]=19); i < j so swap
    [10 5 6 8 23 19 12 17]
  scan 4: j stops at index 3 (A[j]=8), i stops at index 4 (A[i]=23); i >= j so stop and
return j = 3
  final array:    [10 5 6 8 23 19 12 17]
  left part A[0..3] = 10 5 6 8
  right part A[4..7] = 23 19 12 17

HOARE PARTITION on [17 12 6 19 23 8 5 10], pivot = first element = 17
  pivot value x = 17
  scan 1: j stops at index 7 (A[j]=10), i stops at index 0 (A[i]=17); i < j so swap
    [10 12 6 19 23 8 5 17]
  scan 2: j stops at index 6 (A[j]=5), i stops at index 3 (A[i]=19); i < j so swap
    [10 12 6 5 23 8 19 17]
  scan 3: j stops at index 5 (A[j]=8), i stops at index 4 (A[i]=23); i < j so swap
    [10 12 6 5 8 23 19 17]
  scan 4: j stops at index 4 (A[j]=8), i stops at index 5 (A[i]=23); i >= j so stop and
return j = 4
  final array:    [10 12 6 5 8 23 19 17]
  left part A[0..4] = 10 12 6 5 8
  right part A[5..7] = 23 19 17
```

Setiap langkah kedua penelusuran Hoare itu cocok dengan pekerjaan tangannya, dan yang pertama di antaranya cocok dengan **Gambar 5.17** secara persis.

5.8.7 Prosedur Quicksort itu sendiri

Dengan `PARTITION` di tangan, Quicksort hanyalah keempat baris **Pseudokode 5.11**.

Pseudokode 5.11 `QUICKSORT` dengan pemartisian Lomuto: partisikan, lalu urutkan kedua sisi *pivot*-nya secara rekursif.

```
QUICKSORT(A, p, r):
 1.  if p < r
 2.      q ← PARTITION(A, p, r)
 3.      QUICKSORT(A, p, q − 1)          // Lomuto: pivot di q sudah final
 4.      QUICKSORT(A, q + 1, r)
```

Pemanggilan awalnya adalah `QUICKSORT(A, 1, n)`.

Kasus basisnya adalah syarat pada baris 1. Jika $p \geq r$, subarray-nya beranggotakan nol atau satu elemen, yang sudah terurut, dan prosedurnya langsung kembali. Itulah seluruh hujah terminasinya: setiap pemanggilan rekursif menerima subarray yang benar-benar lebih pendek, sebab *pivot* di indeks q bukan milik salah satu pun di antaranya, sehingga rekursinya pasti mencapai panjang ≤ 1 sesudah berhingga langkah.

> **KESALAHAN SATU-ANGKA YANG MERUSAK SEGALANYA.** Kalau Anda memakai pemartisian Lomuto tetapi menulis `QUICKSORT(A, p, q)` pada baris 3, *pivot*-nya ikut termuat dalam pemanggilan rekursif kirinya. Ketika *pivot*-nya kebetulan merupakan elemen terbesar dalam subarray-nya, pemartisiannya mengembalikan $q = r$, pemanggilan kirinya menjadi `QUICKSORT(A, p, r)` — subarray yang *sama* — dan programnya berekursi selamanya sampai tumpukan panggilannya habis. Aturannya sederhana: **Lomuto mengecualikan pivotnya dari kedua pemanggilan; Hoare tidak mengecualikannya dari satu pun.** Cocokkan rekursinya dengan pemartisian yang sungguh Anda tulis.

5.8.8 Quicksort, ditelusuri tuntas

Kode 5.8 mengurutkan [2,8,7,1,3,5,6,4] dan mencetak seluruh lariknya sesudah tiap pemartisian, dengan kurung siku menandai subarray yang sedang dikerjakan.

Kode 5.8 Quicksort dengan pemartisian Lomuto, yang menarasikan rekursinya. Indentasinya menunjukkan kedalaman pemanggilannya.

```
// QUICKSORT dengan pemartisian Lomuto, mencetak seluruh lariknya sesudah tiap pemartisian.
#include <iostream>
#include <vector>
#include <string>
using namespace std;

vector<int> A;
int calls = 0;

void showWindow(int p, int r, const string& note) {
    cout << "  ";
    for (int k = 0; k < (int)A.size(); ++k) {
        if (k == p) cout << "[";
        cout << A[k];
        if (k == r) cout << "]"; else cout << " ";
    }
    cout << "  " << note << "\n";
}

int partition(int p, int r) {
    int x = A[r], i = p - 1;
    for (int j = p; j <= r - 1; ++j)
        if (A[j] <= x) swap(A[++i], A[j]);
    swap(A[i + 1], A[r]);
    return i + 1;
}

void quicksort(int p, int r, int depth) {
    if (p >= r) {                                // nol atau satu elemen: tidak ada yang
perlu dikerjakan
        if (p == r) cout << string(2 * depth + 2, ' ') << "QUICKSORT(" << p << "," << r
                         << "): one element, return\n";
        return;
    }
    ++calls;
    cout << string(2 * depth + 2, ' ') << "QUICKSORT(" << p << "," << r
         << "): pivot = A[" << r << "] = " << A[r] << "\n";
    int q = partition(p, r);
    cout << string(2 * depth + 2, ' ');
    showWindow(p, r, "after PARTITION, pivot " + to_string(A[q]) + " is final at index " +
to_string(q));
    quicksort(p, q - 1, depth + 1);
    quicksort(q + 1, r, depth + 1);
}
```

Kode 5.8 (Lanjutan) *Quicksort dengan pemartisian Lomuto, yang menarasikan rekursinya. Indentasinya menunjukkan kedalaman pemanggilannya.*

```
int main() {
    A = {2, 8, 7, 1, 3, 5, 6, 4};
    cout << "input: ";
    for (int x : A) cout << x << " ";
    cout << "\n\n";
    quicksort(0, (int)A.size() - 1, 0);
    cout << "\nsorted: ";
    for (int x : A) cout << x << " ";
    cout << "\npartition calls: " << calls << "\n";
    return 0;
}
```

Keluarannya:

```
input: 2 8 7 1 3 5 6 4

  QUICKSORT(0,7): pivot = A[7] = 4
    [2 1 3 4 7 5 6 8]  after PARTITION, pivot 4 is final at index 3
    QUICKSORT(0,2): pivot = A[2] = 3
      [2 1 3]4 7 5 6 8   after PARTITION, pivot 3 is final at index 2
      QUICKSORT(0,1): pivot = A[1] = 1
        [1 2]3 4 7 5 6 8   after PARTITION, pivot 1 is final at index 0
        QUICKSORT(1,1): one element, return
    QUICKSORT(4,7): pivot = A[7] = 8
      1 2 3 4 [7 5 6 8]  after PARTITION, pivot 8 is final at index 7
      QUICKSORT(4,6): pivot = A[6] = 6
        1 2 3 4 [5 6 7]8   after PARTITION, pivot 6 is final at index 5
        QUICKSORT(4,4): one element, return
        QUICKSORT(6,6): one element, return

sorted: 1 2 3 4 5 6 7 8
partition calls: 5
```

Bacalah sebagai sebuah pohon. Pemanggilan akarnya memartisi di sekitar 4, menghasilkan potongan kiri [2,1,3] dan potongan kanan [7,5,6,8]. Potongan kirinya memartisi di sekitar 3, lalu di sekitar 1. Potongan kanannya memartisi di sekitar 8 — pilihan yang sial, sebab 8 adalah elemen terbesar di sana dan sub-potongan kanannya kosong — lalu di sekitar 6.

Perhatikan sesuatu yang dibuat gamblang oleh penelusuran itu: **sebuah pivot tidak pernah bergerak lagi.** Begitu `PARTITION` kembali, $A[q]$ memuat sebuah nilai yang $\geq$ segala yang di kirinya dan $\leq$ segala yang di kanannya, dan kedua pemanggilan rekursifnya tetap benar-benar berada di satu sisinya. Lima pemartisian menempatkan lima elemen secara permanen; tiga sisanya rontok sebagai kasus basis satu elemen. Itulah invarian yang membuat Quicksort benar, dan ia tidak memerlukan pemaduan untuk menegakkannya.

5.9 Menganalisis Quicksort: Kasus Terbaik, Terburuk, dan Rata-Rata

5.9.1 Segalanya bergantung pada belahannya

`PARTITION` selalu berongkos $\Theta(n)$ pada subarray berisi n elemen — satu lintasan, satu pembandingan per elemen. Jadi waktu jalan Quicksort sepenuhnya ditentukan oleh *bagaimana lariknya terbagi*. Tuliskan rekurensi umumnya: kalau pemartisiannya menaruh k elemen di kiri, maka

$$T(n) = T(k) + T(n-k-1) + \Theta(n), \tag{5.10}$$

dengan -1 itu memperhitungkan *pivot*-nya, yang tidak masuk ke satu pun pemanggilan rekursifnya. Segala isi subbab ini adalah Persamaan (5.10) dengan k yang berbeda-beda.

5.9.2 Kasus terbaik: belahan yang seimbang sempurna

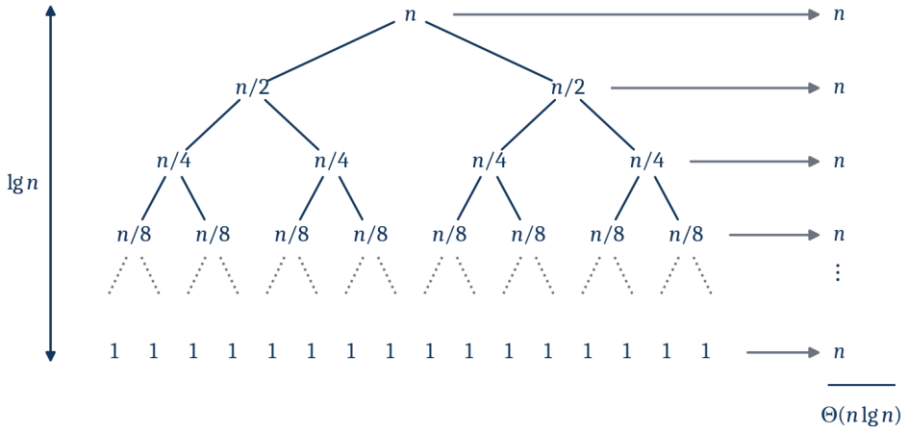
Andaikan *pivot*-nya selalu median, sehingga setiap pemartisian membelah subarray-nya menjadi dua paruh berukuran $\lfloor n/2 \rfloor$ dan $\lceil n/2 \rceil - 1$. Dengan mengabaikan lantai dan langit-langitnya, sebagaimana ditunjukkan Bab 3 boleh kita lakukan,

$$T(n) = 2T(n/2) + \Theta(n). \tag{5.11}$$

Itu persis rekurensi Merge Sort. Terapkan Teorema Master pada Persamaan (5.11): $a = 2$, $b = 2$, sehingga $n^{\log_b a} = n^{\log_2 2} = n^1 = n$, dan $f(n) = \Theta(n) = \Theta(n^{\log_b a})$. Ini Kasus 2 dengan $k = 0$, memberikan

$$T(n) = \Theta(n\lg n).$$

Gambar 5.18 adalah pohon rekursinya, yang menunjukkan hasil yang sama tanpa teorema apa pun.



Gambar 5.18 Pohon rekursi kasus terbaik bagi Quicksort. Ukuran submasalah tiap tingkat membelah dua — n, lalu $n/2$ dua kali, lalu $n/4$ empat kali — tetapi pekerjaan *total* pada tiap tingkatnya tetap n, sebab memartisi 2^d subarray berukuran $n/2^d$ berongkos $2^d \times n/2^d = n$. Ada $\lg n + 1$ tingkat, jadi totalnya $\Theta(n\lg n)$.

Bacalah **Gambar 5.18** dengan cermat, sebab gambar yang sama menjelaskan setiap kasus lain dalam subbab ini. Panah-panah di sebelah kanan tiap tingkat membawa biaya total tingkat itu. Setiap satu di antaranya berbunyi n. Tinggi pohonnya adalah $\lg n$ sebab ukurannya membelah dua. Kalikan: $n \times \lg n$.

5.9.3 Kasus terburuk: belahan yang setakseimbang mungkin

Sekarang andaikan *pivot*-nya selalu elemen terkecil (atau selalu terbesar) dalam subarray-nya. Maka satu sisi mendapat 0 elemen dan sisi lainnya mendapat $n - 1$. Memasukkan $k = 0$ ke dalam rekurensi umumnya,

$$T(n) = T(0) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n), \tag{5.12}$$

sebab $T(0) = \Theta(1)$ terserap. **Gambar 5.19** menunjukkan penurunan bentuk tertutupnya secara ringkas.

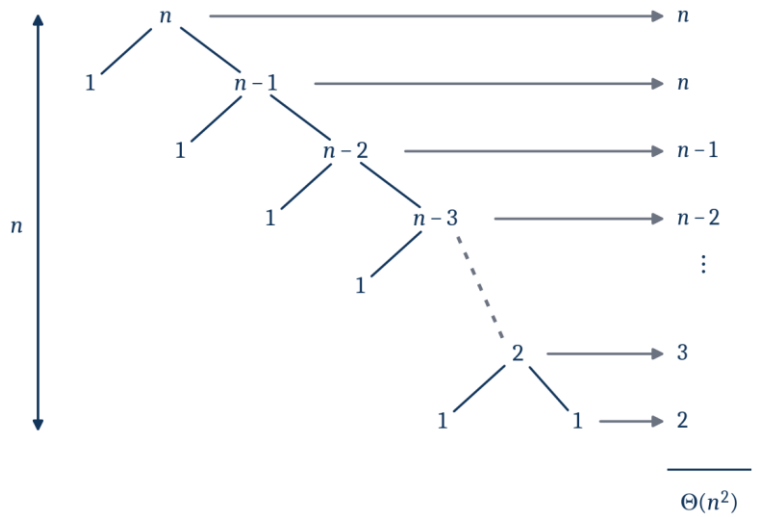
$$
\begin{aligned}
T(n) = T(1) + T(n-1) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \sum_{k=1}^{n} \Theta(k) \\
&= \Theta\left(\sum_{k=1}^{n} k\right) \\
&= \Theta(n^2)
\end{aligned}
$$

Gambar 5.19 Rekurensi kasus terburuknya dibuka gulungannya: $T(n) = T(1) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \Theta(n^2)$.

Mari kita buka gulungan Persamaan (5.12) secara terang-terangan, satu baris demi satu baris, sebab pekerjaannya pendek dan sepenuhnya meyakinkan:

$$
\begin{aligned}
T(n) &= T(n-1) + cn \\
&= [T(n-2) + c(n-1)] + cn \\
&= [T(n-3) + c(n-2)] + c(n-1) + cn \\
&\vdots \\
&= T(0) + c(1 + 2 + 3 + \cdots + (n-1) + n) \\
&= \Theta(1) + c \cdot \frac{n(n+1)}{2} \\
&= \Theta(n^2),
\end{aligned}
$$

dengan memakai deret aritmetika $\sum_{k=1}^{n} k = n(n+1)/2$ dari Bab 3. **Gambar 5.20** menggambar perhitungan yang sama sebagai pohon rekursi.



Gambar 5.20 Pohon rekursi kasus terburuknya. Alih-alih bercabang, pohonnya merosot menjadi satu rantai tunggal sepanjang n — setiap simpul memperanakkan satu anak berukuran satu lebih kecil dan satu anak berukuran 0. Biaya tingkat-tingkatnya adalah $n, n, n-1, n-2, \ldots, 2$, dan menjumlahkannya memberikan $\Theta(n^2)$.

Pohon di **Gambar 5.20** adalah definisi visual kasus terburuknya: **sebuah pohon yang bukan pohon.** Kedalamannya n alih-alih lg$n$, dan Quicksort sudah merosot menjadi sesuatu yang sangat mirip Selection Sort.

Kapan ini sungguh terjadi? Dengan pemartisian Lomuto dan elemen terakhir sebagai *pivot*, ia terjadi ketika lariknya sudah terurut, atau terurut terbalik. Pada larik yang sudah terurut, $A[r]$ adalah elemen terbesar, sehingga setiap elemen lolos pengujian $A[j] \leq x$, setiap elemen bergabung ke wilayah kiri, dan pemartisiannya mengembalikan $q = r$ — belahan yang ekstrem, setiap kali.

> **IRONINYA.** Masukan yang sudah terurut adalah kasus **terbaik** bagi Insertion Sort, $\Theta(n)$. Ia adalah kasus **terburuk** bagi Quicksort yang naif, $\Theta(n^2)$. Dan masukan terurut atau hampir terurut sama sekali tidak eksotis: itulah yang Anda peroleh dari basis data yang mengembalikan baris dalam urutan indeks, dari sebuah berkas log, dari sebuah daftar tanggal, atau dari larik yang sudah pernah diurutkan orang lain. Jangan pernah menggelar Quicksort ber-*pivot* tetap pada data yang tidak Anda bangkitkan sendiri. Subbab 5.10 memberikan perbaikannya.

Sebuah batas atas, dibuktikan dengan substitusi. Pohon rekursinya menunjukkan bahwa kasus terburuknya sekurang-kurangnya $\Theta(n^2)$; supaya ketat, kita juga harus membuktikan bahwa ia tidak pernah lebih buruk daripada itu. Ambil rekurensi kasus terburuknya yang penuh, dengan memaksimumkan atas semua belahan yang mungkin:

$$T(n) = \max_{0 \leq k \leq n-1} (T(k) + T(n-k-1)) + \Theta(n).$$

Tebak $T(n) \leq cn^2$ untuk suatu konstanta $c > 0$, lalu substitusikan:

$$\begin{aligned} T(n) &\leq \max_{0 \leq k \leq n-1} (ck^2 + c(n-k-1)^2) + dn \\ &= c \cdot \max_{0 \leq k \leq n-1} (k^2 + (n-k-1)^2) + dn. \end{aligned}$$

Ungkapan $k^2 + (n-k-1)^2$ adalah parabola yang terbuka ke atas dalam k, sehingga pada selang $[0, n-1]$ maksimumnya berada di salah satu titik ujungnya. Di $k = 0$ nilainya $(n-1)^2$; di $k = n-1$ nilainya juga $(n-1)^2$. Jadi

$$T(n) \le c(n-1)^2 + dn = cn^2 - 2cn + c + dn = cn^2 - c(2n-1) + dn \le cn^2,$$

asalkan $c(2n-1) \ge dn$, yang berlaku untuk semua $n \ge 1$ begitu kita memilih $c \ge d$. Jadi $T(n) = O(n^2)$, dan digabungkan dengan $\Omega(n^2)$ dari pohonnya, kita peroleh $T(n) = \Theta(n^2)$ pada kasus terburuk. ■

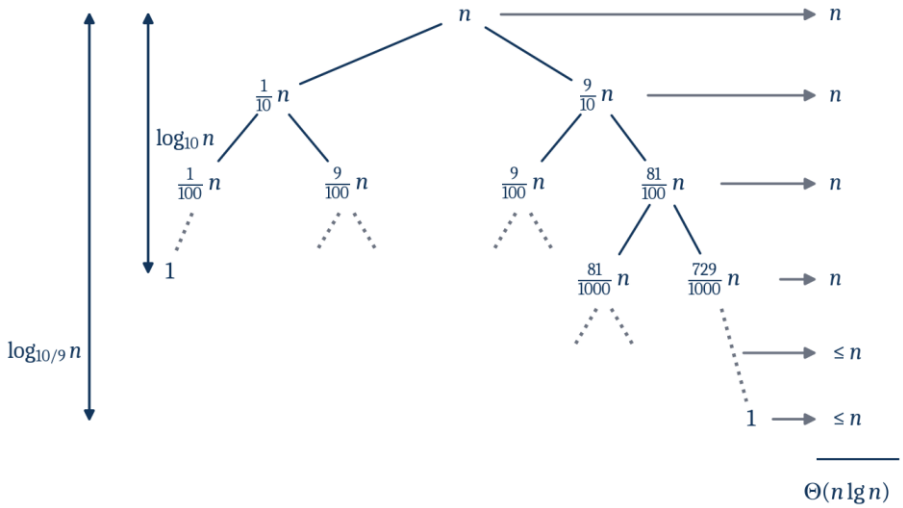
5.9.4 Gagasan belahan yang seimbang, dan belahan 9: 1

Inilah fakta yang menyelamatkan Quicksort, dan ia sungguh-sungguh mengejutkan pada perjumpaan pertama.

Andaikan pemartisiannya *selalu* berat sebelah, tetapi berat sebelah menurut nisbah yang tetap — katakanlah ia selalu menaruh 10% elemennya di satu sisi dan 90% di sisi lain. Itu kedengarannya buruk. Rekurensinya adalah

$$T(n) = T(n/10) + T(9n/10) + \Theta(n). \tag{5.13}$$

Gambar 5.21 adalah pohon rekursi bagi Persamaan (5.13).



Gambar 5.21 Pohon rekursi bagi belahan tetap 1: 9. Tulang punggung kirinya, yang ukurannya menyusut sepersepuluh setiap tingkat, menyentuh dasar sesudah $\log_{10} n$ tingkat. Tulang punggung kanannya, yang ukurannya hanya menyusut dengan faktor 10/9, bertahan sampai kedalaman $\log_{10/9} n$. Di antara kedua kedalaman itu, beberapa tingkat tidak lengkap dan berongkos kurang dari n; di atas kedalaman yang lebih dangkal, setiap tingkat berongkos tepat n. Totalnya karena itu berada di antara $n\log_{10} n$ dan $n\log_{10/9} n$, dan keduanya $\Theta(n\lg n)$.

Kerjakan **Gambar 5.21** simpul demi simpul. Akarnya berukuran n dan berongkos n. Anak-anaknya berukuran $n/10$ dan $9n/10$; bersama-sama mereka berongkos $n/10 + 9n/10 = n$. Anak-anak mereka berukuran $n/100$, $9n/100$, $9n/100$, dan $81n/100$; bersama-sama $\frac{1+9+9+81}{100} n = n$. **Setiap tingkat yang lengkap berongkos tepat** n, sebab pemartisian tidak pernah kehilangan satu elemen pun.

Jadi satu-satunya pertanyaannya adalah: ada berapa tingkat? Dua jawaban, sebab pohonnya berat sebelah.

- Menyusuri lintasan *terpendek* — selalu ke sisi yang kecil — ukurannya adalah $n, n/10, n/100, \ldots$, mencapai 1 sesudah $\log_{10} n$ langkah.
- Menyusuri lintasan *terpanjang* — selalu ke sisi yang besar — ukurannya adalah $n, \frac{9}{10}n, \left(\frac{9}{10}\right)^2 n, \ldots$, mencapai 1 ketika $\left(\frac{9}{10}\right)^d n = 1$, yaitu ketika $d = \log_{10/9} n$.

Setiap tingkat di atas kedalaman $\log_{10} n$ lengkap dan berongkos tepat n; setiap tingkat di bawahnya berongkos paling banyak n. Jadi

$$n\log_{10} n \;\leq\; T(n) \;\leq\; n\log_{10/9} n + \Theta(n).$$

Sekarang ubah kedua logaritmanya ke basis 2 dengan memakai $\log_b n = \lg n / \lg b$:

$$\log_{10} n = \frac{\lg n}{\lg 10} \approx \frac{\lg n}{3.32} \approx 0.30 \lg n, \qquad \log_{10/9} n = \frac{\lg n}{\lg(10/9)} \approx \frac{\lg n}{0.152} \approx 6.58 \lg n.$$

Jadi $T(n)$ terletak di antara kira-kira $0.30\, n\lg n$ dan kira-kira $6.58\, n\lg n$. Kedua ujungnya adalah $\Theta(n\lg n)$. Keberatsebelahan itu berongkos bagi kita **sebuah faktor konstanta sebesar kira-kira dua puluh dua** — dan tidak lebih.

> **FAKTA YANG MENENTUKAN. Sembarang** belahan dengan nisbah yang tetap memberikan $\Theta(n\lg n)$. Lima puluh–lima puluh, sembilan puluh–sepuluh, sembilan puluh sembilan berbanding satu — tidak ada bedanya bagi jawaban asimtotiknya, hanya bagi konstantanya. Pohon rekursinya berkedalaman $\log_{1/\alpha} n = \Theta(\lg n)$ untuk pecahan tetap α mana pun, dan tiap tingkatnya berongkos n. **Hanya belahan yang nisbahnya tidak tetap — belahan merosot** $1:(n-1)$**, dengan sisi kecilnya berisi sejumlah elemen yang tetap alih-alih sebuah pecahan yang tetap — yang menghasilkan** $\Theta(n^2)$**.** Itulah sebabnya Quicksort begitu sukar dibuat lambat secara tak sengaja: Anda harus sial pada hakikatnya di setiap pemartisian, bukan sekadar pada sebagian besarnya.

Mari kita konkretkan dengan angka. Untuk $n = 1{,}000{,}000$:

- Belahan 99: 1 setiap kali: kedalaman $\log_{100/99} 10^6 = \lg 10^6 / \lg(100/99) \approx 19.93/0.0145 \approx 1{,}375$ tingkat. Pekerjaan total kira-kira 1.4×10^9 operasi — satu atau dua detik.
- Belahan merosot $1:(n-1)$ setiap kali: kedalaman 10^6, pekerjaan total 5×10^{11} operasi — kira-kira delapan menit.

Sebuah faktor lebih dari tiga ratus memisahkan "belahan yang buruk secara konsisten" dari "belahan yang merosot". Jurang itulah batas keselamatan Quicksort.

5.9.5 Kasus rata-rata, dihujahkan dengan dua cara

Kasus terbaiknya $\Theta(n\lg n)$ dan kasus terburuknya $\Theta(n^2)$. Mana yang kita peroleh dalam praktik? Kasus rata-ratanya — dan kasus itu sangat dekat dengan kasus terbaiknya.

Hujah pertama: belahan yang buruk diserap oleh belahan baik yang menyusulnya. Inilah hujah yang perlu Anda pegang di kepala. Andaikan nasibnya berselang-seling dengan sempurna: satu tingkat membelah tepat di tengah (mujur), tingkat berikutnya membelah 1: $(n-1)$ (sial), tingkat berikutnya mujur lagi, dan seterusnya. Tuliskan $L(n)$ untuk ongkosnya ketika tingkat teratasnya mujur dan $U(n)$ untuk ongkosnya ketika tingkat teratasnya sial. **Gambar 5.22** menjabarkan penurunannya.

$$L(n) = 2U(n/2) + \Theta(n) \qquad \text{beruntung}$$

$$U(n) = L(n-1) + \Theta(n) \qquad \text{sial}$$

maka diperoleh

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

Gambar 5.22 Analisis nasib berselang-seling. Dengan $L(n) = 2U(n/2) + \Theta(n)$ untuk tingkat yang mujur dan $U(n) = L(n-1) + \Theta(n)$ untuk tingkat yang sial, mensubstitusikan yang kedua ke dalam yang pertama memberikan $L(n) = 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) = 2L(n/2-1) + \Theta(n) = \Theta(n\lg n)$.

Kedua rekurensinya adalah

$$L(n) = 2\,U(n/2) + \Theta(n) \qquad \text{(mujur: dua paruh, masing-masing lalu sial)}$$

$$= L(n-1) + \Theta(n) \qquad \text{(sial: satu sisi kosong, sisi lainnya lalu mujur)}$$

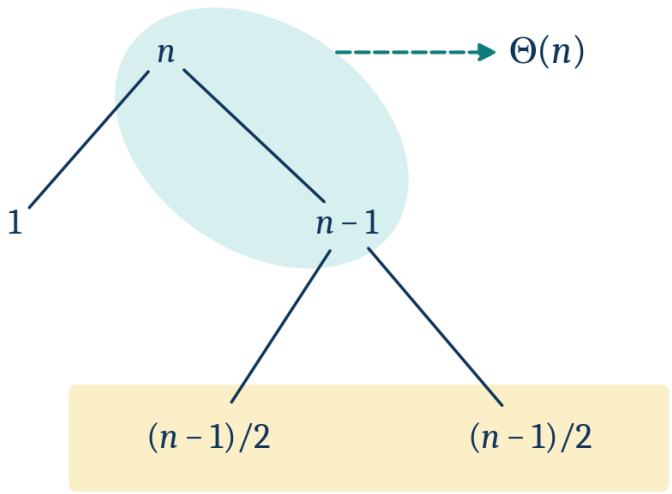
$U(n)$

Substitusikan yang kedua ke dalam yang pertama:

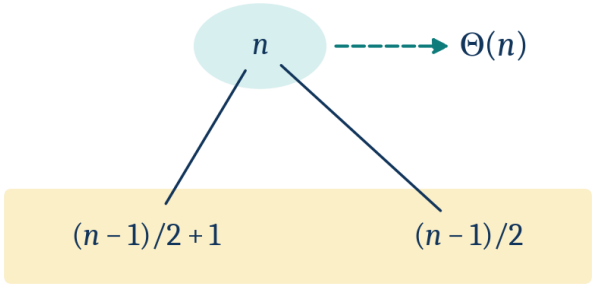
$$\begin{aligned} L(n) &= 2\,U(n/2) + \Theta(n) \\ &= 2[\,L(n/2-1) + \Theta(n/2)\,] + \Theta(n) \\ &= 2\,L(n/2-1) + \Theta(n) + \Theta(n) \\ &= 2\,L(n/2-1) + \Theta(n). \end{aligned}$$

Baris terakhirnya lagi-lagi rekurensi Merge Sort — angka -1 di dalamnya adalah gangguan bersuku rendah yang menurut Bab 3 boleh kita abaikan — sehingga $L(n) = \Theta(n\lg n)$.

Gambar 5.23 dan **Gambar 5.24** menaruh pokok yang sama dalam gambar, dan keduanya layak ditelaah berdampingan.



Gambar 5.23 Sebuah belahan sial disusul sebuah belahan mujur. Pemartisian sial atas n elemen menghasilkan potongan berukuran 1 dan potongan berukuran $n-1$; pemartisian mujur atas potongan itu menghasilkan dua potongan berukuran $(n-1)/2$. Daerah yang diarsir menunjukkan biaya gabungannya, $\Theta(n) + \Theta(n-1) = \Theta(n)$.



Gambar 5.24 Sebuah belahan mujur tunggal, sebagai pembanding: satu pemartisian atas n elemen, berongkos $\Theta(n)$, menghasilkan dua potongan berukuran kira-kira $(n-1)/2$.

Bandingkan **Gambar 5.23** dengan **Gambar 5.24**. Keduanya berakhir di tempat yang sama — dua submasalah berukuran kira-kira $(n-1)/2$ — dan keduanya berongkos sama hingga sebuah faktor konstanta: dua tingkat pekerjaan $\Theta(n)$ alih-alih satu. Jadi **sebuah belahan yang sial tidak melipatduakan waktu jalannya; ia semata-mata menambahkan satu tingkat tambahan.** Biaya nasib buruk adalah sebuah faktor konstanta, bukan perubahan laju pertumbuhan. Itulah arti "diserap".

Hujah kedua: banyaknya pembandingan yang diharapkan, dicacah secara eksak. Hujah berselang-seling tadi adalah gagasan. Berikut ini teorema yang sesungguhnya, dan ia salah satu hujah tercantik di seluruh analisis algoritma.

Namai ulang elemen-elemen lariknya $z_1, z_2, \ldots, z_n$ dalam urutan *terurut*, sehingga z_1 yang terkecil dan z_n yang terbesar. Tuliskan $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ untuk himpunan elemen yang posisi terurutnya terletak di antara i dan j, keduanya termasuk.

Sekarang buatlah satu pengamatan yang mengerjakan seluruh pekerjaannya:

> **PENGAMATAN KUNCINYA.** Dua elemen z_i dan z_j dibandingkan **paling banyak sekali** dalam seluruh jalannya Quicksort. Keduanya dibandingkan hanya ketika salah satunya menjadi *pivot*, dan sebuah *pivot* dibandingkan dengan segala yang ada di subarray-nya lalu dikeluarkan dari rekursinya untuk selamanya.

Jadi total banyaknya pembandingan semata-mata adalah banyaknya pasangan yang pernah dibandingkan. Definisikan peubah acak indikator

$$X_{ij} = \begin{cases} 1 & \text{jika } z_i \text{ dibandingkan dengan } z_j \text{ pada suatu saat,} \\ 0 & \text{jika tidak,} \end{cases}$$

sehingga cacah pembandingan totalnya adalah $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$, dan menurut kelinearan nilai harapan

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ dibandingkan dengan } z_j\}.$$

Kapan z_i dan z_j dibandingkan? Ikuti rekursinya. Selama belum ada *pivot* yang dipilih dari Z_{ij}, seluruh Z_{ij} tinggal bersama-sama dalam subarray yang sama — sebab sembarang *pivot* di luar Z_{ij} entah lebih kecil daripada setiap elemen Z_{ij} atau lebih besar daripada setiap elemennya, sehingga mengirim seluruh himpunan itu ke satu sisi. Pada saat suatu elemen Z_{ij} terpilih sebagai *pivot*, salah satu dari tiga hal terjadi:

- *pivot*-nya adalah z_i — maka z_i dibandingkan dengan segala yang ada di subarray-nya, termasuk z_j. **Dibandingkan.**
- *pivot*-nya adalah z_j — sama saja, menurut kesetangkupan. **Dibandingkan.**
- *pivot*-nya adalah suatu z_k dengan $i < k < j$ — maka z_i pergi ke potongan kiri, z_j pergi ke potongan kanan, dan keduanya tidak pernah berada dalam subarray yang sama lagi. **Tidak pernah dibandingkan.**

Jadi z_i dan z_j dibandingkan **tepat ketika pivot pertama yang ditarik dari Z_{ij} adalah z_i atau z_j.** Himpunan Z_{ij} beranggotakan $j - i + 1$ elemen, dan — dengan *pivot* yang dipilih secara acak, yang menjadi pokok Subbab 5.10 — masing-masing sama mungkinnya menjadi yang pertama terpilih. Karena itu

$$\Pr\{z_i \text{ dibandingkan dengan } z_j\} = \frac{2}{j - i + 1}. \tag{5.14}$$

Mensubstitusikan Persamaan (5.14) ke dalam jumlahan gandanya, dan menuliskan $k = j - i$:

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = 2 \sum_{i=1}^{n-1} H_n,$$

dengan $H_n = \sum_{k=1}^{n} 1/k$ adalah bilangan harmonik dari Bab 3. Jadi

$$\mathbb{E}[X] < 2(n-1)H_n = O(n\lg n),$$

sebab $H_n = \ln n + O(1)$ dan $\ln n = \lg n \cdot \ln 2 \approx 0.693 \lg n$. Perhitungan jumlahan gandanya yang lebih cermat, dengan mempertahankan suku-suku bersuku rendahnya, memberikan jawaban eksaknya [8]

$$\mathbb{E}[X] = 2(n+1)H_n - 4n \approx 2n\ln n - 2.85n \approx 1.386\, n\lg n. \tag{5.15}$$

APA ARTI $1.386\,n\lg n$**.** Pengurutan berbasis pembandingan yang terbaik yang mungkin memerlukan kira-kira $n\lg n$ pembandingan (Subbab 5.11). Quicksort teracak rata-rata memakai kira-kira 1.39 kali angka itu. Merge Sort memakai kira-kira $n\lg n$. Jadi Quicksort melakukan kira-kira **tiga puluh sembilan persen lebih banyak pembandingan** daripada Merge Sort — dan masih lebih cepat dalam praktik, sebab satu pembandingan itu murah sedangkan satu *cache miss* tidak. Subbab 5.12.3 mengukurnya.

5.9.6 Memeriksa teorinya terhadap sebuah mesin

Kode 5.9 mencacah pembandingan yang sungguh dilakukan Quicksort pada tiga macam masukan, dan membandingkannya terhadap dua rumus yang sudah kita turunkan: $n(n-1)/2$ untuk kasus terburuknya dan $2(n+1)H_n - 4n$, dari Persamaan (5.15), untuk kasus rata-ratanya.

Kode 5.9 Mencacah pembandingan Quicksort pada masukan terurut dan teracak, dengan dan tanpa *pivot* acak, terhadap kedua rumus teoretisnya.

```
// Berapa pembandingan yang sungguh dilakukan quicksort? Pivot tetap lawan pivot acak.
#include <iostream>
#include <iomanip>
#include <vector>
#include <numeric>
#include <algorithm>
#include <random>
#include <cmath>
using namespace std;

long long comparisons;
mt19937 rng(20260919);

int partition(vector<int>& A, int p, int r) {
    int x = A[r], i = p - 1;
    for (int j = p; j <= r - 1; ++j) {
        ++comparisons;                          // inilah PEMBANDINGAN milik algoritmanya
        if (A[j] <= x) swap(A[++i], A[j]);
    }
    swap(A[i + 1], A[r]);
    return i + 1;
}

int randomizedPartition(vector<int>& A, int p, int r) {
    uniform_int_distribution<int> pick(p, r);
    swap(A[r], A[pick(rng)]);                   // sebuah elemen acak seragam menjadi
pivotnya
    return partition(A, p, r);
}

void quicksort(vector<int>& A, int p, int r, bool randomized) {
    if (p >= r) return;
    int q = randomized ? randomizedPartition(A, p, r) : partition(A, p, r);
    quicksort(A, p, q - 1, randomized);
    quicksort(A, q + 1, r, randomized);
}

int main() {
    cout << fixed << setprecision(2);
    cout << "        n |  sorted input, fixed pivot |  random input, fixed pivot |"
            "  sorted input, random pivot |   n(n-1)/2 |  2(n+1)H_n-4n\n";
    for (int n : {100, 500, 1000, 2000, 4000}) {
        vector<int> sorted(n);
        iota(sorted.begin(), sorted.end(), 1);   // 1, 2, 3, ..., n -- sudah terurut
```

Kode 5.9 (Lanjutan) *Mencacah pembandingan Quicksort pada masukan terurut dan teracak, dengan dan tanpa pivot acak, terhadap kedua rumus teoretisnya.*

```
        vector<int> shuffled = sorted;
        shuffle(shuffled.begin(), shuffled.end(), rng);

        vector<int> a = sorted;   comparisons = 0; quicksort(a, 0, n - 1, false);
        long long worst = comparisons;

        vector<int> b = shuffled; comparisons = 0; quicksort(b, 0, n - 1, false);
        long long avg = comparisons;

        vector<int> c = sorted;   comparisons = 0; quicksort(c, 0, n - 1, true);
        long long rnd = comparisons;

        double half = (double)n * (n - 1) / 2.0;
        double H = 0;                            // bilangan harmonik ke-n, dijumlahkan
langsung
        for (int k = 1; k <= n; ++k) H += 1.0 / k;
        double pred = 2.0 * (n + 1) * H - 4.0 * n;   // cacah pembandingan harapan yang
eksak
        cout << setw(9) << n << " | " << setw(26) << worst << " | " << setw(26) << avg
             << " | " << setw(27) << rnd << " | " << setw(10) << half
             << " | " << setw(12) << pred << "\n";
    }
    return 0;
}
```

Keluarannya:

```
        n |  sorted input, fixed pivot |  random input, fixed pivot |  sorted input, random
pivot |   n(n-1)/2 |  2(n+1)H_n-4n
      100 |                       4950 |                        623 |
634 |    4950.00 |       647.85
      500 |                     124750 |                       5146 |
5144 |  124750.00 |      4806.41
     1000 |                     499500 |                      10452 |
11176 |  499500.00 |     10985.91
     2000 |                    1999000 |                      22672 |
23687 | 1999000.00 |     24729.83
     4000 |                    7998000 |                      55683 |
57639 | 7998000.00 |     54988.87
```

Tiga hal yang perlu dibaca dari tabel itu.

Satu. Kolom "sorted input, fixed pivot" *tepat* sama dengan $n(n-1)/2$ — 4950 berbanding 4950.00, 499500 berbanding 499500.00, di setiap barisnya. Bukan kira-kira; tepat. Itulah kasus terburuknya, tercapai, dan ia membenarkan penurunan Subbab 5.9.3 sampai ke pembandingan yang terakhir. Setiap pasangan elemen dibandingkan, dan itu $\binom{n}{2} = n(n-1)/2$ pasangan.

Dua. Kedua kolom kasus rata-ratanya mengikuti rumus $2(n+1)H_n - 4n$ dengan rapat — 55,683 dan 57,639 berbanding ramalan 54,989 pada $n = 4000$, dalam jarak kira-kira lima persen. Keacakan berarti satu kali jalan tidak akan mendarat tepat pada nilai harapannya, tetapi ia mendarat di dekatnya.

Tiga, dan yang paling penting. Bandingkan kolom pertama dengan kolom ketiga pada $n = 4000$: 7,998,000 pembandingan berbanding 57,639. **Masukan yang sama, algoritma yang sama, satu baris berbeda** — dan ia berjalan seratus tiga puluh delapan kali lebih cepat. Satu baris itulah pokok subbab berikutnya.

5.10 Quicksort Teracak

5.10.1 Perbaikannya

Persoalan `PARTITION` sebagaimana tertulis adalah bahwa *pivot*-nya dipilih menurut *posisi* — selalu elemen terakhir. Seorang lawan yang mengetahui hal ini dapat menyodorkan kepada Anda sebuah masukan yang membuat setiap pemartisiannya merosot. Dan Anda bahkan tidak memerlukan lawan yang jahat: "sudah terurut" adalah masukan yang sepenuhnya biasa yang berakibat sama.

Perbaikannya adalah satu penukaran, dan **Pseudokode 5.12** menunjukkannya.

Pseudokode 5.12 `RANDOMIZED-PARTITION` dan `RANDOMIZED-QUICKSORT`: tukarkan sebuah elemen acak seragam ke slot *pivot*, lalu partisikan seperti biasa.

```
RANDOMIZED-PARTITION(A, p, r):
 1.  i ← RANDOM(p, r)                    // indeks acak seragam dalam [p, r]
 2.  swap A[r] ↔ A[i]
 3.  return PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r):
 1.  if p < r
 2.      q ← RANDOMIZED-PARTITION(A, p, r)
 3.      RANDOMIZED-QUICKSORT(A, p, q - 1)
 4.      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Baris 1 memilih sebuah indeks acak; baris 2 memindahkan elemen itu ke slot *pivot*-nya; baris 3 berjalan persis seperti sebelumnya. Dua baris tambahan, dan segalanya berubah.

5.10.2 Apa yang sesungguhnya dibeli oleh pengacakan

Penting untuk bersikap tepat di sini, sebab mahasiswa rutin mengklaim terlalu banyak.

> **APA YANG DIKERJAKAN DAN TIDAK DIKERJAKAN PENGACAKAN.** 1. **Ia membuat waktu jalannya tidak bergantung pada urutan masukan.** *Pivot*-nya tidak lagi ditentukan oleh di mana elemen-elemennya duduk, sehingga tidak ada susunan data yang secara hakiki buruk. Keacakannya datang dari lemparan koin algoritmanya sendiri, bukan dari andaian tentang masukannya. 2. **Tidak ada masukan tertentu yang memicu kasus terburuknya.** Tidak ada lagi persoalan "larik terurut", dan lawan yang dapat melihat data Anda tidak dapat menyusun kasus yang buruk, sebab lawan itu tidak dapat melihat bilangan-bilangan acak Anda. 3. **Kasus terburuknya masih dapat terjadi — karena nasib buruk.** Pengacakan tidak menghapus cabang $\Theta(n^2)$-nya; ia membuat cabang itu mustahil secara astronomis. 4. **Waktu jalan harapannya adalah $O(n\lg n)$, dengan kira-kira $1.39\,n\lg n$ pembandingan** — hasil yang diturunkan di Subbab 5.9.5, yang memang mengandaikan *pivot* acak seragam semacam inilah.

Butir 3 pantas mendapat sebuah angka. Supaya setiap pemartisiannya merosot, *pivot* yang terpilih secara acak harus merupakan minimum atau maksimum subarray-nya, setiap kali tanpa kecuali. Pada tingkat teratas, peluangnya $2/n$; pada tingkat berikutnya $2/(n-1)$; dan seterusnya. Peluang bencana penuhnya kira-kira

$$\frac{2}{n}\cdot\frac{2}{n-1}\cdots\frac{2}{2}=\frac{2^{n-1}}{n!}. \tag{5.16}$$

Untuk $n = 100$ itu kira-kira 10^{-128}. Sebagai pembanding, banyaknya atom di alam semesta teramati kira-kira 10^{80}. Memilih satu atom tertentu dari seluruh alam semesta teramati secara membabi buta masih lebih mungkin daripada bencana ini, dengan faktor kira-kira 10^{48}.

POKOK YANG LEBIH DALAM. Sebelum diacak, waktu jalan Quicksort merupakan fungsi dari masukannya: sebagian masukan cepat, sebagian lambat, dan mana yang mana merupakan sifat tetap yang dapat ditemukan orang. Sesudah diacak, waktu jalannya menjadi fungsi dari lemparan koin algoritmanya sendiri, dan **setiap** masukan mempunyai sebaran waktu jalan yang sama. Kita tidak membuat algoritmanya lebih cepat; kita membuatnya *dapat diramalkan*. Dalam sistem produksi, beda itu lebih berharga daripada kecepatan.

5.10.3 Strategi pivot yang lain

Pengacakan bukan satu-satunya pertahanan, dan pustaka sungguhan memadukan beberapa di antaranya.

- **Median dari tiga.** Ambil median dari $A[p]$, $A[\lfloor (p+r)/2 \rfloor]$, dan $A[r]$ lalu pakai itu sebagai *pivot*-nya. Ia berongkos tiga pembandingan dan mengalahkan masukan terurut maupun terurut terbalik secara langsung, sebab elemen tengah larik terurut adalah mediannya. Ia tidak kebal terhadap masukan lawan yang dirancang dengan sengaja, tetapi ia sangat murah dan sangat mujarab pada data sungguhan.
- **Dwi-pivot (Yaroslavskiy, 2009).** Pilih dua *pivot* $x \leq y$ dan partisikan menjadi tiga wilayah: $<x$, di antara x dan y, dan $>y$. Lebih sedikit lintasan atas datanya untuk jumlah pengurutan yang sama. Inilah yang dipakai `Arrays.sort` milik Java untuk tipe primitif; dokumentasi JDK 25 menyebutnya menawarkan kinerja $O(n \lg n)$ pada semua data [9].
- **Introsort (Musser, 1997)** [10]**.** Mulailah dengan Quicksort. Lacak kedalaman rekursinya; kalau melampaui kira-kira $2 \lg n$ — pertanda bahwa belahannya sedang berjalan buruk — alihkan submasalah itu ke Heapsort, yang tidak dapat dibuat lambat. Berhenti pula berekursi pada subarray yang lebih kecil daripada kira-kira enam belas elemen dan selesaikan dengan satu lintasan Insertion Sort atas seluruh lariknya. Inilah yang dikerjakan `std::sort` milik C++, dan inilah algoritma yang paling penting secara praktis dalam bab ini: ia punya kecepatan Quicksort, jaminan Heapsort, dan overhead rendah Insertion Sort pada masukan kecil, semuanya sekaligus.
- **Pemartisian tiga arah** — persoalan *Dutch national flag* (bendera kebangsaan Belanda) [11]. Kalau datanya banyak memuat kunci kembar, partisikan menjadi $<x$, $=x$, $>x$ dan berekursilah hanya pada dua wilayah terluarnya. Larik sejuta elemen yang hanya punya sepuluh nilai berbeda terurut dalam waktu $\Theta(n)$ dengan cara ini.

DALAM LOMBA, DAN DALAM PRODUKSI. Pakailah pengurutan bawaan bahasanya. `std::sort` milik C++ adalah Introsort; Java memakai Quicksort dwi-*pivot* untuk tipe primitif dan TimSort untuk objek; Python memakai Timsort, yang aturan penggabungan *run*-nya sejak Python 3.11 diganti dengan kebijakan Powersort dari Munro dan Wild yang hampir optimal [12]. Semuanya lebih cepat daripada apa pun yang akan Anda tulis dalam waktu yang tersedia, dan semuanya benar [4], [5]. Pelajari analisis dalam bab ini karena ia memberitahu Anda *apa yang sedang dikerjakan pustaka itu* dan *kapan ia akan lambat* — bukan supaya Anda dapat mengimplementasikannya ulang di bawah tekanan waktu.

5.10.4 Rekursi ekor dan tumpukan panggilan

Ada satu bahaya praktis lagi, dan bahaya itu sama sekali bukan tentang waktu: ia tentang memori.

Setiap pemanggilan rekursif memakan sebuah **bingkai tumpukan** (*stack frame*) — ruang untuk parameternya, alamat kembalinya, dan peubah lokalnya. Kedalaman rekursi Quicksort adalah tinggi pohon rekursinya. Pada kasus terbaiknya itu $\Theta(\lg n)$, segenggam bingkai. Pada kasus terburuknya itu $\Theta(n)$ — dan sejuta pemanggilan bersarang akan menghabiskan tumpukan panggilannya lalu merusak programnya, jauh sebelum waktu $\Theta(n^2)$-nya menjadi soal.

Perbaikannya memakai sebuah sifat baris 4 `QUICKSORT` pada **Pseudokode 5.11**: baris itu merupakan **pemanggilan ekor** (*tail call*), yaitu hal terakhir yang dikerjakan prosedurnya. Pemanggilan ekor tidak memerlukan bingkai tumpukan baru — alih-alih berekursi, kita cukup memperbarui parameternya dan menggelung. **Pseudokode 5.13** mengerjakan persis hal itu.

Pseudokode 5.13 `QUICKSORT-TAIL`: berekursi hanya pada potongan yang lebih kecil dan menggelung pada yang lebih besar, sehingga kedalaman tumpukannya tetap $O(\lg n)$.

```
QUICKSORT-TAIL(A, p, r):
 1.  while p < r
 2.      q ← PARTITION(A, p, r)
 3.      if (q - p) < (r - q)                    // potongan KIRI yang lebih kecil
 4.          QUICKSORT-TAIL(A, p, q - 1)         //   berekursi padanya ...
 5.          p ← q + 1                           //   ... lalu menggelung ke kanan
 6.      else                                    // potongan KANAN yang lebih kecil
 7.          QUICKSORT-TAIL(A, q + 1, r)         //   berekursi padanya ...
 8.          r ← q - 1                           //   ... lalu menggelung ke kiri
```

Dua perubahan, dan keduanya diperlukan:

1. **Hanya satu di antara dua pemanggilannya yang merupakan rekursi sungguhan**; yang lain menjadi satu iterasi gelung. Itu saja sudah memangkas separuh banyaknya bingkai.
2. **Rekursinya selalu pada potongan yang *lebih kecil*.** Inilah perubahan yang menentukan. Setiap pemanggilan rekursif yang sungguhan sekurang-kurangnya membelahdua ukuran submasalahnya — sebab yang lebih kecil di antara dua potongan paling banyak separuh jumlah keduanya — sehingga kedalamannya tidak dapat melampaui $\lg n$, seberapa pun buruknya pemartisiannya berjalan.

TEOREMA. `QUICKSORT-TAIL` memakai $O(\lg n)$ ruang tumpukan pada kasus terburuk. **Bukti.** Setiap pemanggilan rekursif dilakukan pada subarray yang ukurannya paling banyak separuh ukuran pemanggilnya, sebab ia merupakan yang lebih kecil di antara dua potongan yang jumlah ukurannya paling banyak sebesar ukuran pemanggilnya dikurangi satu. Mulai dari n, dengan membelahdua pada setiap tingkat rekursi yang sungguhan, kedalaman d memenuhi $n/2^d \geq 1$, sehingga $d \leq \lg n$. Setiap bingkainya $O(1)$, jadi totalnya $O(\lg n)$. ■

Kode 5.10 mengukur bedanya.

Kode 5.10 Mengukur kedalaman rekursi Quicksort naif dan Quicksort rekursi-ekor pada masukan terurut dan teracak.

```
// Kedalaman rekursi: quicksort naif melawan versi rekursi-ekor yang selalu
// berekursi pada sisi yang LEBIH KECIL lebih dulu.
#include <iostream>
#include <iomanip>
#include <vector>
#include <numeric>
#include <algorithm>
#include <random>
using namespace std;

int depthNow, depthMax;

struct DepthGuard {                             // mencacah seberapa dalam tumpukan
panggilan C++ turun
    DepthGuard()  { if (++depthNow > depthMax) depthMax = depthNow; }
    ~DepthGuard() { --depthNow; }
};

int partition(vector<int>& A, int p, int r) {
    int x = A[r], i = p - 1;
    for (int j = p; j <= r - 1; ++j) if (A[j] <= x) swap(A[++i], A[j]);
    swap(A[i + 1], A[r]);
    return i + 1;
}

void quicksortNaive(vector<int>& A, int p, int r) {
    DepthGuard g;
    if (p >= r) return;
    int q = partition(A, p, r);
    quicksortNaive(A, p, q - 1);
    quicksortNaive(A, q + 1, r);
}

void quicksortTail(vector<int>& A, int p, int r) {
    DepthGuard g;
    while (p < r) {
        int q = partition(A, p, r);
        if (q - p < r - q) {                    // sisi KIRI lebih kecil: berekursilah
padanya
            quicksortTail(A, p, q - 1);
            p = q + 1;                          // lalu menggelunglah pada sisi kanannya
        } else {                                // sisi KANAN lebih kecil: berekursilah
padanya
            quicksortTail(A, q + 1, r);
            r = q - 1;                          // lalu menggelunglah pada sisi kirinya
        }
    }
}
```

Kode 5.10 (Lanjutan) *Mengukur kedalaman rekursi Quicksort naif dan Quicksort rekursi-ekor pada masukan terurut dan teracak.*

```
int main() {
    mt19937 rng(20260919);
    cout << "        n | sorted: naive | sorted: tail | shuffled: naive | shuffled: tail |
lg n\n";
    for (int n : {1000, 4000, 16000, 64000}) {
        vector<int> sorted(n); iota(sorted.begin(), sorted.end(), 1);
        vector<int> shuffled = sorted; shuffle(shuffled.begin(), shuffled.end(), rng);

        vector<int> a = sorted;   depthNow = depthMax = 0; quicksortNaive(a, 0, n - 1); int
sn = depthMax;
        vector<int> b = sorted;   depthNow = depthMax = 0; quicksortTail (b, 0, n - 1); int
st = depthMax;
        vector<int> c = shuffled; depthNow = depthMax = 0; quicksortNaive(c, 0, n - 1); int
rn = depthMax;
        vector<int> d = shuffled; depthNow = depthMax = 0; quicksortTail (d, 0, n - 1); int
rt = depthMax;

        int lg = 0; { int m = n; while (m > 1) { m >>= 1; ++lg; } }
        cout << setw(9) << n << " | " << setw(13) << sn << " | " << setw(12) << st
             << " | " << setw(15) << rn << " | " << setw(14) << rt
             << " | " << setw(6) << lg << "\n";
    }
    return 0;
}
```

Keluarannya:

```
        n | sorted: naive | sorted: tail | shuffled: naive | shuffled: tail |  lg n
     1000 |          1000 |            2 |              24 |              7 |      9
     4000 |          4000 |            2 |              31 |              8 |     11
    16000 |         16000 |            2 |              32 |             10 |     13
    64000 |         64000 |            2 |              39 |             11 |     15
```

Versi naifnya pada masukan terurut mencapai kedalaman tepat n: enam puluh empat ribu bingkai tumpukan bersarang. Pada tumpukan delapan megabita yang lazim dengan bingkai sekitar seratus bita, itu sudah dekat sekali ke batasnya, dan larik terurut sejuta elemen pasti akan merusaknya. Versi rekursi-ekornya mencapai kedalaman **dua** pada masukan yang sama — satu rekursi sungguhan ke dalam potongan kiri yang kosong, lalu sebuah gelung — dan tidak pernah melampaui $\lg n$ pada masukan teracak juga: $7 \leq 9$, $8 \leq 11$, $10 \leq 13$, $11 \leq 15$, persis seperti yang dijanjikan teoremanya.

> **BEREKURSILAH PADA SISI YANG LEBIH KECIL.** Kalau Anda mengambil satu kebiasaan implementasi dari bab ini, ambillah yang ini. Ia berongkos satu pembandingan tunggal per pemartisian dan ia mengubah sebuah kerusakan menjadi peristiwa yang tak terjadi. Setiap Quicksort produksi melakukannya.

5.11 Batas Bawah $\Omega(n\lg n)$ bagi Pengurutan Berbasis Pembandingan

Kita berjanji di Subbab 5.1.3 untuk membenarkan klaim bahwa $n\lg n$ adalah sebuah lantai, bukan sekadar yang terbaik yang sejauh ini berhasil kita capai. Inilah hujahnya. Ia pendek, ia lengkap, dan ia salah satu dari sedikit tempat dalam buku ini yang di dalamnya kita membuktikan bahwa sesuatu **mustahil**.

5.11.1 Apa yang tergolong pengurutan berbasis pembandingan

Setiap algoritma dalam bab ini, dan dalam Bab 2 dan 3, mempelajari masukannya dengan satu cara saja: ia mengajukan pertanyaan berbentuk "apakah $a_i \leq a_j$?" Ia tidak pernah melihat *nilai*-nya — tidak pernah memakai kenyataan bahwa sebuah kunci adalah bilangan 47 dan bukan untai `"pineapple"`. Algoritma semacam itu disebut **pengurutan berbasis pembandingan** (*comparison sort*).

Insertion Sort, Selection Sort, Bubble Sort, Merge Sort, Heapsort, dan Quicksort semuanya adalah pengurutan berbasis pembandingan. Alur kendalinya ditentukan sepenuhnya oleh jawaban atas pembandingan ya/tidak.

5.11.2 Pohon keputusan

Tetapkan algoritmanya dan tetapkan ukuran masukan n. Sekarang telusuri apa yang dikerjakan algoritmanya, sebagai sebuah pohon:

- Setiap **simpul dalam** berlabel sebuah pembandingan $a_i : a_j$ yang dilakukan algoritmanya pada titik itu.
- Kedua anaknya bersesuaian dengan dua jawaban yang mungkin, $\leq$ dan $>$.
- Setiap **daun** berlabel permutasi yang dikeluarkan algoritmanya ketika ia mencapai daun itu.

Inilah **pohon keputusan** (*decision tree*) algoritmanya untuk masukan berukuran n. Menjalankan algoritmanya pada suatu masukan tertentu berarti berjalan dari akar ke sebuah daun, berbelok ke kiri atau ke kanan pada tiap simpul menurut jawabannya.

Dua fakta langsung menyusul.

Fakta 1: pohonnya harus punya sekurang-kurangnya $n!$ daun. Ada $n!$ keterurutan yang mungkin atas n elemen yang berbeda, dan algoritma pengurutan yang benar harus menghasilkan permutasi keluaran yang berbeda untuk masing-masingnya. Kalau dua keterurutan yang berbeda menuju daun yang sama, algoritmanya akan mengeluarkan permutasi yang sama bagi keduanya — dan paling banyak satu di antara keluaran itu yang dapat benar.

Fakta 2: banyaknya pembandingan pada kasus terburuk adalah tinggi pohonnya. Banyaknya pembandingan pada suatu masukan adalah banyaknya simpul dalam pada lintasan dari akar ke daunnya, dan kasus terburuk atas semua masukan adalah lintasan semacam itu yang terpanjang.

5.11.3 Pembuktiannya

> **TEOREMA.** Setiap pengurutan berbasis pembandingan memerlukan $\Omega(n \lg n)$ pembandingan pada kasus terburuk.

Bukti. Misalkan pohon keputusannya bertinggi h dan berdaun ℓ.

Sebuah pohon biner bertinggi h mempunyai paling banyak 2^h daun — satu daun bagi masing-masing dari paling banyak 2^h lintasan akar-ke-daun berbeda sepanjang h. Menurut Fakta 1, $\ell \geq n!$. Karena itu

$$n! \leq \ell \leq 2^h. \tag{5.17}$$

Ambil lg dari kedua ujung Persamaan (5.17), yang mempertahankan ketaksamaannya sebab lg naik:

$$h \geq \lg(n!).$$

Sekarang batasi $\lg(n!)$ dari bawah. Hampiran Stirling dari Bab 3 mengatakan $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + \Theta(1/n))$, sehingga

$$\lg(n!) \geq \lg\left(\left(\frac{n}{e}\right)^n\right) = n\lg n - n\lg e = n\lg n - 1.443n = \Omega(n\lg n). \tag{5.18}$$

Karena itu $h = \Omega(n\lg n)$: ada masukan yang memaksa algoritmanya melakukan sekurang-kurangnya sebanyak itu pembandingan. ■

Kalau Anda lebih suka menghindari Stirling, inilah versi dasarnya yang memberikan konstanta sedikit lebih lemah. Di antara n faktor dalam $n! = 1 \cdot 2 \cdots n$, sebanyak $n/2$ faktor yang terbesar masing-masing sekurang-kurangnya $n/2$. Jadi $n! \geq (n/2)^{n/2}$, dan

$$\lg(n!) \geq \frac{n}{2}\lg\frac{n}{2} = \frac{n}{2}\lg n - \frac{n}{2} = \Omega(n\lg n).$$

5.11.4 Apa yang dilarang dan tidak dilarang oleh batas itu

BACALAH HIPOTESISNYA. Teorema itu mengatakan bahwa pengurutan *berbasis pembandingan* tidak dapat mengalahkan $\Omega(n\lg n)$. Ia tidak mengatakan apa pun tentang algoritma yang melihat kuncinya sendiri. Counting Sort, Radix Sort, dan Bucket Sort, yang dibahas tuntas di Bab 8 CLRS [3], mengurutkan dalam waktu $\Theta(n)$ dengan memakai nilai kuncinya sebagai indeks larik alih-alih sekadar membandingkannya. Mereka tidak bertentangan dengan teoremanya; mereka jatuh di luar hipotesisnya. Ongkosnya adalah bahwa mereka perlu mengetahui sesuatu tentang kuncinya di muka: bahwa kuncinya bilangan bulat kecil, atau bahwa kuncinya ditarik dari sebaran yang diketahui.

Kode 5.11 membuat hujahnya konkret untuk n kecil, dan menabelkan batasnya.

Kode 5.11 Batas pohon keputusan yang dihitung untuk n kecil: banyaknya daun yang diperlukan, dan tinggi minimum yang dihasilkannya.

```
// Batas pohon keputusan, dikonkretkan: berapa daun yang harus dimiliki pengurutan
pembandingan,
// dan setinggi apa pohon biner berdaun sebanyak itu harus jadinya?
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
#include <cmath>
using namespace std;

long long factorial(int n) { long long f = 1; for (int i = 2; i <= n; ++i) f *= i; return
f; }

int main() {
    cout << "All 3! = 6 orderings of three distinct keys a, b, c, and the comparisons\n"
            "a comparison sort must make to tell each one apart:\n\n";
    vector<int> p = {1, 2, 3};
    do {
        cout << "  permutation (" << p[0] << "," << p[1] << "," << p[2] << ")"
             << "  a<b? " << (p[0] < p[1] ? "yes" : "no ")
             << "  b<c? " << (p[1] < p[2] ? "yes" : "no ")
             << "  a<c? " << (p[0] < p[2] ? "yes" : "no ") << "\n";
    } while (next_permutation(p.begin(), p.end()));

    cout << "\nSix outcomes need six leaves. A binary tree of height h has at most 2^h\n"
            "leaves, so h >= lg 6 = " << fixed << setprecision(4) << log2(6.0)
         << ", and since h is an integer, h >= 3.\n\n";

    cout << "    n |                  n! | lg(n!) | ceil(lg n!) | n lg n\n";
    for (int n : {2, 3, 4, 5, 8, 10, 12, 16, 20}) {
        long long f = factorial(n);
        double lgf = 0;                          // lg(n!) = jumlah lg k, dijumlahkan agar
tak melimpah
        for (int k = 2; k <= n; ++k) lgf += log2((double)k);
        cout << setw(5) << n << setw(22) << f
             << setw(9) << setprecision(1) << lgf
             << setw(14) << (long long)ceil(lgf)
             << setw(9) << n * log2((double)n) << setprecision(4) << "\n";
    }
    return 0;
}
```

Keluarannya:

```
All 3! = 6 orderings of three distinct keys a, b, c, and the comparisons
a comparison sort must make to tell each one apart:

  permutation (1,2,3)  a<b? yes  b<c? yes  a<c? yes
  permutation (1,3,2)  a<b? yes  b<c? no   a<c? yes
  permutation (2,1,3)  a<b? no   b<c? yes  a<c? yes
  permutation (2,3,1)  a<b? yes  b<c? no   a<c? no
  permutation (3,1,2)  a<b? no   b<c? yes  a<c? no
  permutation (3,2,1)  a<b? no   b<c? no   a<c? no

Six outcomes need six leaves. A binary tree of height h has at most 2^h
leaves, so h >= lg 6 = 2.5850, and since h is an integer, h >= 3.

    n |                  n! | lg(n!) | ceil(lg n!) | n lg n
    2                     2      1.0             1      2.0
    3                     6      2.6             3      4.8
    4                    24      4.6             5      8.0
    5                   120      6.9             7     11.6
    8                 40320     15.3            16     24.0
   10               3628800     21.8            22     33.2
   12             479001600     28.8            29     43.0
   16         20922789888000     44.3            45     64.0
   20    2432902008176640000     61.1            62     86.4
```

Bacalah blok pertamanya: tidak ada dua di antara keenam permutasi itu yang sepakat pada ketiga pembandingannya, sehingga tiga pertanyaan ya/tidak tepat cukup untuk membedakan semuanya — dan dua pertanyaan tidak akan cukup, sebab dua pertanyaan hanya punya empat jawaban yang mungkin sedangkan kasusnya ada enam. Batasnya tercapai.

Bacalah tabelnya: untuk $n = 20$, tidak ada pengurutan berbasis pembandingan yang selalu dapat selesai dalam kurang dari 62 pembandingan. Kolom terakhirnya menunjukkan $n\lg n = 86.4$, sehingga batas $\lceil \lg(n!) \rceil$ kira-kira $0.7\, n\lg n$ — konstantanya kurang dari satu, tetapi laju pertumbuhannya sama, dan itulah yang diklaim $\Omega(n\lg n)$.

> **APA ARTINYA BAGI BAB INI.** Merge Sort, Heapsort, dan Quicksort teracak semuanya berjalan dalam $\Theta(n\lg n)$. Teoremanya mengatakan tidak ada yang dapat berbuat lebih baik dengan pembandingan. Jadi ketiga algoritma itu **optimal secara asimtotik** — dan satu-satunya pertanyaan yang tersisa adalah tentang konstanta, memori, jaminan, dan perilaku *cache*. Pertanyaan-pertanyaan itulah pokok subbab berikutnya, dan pertanyaan itu bukan pertanyaan kecil: merekalah yang menentukan algoritma mana yang sungguh-sungguh dikirim ke produksi.

5.12 Memilih Pengurutan: Perbandingan yang Jujur

5.12.1 Lanskap lengkapnya

Tabel 5.7 adalah seluruh isi buku ini sejauh ini dalam satu tabel.

Tabel 5.7 Setiap algoritma pengurutan dalam buku ini sejauh ini, dengan batas Θ atas banyaknya pembandingan (kasus terbaik Heapsort mengandaikan kunci yang berbeda-beda). Kolom ruangnya mencacah memori tambahan di luar larik masukannya.

Algoritma	Terburuk	Rata-rata	Terbaik	Ruang tambahan	Stabil?	Di tempat?	CLRS	Bab
Insertion Sort	n^2	n^2	n	1	Ya	Ya	Bab 2	2
Selection Sort	n^2	n^2	n^2	1	Tidak	Ya	Bab 2	2
Bubble Sort	n^2	n^2	n	1	Ya	Ya	Bab 2	2
Merge Sort	$n\lg n$	$n\lg n$	$n\lg n$	n	Ya	Tidak	Bab 2	3
Heapsort	$n\lg n$	$n\lg n$	$n\lg n$	1	Tidak	Ya	Bab 6	5
Quicksort	n^2	$n\lg n$	$n\lg n$	$\lg n$	Tidak	Ya	Bab 7	5

Dua entri dalam tabel itu mudah disalahbaca.

Ruang Quicksort adalah $\lg n$**, bukan** 1**.** Quicksort tidak mengalokasikan larik apa pun, sehingga ia bekerja di tempat dalam pengertian yang lazim — tetapi rekursinya sendiri memakan tumpukan, sebanyak $\Theta(\lg n)$ bingkai kalau ditulis seperti pada Subbab 5.10.4. Itulah angka yang jujur.

Kasus terburuk Quicksort sungguh-sungguh n^2**.** Tabelnya tidak berdusta tentang hal ini, dan n^2-nya tidak lenyap dengan pengacakan. Yang diubah pengacakan adalah *peluang*-nya, bukan *kemungkinan*-nya.

5.12.2 Kestabilan, dan mengapa ada yang peduli

Kita sudah mendefinisikan kestabilan di Subbab 5.6.5: sebuah pengurutan disebut stabil jika elemen-elemen berkunci sama tetap pada urutan relatifnya yang semula. Mengapa itu penting?

Sebab rekaman yang sungguhan punya lebih dari satu medan, dan pengguna yang sungguhan mengurutkannya lebih dari sekali. Andaikan daftar pesanan sebuah binatu diurutkan menurut nama pelanggan, lalu operatornya mengurutkannya menurut tanggal pengambilan. Dengan pengurutan yang **stabil**, pesanan yang jatuh tempo pada tanggal yang sama tetap dalam urutan nama — sebuah pengurutan dua kunci dengan harga dua kali pengurutan satu kunci. Dengan pengurutan yang **tidak stabil**, keterurutan namanya musnah dan hasilnya sembarang di dalam tiap tanggal.

Kestabilan juga yang memungkinkan Radix Sort: ia mengurutkan menurut digit paling tidak berarti lebih dulu, lalu menurut digit berikutnya, dengan bersandar pada kenyataan bahwa tiap lintasan memelihara hasil kerja lintasan sebelumnya.

Kode 5.12 memperagakan sifat itu pada masukan mungil yang tiap elemennya dapat Anda periksa dengan mata.

Kode 5.12 Sebuah uji kestabilan. Tiap rekaman membawa sebuah tanda yang menunjukkan dari mana ia berangkat, sehingga penataan ulang di antara kunci yang sama menjadi kasatmata.

```
// Kestabilan: apakah pengurutannya menjaga kunci yang sama pada urutan relatif semulanya?
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

struct Rec { int key; string tag; };             // tag mencatat dari mana butirnya
berangkat

void show(const string& label, const vector<Rec>& v) {
    cout << label;
    for (const Rec& r : v) cout << r.key << r.tag << " ";
    cout << "\n";
}

void insertionSort(vector<Rec>& A) {
    for (size_t j = 1; j < A.size(); ++j) {
        Rec key = A[j]; size_t i = j;
        while (i > 0 && A[i - 1].key > key.key) { A[i] = A[i - 1]; --i; }   // > , bukan >=
        A[i] = key;
    }
}
void mergeSortRec(vector<Rec>& A, vector<Rec>& b, int p, int r) {
    if (p >= r) return;
    int q = p + (r - p) / 2;
    mergeSortRec(A, b, p, q); mergeSortRec(A, b, q + 1, r);
    int i = p, j = q + 1, k = p;
    while (i <= q && j <= r) b[k++] = (A[i].key <= A[j].key) ? A[i++] : A[j++];   // <=
menjaga yang kiri lebih dulu
    while (i <= q) b[k++] = A[i++];
    while (j <= r) b[k++] = A[j++];
    for (int t = p; t <= r; ++t) A[t] = b[t];
}
void mergeSort(vector<Rec>& A) { vector<Rec> b(A.size()); mergeSortRec(A, b, 0,
(int)A.size() - 1); }

void siftDown(vector<Rec>& A, int i, int m) {
    while (true) {
        int l = 2 * i + 1, r = 2 * i + 2, big = i;
        if (l < m && A[l].key > A[big].key) big = l;
        if (r < m && A[r].key > A[big].key) big = r;
        if (big == i) return;
        swap(A[i], A[big]); i = big;
    }
}
void heapSort(vector<Rec>& A) {
    int m = (int)A.size();
    for (int i = m / 2 - 1; i >= 0; --i) siftDown(A, i, m);
```

Kode 5.12 (Lanjutan) *Sebuah uji kestabilan. Tiap rekaman membawa sebuah tanda yang menunjukkan dari mana ia berangkat, sehingga penataan ulang di antara kunci yang sama menjadi kasatmata.*

```
    for (int i = m - 1; i >= 1; --i) { swap(A[0], A[i]); siftDown(A, 0, i); }
}

void quickSort(vector<Rec>& A, int p, int r) {
    if (p >= r) return;
    int x = A[r].key, i = p - 1;
    for (int j = p; j <= r - 1; ++j) if (A[j].key <= x) swap(A[++i], A[j]);
    swap(A[i + 1], A[r]);
    quickSort(A, p, i); quickSort(A, i + 2, r);
}

int main() {
    vector<Rec> original = {{3,"a"},{1,"b"},{3,"c"},{2,"d"},{1,"e"},{3,"f"},{2,"g"}};
    show("original:       ", original);
    vector<Rec> v;
    v = original; insertionSort(v);              show("insertion sort: ", v);
    v = original; mergeSort(v);                  show("merge sort:     ", v);
    v = original; heapSort(v);                   show("heap sort:      ", v);
    v = original; quickSort(v, 0, (int)v.size()-1); show("quicksort:      ", v);
    cout << "\nRead the letters within each group of equal numbers. If they are in\n"
            "alphabetical order the sort was stable on this input.\n";
    return 0;
}
```

Keluarannya:

```
original:       3a 1b 3c 2d 1e 3f 2g
insertion sort: 1b 1e 2d 2g 3a 3c 3f
merge sort:     1b 1e 2d 2g 3a 3c 3f
heap sort:      1e 1b 2g 2d 3f 3c 3a
quicksort:      1b 1e 2d 2g 3c 3f 3a

Read the letters within each group of equal numbers. If they are in
alphabetical order the sort was stable on this input.
```

Bacalah tandanya. Insertion Sort dan Merge Sort sama-sama memberikan `1b 1e`, `2d 2g`, `3a 3c 3f` — setiap kelompok dalam urutan semulanya. Heapsort memberikan `1e 1b`, `2g 2d`, `3f 3c 3a` — setiap kelompok **terbalik**. Quicksort memberikan `3c 3f 3a` — bukan urutan semula dan bukan pula kebalikannya, semata-mata apa pun yang rontok. Itulah arti praktis "tidak stabil": bukan sekadar "boleh jadi menata ulang", melainkan "pasti menata ulang, tanpa dapat diramalkan".

> **CARA MEMPEROLEH KESTABILAN DARI PENGURUTAN YANG TIDAK STABIL.** Lekatkan indeks semula pada tiap rekaman sebagai pemutus seri, lalu bandingkan pasangan (key,index) secara leksikografis. Kini tidak ada dua rekaman yang kuncinya sama, sehingga tidak ada yang perlu ditata ulang, dan pengurutan apa pun menjadi stabil. Ongkosnya adalah satu bilangan bulat tambahan per rekaman dan satu pembandingan tambahan per seri. Ini layak diketahui: artinya kestabilan tidak pernah menjadi alasan Anda *tidak dapat* memakai Quicksort, hanya alasan ia berongkos sedikit lebih mahal.

5.12.3 Perilaku cache, dan mengapa Quicksort menang dalam praktik

Inilah fakta yang tidak dapat ditunjukkan **Tabel 5.7**, dan inilah pokok praktis tunggal yang paling penting dalam bab ini.

Model RAM dari Bab 2 mengatakan setiap akses memori berongkos satu satuan waktu. Perangkat keras yang sungguhan tidak bekerja begitu. Prosesor modern membaca memori dalam **baris cache** selebar enam puluh empat bita, dan menyimpan baris-baris yang baru saja dipakai di dalam *cache* kecil yang cepat. Membaca nilai yang sudah ada di *cache* berongkos kira-kira satu

nanodetik; membaca nilai yang harus didatangkan dari memori utama berongkos kira-kira seratus. Sebuah faktor seratus, yang tak kasatmata bagi setiap analisis dalam bab ini.

Sekarang lihatlah bagaimana kedua algoritma $\Theta(n\lg n)$ kita yang bekerja di tempat itu menyentuh memori.

Quicksort memindai. `PARTITION` berjalan dari p ke r, satu elemen sesudah yang lain, berurutan. Pra-pengambil prosesornya langsung mengenali polanya dan mendatangkan baris *cache* berikutnya sebelum ia dibutuhkan. Hampir setiap aksesnya adalah *cache hit*. Lebih jauh lagi, ketika rekursinya menurun, subarray-nya menyusut, dan begitu sebuah subarray muat seluruhnya di dalam *cache*, setiap operasi yang tersisa padanya menjadi gratis.

Heapsort melompat. `MAX-HEAPIFY` berpindah dari indeks i ke indeks $2i$ lalu ke indeks $4i$ — berlipat dua setiap langkah. Untuk *heap* berisi sejuta elemen, perjalanan dari akarnya mengunjungi indeks $1,2,5,11,23,47,\ldots,500000$ atau sekitarnya, dan beberapa yang terakhir itu berada di wilayah memori yang sama sekali berbeda, terpaut kilobita atau megabita. Setiap akses semacam itu adalah *cache miss*. Lebih buruk lagi, polanya tidak dapat diramalkan, sehingga pra-pengambilnya tidak dapat menolong.

Kode 5.13 mengukur bedanya pada mesin yang sama, dengan data yang sama.

Kode 5.13 Adu langsung keempat pengurutan pada data acak, dengan tiap pengurutan diberi salinan yang segar dan keluarannya diperiksa.

```
// Adu waktu langsung keempat pengurutan pada data yang sama, pada satu mesin.
#include <iostream>
#include <iomanip>
#include <vector>
#include <numeric>
#include <algorithm>
#include <random>
#include <chrono>
using namespace std;
using namespace std::chrono;

void insertionSort(vector<int>& A) {
    for (size_t j = 1; j < A.size(); ++j) {
        int key = A[j];
        size_t i = j;
        while (i > 0 && A[i - 1] > key) { A[i] = A[i - 1]; --i; }
        A[i] = key;
    }
}

void mergeSortRec(vector<int>& A, vector<int>& buf, int p, int r) {
    if (p >= r) return;
    int q = p + (r - p) / 2;
    mergeSortRec(A, buf, p, q);
    mergeSortRec(A, buf, q + 1, r);
    int i = p, j = q + 1, k = p;
    while (i <= q && j <= r) buf[k++] = (A[i] <= A[j]) ? A[i++] : A[j++];
    while (i <= q) buf[k++] = A[i++];
    while (j <= r) buf[k++] = A[j++];
    for (int t = p; t <= r; ++t) A[t] = buf[t];
}
void mergeSort(vector<int>& A) { vector<int> buf(A.size()); mergeSortRec(A, buf, 0,
(int)A.size() - 1); }

void siftDown(vector<int>& A, int i, int m) {    // MAX-HEAPIFY berbasis 0
    while (true) {
        int l = 2 * i + 1, r = 2 * i + 2, largest = i;
        if (l < m && A[l] > A[largest]) largest = l;
        if (r < m && A[r] > A[largest]) largest = r;
        if (largest == i) return;
        swap(A[i], A[largest]);
        i = largest;
    }
}
void heapSort(vector<int>& A) {
    int m = (int)A.size();
    for (int i = m / 2 - 1; i >= 0; --i) siftDown(A, i, m);
```

Kode 5.13 (Lanjutan) *Adu langsung keempat pengurutan pada data acak, dengan tiap pengurutan diberi salinan yang segar dan keluarannya diperiksa.*

```
    for (int i = m - 1; i >= 1; --i) { swap(A[0], A[i]); siftDown(A, 0, i); }
}

mt19937 rng(20260919);
void quickSortRec(vector<int>& A, int p, int r) {
    while (p < r) {
        uniform_int_distribution<int> pick(p, r);
        swap(A[r], A[pick(rng)]);
        int x = A[r], i = p - 1;
        for (int j = p; j <= r - 1; ++j) if (A[j] <= x) swap(A[++i], A[j]);
        swap(A[i + 1], A[r]);
        int q = i + 1;
        if (q - p < r - q) { quickSortRec(A, p, q - 1); p = q + 1; }
        else               { quickSortRec(A, q + 1, r); r = q - 1; }
    }
}
void quickSort(vector<int>& A) { quickSortRec(A, 0, (int)A.size() - 1); }

template <class F>
double timeIt(F f, vector<int> data) {           // salinannya disengaja: tiap pengurutan
mendapat data yang segar
    auto t0 = high_resolution_clock::now();
    f(data);
    auto t1 = high_resolution_clock::now();
    for (size_t i = 1; i < data.size(); ++i) if (data[i - 1] > data[i]) return -1;  //
pemeriksaan akal sehat
    return duration<double, milli>(t1 - t0).count();
}

int main() {
    cout << fixed << setprecision(2);
    cout << "  n       | insertion |   merge |    heap |   quick   (milliseconds)\n";
    for (int n : {1000, 10000, 100000, 1000000}) {
        vector<int> data(n);
        for (int& x : data) x = (int)(rng() % 1000000000u);
        double ti = (n <= 100000) ? timeIt(insertionSort, data) : -2;
        double tm = timeIt(mergeSort, data);
        double th = timeIt(heapSort,  data);
        double tq = timeIt(quickSort, data);
        cout << setw(8) << n << " | ";
        if (ti < -1) cout << setw(9) << "skipped"; else cout << setw(9) << ti;
        cout << " | " << setw(7) << tm << " | " << setw(7) << th << " | " << setw(7) << tq
<< "\n";
    }
    return 0;
}
```

Ini sebuah listing *pengukuran waktu*, jadi angkanya adalah milidetik jam dinding dan akan berbeda pada setiap mesin dan antar-jalan; yang dapat direproduksi adalah polanya, bukan angkanya. Satu kali jalan, pada mesin tempat buku ini dibangun, menghasilkan:

```
  n       | insertion |   merge |    heap |   quick   (milliseconds)
    1000 |      0.09 |    0.05 |    0.03 |    0.04
   10000 |      7.82 |    0.59 |    0.37 |    0.48
  100000 |    806.91 |    7.48 |    6.41 |    6.93
 1000000 |   skipped |   97.02 |  122.72 |   66.79
```

Waktu mutlaknya berbeda-beda antar-mesin, tetapi *bentuk* tabel ini tidak, dan ia menuturkan tiga cerita.

Cerita satu: kuadratik memang sungguh kuadratik. Insertion Sort bergerak 0.09 → 7.82 → 806.91 milidetik. Setiap kenaikan sepuluh kali lipat pada n mengalikan waktunya dengan kira-

kira seratus, yaitu 10^2 — persis seperti yang diramalkan $\Theta(n^2)$. Pengurutan $n\lg n$-nya bergerak dari 7.48 ke 97.02 pada langkah yang sama: sebuah faktor tiga belas, dan ramalannya adalah $10 \times \lg(10^6)/\lg(10^5) = 10 \times 1.2 = 12$. Persis seperti yang diramalkan $\Theta(n\lg n)$.

Cerita dua: Heapsort menang selama datanya muat di cache, dan kalah ketika tidak. Pada $n = 100{,}000$ lariknya berukuran empat ratus kilobita, yang muat dengan nyaman di *cache* L2 modern, dan Heapsort menjadi yang tercepat di antara ketiganya pada 6.41 ms berbanding 6.93 milik Quicksort. Ia menang di sana walaupun melakukan lebih banyak pembandingan daripada Quicksort (kira-kira $2n\lg n$ berbanding sekitar $1.39\, n\lg n$), sebab seluruh lariknya duduk di *cache* dan ia memindahkan lebih sedikit data daripada Merge Sort. Pada $n = 1{,}000{,}000$ lariknya berukuran empat megabita, lompatan *heap*-nya mulai meleset dari *cache*, dan Heapsort menjadi yang **paling lambat** di antara ketiganya pada 122.72 ms sementara Quicksort tercepat pada 66.79 ms. Tidak ada yang berubah dalam analisis asimtotiknya. Perangkat kerasnyalah yang berubah.

Cerita tiga: inilah sebabnya pustaka memilih Quicksort. Heapsort punya jaminan yang lebih baik dan konstanta yang lebih buruk. Pada ukuran data yang penting — yaitu yang besar — konstantanyalah yang menang: di sini Heapsort memakan $122.72/66.79 \approx 1.84$ kali lipat waktu Quicksort pada sejuta elemen. Pustaka yang mengirimkan Heapsort akan hampir dua kali lebih lambat pada setiap pengurutan besar, sebagai tukaran bagi perlindungan terhadap kasus yang sudah dibuat sangat jarang oleh pengacakan. Introsort (Subbab 5.10.3) mengambil keduanya: kecepatan Quicksort sebagai bawaan, jaminan Heapsort sebagai cadangan.

5.12.4 Kapan memakai yang mana

Tabel 5.8 adalah tabel keputusannya. Bacalah sebagai nasihat, bukan undang-undang: ukurlah data Anda sendiri.

Tabel 5.8 Memilih algoritma pengurutan. Baris terakhirnya adalah yang paling sering akan Anda pakai dalam praktik.

Situasi	Pilihan terbaik, dan sebabnya
Larik kecil, $n < 50$	**Insertion Sort** — overhead per elemennya paling rendah; asimtotiknya belum jadi soal
Data hampir terurut	**Insertion Sort** — $\Theta(n)$ ketika tiap elemen dekat dengan tempat finalnya
Serbaguna, kecepatan nomor satu	**Quicksort teracak** (atau Introsort milik pustakanya) — tercepat dalam praktik, berkat perilaku *cache*-nya
Jaminan $O(n\lg n)$ yang keras diperlukan	**Heapsort** atau **Merge Sort** — keduanya sama sekali tidak punya kasus n^2
Kestabilan diperlukan	**Merge Sort** — satu-satunya pengurutan $\Theta(n\lg n)$ yang stabil yang sudah kita temui
Memori sangat sempit	**Heapsort** — $O(1)$ ruang tambahan dengan $\Theta(n\lg n)$ yang terjamin
Data lebih besar daripada memori	**Merge Sort** — ia dasar bagi setiap pengurutan eksternal multi-lintasan
Penyisipan dan pengambilan-maks berulang	**Binary heap / antrean prioritas** — jangan mengurutkan sama sekali; $O(\lg n)$ per operasi
Sistem waktu nyata, tenggat keras	**Heapsort** — kasus terburuk yang dapat Anda tuliskan di dalam spesifikasi
Pemrograman kompetitif	Pengurutan bawaan bahasanya: `std::sort`, `Arrays.sort`, `sorted()`

RINGKASAN SATU KALIMAT. Pakailah pengurutan milik pustakanya kecuali Anda punya alasan khusus untuk tidak; kalau Anda memerlukan jaminan kasus terburuk, pakailah Heapsort; kalau Anda memerlukan kestabilan, pakailah Merge Sort; kalau lariknya mungil atau

> hampir terurut, pakailah Insertion Sort; dan kalau Anda berulang kali meminta elemen terbesar yang tersisa, janganlah mengurutkan sama sekali — pakailah *heap*.

5.13 Profitina dalam Praktik

5.13.1 Di mana bab ini muncul pada produk yang sungguhan

Profitina adalah platform visibilitas AI — sebuah perkakas *Generative Engine Optimization* (GEO). Ia memindai situs web sebuah merek, menilai seberapa siap tiap halamannya untuk **dikutip oleh mesin jawaban AI**, dan menjelaskan apa yang perlu diperbaiki. Produk publiknya berjalan di `app.profitina.com`; ia adalah aplikasi web Python di atas PostgreSQL, yang memakai Ollama dengan model Qwen2.5 yang di-*host* secara lokal untuk asisten konten dan teknisnya, sehingga gelung intinya tidak bergantung pada model yang di-*host* pihak lain [13].

Tiga bagian produk itu adalah bab ini, hampir baris demi baris.

Antrean perayapannya adalah antrean prioritas. Sebuah pemindaian situs bukanlah "ambil setiap halaman menurut urutan kita menemukannya". Sebagian halaman jauh lebih penting daripada yang lain: halaman muka, halaman harga, halaman yang paling mungkin ditanyakan kepada mesin jawaban. Sebagian lain kurang penting: arsip berhalaman dari empat tahun lalu. Dan perayapnya punya anggaran — batas per tingkatan langganan atas berapa halaman yang akan diambilnya dalam satu pemindaian. Jadi perayapnya menyimpan batas depannya di dalam **antrean prioritas-maks** berkunci skor kegentingan, dan berulang kali melakukan `EXTRACT-MAX` sampai anggarannya habis. Ketika sebuah halaman ternyata menaut ke sesuatu yang penting, kegentingan halaman yang ditaut itu dinaikkan — itulah `INCREASE-KEY`. Ketika sebuah URL baru ditemukan, itulah `INSERT`. Ketiganya $O(\lg n)$ pada sebuah *heap*, dan ketiganya akan menjadi $O(n)$ pada alternatif berbasis larik mana pun. Dengan batas depan berisi seratus ribu URL, $\lg n \approx 17$; alternatifnya adalah 100,000. Pilihan struktur datanya adalah beda antara pemindaian yang selesai dan pemindaian yang kehabisan waktu.

Penjadwal pemindaian ulangnya adalah antrean prioritas-min. Setiap merek pada tingkatan berbayar dipindai ulang secara otomatis dengan irama yang ditetapkan tingkatannya. Itu antrean kejadian yang klasik: tiap merek membawa cap waktu "jatuh tempo berikutnya", dan penjadwalnya berulang kali mengeluarkan butir yang jatuh temponya *paling awal*, menjalankan pemindaiannya, menghitung ulang waktu jatuh tempo berikutnya, lalu menyisipkannya kembali. Inilah pola simulasi kejadian diskret dari Subbab 5.7.2, yang berjalan sungguhan. Sebuah *min-heap* menjaga pertanyaan "apa yang jatuh tempo berikutnya?" tetap $O(1)$ dan penjadwalan ulangnya $O(\lg n)$, berapa pun banyaknya merek yang terdaftar.

Papan riwayatnya mengurutkan. Tampilan tren skor mingguan, dan ekspor CSV yang menyertainya, sama-sama mengurutkan baris — menurut tanggal, menurut skor, menurut sinyal. Ini pengurutan biasa atas beberapa ribu baris, dan jawaban yang benar adalah membiarkan PostgreSQL atau pustaka bakunya yang mengerjakannya. Alasan memahami bab ini bukanlah untuk mengimplementasikannya ulang; alasannya adalah untuk mengetahui bahwa mengurutkan seribu baris berongkos mikrodetik sedangkan mengurutkan sepuluh juta berongkos detik, sehingga Anda dapat memberi tahu kueri mana yang perlu indeks dan mana yang tidak.

Yang diperikan subbab ini adalah *bentuk* persoalannya — penelusuran berprioritas yang beranggaran terbatas, sebuah penjadwal berkala, sebuah tabel yang perlu diurutkan — dan kelas penyelesaiannya, yang merupakan materi bab ini. Itulah bagian yang berpindah ke sistem apa pun yang Anda bangun [13].

Satu kendala lagi yang layak disebut, sebab ia membentuk rancangan lebih daripada kinerja. Profitina beroperasi di bawah undang-undang perlindungan data pribadi Indonesia, **UU PDP 27/2022**. Setiap antrean, log, atau ekspor yang menyentuh data pribadi harus dibangun sedemikian rupa sehingga data seorang pengguna dapat ditemukan, diekspor, dan dimusnahkan atas permintaan. Dalam praktik itu berarti: kuncilah antrean Anda pada pengenal, bukan pada medan pribadi; jangan pernah menaruh nama, surel, atau nomor telepon ke dalam kunci pengurutan atau kunci *cache*, tempat ia akan tersalin ke tempat-tempat yang tidak dapat Anda daftar satu per satu. Studi kasus di bawah ini menaati aturan itu.

5.13.2 Studi kasus yang dikerjakan: Profitina Laundry

Studi kasus langsung kita adalah **Profitina Laundry**, usaha pengajaran yang kita pakai sepanjang buku ini. Ia persoalan berukuran sungguhan.

Toko itu menerima cucian sepanjang hari. Setiap pesanan punya berat dalam kilogram, jam siap yang dijanjikan, dan mungkin biaya tambahan kilat. Mesinnya lebih sedikit daripada pesanannya, sehingga tokonya terus-menerus harus menjawab satu pertanyaan: **apa yang kita cuci berikutnya?**

Itu persis `EXTRACT-MAX` pada sebuah antrean prioritas. Kita definisikan sebuah skor kegentingan yang memadukan tiga fakta yang kita ketahui:

$$\text{urgency}(o) = (48 - \text{dueHour}) \times 10 + 40 \cdot [\, o \text{ kilat} \,] - \text{kg}. \tag{5.19}$$

Bacalah ketiga suku Persamaan (5.19):

- $(48 - \text{dueHour}) \times 10$ — makin awal sebuah pesanan jatuh tempo, makin besar suku ini. Pengalinya 10 membuat tiap jam bernilai sepuluh angka, sehingga waktu yang mendominasi.
- $+40$ untuk pesanan kilat — setara keunggulan empat jam, yang merupakan kebijakan komersial tokonya yang dinyatakan sebagai angka.
- $-\text{kg}$ — sebuah potongan kecil bagi muatan berat, sehingga ketika dua pesanan selebihnya sama, yang lebih ringan jalan lebih dulu dan mengosongkan mesinnya lebih cepat.

Pada akhir hari, tokonya ingin laporan penerimaan yang terurut menurut nilai, dan itu pengurutan biasa — dan kita memakai Quicksort untuknya, sebab tidak ada apa pun di sini yang bersifat melawan dan n-nya kecil.

Kode 5.14 adalah keseluruhannya, dan ia berjalan. Perhatikan apa yang *tidak* ada di dalam rekaman `Order`: tidak ada nama pelanggan, tidak ada nomor telepon, tidak ada alamat. Pesanan dikenali lewat kode tiket anonim yang tercetak pada slip pelanggannya. Ini bukan hiasan; inilah rupa kepatuhan UU PDP 27/2022 pada tataran sebuah `struct` tunggal [13].

Kode 5.14 Profitina Laundry: sebuah *heap* memutuskan urutan mencucinya, dan Quicksort menghasilkan laporan penerimaannya.

```
// Profitina Laundry: antrean kerja harinya sebagai heap, dan laporan harinya sebagai
quicksort.
// Tidak ada data pribadi di mana pun: pesanan dikenali hanya lewat kode tiketnya.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

struct Order {
    string ticket;      // kode tiket anonim yang tercetak pada slip pelanggan
    int    kg;          // berat muatannya
    int    dueHour;     // jam hari saat muatannya dijanjikan, jam 24
    bool   express;     // pesanan kilat dibayar dengan tarif yang lebih tinggi
};

// Kegentingan: pesanan kilat setara keunggulan empat jam, dan tiap kilogram
// yang akan menyibukkan mesin mendorong pesanannya sedikit ke belakang antrean.
int urgency(const Order& o) {
    return (48 - o.dueHour) * 10 + (o.express ? 40 : 0) - o.kg;
}

// ---------- max-heap berisi pesanan, berkunci kegentingan ----------
vector<Order> H;                                  // H[0] tak terpakai; heap-nya adalah
H[1..heapSize]
int heapSize = 0;

void maxHeapify(int i) {
    int l = 2 * i, r = 2 * i + 1, largest = i;
    if (l <= heapSize && urgency(H[l]) > urgency(H[largest])) largest = l;
    if (r <= heapSize && urgency(H[r]) > urgency(H[largest])) largest = r;
    if (largest != i) { swap(H[i], H[largest]); maxHeapify(largest); }
}

void insertOrder(const Order& o) {
    ++heapSize;
    if ((int)H.size() <= heapSize) H.push_back(o); else H[heapSize] = o;
    int i = heapSize;
    while (i > 1 && urgency(H[i / 2]) < urgency(H[i])) { swap(H[i], H[i / 2]); i /= 2; }
}
```

Kode 5.14 (Lanjutan) *Profitina Laundry: sebuah heap memutuskan urutan mencucinya, dan Quicksort menghasilkan laporan penerimaannya.*

```
Order extractMax() {
    Order best = H[1];
    H[1] = H[heapSize];
    --heapSize;
    maxHeapify(1);
    return best;
}

// ---------- quicksort, dipakai untuk laporan akhir hari ----------
int priceOf(const Order& o) { return o.kg * (o.express ? 12000 : 7000); }   // rupiah

int partition(vector<Order>& A, int p, int r) {
    int x = priceOf(A[r]), i = p - 1;
    for (int j = p; j <= r - 1; ++j) if (priceOf(A[j]) <= x) swap(A[++i], A[j]);
    swap(A[i + 1], A[r]);
    return i + 1;
}
void quicksortByPrice(vector<Order>& A, int p, int r) {
    if (p >= r) return;
    int q = partition(A, p, r);
    quicksortByPrice(A, p, q - 1);
    quicksortByPrice(A, q + 1, r);
}

int main() {
    vector<Order> arrivals = {
        {"PL-0341", 6, 17, false}, {"PL-0342",  3, 14, true },
        {"PL-0343", 12, 20, false}, {"PL-0344", 4, 13, false},
        {"PL-0345",  8, 16, true }, {"PL-0346", 2, 22, false},
        {"PL-0347",  5, 15, false}, {"PL-0348", 9, 18, true }
    };

    H.assign(1, arrivals[0]);                    // slot 0 hanya pengisi tempat, tak pernah
dibaca

    cout << "Orders as they arrive at the counter:\n";
    cout << "  ticket    kg  due  express  urgency\n";
    for (const Order& o : arrivals) {
        cout << "  " << o.ticket << setw(5) << o.kg << setw(5) << o.dueHour
             << setw(9) << (o.express ? "yes" : "no") << setw(9) << urgency(o) << "\n";
        insertOrder(o);
    }
```

Kode 5.14 (Lanjutan) *Profitina Laundry: sebuah heap memutuskan urutan mencucinya, dan Quicksort menghasilkan laporan penerimaannya.*

```
    cout << "\nThe machine asks the queue what to wash next:\n";
    vector<Order> done;
    int step = 1;
    while (heapSize > 0) {
        Order o = extractMax();
        cout << "  wash " << step++ << ": " << o.ticket
             << "  (urgency " << urgency(o) << ", due " << o.dueHour << ":00"
             << (o.express ? ", express" : "") << ")\n";
        done.push_back(o);
    }

    cout << "\nEnd-of-day report, sorted by value with quicksort:\n";
    quicksortByPrice(done, 0, (int)done.size() - 1);
    long long total = 0;
    for (const Order& o : done) {
        cout << "  " << o.ticket << setw(4) << o.kg << " kg"
             << setw(12) << priceOf(o) << " IDR\n";
        total += priceOf(o);
    }
    cout << "  " << string(7, ' ') << setw(12) << " " << "------------\n";
    cout << "  takings" << setw(16) << total << " IDR\n";
    return 0;
}
```

Keluarannya:

```
Orders as they arrive at the counter:
  ticket    kg  due  express  urgency
  PL-0341    6   17       no      304
  PL-0342    3   14      yes      377
  PL-0343   12   20       no      268
  PL-0344    4   13       no      346
  PL-0345    8   16      yes      352
  PL-0346    2   22       no      258
  PL-0347    5   15       no      325
  PL-0348    9   18      yes      331

The machine asks the queue what to wash next:
  wash 1: PL-0342  (urgency 377, due 14:00, express)
  wash 2: PL-0345  (urgency 352, due 16:00, express)
  wash 3: PL-0344  (urgency 346, due 13:00)
  wash 4: PL-0348  (urgency 331, due 18:00, express)
  wash 5: PL-0347  (urgency 325, due 15:00)
  wash 6: PL-0341  (urgency 304, due 17:00)
  wash 7: PL-0343  (urgency 268, due 20:00)
  wash 8: PL-0346  (urgency 258, due 22:00)

End-of-day report, sorted by value with quicksort:
  PL-0346   2 kg       14000 IDR
  PL-0344   4 kg       28000 IDR
  PL-0347   5 kg       35000 IDR
  PL-0342   3 kg       36000 IDR
  PL-0341   6 kg       42000 IDR
  PL-0343  12 kg       84000 IDR
  PL-0345   8 kg       96000 IDR
  PL-0348   9 kg      108000 IDR
                     ------------
  takings          443000 IDR
```

Lihatlah urutan mencucinya dan periksalah terhadap aturan skornya, sebab di sinilah aritmetika berubah menjadi keputusan bisnis.

Pesanan `PL-0344` jatuh tempo pukul 13:00 — **lebih awal daripada apa pun yang ada di toko** — dan toh ia dicuci *ketiga*. Dua pesanan kilat yang jatuh tempo pukul 14:00 dan 16:00 mendahuluinya. Itu bukan kutu; itu kebijakan yang dikodekan oleh skornya. Kilat bernilai 40 angka, dan satu jam bernilai 10, sehingga sebuah pesanan kilat mengalahkan pesanan biasa yang jatuh temponya sampai empat jam lebih awal. `PL-0342` kilat dan jatuh tempo pukul 14:00: skornya $(48 - 14) \times 10 + 40 - 3 = 340 + 40 - 3 = 377$. `PL-0344` biasa dan jatuh tempo pukul 13:00: $(48 - 13) \times 10 + 0 - 4 = 350 - 4 = 346$. Jadi $377 > 346$ dan pesanan kilatnya menang.

> **PELAJARANNYA BAGI SISTEM SUNGGUHAN.** *Heap*-nya tidak memutuskan apa pun; *fungsi skornya* yang memutuskan. *Heap*-nya semata-mata melaksanakan keputusan itu secara efisien. Kalau pemilik tokonya memutuskan bahwa tenggat yang terlewat lebih buruk daripada pelanggan kilat yang kecewa, ia mengubah pengalinya dari 40 menjadi 15, dan antreannya berperilaku berbeda pada sore yang sama — tanpa perubahan apa pun pada struktur datanya. **Betulkan dulu fungsi prioritasnya; baru raih heap-nya untuk membuatnya cepat.** Keempat operasi Subbab 5.7 adalah mekanismenya; skornya adalah kebijakannya. Pemeringkatan Profitina sendiri bekerja dengan cara yang sama: tugas pemindainya adalah mengukur sinyal, dan tugas fungsi skornya adalah menyatakan berapa nilai sinyal itu. Aplikasinya menasihati; penggunanya yang memutuskan.

5.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. Mengira sebuah heap itu terurut. Ia tidak terurut. $A[1]$ adalah maksimumnya, dan itulah satu-satunya jaminan global. Mencetak sebuah *heap* lalu mengharapkan urutan menaik adalah kesalahan pemula yang paling sering terjadi. *Perbaikan:* kalau Anda menginginkan keluaran terurut, jalankan Heapsort, atau panggil `EXTRACT-MAX` berulang kali.

2. Mencampuradukkan `heap-size` dan panjang lariknya. Keduanya bilangan yang berbeda, dan Heapsort bergantung pada perbedaan keduanya. Kalau `MAX-HEAPIFY` memakai panjang lariknya alih-alih `heap-size`, ia akan dengan senang hati menarik kembali elemen yang sudah terurut ke dalam *heap*-nya dan pengurutan Anda akan menghasilkan sampah. *Perbaikan:* jadikan `heap-size` peubah yang eksplisit, dan jangan pernah menulis `A.size()` di dalam `MAX-HEAPIFY`.

3. Memeriksa batas sesudah mengakses lariknya. Menulis `if (A[l] > A[largest] && l <= heapSize)` membaca di luar rentang sebelum mengujinya. C++ mengevaluasi dari kiri ke kanan dan `&&` berhubung-singkat, sehingga uji batasnya harus datang **lebih dulu**. *Perbaikan:* `if (l <= heapSize && A[l] > A[largest])`, selalu dalam urutan itu.

4. Membandingkan anak kanan terhadap `A[i]` alih-alih `A[largest]`. Sesudah uji anak kirinya mungkin memperbarui `largest`, uji anak kanannya harus membandingkan terhadap juara yang sekarang. *Perbaikan:* `if` yang kedua selalu berbunyi `A[r] > A[largest]`.

5. Membangun sebuah heap dengan menyisipkan n kali. Cara itu bekerja, dan berongkos $O(n\lg n)$. `BUILD-MAX-HEAP` berongkos $O(n)$. *Perbaikan:* kalau seluruh datanya sudah ada di depan Anda, bangunlah; sisipkan hanya ketika butirnya sungguh-sungguh datang satu per satu. Dalam C++, `make_heap` adalah versi $O(n)$-nya.

6. Memulai `BUILD-MAX-HEAP` di n alih-alih $\lfloor n/2 \rfloor$. Tak berbahaya tetapi boros — paruh kedua lariknya adalah daun, dan memanggil `MAX-HEAPIFY` pada daun tidak mengerjakan apa pun. *Perbaikan:* mulailah di $\lfloor n/2 \rfloor$. Memulai di 1 dan mencacah *menaik* adalah versi serius kesalahan ini: prasyaratnya gagal dan hasilnya salah, bukan sekadar lambat.

7. Salah menetapkan batas rekursi Quicksort. Dengan pemartisian Lomuto, pemanggilannya adalah `(p, q-1)` dan `(q+1, r)`. Dengan pemartisian Hoare, pemanggilannya adalah `(p, q)` dan `(q+1, r)` — dan nilai *pivot*-nya lalu harus diambil dari $A[p]$, bukan $A[r]$ (Subbab 5.8.5). Mencampur keduanya menghasilkan entah rekursi tak berkesudahan atau elemen yang hilang. *Perbaikan:* tulislah pemartisiannya dan kedua pemanggilan rekursifnya pada saat yang sama, jangan pernah terpisah.

8. Menggelar Quicksort ber-pivot tetap. Masukan terurut lalu memicu kasus terburuk $\Theta(n^2)$-nya, dan masukan terurut amat lazim. *Perbaikan:* acaklah *pivot*-nya — dua baris — atau pakai median dari tiga, atau pakai pustakanya.

9. Lupa bahwa rekursinya memakan tumpukan. Quicksort naif pada larik terurut sejuta elemen berekursi sedalam sejuta dan rusak sebelum waktu kuadratiknya sempat kasatmata. *Perbaikan:* selalu berekursi pada sisi yang lebih kecil dan menggelung pada sisi yang lebih besar, seperti di Subbab 5.10.4.

10. Mengira pengacakan menghapus kasus terburuknya. Ia tidak menghapusnya; ia membuatnya tak mungkin. Kalau Anda memerlukan sebuah *jaminan* — tenggat waktu nyata, masukan lawan, sebuah perjanjian tingkat layanan — pakailah Heapsort atau Introsort, bukan Quicksort teracak. *Perbaikan:* jernihkan dulu apakah Anda memerlukan "cepat rata-rata" atau "tidak pernah lambat". Keduanya syarat yang berbeda.

11. Mengira Quicksort atau Heapsort itu stabil. Keduanya tidak. Kalau dua rekaman berkunci sama harus mempertahankan urutannya, entah pakailah Merge Sort atau tambahkan indeks semula sebagai pemutus seri. *Perbaikan:* lihat catatan di akhir Subbab 5.12.2.

12. Menilai dari asimtotiknya saja. **Tabel 5.7** mengatakan Heapsort dan Quicksort punya kompleksitas rata-rata yang sama. Tolok ukur di Subbab 5.12.3 menunjukkan Quicksort menyelesaikan sejuta elemen dalam 67 ms dan Heapsort dalam 123 ms. Tabel tidak mengukur *cache*. *Perbaikan:* ukurlah, pada data Anda, pada mesin Anda.

13. Mengurutkan padahal seharusnya memakai heap. Kalau tugasnya adalah "berulang kali ambil butir terbesar yang tersisa", jangan mengurutkan lariknya — itu $\Theta(n\lg n)$ di muka dan boleh jadi Anda hanya memerlukan beberapa butir yang pertama. Sebuah *heap* memberi Anda k yang pertama dalam $O(n + k\lg n)$. *Perbaikan:* tanyakan apakah Anda memerlukan seluruh urutannya atau hanya ujung-ujungnya.

14. Memakai `int` padahal cacahnya dapat melimpah. Cacah pembandingan sebesar $n(n-1)/2$ untuk $n = 100{,}000$ adalah kira-kira 5×10^9, yang melimpah pada `int` 32-bit. *Perbaikan:* cacahlah dengan `long long`, seperti yang dilakukan setiap program dalam bab ini.

15. Menulis pengurutan sendiri dalam sebuah lomba. `std::sort` adalah Introsort, ia benar, dan ia lebih cepat daripada yang akan Anda tulis. *Perbaikan:* pakailah ia, dan belanjakan waktu yang Anda hemat itu pada bagian persoalan yang sungguh-sungguh sukar.

5.15 Ringkasan Bab

1. **Sebuah binary heap menyimpan pohon biner hampir lengkap di dalam larik yang datar**, sama sekali tanpa penunjuk. Navigasinya aritmetika murni: $\text{Parent}(i) = \lfloor i/2 \rfloor$, $\text{Left}(i) = 2i$, $\text{Right}(i) = 2i + 1$, masing-masing sebuah instruksi geser tunggal. Karena lariknya bersambungan, *heap* bersahabat dengan *cache* dengan cara yang tidak dimiliki pohon berbasis penunjuk.

2. **Sifat max-heap itu lokal**: setiap induk sekurang-kurangnya sebesar anak-anaknya, dan tidak ada lagi yang dijamin. Dari satu aturan lokal itu menyusul satu fakta global tunggal melalui ketransitifan: $A[1]$ adalah maksimumnya.

3. **Sebuah heap berisi n elemen bertinggi tepat** $\lfloor \lg n \rfloor$, daun-daunnya menempati indeks $\lfloor n/2 \rfloor + 1$ sampai n, dan paling banyak $\lceil n/2^{h+1} \rceil$ simpulnya bertinggi h. Setiap biaya dalam bab ini diturunkan dari ketiga cacahan itu.

4. `MAX-HEAPIFY` **menenggelamkan satu nilai menuruni satu lintasan akar-ke-daun** dalam $O(\lg n)$, dan ia mesin bagi segala yang lain. Rekurensinya, Persamaan (5.5), yaitu $T(n) \leq T(2n/3) + \Theta(1)$, terselesaikan menjadi $O(\lg n)$ lewat Kasus 2 Teorema Master.

5. `BUILD-MAX-HEAP` **berjalan dalam $\Theta(n)$, bukan $\Theta(n \lg n)$** — dibuktikan dengan menjumlahkan $\lceil n/2^{h+1} \rceil \cdot O(h)$ atas semua tinggi dan memakai $\sum_{h \geq 0} h/2^h = 2$ (Persamaan (5.8)), yang sendirinya datang dari menurunkan deret geometri. Pengukuran membenarkannya: banyaknya penukaran per elemen tetap di dekat 0.74 dari seribu elemen sampai sepuluh juta, sementara $\lg n$ merangkak dari 9 ke 23.

6. **Heapsort adalah** `BUILD-MAX-HEAP` **ditambah $n - 1$ pengeluaran**: $\Theta(n \lg n)$ pada setiap kasus bila kuncinya berbeda-beda, $O(1)$ ruang tambahan, tidak stabil. Ialah algoritma yang mengisi lubang di **Tabel 5.1**.

7. **Antrean prioritas di atas heap melakukan** `MAXIMUM` **dalam $O(1)$ dan** `INSERT`**,** `EXTRACT-MAX`**, serta** `INCREASE-KEY` **dalam $O(\lg n)$** — satu-satunya implementasi di **Tabel 5.4** yang tidak punya operasi linear. Inilah struktur data di balik Dijkstra, Prim, Huffman, simulasi kejadian, dan setiap penjadwal yang akan pernah Anda tulis.

8. **Quicksort memartisi di sekitar sebuah pivot lalu berekursi**, tanpa langkah pemaduan. Pemartisian Lomuto memelihara invarian empat wilayah dan mudah dibuktikan; pemartisian Hoare menjalankan dua penunjuk yang saling mendekat, melakukan lebih sedikit penukaran, dan itulah yang dipakai pustaka sungguhan.

9. **Biaya Quicksort sepenuhnya merupakan fungsi dari seberapa seimbang belahannya.** Belahan yang sempurna memberikan $\Theta(n \lg n)$; belahan merosot $1:(n-1)$ memberikan $\Theta(n^2)$; dan **sembarang belahan dengan nisbah konstanta tetap — bahkan** $99:1$ **— memberikan** $\Theta(n \lg n)$, hanya berbeda pada konstantanya. Ketegaran itulah sebabnya Quicksort bekerja.

10. **Banyaknya pembandingan yang diharapkan adalah** $2(n+1)H_n - 4n \approx 1.39\, n \lg n$, dibuktikan dengan hujah peubah indikator: z_i dan z_j dibandingkan tepat ketika *pivot* pertama yang dipilih dari Z_{ij} adalah salah satu di antara keduanya, yang berpeluang $2/(j - i + 1)$.

11. **Mengacak pivotnya berongkos dua baris dan membuat waktu jalannya tidak bergantung pada masukannya.** Pengacakan tidak melenyapkan kasus terburuknya; ia menurunkan peluangnya menjadi kira-kira $2^{n-1}/n!$ (Persamaan (5.16)), yang untuk $n = 100$ adalah sekitar 10^{-128}.

12. **Selalu berekursilah pada sisi yang lebih kecil.** Itu membatasi kedalaman tumpukannya pada $\lg n$ tak peduli bagaimana pemartisiannya berjalan, dan itu mengubah sebuah kerusakan menjadi peristiwa yang tak terjadi.

13. **Tidak ada pengurutan berbasis pembandingan yang dapat mengalahkan** $\Omega(n \lg n)$, sebab pohon keputusannya memerlukan $n!$ daun dan pohon biner berdaun $n!$ bertinggi sekurang-kurangnya $\lg(n!) = \Omega(n \lg n)$ menurut Persamaan (5.18). Merge Sort, Heapsort, dan

Quicksort teracak karena itu optimal secara asimtotik, dan persaingan yang tersisa adalah tentang konstanta, memori, dan jaminan.

14. **Dalam praktik Quicksort mengalahkan Heapsort** kendati jaminannya lebih buruk, sebab ia memindai memori secara berurutan sementara sebuah *heap* melompat-lompat di antara indeks i, $2i$, dan $4i$. Terukur: 67 ms berbanding 123 ms pada sejuta elemen — sementara pada seratus ribu elemen, ketika segalanya muat di *cache*, Heapsort-lah yang lebih cepat di antara keduanya.

15. **Mengganti struktur datanya dapat mengganti kelas kompleksitasnya.** Selection Sort dan Heapsort adalah algoritma yang sama; satu-satunya bedanya adalah di mana datanya tinggal. Itulah gagasan yang paling dapat dipindahkan dalam bab ini.

5.15.1 Menengok ke depan

Bab 6 meninggalkan larik dan beralih ke **pohon pencarian biner** serta bentuk swaseimbangnya, **pohon AVL**, yang mendukung pencarian, penyisipan, *dan* penghapusan dalam $O(\lg n)$ — sesuatu yang tidak dapat dilakukan sebuah *heap*, sebab *heap* hanya dapat menemukan maksimumnya, bukan kunci sembarang. Bab 7 bergerak dari bagi dan kuasai ke **pemrograman dinamis**, lalu Bab 8, *Evaluasi dan Pembahasan II*, menggarap soal-soal yang diambil dari Bab 5 sampai 7 — termasuk *heap* dan pemartisian dari bab ini. Bab 11 dan 13 akhirnya mencairkan antrean prioritasnya: pohon merentang minimum Prim dan lintasan terpendek Dijkstra keduanya, secara struktural, adalah Subbab 5.7 yang dibungkuskan pada sebuah graf.

Satu topik layak mendapat penunjuk tersendiri, sebab bab ini membuktikannya mustahil lalu diam-diam mengecualikannya: **pengurutan waktu linear**. Counting Sort, Radix Sort, dan Bucket Sort mengurutkan n butir dalam $\Theta(n)$ dengan menolak menjadi pengurutan berbasis pembandingan sama sekali. Ketiganya berada di luar alur utama buku ini, dan Bab 8 CLRS [3] membahasnya secara tuntas.

5.16 Soal Latihan

5.1 Berapa banyak elemen minimum dan maksimum dalam sebuah *heap* bertinggi h?

5.2 Tunjukkan bahwa sebuah *heap* berisi n elemen bertinggi $\lfloor \lg n \rfloor$ dengan menghitung tingginya secara langsung untuk $n = 1,2,3,4,7,8,15,16,17$ dan memeriksa masing-masingnya terhadap rumusnya.

5.3 Apakah larik $A = [23,17,14,6,13,10,1,5,7,12]$ merupakan *max-heap*? Kalau bukan, sebutkan setiap pelanggarannya.

5.4 Di bagian mana sebuah *max-heap* elemen **terkecil** dapat berada, dengan mengandaikan semua elemennya berbeda? Buktikan jawaban Anda.

5.5 Peragakan cara kerja `MAX-HEAPIFY(A, 3)` pada $A = [27,17,3,16,13,10,1,5,7,12,4,8,9,0]$ dengan heap-size $= 14$. Tunjukkan lariknya sesudah setiap penukaran.

5.6 Peragakan `BUILD-MAX-HEAP` pada $A = [5,3,17,10,84,19,6,22,9]$. Tunjukkan lariknya sesudah setiap iterasi gelungnya.

5.7 Mulai dari *heap* yang Anda bangun pada Soal 5.6, tunjukkan tiga iterasi pertama `HEAPSORT`.

5.8 Berapa waktu jalan `HEAPSORT` pada larik A sepanjang n yang **sudah terurut menaik**? Bagaimana kalau **menurun**? Benarkan kedua jawabannya.

5.9 Hujahkan bahwa untuk sebuah *heap* dengan elemen-elemen yang berbeda, maksimum dari *daun-daunnya* saja belum tentu merupakan elemen terbesar kedua dalam *heap*-nya. Lalu nyatakan, dan buktikan, di mana elemen terbesar kedua itu pasti berada.

5.10 Peragakan `MAX-HEAP-INSERT(A, 10)` pada *heap* $A = [15,13,9,5,12,8,7,4,0,6,2,1]$. Tunjukkan setiap pembandingannya.

5.11 Dengan memakai `PARTITION` versi Lomuto, peragakan cara kerjanya pada larik $A = [13,19,9,5,12,8,7,4,21,2,6,11]$. Tunjukkan nilai i dan j serta lariknya pada setiap iterasi.

5.12 Nilai q apa yang dikembalikan `PARTITION` ketika semua elemen $A[p \ldots r]$ bernilai **sama**? Ubahlah pemartisian Lomuto sehingga ia membelah larik semacam itu menjadi dua paruh yang kira-kira sama, dan sebutkan mengapa hal itu penting.

5.13 Selesaikan rekurensi $T(n) = T(n-1) + \Theta(n)$ dengan metode substitusi, dengan membuktikan $T(n) = O(n^2)$ beserta konstanta yang eksplisit.

5.14 Andaikan `PARTITION` milik Quicksort selalu menghasilkan belahan $1:99$. Tuliskan rekurensinya, gambarkan bentuk pohon rekursinya, hitung kedalaman minimum dan maksimumnya, lalu simpulkan waktu jalannya.

5.15 Berapa peluang Quicksort teracak menghasilkan belahan yang *seimbang sempurna* pada tingkat teratas sebuah larik berisi n elemen berbeda? Untuk $n = 1000$, seberapa besar peluang itu?

5.16 Tunjukkan, dengan larik empat elemen yang konkret, bahwa Heapsort tidak stabil, dan bahwa Quicksort tidak stabil.

5.17 Buktikan bahwa setiap pengurutan berbasis pembandingan memerlukan sekurang-kurangnya $\lceil \lg(n!) \rceil$ pembandingan pada kasus terburuk, lalu hitunglah bilangan itu untuk $n = 5$. Kemudian rancanglah, dengan tangan, sebuah pohon keputusan yang mengurutkan tiga elemen dalam paling banyak tiga pembandingan.

5.18 *(Pemrograman.)* Tulislah program C++ yang membaca n dan k, membangkitkan n bilangan bulat acak, lalu mencetak k terbesar dengan memakai sebuah *heap* dalam waktu $O(n + k\lg n)$ — tanpa mengurutkan seluruh lariknya. Bandingkan waktu jalannya terhadap `std::sort` untuk $n = 10^7$, $k = 10$.

5.19 *(Pemrograman.)* Implementasikan antrean prioritas-min yang mendukung `INSERT`, `MINIMUM`, `EXTRACT-MIN`, dan `DECREASE-KEY`, lalu pakailah untuk memadukan k senarai terurut yang panjang totalnya n dalam waktu $O(n\lg k)$.

5.20 *(Analisis.)* Pengukuran `BUILD-MAX-HEAP` di **Kode 5.3** melaporkan kira-kira 0.74 penukaran per elemen. Jelaskan mengapa konstanta itu kurang dari 1, dan taksirlah nilainya dari tinggi simpul-simpulnya.

5.17 Penyelesaian Soal Latihan

5.1 Menurut Persamaan (5.4) di Subbab 5.3.6, sebuah *heap* bertinggi h memenuhi $2^h \le n < 2^{h+1}$, yaitu $2^h \le n \le 2^{h+1} - 1$, sebab n bilangan bulat. **Minimumnya** adalah 2^h: tingkat 0 sampai $h - 1$ penuh ($2^h - 1$ simpul) ditambah satu simpul di tingkat h. **Maksimumnya** adalah $2^{h+1} - 1$: setiap tingkat penuh, termasuk tingkat h. Sebagai contoh, tinggi 3 menerima antara 8 dan 15 elemen.

5.2 Hitung $\lfloor \lg n \rfloor$ lalu bandingkan dengan pohon yang digambar; **Tabel 5.9** mengumpulkan kesembilan kasusnya.

Tabel 5.9 Tinggi-tinggi untuk Soal 5.2. Tingginya bertambah satu tepat pada tiap pangkat dua, dan itulah yang dikerjakan lantai sebuah logaritma basis dua.

n	$\lg n$	$\lfloor \lg n \rfloor$	Tinggi sebenarnya
1	0	0	0 ✓
2	1	1	1 ✓
3	1.58	1	1 ✓
4	2	2	2 ✓
7	2.81	2	2 ✓
8	3	3	3 ✓
15	3.91	3	3 ✓
16	4	4	4 ✓
17	4.09	4	4 ✓

5.3 Periksa setiap induknya. $A[1] = 23 \geq 17{,}14$ ✓. $A[2] = 17 \geq 6{,}13$ ✓. $A[3] = 14 \geq 10{,}1$ ✓. $A[4] = 6$: anak-anaknya $A[8] = 5$ ✓ dan $A[9] = 7$ — **pelanggaran**, $6 < 7$. $A[5] = 13$: anaknya $A[10] = 12$ ✓. Jadi ada tepat satu pelanggaran, pada pasangan $(A[4], A[9]) = (6{,}7)$. Lariknya **bukan** *max-heap*. Satu pemanggilan `MAX-HEAPIFY(A, 4)` memperbaikinya, memberikan [23,17,14,7,13,10,1,5,6,12].

5.4 Elemen terkecilnya pasti sebuah **daun**, yaitu berada di suatu indeks dalam $\lfloor n/2 \rfloor + 1, \ldots, n$. *Bukti:* andaikan elemen terkecilnya duduk di simpul dalam i. Maka i punya sekurang-kurangnya satu anak c, dan sifat *heap* menuntut $A[i] \geq A[c]$. Karena semua elemennya berbeda, $A[i] > A[c]$, sehingga $A[i]$ bukan yang terkecil — sebuah kontradiksi. Karena itu ia sebuah daun. Sembarang daun mungkin: taruh nilai terkecilnya di mana saja pada paruh kedua sebuah *heap* yang selebihnya berisi nilai-nilai besar, dan sifatnya tetap berlaku.

5.5 `MAX-HEAPIFY(A, 3)` pada $A = [27{,}17{,}3{,}16{,}13{,}10{,}1{,}5{,}7{,}12{,}4{,}8{,}9{,}0]$, heap-size $= 14$.

- $i = 3$: $A[3] = 3$, anak-anaknya $A[6] = 10$, $A[7] = 1$. Terbesarnya 10 di indeks 6. Tukar: [27,17,**10**,16,13,**3**,1,5,7,12,4,8,9,0].
- $i = 6$: $A[6] = 3$, anak-anaknya $A[12] = 8$, $A[13] = 9$. Terbesarnya 9 di indeks 13. Tukar: [27,17,10,16,13,**9**,1,5,7,12,4,8,**3**,0].
- $i = 13$: $2 \times 13 = 26 > 14$, sebuah daun. Berhenti.

Akhirnya: $A = [27{,}17{,}10{,}16{,}13{,}9{,}1{,}5{,}7{,}12{,}4{,}8{,}3{,}0]$.

5.6 $A = [5{,}3{,}17{,}10{,}84{,}19{,}6{,}22{,}9]$, $n = 9$, $\lfloor n/2 \rfloor = 4$.

- $i = 4$: $A[4] = 10$, anak-anaknya $A[8] = 22$, $A[9] = 9$. Tukar dengan 22: [5,3,17,**22**,84,19,6,**10**,9]. Indeks 8 adalah daun. Berhenti.
- $i = 3$: $A[3] = 17$, anak-anaknya $A[6] = 19$, $A[7] = 6$. Tukar dengan 19: [5,3,**19**,22,84,**17**,6,10,9]. Daun. Berhenti.
- $i = 2$: $A[2] = 3$, anak-anaknya $A[4] = 22$, $A[5] = 84$. Tukar dengan 84: [5,**84**,19,22,**3**,17,6,10,9]. Di indeks 5: $2 \times 5 = 10 > 9$, sebuah daun. Berhenti.
- $i = 1$: $A[1] = 5$, anak-anaknya $A[2] = 84$, $A[3] = 19$. Tukar dengan 84: [**84**,**5**,19,22,3,17,6,10,9]. Di indeks 2: anak-anaknya $A[4] = 22$, $A[5] = 3$; tukar dengan 22: [84,**22**,19,**5**,3,17,6,10,9]. Di indeks 4: anak-anaknya $A[8] = 10$, $A[9] = 9$; tukar dengan 10: [84,22,19,**10**,3,17,6,**5**,9]. Indeks 8 adalah daun. Berhenti.

Hasilnya: $A = [84{,}22{,}19{,}10{,}3{,}17{,}6{,}5{,}9]$. Verifikasi: $84 \geq 22{,}19$; $22 \geq 10{,}3$; $19 \geq 17{,}6$; $10 \geq 5{,}9$ ✓.

5.7 Mulai dari [84,22,19,10,3,17,6,5,9], $n = 9$.

- $i = 9$: tukarkan $A[1] = 84$ dengan $A[9] = 9$; susutkan menjadi 8; *heapify* akarnya: 9 melawan 22 dan 19 → tukar dengan 22; lalu 9 melawan $A[4] = 10$ dan $A[5] = 3$ → tukar dengan 10; lalu indeks 4 hanya punya anak $A[8] = 5 \rightarrow 9 > 5$, berhenti. Hasilnya [22,10,19,9,3,17,6,5 | 84].
- $i = 8$: tukarkan 22 dengan $A[8] = 5$; susutkan menjadi 7; *heapify*: 5 melawan 10 dan 19 → tukar dengan 19; di indeks 3, anak-anaknya $A[6] = 17$ dan $A[7] = 6$ → tukar dengan 17; indeks 6 kini sebuah daun. Hasilnya [19,10,17,9,3,5,6 | 22,84].
- $i = 7$: tukarkan 19 dengan $A[7] = 6$; susutkan menjadi 6; *heapify*: 6 melawan 10 dan 17 → tukar dengan 17; di indeks 3, hanya anak $A[6] = 5$, dan $6 > 5$, berhenti. Hasilnya [17,10,6,9,3,5 | 19,22,84].

5.8 Pada **kedua** kasusnya waktu jalannya $\Theta(n\lg n)$. Biaya Heapsort tidak bergantung pada urutan masukannya. `BUILD-MAX-HEAP` adalah $\Theta(n)$ pada keduanya, dan gelung pengeluarannya melakukan $n - 1$ pemanggilan `MAX-HEAPIFY`, yang masing-masingnya harus menenggelamkan sebuah elemen kecil — yang diambil dari dasar *heap*-nya — menembus *heap* bertinggi $\Theta(\lg n)$. Tidak ada masukan yang memendekkan perjalanan itu secara sistematis. Inilah keutamaan Heapsort sekaligus kutukannya: ia tidak pernah terkejut, dan ia tidak pernah mujur. (Bandingkan dengan Quicksort, yang bagi masukan menaik justru merupakan kasus terburuk $\Theta(n^2)$.)

5.9 Maksimum dari daun-daunnya belum tentu terbesar kedua: dalam [16,14,10,8,7,9,3,2,4,1] daun-daunnya adalah {9,3,2,4,1} dengan maksimum 9, sedangkan elemen terbesar kedua *heap*-nya adalah 14, sebuah simpul dalam. Pernyataan yang benar adalah: **elemen terbesar kedua pasti merupakan anak akarnya**, yaitu berada di indeks 2 atau 3. *Bukti:* misalkan x elemen terbesar kedua, dan andaikan ia duduk di kedalaman 2 atau lebih. Maka x punya induk p, dan p bukan akarnya. Sifat *heap* memberikan $A[p] > x$ (secara tegas, sebab elemennya berbeda-beda), dan akarnya, sebagai maksimum, juga lebih besar daripada x. Jadi sekurang-kurangnya dua elemen yang berbeda, yaitu akarnya dan p, lebih besar daripada x, sehingga x paling banter terbesar ketiga — sebuah kontradiksi. Karena itu x duduk di kedalaman 1, di indeks 2 atau 3. ■

5.10 Sisipkan 10 ke dalam $A = [15,13,9,5,12,8,7,4,0,6,2,1]$ ($n = 12$).

- heap-size ← 13; $A[13] \leftarrow -\infty$; lalu naikkan menjadi 10.
- $i = 13$. Parent(13) = 6. Apakah $A[6] = 8 < 10$? Ya → tukar. Kini lariknya berbunyi … $A[6] = 10$ … $A[13] = 8$, dan $i = 6$.
- $i = 6$. Parent(6) = 3. Apakah $A[3] = 9 < 10$? Ya → tukar. Kini $A[3] = 10$, $A[6] = 9$, dan $i = 3$.
- $i = 3$. Parent(3) = 1. Apakah $A[1] = 15 < 10$? **Tidak.** Berhenti.

Hasilnya: $A = [15,13,10,5,12,9,7,4,0,6,2,1,8]$. Tiga pembandingan, dua penukaran, dan kunci barunya naik dua tingkat.

5.11 Lomuto pada $A = [13,19,9,5,12,8,7,4,21,2,6,11]$, $p = 1$, $r = 12$, *pivot* $x = A[12] = 11$, $i = 0$. **Tabel 5.10** membawa gelungnya.

Tabel 5.10 Pemartisian Lomuto pada Soal 5.11, iterasi demi iterasi.

j	$A[j]$	≤11?	i Sesudahnya	Larik
1	13	tidak	0	[13,19,9,5,12,8,7,4,21,2,6,11]
2	19	tidak	0	tak berubah
3	9	ya	1	[9,19,13,5,12,8,7,4,21,2,6,11]
4	5	ya	2	[9,5,13,19,12,8,7,4,21,2,6,11]
5	12	tidak	2	tak berubah
6	8	ya	3	[9,5,8,19,12,13,7,4,21,2,6,11]
7	7	ya	4	[9,5,8,7,12,13,19,4,21,2,6,11]
8	4	ya	5	[9,5,8,7,4,13,19,12,21,2,6,11]
9	21	tidak	5	tak berubah
10	2	ya	6	[9,5,8,7,4,2,19,12,21,13,6,11]
11	6	ya	7	[9,5,8,7,4,2,6,12,21,13,19,11]

Akhirnya tukarkan $A[i+1] = A[8]$ dengan $A[12]$: [9,5,8,7,4,2,6, **11**, 21,13,19,12] lalu kembalikan $q = 8$. Bagian kiri {9,5,8,7,4,2,6}, semuanya ≤ 11 ✓; bagian kanan {21,13,19,12}, semuanya >11 ✓.

5.12 Kalau setiap elemennya sama dengan *pivot*-nya maka setiap uji $A[j] \leq x$ berhasil, i bertambah pada setiap iterasi, dan prosedurnya mengembalikan $q = r$ — belahan yang ekstrem. Quicksort lalu berekursi pada $n-1$ elemen dan ongkosnya $\Theta(n^2)$: larik sejuta nilai yang identik adalah kasus terburuk. **Perbaikan:** perubahan paling kecil pada prosedur Lomuto sendiri adalah membagi rata kunci-kunci kembarnya. Peliharalah sebuah bendera yang berbalik setiap kali $A[j] = x$, dan biarkan elemen yang sama dengan *pivot* bergabung ke wilayah kiri hanya ketika benderanya terpasang. Pada larik serba-sama, separuh dari $n-1$ elemen yang disapu lalu masuk ke kiri dan separuhnya tetap di kanan, sehingga belahannya menjadi dua paruh berukuran $\lfloor (n-1)/2 \rfloor$ dan $\lceil (n-1)/2 \rceil$; pengurutannya tetap benar, sebab elemen yang sama dengan *pivot* boleh berada di sisi mana pun. Obat kedua adalah skema Hoare, yang menghentikan kedua penunjuknya pada nilai yang sama dengan *pivot*-nya dan karena itu juga membelah larik serba-sama hampir tepat di tengah. Perbaikan yang semestinya adalah pemartisian *Dutch national flag*: hasilkan tiga wilayah $<x$, $=x$, $>x$ lalu berekursilah hanya pada dua wilayah terluarnya. Larik dengan d nilai berbeda lalu terurut dalam $O(n \lg d)$. Ini penting sebab data yang penuh kembaran amat lazim — status, kategori, tanggal, bendera boolean.

5.13 Tebak $T(n) \leq cn^2$ dan misalkan suku $\Theta(n)$-nya paling banyak dn.

Andaikan $T(n-1) \leq c(n-1)^2$. Maka

$$T(n) \leq c(n-1)^2 + dn = cn^2 - 2cn + c + dn = cn^2 - (2c-d)n + c.$$

Ini paling banyak cn^2 asalkan $(2c-d)n \geq c$, yaitu $n \geq \frac{c}{2c-d}$. Memilih $c = d$ membuat syaratnya menjadi $n \geq \frac{d}{d} = 1$, sehingga batasnya berlaku untuk semua $n \geq 1$ dengan $c = d$ dan kasus basis $T(1) \leq c$. Karena itu $T(n) = O(n^2)$. ■

5.14 Rekurensinya adalah $T(n) = T(n/100) + T(99n/100) + \Theta(n)$.

Setiap tingkat yang lengkap berongkos n, sebab $\frac{1}{100}n + \frac{99}{100}n = n$. Daun yang paling dangkal dicapai dengan selalu mengambil cabang $1/100$: kedalaman $\log_{100} n$. Yang paling dalam dicapai dengan selalu mengambil cabang $99/100$: kedalaman $\log_{100/99} n$. Jadi

$$n\log_{100} n \leq T(n) \leq n\log_{100/99} n + \Theta(n).$$

Mengubahnya ke basis dua: $\log_{100} n = \lg n/\lg 100 \approx 0.15\lg n$ dan $\log_{100/99} n = \lg n/\lg(100/99) \approx \lg n/0.0145 \approx 69\lg n$. Keduanya $\Theta(n\lg n)$, sehingga

$$T(n) = \Theta(n\lg n),$$

dengan konstanta di suatu tempat antara kira-kira 0.15 dan kira-kira 69 — konstanta yang besar, dan tetap saja bukan perubahan laju pertumbuhan. Inilah pokok Subbab 5.9.4.

5.15 Belahan yang seimbang sempurna menuntut *pivot* yang terpilih merupakan **median**-nya. Dengan *pivot* acak seragam dari n elemen yang berbeda, tepat satu elemen merupakan mediannya, sehingga peluangnya $1/n$. Untuk $n = 1000$ itu 0.001, satu berbanding seribu.

Pokok pertanyaan ini adalah bahwa peluang mungil itu **tidak** penting. Quicksort tidak memerlukan *pivot* median; menurut Subbab 5.9.4 ia hanya memerlukan belahan bernisbah tetap. Peluang *pivot*-nya jatuh di paruh tengah urutan terurutnya — yang paling buruk memberikan belahan $1:3$ — adalah $1/2$, dan belahan $1:3$ sudah $\Theta(n\lg n)$ dengan kedalaman $\log_{4/3} n \approx 2.4\lg n$. Jadi separuh dari semua *pivot* sudah "cukup baik", dan itulah sebabnya Quicksort berperilaku baik pada hakikatnya selalu.

5.16 Ambil rekaman $\langle 2, a\rangle, \langle 1, b\rangle, \langle 2, c\rangle, \langle 1, d\rangle$, diurutkan menurut bilangannya.

Heapsort. `BUILD-MAX-HEAP` pada $[2a, 1b, 2c, 1d]$: pada $i = 2$, $A[2] = 1b$ dibandingkan dengan anaknya $A[4] = 1d$ — sama, jadi tanpa penukaran (ujinya adalah $>$ yang tegas); pada $i = 1$, $2a$ dibandingkan dengan $1b$ dan $2c$ — ia tidak dikalahkan, jadi tanpa penukaran. *Heap*-nya tak berubah. Sekarang gelung pengeluarannya, satu iterasi demi satu iterasi:

- $i = 4$: tukarkan $A[1] = 2a$ dengan $A[4] = 1d$, memberikan $[1d, 1b, 2c \mid 2a]$. *Heapify* akarnya: $2c > 1d$, jadi tukar, memberikan $[2c, 1b, 1d \mid 2a]$.
- $i = 3$: tukarkan $A[1] = 2c$ dengan $A[3] = 1d$, memberikan $[1d, 1b \mid 2c, 2a]$. *Heapify*: $1b$ tidak lebih besar daripada $1d$, jadi tidak ada yang bergerak.
- $i = 2$: tukarkan $A[1] = 1d$ dengan $A[2] = 1b$, memberikan $[1b \mid 1d, 2c, 2a]$.

Hasilnya $[1b, 1d, 2c, 2a]$. Kedua angka 2 kini muncul dalam urutan c, a, kebalikan dari urutan semulanya. **Tidak stabil.**

Quicksort (Lomuto, pivot terakhir). $x = 1d$. $j = 1$: $2a > 1$, lewati. $j = 2$: $1b \leq 1$, $i \to 1$, tukarkan $A[1] \leftrightarrow A[2] \to [1b, 2a, 2c, 1d]$. $j = 3$: $2c > 1$, lewati. Lalu tukarkan $A[2] \leftrightarrow A[4] \to [1b, 1d, 2c, 2a]$. Kedua angka 2 kini berurutan c, a — terbalik. **Tidak stabil.**

Alasan umumnya sama pada kedua kasus: kedua algoritma memindahkan elemen sejauh jarak yang panjang dalam satu penukaran, melewati elemen-elemen yang tidak pernah mereka bandingkan.

5.17 *Batasnya.* Menurut Subbab 5.11.3, pohon keputusan sebuah pengurutan berbasis pembandingan yang benar berdaun sekurang-kurangnya $n!$ dan pohon biner bertinggi h berdaun paling banyak 2^h, sehingga $2^h \geq n!$, karena itu $h \geq \lg(n!)$. Karena h mencacah pembandingan dan merupakan bilangan bulat, $h \geq \lceil \lg(n!) \rceil$.

Untuk $n = 5$: $5! = 120$ dan $\lg 120 \approx 6.91$, sehingga diperlukan sekurang-kurangnya $\lceil 6.91 \rceil = \mathbf{7}$ pembandingan. (Tujuh juga memadai — *merge-insertion sort* mencapainya — sehingga batasnya ketat di sini.)

Sebuah pohon keputusan untuk tiga elemen a, b, c. Tiga pembandingan memadai, dan menurut batasnya $\lceil \lg 6 \rceil = 3$ pembandingan diperlukan.

```
                    a : b
            ≤ /             \ >
          b : c             a : c
      ≤ /       \ >     ≤ /       \ >
  (a,b,c)      a : c   (b,a,c)    b : c
            ≤ /   \ >          ≤ /    \ >
        (a,c,b) (c,a,b)    (b,c,a)  (c,b,a)
```

Enam daun, satu bagi tiap permutasi, dan tidak ada lintasan akar-ke-daun yang lebih panjang daripada tiga. ✓

5.18 *Kerangka penyelesaiannya.* Dua pendekatan: yang pertama persis yang diminta soalnya, dan yang kedua adalah alternatif yang berguna.

Pendekatan A ($O(n + k\lg n)$). Bangun sebuah *max-heap* atas seluruh n elemennya dengan `make_heap` — $O(n)$ — lalu panggil `pop_heap` sebanyak k kali, $O(k\lg n)$. Totalnya $O(n + k\lg n)$, yang persis diminta soalnya.

Pendekatan B ($O(n\lg k)$, lebih baik ketika $k \ll n$). Peliharalah sebuah **min**-heap berukuran tepat k. Untuk masing-masing dari n elemennya: kalau *heap*-nya berisi kurang dari k butir, dorong elemennya masuk; kalau tidak, dan kalau elemennya melampaui minimum *heap*-nya, keluarkan minimumnya lalu dorong elemennya masuk. Pada akhirnya *heap*-nya memuat k elemen terbesar. Ongkosnya $O(n\lg k)$, dan — ini yang menentukan — memorinya $O(k)$ alih-alih $O(n)$, sehingga ia bekerja pada aliran data yang tidak dapat Anda simpan.

Dalam C++, Pendekatan A adalah `make_heap(v.begin(), v.end()); for (int t=0;t<k;++t){ pop_heap(v.begin(), v.end()-t); }`, dan Pendekatan B adalah sebuah `priority_queue<int, vector<int>, greater<int>>` berkapasitas k. Keduanya akan menyelesaikan $n = 10^7$, $k = 10$ jauh di bawah satu detik, sementara `std::sort` pada 10^7 bilangan bulat memakan kira-kira satu detik — sebab mengurutkan menghitung seluruh urutannya padahal hanya sepuluh nilai yang diinginkan. Pelajarannya adalah Kesalahan yang Sering Terjadi nomor 13.

5.19 *Kerangka.* Taruh satu entri per senarai ke dalam sebuah *min-heap*, tiap entri mencatat `(value, whichList, positionInList)`. Berulang kali lakukan `EXTRACT-MIN`, tambahkan nilainya ke keluarannya, lalu `INSERT` elemen berikutnya dari senarai yang sama itu, kalau masih ada. *Heap*-nya tidak pernah memuat lebih dari k entri, sehingga masing-masing dari n pengeluaran dan n penyisipannya berongkos $O(\lg k)$: total $O(n\lg k)$.

Ini benar-benar lebih baik daripada menyambung lalu mengurutkan, yang berongkos $O(n\lg n)$, dan inilah cara pengurutan eksternal memadukan rentetan dari cakram. `DECREASE-KEY` tidak diperlukan untuk pemaduannya sendiri, tetapi diperlukan pada klien termasyhur lain dari kelas yang sama, yaitu algoritma Dijkstra (Bab 13). Implementasi `DECREASE-KEY` adalah cermin persis `INCREASE-KEY` di **Kode 5.5**: setel kunci yang lebih kecil, lalu apungkan elemennya *ke atas* selama induknya lebih besar.

5.20 Cacah penukarannya adalah total jarak tenggelam semua nilainya selama `BUILD-MAX-HEAP`. Kelompokkan simpulnya menurut tinggi:

- Tinggi 0 — kira-kira $n/2$ simpul, daun-daunnya. `MAX-HEAPIFY` tidak pernah dipanggil padanya. **Nol** penukaran.
- Tinggi 1 — kira-kira $n/4$ simpul. Masing-masing tenggelam paling banyak satu tingkat, dan pada data acak tenggelam kira-kira $\frac{2}{3}$ tingkat rata-rata (akar sebuah subpohon tiga elemen sudah menjadi yang terbesar dengan peluang $1/3$). Sumbangannya $\approx \frac{n}{4} \cdot \frac{2}{3} \approx 0.167n$.
- Tinggi 2 — kira-kira $n/8$ simpul, tenggelam paling banyak dua tingkat; pada data acak sedikit lebih dari satu tingkat. Sumbangannya $\approx \frac{n}{8} \cdot 1.2 \approx 0.15n$.
- Tinggi h — kira-kira $n/2^{h+1}$ simpul yang tenggelam $O(h)$ tingkat.

Menjumlahkannya memberikan $n \sum_{h \geq 0} c_h / 2^{h+1}$ dengan $c_h < h$, dan deretnya konvergen ke sebuah konstanta yang benar-benar kurang dari $\sum_h h / 2^{h+1} = 1$. Angka terukur 0.74 duduk dengan nyaman di bawah langit-langit itu — dan langit-langitnya sendiri tepat separuh dari identitas $\sum h/2^h = 2$ di Subbab 5.5.6. Alasan konstantanya di bawah 1 adalah butir pertama tadi: **lebih dari separuh simpulnya sama sekali tidak pernah disentuh.**

Daftar Pustaka

[1] J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, Jun. 1964, doi: 10.1145/512274.512284.

[2] R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, vol. 7, no. 12, p. 701, Dec. 1964, doi: 10.1145/355588.365103.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, chs. 6–8.

[4] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1, 4th ed. Lulu Press, 2020.

[5] ISO/IEC 14882:2024, *Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2024.

[6] Python Software Foundation, "heapq — Heap queue algorithm," *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/heapq.html. [Accessed: 24-Sep-2026].

[7] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, Jan. 1962, doi: 10.1093/comjnl/5.1.10.

[8] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[9] Oracle, "Class Arrays," *Java SE 25 & JDK 25 API Specification*. [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Arrays.html. [Accessed: 24-Sep-2026].

[10] D. R. Musser, "Introspective sorting and selection algorithms," *Software: Practice and Experience*, vol. 27, no. 8, pp. 983–993, Aug. 1997, doi: 10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#.

[11] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011, ch. 2.

[12] S. Wild, "Powersort in official Python 3.11 release." [Online]. Available: https://www.wild-inter.net/posts/powersort-in-python-3.11. [Accessed: 24-Sep-2026]. See also J. I. Munro and S. Wild, "Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs," in *Proc. 26th European Symposium on Algorithms (ESA 2018)*, LIPIcs vol. 112, 2018, pp. 63:1–63:16.

[13] Profitina, "Profitina — AI Visibility (GEO) platform." [Online]. Available: https://profitina.com. [Accessed: 19-Sep-2026].

Bab 6
Pohon Pencarian Biner dan Pohon AVL

Di akhir bab ini Anda akan sanggup membangun, dengan tangan dan dengan C++, sebuah struktur data yang menjaga sekumpulan kunci tetap terurut selama-lamanya sambil membiarkan Anda menambah dan membuang kunci sesuka hati. Anda akan sanggup menyatakan sifat pohon pencarian biner dengan tepat, dan mengatakan persis di mana bedanya dari sifat *heap* (tumpukan) yang begitu sering dikacaukan para pemula dengannya. Anda akan sanggup menelusuri sebuah pohon dengan tiga cara yang berbeda, dan membuktikan — bukan sekadar menyatakan — bahwa salah satu dari ketiga penelusuran itu mencetak kuncinya dalam keadaan terurut. Anda akan sanggup mencari di dalam pohon, menemukan kunci terkecil dan terbesarnya, menemukan kunci yang datang tepat sebelum atau tepat sesudah sebuah kunci mana pun, menyisipkan sebuah kunci, dan menghapus sebuah kunci dalam ketiga bentuk yang bisa diambil penghapusan, dan Anda akan tahu mengapa setiap satu dari operasi-operasi itu berongkos $O(h)$ dengan h adalah tingginya. Sesudah itu Anda akan melihat bagaimana h bisa diam-diam tumbuh sampai n dan menghancurkan segalanya, dan Anda akan mempelajari obat yang ditemukan Adelson-Velsky dan Landis: sebuah pohon yang memperbaiki bentuknya sendiri sesudah setiap perubahan. Anda akan menurunkan, dari asas yang paling awal, mengapa sebuah pohon AVL tidak pernah bisa lebih tinggi daripada kira-kira $1.44\lg n$; Anda akan menguasai keempat kasus rotasi lengkap dengan penugasan penunjuk yang persis dilakukan masing-masingnya; Anda akan memutar ulang sebuah contoh penyisipan lima langkah dan sebuah contoh penghapusan, simpul demi simpul; dan Anda akan selesai dengan kesanggupan mengatakan, untuk sebuah pekerjaan tertentu, apakah pohon AVL, pohon merah-hitam, atau tabel hash yang merupakan perkakas yang tepat.

6.1 Dari Mana Kita Melanjutkan

Bab 3 memberi kita sebuah bahasa dan sekotak perkakas.

Bahasanya adalah **notasi asimtotik**. Kita belajar bahwa $O(g(n))$ adalah himpunan fungsi yang pada akhirnya duduk di bawah suatu kelipatan konstanta dari g; bahwa $\Omega(g(n))$ adalah himpunan yang pada akhirnya duduk di atas suatu kelipatan konstanta dari g; dan bahwa $\Theta(g(n))$ adalah himpunan yang melakukan keduanya sekaligus, terjepit di dalam lorong antara $c_1 g(n)$ dan $c_2 g(n)$. Kita belajar menyebut nama para saksi c dan n_0, bukan sekadar melambai ke arah mereka.

Kotak perkakasnya adalah **rekurensi**. Ketika sebuah algoritma memanggil dirinya sendiri, waktu jalannya menuruti sebuah persamaan yang menyebut dirinya sendiri, dan kita belajar tiga cara menyelesaikan persamaan semacam itu: substitusi (tebak lalu buktikan dengan induksi), pohon rekursi (gambar kerjanya aras demi aras lalu jumlahkan), dan Teorema Master (sebuah resep yang menjawab $T(n) = aT(n/b) + f(n)$ dalam tiga kasus).

Perkakas-perkakas itu dibangun untuk menganalisis algoritma. Sekarang kita akan memakainya untuk menganalisis sebuah *struktur data*, dan itu latihan yang sedikit berbeda. Sebuah algoritma

punya satu waktu jalan. Sebuah struktur data punya waktu jalan per operasi, dan waktu jalan operasi-operasinya saling terjalin: bentuk yang membuat satu operasi cepat dihasilkan oleh operasi-operasi yang lain. Jalinan itulah pokok bahasan bab ini.

Inilah celah yang hendak kita tutup. Di Bab 2 kita bertemu dua cara menyimpan sekumpulan kunci.

Sebuah **larik terurut** luar biasa nyaman untuk dicari. Pencarian biner membagi dua rentang calon pada setiap sentuhan, sehingga ia menemukan sebuah kunci di antara n kunci dalam $\Theta(\lg n)$ sentuhan. Tetapi ia menyedihkan untuk diubah: untuk menyisipkan kunci baru pada tempatnya yang benar Anda harus menggeser, rata-rata, separuh lariknya satu posisi ke kanan, dan itu kerja $\Theta(n)$ per penyisipan.

Sebuah **senarai berantai** adalah kebalikannya. Menyambungkan simpul baru ke dalam senarai, begitu Anda sudah memegang simpul yang harus diikutinya, berongkos $\Theta(1)$. Tetapi *sampai* ke simpul itu berarti menyusuri senarainya dari salah satu ujung, dan itu $\Theta(n)$.

Yang satu cepat dibaca dan lambat ditulis; yang lain cepat ditulis dan lambat dibaca. Tak satu pun cukup baik untuk persoalan nyata yang sudah terang di depan mata: sebuah koleksi yang terus-menerus dicari *dan* terus-menerus diubah.

Pohon pencarian biner adalah struktur yang menolak memilih. Ia mempertahankan perilaku membelah-dua milik pencarian biner — setiap pembandingan membuang satu sisi penuh dari sisa koleksinya — sekaligus mempertahankan bedah kecil-kecilan yang murah milik senarai berantai, karena menyisipkan sebuah simpul berarti mengubah dua atau tiga penunjuk dan tidak lebih. Setiap operasi dalam bab ini akan berongkos $O(h)$, dengan h adalah tinggi pohonnya, dan seluruh paruh kedua bab ini adalah tentang satu pertanyaan: **bagaimana kita menghentikan h dari tumbuh?**

6.2 Kosakata Pohon

Sebelum apa pun, kita memerlukan kata-katanya. Pembaca yang belum pernah berjumpa dengan pohon dalam ilmu komputer sebaiknya membaca bagian ini pelan-pelan; semua yang datang sesudahnya bergantung padanya, dan tidak satu pun dari isinya sulit.

Sebuah **pohon** adalah kumpulan **simpul** yang disambungkan oleh **sisi**, digambar terbalik: satu simpul di puncak adalah **akar**, dan pohonnya tumbuh ke bawah. Setiap simpul memuat sebuah **kunci** — nilai yang kita pakai untuk mengurutkan dan mencari — dan biasanya juga **data satelit** yang melekat pada kunci itu, sebagaimana sebuah nomor nota cucian punya pelanggan, berat, dan harga yang melekat padanya. Sepanjang bab ini kita hanya menggambar kuncinya, karena kunci itulah satu-satunya yang dipedulikan strukturnya.

Jika ada sisi yang membentang dari simpul u turun ke simpul v, maka u adalah **induk** dari v dan v adalah **anak** dari u. Akar adalah satu-satunya simpul yang tidak punya induk. Simpul yang tidak punya anak disebut **simpul daun**; simpul yang punya sekurang-kurangnya satu anak disebut **simpul dalam**. Dua simpul dengan induk yang sama disebut **bersaudara**.

Dalam sebuah **pohon biner**, setiap simpul punya paling banyak dua anak, dan keduanya dibedakan dengan nama: **anak kiri** dan **anak kanan**. Pembedaan ini sangat penting. Sebuah simpul yang hanya punya anak kanan adalah pohon yang berbeda dari simpul yang hanya punya anak kiri, walaupun keduanya sama-sama punya satu simpul yang menggantung di bawah satu simpul.

Sebuah **lintasan** dari u ke v adalah barisan simpul yang Anda lewati ketika berjalan dari u turun ke v menyusuri sisi-sisi. Jika lintasan semacam itu ada, u adalah **leluhur** dari v dan v adalah **keturunan** dari u. Setiap simpul adalah leluhur sekaligus keturunan dari dirinya sendiri; ketika kita ingin mengecualikan kasus itu, kita mengatakan leluhur *sejati* atau keturunan *sejati*.

Subpohon yang berakar di x adalah simpul x bersama setiap keturunan x. Inilah gagasan tunggal yang paling berguna dalam bab ini. Sebuah subpohon dengan sendirinya adalah pohon yang sah sepenuhnya, dan hampir setiap algoritma yang akan kita tulis adalah sebuah kalimat berbentuk: *lakukan sesuatu pada akarnya, lalu lakukan hal yang sama pada subpohon kirinya, lalu lakukan hal yang sama pada subpohon kanannya.*

Kedalaman sebuah simpul adalah cacah sisi pada lintasan dari akar turun ke simpul itu. Akar berkedalaman 0; anak-anaknya berkedalaman 1; anak-anak mereka berkedalaman 2. Semua simpul pada kedalaman tertentu bersama-sama membentuk satu **aras**.

Tinggi sebuah simpul adalah cacah sisi pada lintasan terpanjang dari simpul itu turun ke sebuah simpul daun. **Tinggi pohon** adalah tinggi akarnya.

> **Sebuah kesepakatan yang harus Anda pancangkan dalam ingatan.** Kita mendefinisikan tinggi sebuah simpul tunggal tanpa anak sebagai 0, dan tinggi pohon *kosong* — tanpa simpul sama sekali — sebagai -1. Sebagian buku, dan gambar-gambar penghapusan nanti di bab ini (**Gambar 6.21** dan **Gambar 6.22**), justru mencacah *simpul* pada lintasan terpanjang, sehingga satu simpul tunggal bertinggi 1 dan pohon kosong bertinggi 0. Kedua kesepakatan itu berbeda tepat satu di mana-mana. Tidak ada satu pun dalam bab ini yang bergantung pada pilihannya, karena setiap rumus yang kita pedulikan memakai *selisih* dua tinggi, dan $+1$-nya saling meniadakan. Pilih satu, tuliskan di pinggir halaman, dan jangan pernah mencampur keduanya di dalam satu program.

Akhirnya, dua fakta yang membatasi bentuk sebuah pohon biner. Misalkan n adalah cacah simpul dan h adalah tingginya.

Pohon biner tertinggi yang mungkin dengan n simpul adalah satu rantai tunggal: satu simpul per aras, n aras, tinggi $n-1$. Jadi $h \leq n-1$.

Pohon terpendek yang mungkin dengan n simpul mengisi penuh setiap arasnya. Aras 0 memuat paling banyak 1 simpul, aras 1 paling banyak 2, aras 2 paling banyak 4, dan aras d paling banyak 2^d simpul. Sebuah pohon bertinggi h karena itu memuat paling banyak

$$1 + 2 + 4 + \cdots + 2^h = \sum_{d=0}^{h} 2^d = 2^{h+1} - 1$$

simpul, dengan memakai kesamaan jumlahan geometrik dari Bab 3. Dibalik, $n \leq 2^{h+1} - 1$, sehingga $2^{h+1} \geq n + 1$, sehingga

$$h \geq \lg(n+1) - 1.$$

Karena h adalah bilangan bulat, $h \geq \lceil \lg(n+1) \rceil - 1$. Satukan kedua batas itu, dan sebut hasilnya Persamaan (6.1):

$$\lceil \lg(n+1) \rceil - 1 \;\leq\; h \;\leq\; n - 1. \tag{6.1}$$

Persamaan (6.1) adalah seluruh isi bab ini dalam satu baris. Ujung kirinya adalah impian: tinggi kira-kira $\lg n$, jadi kira-kira 20 langkah untuk sejuta kunci. Ujung kanannya adalah mimpi buruk: tinggi n, jadi sejuta langkah untuk sejuta kunci. Sebuah pohon pencarian biner biasa bisa mendarat di mana saja di antara keduanya, bergantung pada tidak lebih daripada urutan kedatangan kuncinya. Pohon AVL adalah pohon pencarian biner yang *dipaksa* tetap berada dekat ujung kiri.

6.3 Apa Itu Pohon Pencarian Biner

Sekarang kita tambahkan aturan yang mengubah pohon biner menjadi pohon yang bisa dicari.

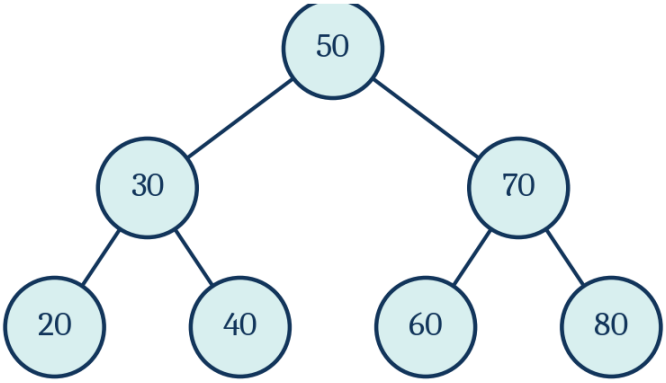
> **DEFINISI (sifat pohon pencarian biner).** Sebuah pohon biner disebut **pohon pencarian biner** jika, untuk setiap simpul x di dalamnya, kedua hal berikut berlaku:

- setiap kunci di subpohon kiri x **lebih kecil daripada** kunci x;
- setiap kunci di subpohon kanan x **lebih besar daripada** kunci x.

> Selanjutnya kita syaratkan, dalam bab ini, bahwa tidak ada dua simpul yang memuat kunci yang sama.

Bacalah definisi itu sekali lagi dan perhatikan dua kata yang mengerjakan seluruh pekerjaannya: **setiap** dan **subpohon**. Aturannya tidak berbunyi "*anak* kirinya lebih kecil". Ia berbunyi setiap kunci di mana pun di bawah dan di sebelah kiri — anak, cucu, cicit, semuanya — lebih kecil. Sebuah simpul tiga aras di bawah, di sebelah kiri akar, harus tetap lebih kecil daripada akar.

Sifat yang sama sering pula dinyatakan secara rekursif dalam empat klausa: cabang kiri hanya memuat kunci yang lebih kecil daripada kunci simpulnya, cabang kanan hanya memuat kunci yang lebih besar, kedua cabang itu sendiri harus juga pohon pencarian biner, dan tidak ada dua simpul yang memuat kunci yang sama. **Gambar 6.1** adalah sebuah ilustrasinya.



Gambar 6.1 Sebuah pohon pencarian biner. Akarnya adalah 50; segala isi subpohon kiri (30,20,40) berada di bawah 50, dan segala isi subpohon kanan (70,60,80) berada di atasnya. Uji yang sama lolos di setiap simpul lainnya.

Mari kita periksa sifat itu pada setiap simpul **Gambar 6.1**, satu simpul satu baris, karena hanya dengan begini cara melihatnya bisa dipelajari.

- Simpul 50 adalah akarnya. Subpohon kirinya memuat {30,20,40}. Apakah setiap satu dari mereka lebih kecil daripada 50? $30 < 50$ ya, $20 < 50$ ya, $40 < 50$ ya. Subpohon kanannya memuat {70,60,80}. Apakah setiap satu lebih besar daripada 50? $70 > 50$ ya, $60 > 50$ ya, $80 > 50$ ya. Simpul 50 lolos.
- Simpul 30 punya subpohon kiri {20} dan subpohon kanan {40}. $20 < 30$ ya; $40 > 30$ ya. Lolos.
- Simpul 70 punya subpohon kiri {60} dan subpohon kanan {80}. $60 < 70$ ya; $80 > 70$ ya. Lolos.

- Simpul 20, 40, 60, dan 80 adalah simpul daun. Simpul daun punya dua subpohon kosong, dan setiap pernyataan tentang anggota himpunan kosong benar dengan sendirinya. Simpul daun selalu lolos.

Setiap simpul lolos, jadi ini adalah pohon pencarian biner.

Sekarang ubah satu hal saja dan saksikan sifat itu patah. Andaikan simpul 40 ternyata 55. Lihat simpul 30: subpohon kanannya akan memuat $\{55\}$, dan $55 > 30$, jadi simpul 30 masih lolos. Tetapi lihat simpul 50: subpohon kirinya akan memuat $\{30,20,55\}$, dan $55 > 50$. Simpul 50 gagal. Inilah jebakan klasiknya, dan ia layak diberi nama sekarang juga: **sebuah pohon bisa memenuhi sifat itu pada setiap pasangan induk–anak dan tetap bukan pohon pencarian biner.** Kita akan menulis program di §6.23 yang menangkap persis kutu ini.

> **Mengapa sifat itu berguna.** Berdirilah di simpul x mana pun lalu bandingkan kunci sasaran k dengan kunci x. Jika k lebih kecil, sifat itu menjamin bahwa k tidak mungkin berada di mana pun di subpohon kanan, sehingga Anda boleh membuang seluruh subpohon kanan tanpa menengoknya sama sekali. Jika k lebih besar, Anda boleh membuang seluruh subpohon kiri. Satu pembandingan menyingkirkan satu sisi penuh. Inilah pencarian biner, tetapi dilakukan pada penunjuk alih-alih pada indeks larik — dan, tidak seperti indeks larik, penunjuk bisa disusun ulang dengan murah ketika koleksinya berubah.

6.4 Sifat Pohon Pencarian Biner Bukan Sifat Heap

Hampir setiap pemula berjumpa dengan pohon pencarian biner dan *binary heap* (tumpukan biner) pada waktu yang hampir bersamaan, dan hampir setiap pemula mengacaukan keduanya sekurang-kurangnya sekali. Lima belas menit untuk memahami bedanya sekarang akan menghemat berjam-jam kebingungan kelak.

Sebuah **tumpukan biner** — yang akan Anda temui selengkapnya ketika kita mempelajari Heapsort — juga berupa pohon biner dengan sebuah aturan tentang kunci, tetapi bentuk aturannya sama sekali berbeda. Dalam **tumpukan-maks**, aturannya berbunyi:

$$\text{kunci(induk)} \;\geq\; \text{kunci(anak)} \quad \text{untuk setiap simpul yang punya induk.}$$

Itulah *seluruh* aturannya. Ia tidak mengatakan apa pun tentang kiri lawan kanan. Anak kiri dan anak kanan sebuah simpul tidak punya urutan tertentu relatif terhadap satu sama lain; mana pun boleh lebih besar. Bandingkan dengan aturan pohon pencarian biner:

$$\text{setiap kunci di kiri}(x) \;<\; \text{kunci}(x) \;<\; \text{setiap kunci di kanan}(x). \tag{6.2}$$

Sifat tumpukan bersifat **tegak dan setempat**: ia mengekang sebuah simpul terhadap anak-anaknya sendiri saja, dan ia tidak mengatakan apa pun tentang kiri dan kanan. Sifat pohon pencarian biner Persamaan (6.2) bersifat **mendatar dan menyeluruh**: ia mengekang sebuah simpul terhadap setiap satu keturunannya, dan kiri dan kanan bermakna berlawanan.

Tabel 6.1 menyandingkan keduanya. Salinlah dengan tangan sekali saja; ini jenis tabel yang hanya melekat kalau dituliskan.

Tabel 6.1 Sifat pohon pencarian biner dibandingkan sifat tumpukan-maks. Keduanya aturan yang berbeda, keduanya mendukung operasi yang berbeda, dan keduanya disimpan dengan cara yang berbeda.

Pertanyaan	Pohon pencarian biner	Tumpukan-maks
Aturannya	subpohon kiri $<$ simpul $<$ subpohon kanan	induk $\geq$ tiap anak
Jangkauan aturannya	semua keturunan	anak langsung saja
Apakah kiri/kanan penting?	ya, keduanya bermakna berlawanan	tidak, kedua anak bisa dipertukarkan
Di mana kunci terkecilnya?	simpul paling kiri, sejauh $O(h)$	entah di mana di antara simpul daun, $O(n)$ untuk menemukannya
Di mana kunci terbesarnya?	simpul paling kanan, sejauh $O(h)$	di akar, $O(1)$
Biaya mencari kunci sembarang k	$O(h)$	$\Theta(n)$ — tidak lebih baik daripada memindai
Apa yang dicetak penelusuran inorder?	kuncinya dalam urutan terurut	tidak ada yang bermakna
Bentuk pohonnya	apa pun yang dihasilkan urutan penyisipan	selalu pohon lengkap, tinggi tepat $\lfloor \lg n \rfloor$
Penyimpanan yang lazim	simpul dan penunjuk	larik biasa, tanpa penunjuk sama sekali

Dua baris terakhir layak diberi satu kalimat masing-masing. Bentuk sebuah tumpukan sudah dipatok oleh cara membangunnya — ia selalu diisi aras demi aras, kiri ke kanan — sehingga tingginya *dengan sendirinya* $\lfloor \lg n \rfloor$ dan penyeimbangan tidak pernah diperlukan. Sebuah pohon pencarian biner membeli kesanggupan mencari kunci sembarang, dan membayarnya dengan bentuk yang bergantung pada riwayat. Pembayaran itulah persoalan yang dipecahkan pohon AVL.

> **Jebakannya dalam satu kalimat.** Jika seseorang menunjukkan kepada Anda sebuah pohon yang setiap induknya lebih besar daripada kedua anaknya lalu meminta Anda mencari kunci 37 di dalamnya, Anda tidak punya pilihan selain menengok setiap simpul. Tumpukan dibuat untuk berulang kali mengambil nilai ekstremnya; pohon pencarian dibuat untuk mencari sesuatu. Memakai yang keliru menghabiskan biaya sebesar faktor n.

6.5 Merepresentasikan Pohon Pencarian Biner dalam C++

Sebuah simpul memerlukan tiga hal: kuncinya, sebuah penunjuk ke subpohon kirinya, dan sebuah penunjuk ke subpohon kanannya. Beberapa algoritma juga menginginkan penunjuk balik ke induknya; kita akan memperkenalkannya hanya di tempat ia sungguh-sungguh diperlukan, yaitu di §6.9. **Kode 6.1** mendeklarasikan simpul semacam itu.

Kode 6.1 Simpul sebuah pohon pencarian biner, dan pohon **Gambar 6.1** yang dirakit dengan tangan dari tujuh simpul. Kompilasi dengan `g++ -std=c++17 -O2 -o node node.cpp`.

```
// Simpul pohon pencarian biner, dan pohon pada gambar pertama yang dirakit dengan tangan.
#include <iostream>

struct Node {
    int   key;
    Node* left;
    Node* right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

void inorder(const Node* x) {                  // subpohon kiri, simpul, subpohon kanan
    if (!x) return;                            // pohon kosong: tidak ada yang dicetak
    inorder(x->left);
    std::cout << x->key << " ";
    inorder(x->right);
}

int main() {
    Node* root = new Node(50);                 // seluruh pohon adalah penunjuk ke akarnya
    root->left  = new Node(30);
    root->right = new Node(70);
    root->left->left   = new Node(20);
    root->left->right  = new Node(40);
    root->right->left  = new Node(60);
    root->right->right = new Node(80);

    std::cout << "root " << root->key << ", left child " << root->left->key
              << ", right child " << root->right->key << "\n";
    std::cout << "left child of 20 is " << (root->left->left->left ? "a node" : "nullptr")
              << " (the empty tree)\n";
    std::cout << "inorder: ";
    inorder(root);
    std::cout << "\n";

    Node* nodes[] = {root->left->left, root->left->right, root->right->left,
                     root->right->right, root->left, root->right, root};
    for (Node* p : nodes) delete p;            // anak dihapus sebelum induknya
    return 0;
}
```

Kode 6.1 mendeklarasikan simpulnya lalu merangkai ketujuh simpul **Gambar 6.1** satu penunjuk demi satu penunjuk, supaya Anda dapat melihat sebuah pohon tersusun dari tidak lebih daripada kunci dan dua penunjuk per simpul. Program itu mencetak:

```
root 50, left child 30, right child 70
left child of 20 is nullptr (the empty tree)
inorder: 20 30 40 50 60 70 80
```

Pohon kosong direpresentasikan oleh penunjuk nol `nullptr`. Ini bukan rincian yang boleh dilewati begitu saja: ia adalah kasus basis dari hampir setiap rekursi dalam bab ini. Subpohon yang tidak ada adalah subpohon bertinggi -1 yang tidak memuat kunci apa pun, dan setiap klaim yang kita buat tentang "semua kunci di subpohon ini" benar dengan sendirinya baginya.

Sebuah pohon utuh kemudian tidak lebih daripada sebuah penunjuk ke simpul akarnya, dan `nullptr` adalah pohon utuh yang sah sepenuhnya yang tidak berisi apa-apa.

6.6 Menelusuri Pohon: Tiga Penelusuran

Menelusuri sebuah pohon berarti mengunjungi setiap simpul tepat satu kali. Untuk pohon biner ada tiga urutan klasik, dan ketiganya hanya berbeda pada *kapan* akar sebuah subpohon dikunjungi relatif terhadap kedua subpohonnya.

DEFINISI (ketiga penelusuran mendalam-dahulu). Untuk menelusuri subpohon yang berakar di x:

- **inorder** — telusuri subpohon kiri, lalu kunjungi x, lalu telusuri subpohon kanan;
- **preorder** — kunjungi x, lalu telusuri subpohon kiri, lalu telusuri subpohon kanan;
- **postorder** — telusuri subpohon kiri, lalu telusuri subpohon kanan, lalu kunjungi x.

Dalam ketiga hal itu, menelusuri pohon kosong tidak melakukan apa-apa sama sekali.

Namanya menggambarkan di mana akarnya jatuh: *pre*-order menaruhnya sebelum kedua subpohonnya, *in*-order menaruhnya di antara keduanya, *post*-order menaruhnya sesudah keduanya. Dalam ketiganya subpohon kiri diurus sebelum subpohon kanan.

Mari kita telusuri ketiganya pada pohon di **Gambar 6.1**, dengan tangan, sampai tuntas.

Inorder, dimulai di akar 50.

1. Kunjungi subpohon kiri 50, yang berakar di 30. Tahan dulu 50.
2. Di 30: kunjungi subpohon kirinya, yang berakar di 20. Tahan dulu 30.
3. Di 20: subpohon kirinya kosong, jadi tidak terjadi apa-apa. Sekarang keluarkan **20**. Subpohon kanannya kosong, jadi tidak terjadi apa-apa. Kembali.
4. Kembali di 30: subpohon kirinya sudah selesai. Keluarkan **30**. Sekarang subpohon kanannya, yang berakar di 40.
5. Di 40: kiri kosong; keluarkan **40**; kanan kosong. Kembali.
6. Kembali di 50: subpohon kirinya sudah tuntas. Keluarkan **50**. Sekarang subpohon kanannya, yang berakar di 70.
7. Di 70: subpohon kirinya dulu, yang berakar di 60. Keluarkan **60**. Lalu keluarkan **70**. Lalu subpohon kanannya, yang berakar di 80: keluarkan **80**.

Keluarannya adalah 20,30,40,50,60,70,80 — terurut.

Preorder, pada pohon yang sama. Akarnya dikeluarkan lebih dulu pada setiap aras: 50; lalu seluruh subpohon kiri secara preorder, yang mengeluarkan 30, lalu 20, lalu 40; lalu seluruh subpohon kanan secara preorder, yang mengeluarkan 70, lalu 60, lalu 80. Keluarannya adalah 50,30,20,40,70,60,80.

Postorder. Setiap akar menunggu kedua subpohonnya. Di 50: subpohon kiri dulu. Di dalamnya, di 30: subpohon kiri memberi 20, subpohon kanan memberi 40, lalu 30 itu sendiri. Jadi paruh kirinya mengeluarkan 20,40,30. Lalu paruh kanannya mengeluarkan 60,80,70. Lalu akhirnya 50. Keluarannya adalah 20,40,30,60,80,70,50.

Kode 6.2 membangun pohonnya dan mencetak ketiga penelusurannya, dan juga menggambar pohonnya dalam keadaan miring supaya Anda bisa mencocokkan bentuknya dengan gambarnya. Putar gambar cetakan itu sembilan puluh derajat searah jarum jam dan ia menjadi **Gambar 6.1**.

Kode 6.2 Ketiga penelusuran dalam C++. Simpan sebagai `traversal.cpp`, kompilasi dengan `g++ -std=c++17 -O2 -o traversal traversal.cpp`, lalu jalankan `./traversal`.

```
// Program: ketiga penelusuran mendalam-dahulu pada pohon pencarian biner.
// Membangun pohon pada gambar pertama lalu menelusurinya dengan tiga cara.
#include <iostream>
#include <string>
#include <vector>

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

// Sisipkan satu kunci, sifat pohon pencarian biner tetap terjaga. Kunci kembar diabaikan.
Node* insertKey(Node* root, int key) {
    if (root == nullptr) return new Node(key);
    if (key < root->key)      root->left  = insertKey(root->left,  key);
    else if (key > root->key) root->right = insertKey(root->right, key);
    return root;
}

void inorder(Node* x, std::vector<int>& out) {
    if (x == nullptr) return;
    inorder(x->left, out);
    out.push_back(x->key);
    inorder(x->right, out);
}

void preorder(Node* x, std::vector<int>& out) {
    if (x == nullptr) return;
    out.push_back(x->key);
    preorder(x->left, out);
    preorder(x->right, out);
}

void postorder(Node* x, std::vector<int>& out) {
    if (x == nullptr) return;
    postorder(x->left, out);
    postorder(x->right, out);
    out.push_back(x->key);
}
```

Kode 6.2 (Lanjutan) *Ketiga penelusuran dalam C++. Simpan sebagai traversal.cpp, kompilasi dengan g++ -std=c++17 -O2 -o traversal traversal.cpp, lalu jalankan ./traversal.*

```
void printSeq(const std::string& label, const std::vector<int>& v) {
    std::cout << label;
    for (size_t i = 0; i < v.size(); ++i) {
        if (i) std::cout << ", ";
        std::cout << v[i];
    }
    std::cout << "\n";
}

// Gambar pohon yang dimiringkan: subpohon kanan di atas, akar, lalu subpohon kiri.
void draw(Node* x, int depth) {
    if (x == nullptr) return;
    draw(x->right, depth + 1);
    std::cout << std::string(4 * depth, ' ') << x->key << "\n";
    draw(x->left, depth + 1);
}

int main() {
    Node* root = nullptr;
    int keys[] = {50, 30, 70, 20, 40, 60, 80};
    for (int k : keys) root = insertKey(root, k);

    std::cout << "The tree, drawn sideways (root at the left margin):\n";
    draw(root, 0);
    std::cout << "\n";

    std::vector<int> a, b, c;
    inorder(root, a);
    preorder(root, b);
    postorder(root, c);
    printSeq("inorder   : ", a);
    printSeq("preorder  : ", b);
    printSeq("postorder : ", c);

    bool sorted = true;
    for (size_t i = 1; i < a.size(); ++i)
        if (a[i - 1] >= a[i]) sorted = false;
    std::cout << "\ninorder output is sorted: " << (sorted ? "yes" : "no") << "\n";
    return 0;
}
```

Menjalankannya mencetak persis ini:

```
The tree, drawn sideways (root at the left margin):
        80
    70
        60
50
        40
    30
        20

inorder   : 20, 30, 40, 50, 60, 70, 80
preorder  : 50, 30, 20, 40, 70, 60, 80
postorder : 20, 40, 30, 60, 80, 70, 50

inorder output is sorted: yes
```

Setiap satu dari ketiga barisnya cocok dengan penelusuran tangan di atas.

6.6.1 Mengapa penelusuran inorder menghasilkan urutan terurut — sebuah bukti

Programnya berkata "yes", tetapi program yang berkata ya adalah bukti empiris, bukan pembuktian. Inilah pembuktiannya, dan ia pendek karena definisinya sendiri yang mengerjakan seluruh pekerjaannya.

Klaim. Jika T adalah pohon pencarian biner, penelusuran inorder atas T mengeluarkan kunci-kuncinya dalam urutan naik tegas.

Bukti, dengan induksi atas cacah simpul n di dalam T.

Kasus basis, $n = 0$. Penelusurannya tidak mengeluarkan apa pun. Barisan kosong naik secara remeh.

Langkah induktif. Andaikan klaim itu berlaku untuk setiap pohon pencarian biner dengan simpul kurang dari n, dan misalkan T punya $n \geq 1$ simpul dengan akar x. Tuliskan L untuk subpohon kiri x dan R untuk subpohon kanannya.

Pertama, L dan R sendiri adalah pohon pencarian biner. Ini langsung terlihat: sifat pohon pencarian biner adalah pernyataan tentang *setiap* simpul, jadi ia berlaku khususnya pada setiap simpul L dan setiap simpul R, dan subpohon sebuah simpul di dalam L adalah subpohon yang sama dengan ketika ia berada di dalam T. Baik L maupun R punya simpul kurang dari n, karena x bukan milik keduanya. Jadi hipotesis induksinya berlaku bagi keduanya.

Penelusurannya mengeluarkan, secara berurutan: penelusuran inorder atas L, lalu kunci x, lalu penelusuran inorder atas R. Menurut hipotesis induksi, blok pertamanya naik dan blok ketiganya naik. Yang tersisa adalah memeriksa kedua sambungannya.

Pada sambungan pertama: kunci terakhir yang dicetak dari L adalah suatu kunci milik L, dan menurut sifat pohon pencarian biner di x, *setiap* kunci L lebih kecil daripada kunci(x). Jadi kunci terakhir blok pertama lebih kecil daripada kunci(x).

Pada sambungan kedua: kunci pertama yang dicetak dari R adalah suatu kunci milik R, dan menurut sifat pohon pencarian biner di x, setiap kunci R lebih besar daripada kunci(x). Jadi kunci(x) lebih kecil daripada kunci pertama blok ketiga.

Seluruh keluarannya karena itu naik, dan itu melengkapkan induksinya. ■

> **Sebuah algoritma pengurutan gratis.** Sisipkan n kunci ke dalam pohon pencarian biner, lalu telusuri ia secara inorder. Kuncinya keluar dalam keadaan terurut. Inilah *tree sort* (pengurutan pohon). Ongkosnya adalah biaya n penyisipan ditambah satu penelusuran $\Theta(n)$, jadi ia $O(nh)$ — yang berarti $\Theta(n\lg n)$ ketika pohonnya tetap pendek, dan $\Theta(n^2)$ ketika tidak. Anda kini punya alasan yang sangat nyata untuk memedulikan tinggi sebuah pohon: ia adalah selisih antara algoritma pengurutan yang sangat baik dan algoritma pengurutan yang buruk.

Tabel 6.2 mencatat untuk apa sesungguhnya tiap penelusuran dipakai, karena ketiganya memang muncul dalam praktik.

Tabel 6.2 Ketiga penelusuran mendalam-dahulu dan pekerjaan yang dikerjakan masing-masingnya.

Penelusuran	Urutan	Gunanya untuk apa
inorder	kiri, simpul, kanan	mendaftar kunci dalam urutan terurut; kueri rentang; memeriksa bahwa sebuah pohon sungguh pohon pencarian biner
preorder	simpul, kiri, kanan	menyalin atau menyerialkan pohon — menyisipkan ulang barisan preorder membangun kembali bentuk yang persis sama
postorder	kiri, kanan, simpul	memusnahkan pohon (bebaskan anak-anaknya sebelum induknya); menghitung nilai di sebuah simpul yang bergantung pada kedua subpohonnya, misalnya tinggi

Tiap penelusuran mengunjungi setiap simpul satu kali dan melakukan kerja $O(1)$ di sana, jadi masing-masingnya berongkos $\Theta(n)$ — apa pun bentuk pohonnya. Penelusuran adalah satu-satunya operasi dalam bab ini yang ongkosnya *tidak* bergantung pada tinggi.

6.7 Pencarian

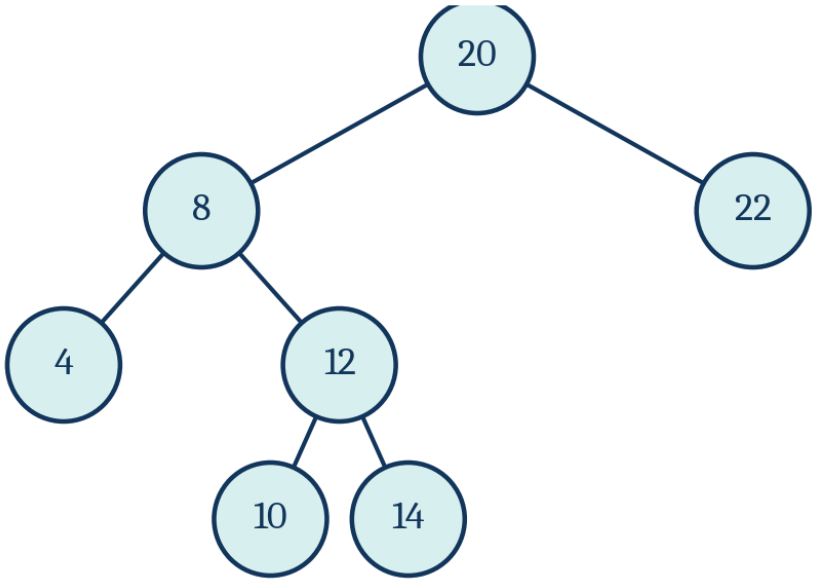
Pencarian adalah operasi yang menjadi asal nama strukturnya, dan panjangnya empat baris.

TREE-SEARCH(x, k)**.** Untuk mencari kunci k di dalam subpohon yang berakar di x:

1. Jika x kosong, kuncinya tidak ada — laporkan gagal.
2. Jika k sama dengan kunci x, laporkan berhasil lalu kembalikan x.
3. Jika k lebih kecil daripada kunci x, cari di subpohon kiri x.
4. Selain itu, cari di subpohon kanan x.

Mengapa langkah 3 boleh mengabaikan subpohon kanan sepenuhnya? Karena sifat pohon pencarian biner berkata *setiap* kunci di subpohon kanan x lebih besar daripada kunci(x), sedangkan $k <$ kunci(x), jadi tidak ada kunci di subpohon kanan yang bisa sama dengan k. Langkah 4 adalah bayangan cerminnya. Tiap pembandingan menyingkirkan satu sisi penuh.

Mari kita telusuri TREE-SEARCH pada pohon **Gambar 6.2**: akar 20, dengan anak kiri 8 dan anak kanan 22; simpul 8 punya anak 4 dan 12; simpul 12 punya anak 10 dan 14.



Gambar 6.2 Sebuah pohon untuk berlatih mencari penerus inorder. Menelusurinya secara inorder memberi 4,8,10,12,14,20,22, dan membaca daftar itu adalah cara termudah untuk memeriksa klaim penerus atau pendahulu mana pun.

Penelusuran SEARCH(10)**.**

1. Mulai di akar, 20. Apakah $10 = 20$? Tidak. Apakah $10 < 20$? Ya, jadi ke kiri.
2. Sekarang di 8. Apakah $10 = 8$? Tidak. Apakah $10 < 8$? Tidak, jadi ke kanan.
3. Sekarang di 12. Apakah $10 = 12$? Tidak. Apakah $10 < 12$? Ya, jadi ke kiri.
4. Sekarang di 10. Apakah $10 = 10$? Ya. Ketemu, sesudah mengunjungi empat simpul: $20 \to 8 \to 12 \to 10$.

Penelusuran SEARCH(13), sebuah kunci yang tidak ada.

1. Di 20: $13 < 20$, ke kiri.

2. Di 8: $13 > 8$, ke kanan.
3. Di 12: $13 > 12$, ke kanan.
4. Di 14: $13 < 14$, ke kiri.
5. Anak kiri 14 kosong. Laporkan gagal, sesudah mengunjungi empat simpul.

Perhatikan apa yang tetap diberitahukan oleh pencarian yang gagal itu: 13 akan menempati persis tempat kita terjatuh dari pohonnya, yaitu sebagai anak kiri 14. Itu bukan kebetulan, dan §6.10 memanfaatkannya.

Biaya. Tiap langkah bergerak tepat satu aras turun dan melakukan sejumlah tetap pembandingan, jadi cacah langkahnya paling banyak sepanjang lintasan akar-ke-daun terpanjang ditambah satu, yaitu $h + 1$. Waktu jalannya $O(h)$.

6.8 Minimum dan Maksimum

Di mana kunci terkecilnya? Tanyakan pada sifatnya. Setiap kunci yang lebih kecil daripada akar tinggal di subpohon kiri, jadi kalau subpohon kirinya tidak kosong, minimumnya ada di sana. Ulangi. Rekursinya baru berhenti ketika tidak ada lagi subpohon kiri untuk dimasuki.

> **TREE-MINIMUM**(x)**.** Selama x punya anak kiri, gantilah x dengan anak kirinya. Kembalikan x.
>
> **TREE-MAXIMUM**(x)**.** Selama x punya anak kanan, gantilah x dengan anak kanannya. Kembalikan x.

Pada **Gambar 6.2**: MINIMUM mulai di 20, ke kiri ke 8, ke kiri ke 4; simpul 4 tidak punya anak kiri, jadi 4 adalah minimumnya. Tiga simpul dikunjungi. MAXIMUM mulai di 20, ke kanan ke 22; simpul 22 tidak punya anak kanan, jadi 22 adalah maksimumnya. Dua simpul dikunjungi.

Keduanya menyusuri satu lintasan menurun tunggal, jadi keduanya berongkos $O(h)$.

> **Jangan mengacaukan "paling kiri" dengan "paling dalam di sebelah kiri."** Minimumnya adalah simpul yang Anda capai dengan selalu membelok ke kiri. Ia tidak harus simpul terdalam di pohonnya, dan ia tidak harus simpul daun — ia sangat mungkin punya anak kanan. Pada **Gambar 6.2**, andaikan simpul 4 punya anak kanan 6, maka 4 akan tetap menjadi minimumnya dan tetap bukan simpul daun.

6.9 Penerus dan Pendahulu

> **DEFINISI. Penerus inorder** sebuah simpul x dalam pohon pencarian biner adalah simpul yang memuat kunci terkecil yang masih lebih besar daripada kunci(x). Setara dengan itu, ia adalah simpul yang datang tepat sesudah x dalam penelusuran inorder. Kunci terbesar di pohonnya tidak punya penerus. **Pendahulu inorder** adalah bayangan cerminnya: simpul dengan kunci terbesar yang lebih kecil daripada kunci(x), simpul yang tepat sebelum x dalam penelusuran inorder, dan kunci terkecil tidak punya pendahulu.

Berikut tiga contoh, yang bisa kita periksa cukup dengan membaca penelusuran inorder **Gambar 6.2**, yaitu 4,8,10,12,14,20,22:

- penerus 8 adalah 10 — dan memang 10 mengikuti 8 dalam daftar itu;
- penerus 10 adalah 12;
- penerus 14 adalah 20.

Menemukan penerus tanpa menuliskan seluruh penelusurannya terbelah menjadi dua kasus, dan belahan itulah bagian yang menarik.

Kasus 1: x punya subpohon kanan. Maka penerusnya adalah *minimum dari subpohon kanan itu*. Mengapa? Setiap kunci di subpohon kanan lebih besar daripada kunci(x), jadi semuanya adalah calon; penerusnya adalah calon terkecil; dan kunci terkecil di sebuah subpohon ditemukan oleh TREE-MINIMUM. Dan tidak ada kunci di luar subpohon kanan yang bisa menyelinap di antaranya: sembarang kunci yang lebih besar daripada kunci(x) yang tidak berada di subpohon kanan pasti terletak di atas x, dan setiap kunci semacam itu lebih besar daripada *segala isi* subpohon kanan x.

Pada **Gambar 6.2**, ambil $x = 8$. Subpohon kanannya berakar di 12. TREE-MINIMUM atas subpohon itu berjalan 12 → 10 lalu berhenti. Penerus 8 adalah 10. ✓

Kasus 2: x tidak punya subpohon kanan. Maka penerusnya adalah leluhur terendah x yang subpohon *kiri*-nya memuat x. Dalam praktik: memanjatlah ke atas dari x; setiap kali Anda tiba di sebuah induk dari anak *kanan*-nya, teruslah memanjat; pertama kali Anda tiba di sebuah induk dari anak *kiri*-nya, induk itulah penerusnya. Jika Anda memanjat sampai lepas dari puncak pohonnya, berarti x adalah maksimumnya dan tidak punya penerus.

Mengapa ini bekerja? Jika Anda datang naik dari anak kiri, kunci induknya lebih besar daripada segala isi subpohon yang baru saja Anda tinggalkan, termasuk kunci(x), dan ia adalah kunci semacam itu yang pertama Anda temui dalam perjalanan naik — jadi ia adalah kunci terkecil yang lebih besar daripada kunci(x). Jika Anda datang naik dari anak kanan, kunci induknya *lebih kecil* daripada kunci(x), jadi ia sama sekali bukan calon.

Pada **Gambar 6.2**, ambil $x = 14$. Ia tidak punya subpohon kanan. Memanjat: 14 adalah anak kanan 12 — datang dari kanan, terus. 12 adalah anak kanan 8 — datang dari kanan, terus. 8 adalah anak *kiri* 20 — berhenti. Penerus 14 adalah 20. ✓

Ambil $x = 22$. Tidak punya subpohon kanan. Memanjat: 22 adalah anak kanan 20 — terus. Di atas 20 tidak ada apa-apa. Jadi 22 tidak punya penerus. ✓ Ia adalah maksimumnya.

Kasus 2 memerlukan cara bergerak *ke atas*, yang tidak bisa dilakukan simpul tiga-medan kita. Ada dua perbaikan baku. Yang pertama adalah menambahkan penunjuk `parent` pada setiap simpul, dan itulah yang dilakukan **Kode 6.3**. Yang kedua adalah membawa "calon terbaik sejauh ini" turun dari akar selama pencarian biasa — setiap kali Anda membelok ke kiri, simpul yang Anda tinggalkan menjadi calon terbaik yang baru. Yang kedua tidak memerlukan memori tambahan dan itulah yang akan kita pakai dalam studi kasus Profitina Laundry di §6.22.

Kode 6.3 mengimplementasikan kelima operasi kueri itu dan mencetak lintasan yang ditempuh masing-masingnya.

Kode 6.3 Pencarian, minimum, maksimum, penerus, dan pendahulu, masing-masing melaporkan lintasan yang ditempuhnya. Kompilasi dengan `g++ -std=c++17 -O2 -o query query.cpp`.

```
// Program: kelima operasi kueri pohon pencarian biner, masing-masing
// mencetak lintasan yang ditempuhnya, pada pohon penerus di Subbab 6.7.
#include <iostream>
#include <string>
#include <vector>

struct Node {
    int key;
    Node* left;
    Node* right;
    Node* parent;                       // dibutuhkan oleh SUCCESSOR dan PREDECESSOR
    Node(int k) : key(k), left(nullptr), right(nullptr), parent(nullptr) {}
};

Node* insertKey(Node* root, int key) {
    Node* z = new Node(key);
    Node* y = nullptr;                  // y membuntuti x, jadi ia berakhir di induk z
    Node* x = root;
    while (x != nullptr) {
        y = x;
        if (key < x->key) x = x->left;
        else              x = x->right;
    }
    z->parent = y;
    if (y == nullptr)            return z;      // pohonnya tadi kosong
    if (key < y->key) y->left  = z;
    else              y->right = z;
    return root;
}

Node* treeSearch(Node* x, int k, std::vector<int>& path) {
    while (x != nullptr && x->key != k) {
        path.push_back(x->key);
        if (k < x->key) x = x->left;
        else            x = x->right;
    }
    if (x != nullptr) path.push_back(x->key);
    return x;
}

Node* treeMinimum(Node* x, std::vector<int>& path) {
    while (x->left != nullptr) { path.push_back(x->key); x = x->left; }
    path.push_back(x->key);
    return x;
}
```

Kode 6.3 (Lanjutan) *Pencarian, minimum, maksimum, penerus, dan pendahulu, masing-masing melaporkan lintasan yang ditempuhnya. Kompilasi dengan g++ -std=c++17 -O2 -o query query.cpp.*

```
Node* treeMaximum(Node* x, std::vector<int>& path) {
    while (x->right != nullptr) { path.push_back(x->key); x = x->right; }
    path.push_back(x->key);
    return x;
}

Node* treeSuccessor(Node* x) {
    if (x->right != nullptr) {          // kasus 1: kunci terkecil di subpohon kanan
        Node* m = x->right;
        while (m->left != nullptr) m = m->left;
        return m;
    }
    Node* y = x->parent;                // kasus 2: naik sampai kita datang dari anak kiri
    while (y != nullptr && x == y->right) { x = y; y = y->parent; }
    return y;
}

Node* treePredecessor(Node* x) {
    if (x->left != nullptr) {
        Node* m = x->left;
        while (m->right != nullptr) m = m->right;
        return m;
    }
    Node* y = x->parent;
    while (y != nullptr && x == y->left) { x = y; y = y->parent; }
    return y;
}

void show(const std::string& label, const std::vector<int>& path) {
    std::cout << label;
    for (size_t i = 0; i < path.size(); ++i) {
        if (i) std::cout << " -> ";
        std::cout << path[i];
    }
    std::cout << "   (" << path.size() << " nodes visited)\n";
}

int main() {
    Node* root = nullptr;
    int keys[] = {20, 8, 22, 4, 12, 10, 14};
    for (int k : keys) root = insertKey(root, k);
```

Kode 6.3 (Lanjutan) *Pencarian, minimum, maksimum, penerus, dan pendahulu, masing-masing melaporkan lintasan yang ditempuhnya. Kompilasi dengan g++ -std=c++17 -O2 -o query query.cpp.*

```
    std::vector<int> p;
    treeSearch(root, 10, p);  show("SEARCH(10)  path: ", p);
    p.clear();
    treeSearch(root, 13, p);  show("SEARCH(13)  path: ", p);
    p.clear();
    treeMinimum(root, p);     show("MINIMUM     path: ", p);
    p.clear();
    treeMaximum(root, p);     show("MAXIMUM     path: ", p);

    std::cout << "\n";
    int probe[] = {8, 10, 14, 4, 22};
    for (int k : probe) {
        std::vector<int> q;
        Node* x = treeSearch(root, k, q);
        Node* s = treeSuccessor(x);
        Node* t = treePredecessor(x);
        std::cout << "key " << k
                  << ": successor = "   << (s ? std::to_string(s->key) :
std::string("none"))
                  << ", predecessor = " << (t ? std::to_string(t->key) :
std::string("none"))
                  << "\n";
    }
    return 0;
}
```

Programnya mencetak:

```
SEARCH(10)  path: 20 -> 8 -> 12 -> 10   (4 nodes visited)
SEARCH(13)  path: 20 -> 8 -> 12 -> 14   (4 nodes visited)
MINIMUM     path: 20 -> 8 -> 4   (3 nodes visited)
MAXIMUM     path: 20 -> 22   (2 nodes visited)

key 8: successor = 10, predecessor = 4
key 10: successor = 12, predecessor = 8
key 14: successor = 20, predecessor = 12
key 4: successor = 8, predecessor = none
key 22: successor = none, predecessor = 20
```

Setiap lintasannya cocok dengan penelusuran tangan tadi, dan ketiga klaim penerusnya cocok persis dengan contoh-contoh di §6.9. Perhatikan dua baris terakhirnya: minimum 4 tidak punya pendahulu dan maksimum 22 tidak punya penerus, sebagaimana dituntut definisinya.

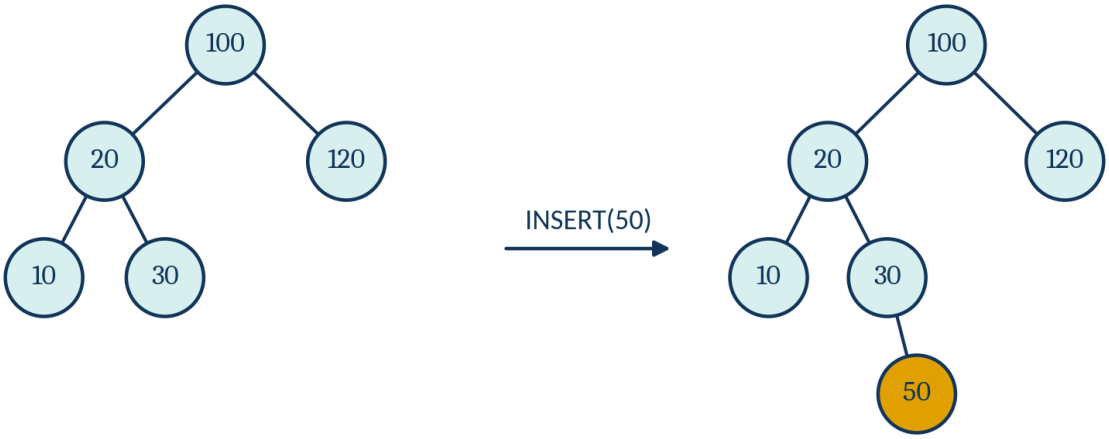
Biaya. Kasus 1 menyusuri satu lintasan ke bawah; kasus 2 menyusuri satu lintasan ke atas. Tak satu pun pernah berganti arah. Jadi SUCCESSOR dan PREDECESSOR keduanya berongkos $O(h)$.

6.10 Penyisipan

Pencarian yang gagal memberi tahu Anda di mana sebuah kunci *akan* berada. Penyisipan tidak lebih daripada melakukan pencarian yang gagal itu lalu menempelkan simpul baru di tempat Anda terjatuh.

> **TREE-INSERT**(T, k)**.** Menyusurlah turun dari akar persis sebagaimana TREE-SEARCH akan melakukannya, sambil mengingat simpul yang sedang Anda pijak ketika Anda meninggalkannya. Ketika Anda melangkah lepas dari dasar pohonnya — yaitu ketika anak yang hendak Anda kunjungi ternyata kosong — ciptakan simpul baru yang memuat k lalu jadikan ia anak tersebut. Simpul yang baru disisipkan selalu berupa simpul daun.

Tiga fakta merangkumnya: unsur baru selalu disisipkan sebagai simpul daun; tempatnya ditemukan dengan menyusur turun dari akar sampai sebuah posisi kosong tercapai; dan simpul barunya lalu ditempelkan sebagai anak di sana. **Gambar 6.3** mengilustrasikan `Insert(50)`.



Gambar 6.3 Sebuah contoh penyisipan. Kunci 50 dijahitkan turun menyusuri pohon — ke kiri di 100, ke kanan di 20, ke kanan di 30 — dan mendarat sebagai anak kanan baru milik 30, ditandai dengan warna kuning tua. Tidak ada hal lain di pohon itu yang bergeser.

Telusuri `Insert(50)` pada pohon sebelah kiri **Gambar 6.3**, satu pembandingan satu baris.

1. Mulai di akar, 100. Apakah $50 < 100$? Ya. Ke kiri. Ingat 100.
2. Sekarang di 20. Apakah $50 < 20$? Tidak. Ke kanan. Ingat 20.
3. Sekarang di 30. Apakah $50 < 30$? Tidak. Ke kanan. Ingat 30.
4. Anak kanan 30 kosong. Berhenti. Ciptakan simpul yang memuat 50 lalu jadikan ia anak kanan 30.

Hasilnya adalah pohon sebelah kanan **Gambar 6.3**, dengan 50 diarsir untuk menandainya sebagai yang baru.

Periksa sifatnya sesudah itu, karena inilah langkah yang sering dilewati orang. Simpul 30 kini punya subpohon kanan $\{50\}$ dan $50 > 30$ ✓. Simpul 20 kini punya subpohon kanan $\{30,50\}$ dan keduanya lebih besar daripada 20 ✓. Simpul 100 kini punya subpohon kiri $\{20,10,30,50\}$ dan keempatnya lebih kecil daripada 100 ✓. Sifatnya selamat, dan alasan ia selamat justru karena lintasan menurunnya dipilih oleh pembandingan yang sama dengan yang dilakukan sifat itu.

> **Mengapa simpul baru selalu berupa simpul daun.** Penyusurannya baru berhenti ketika ia hendak melangkah masuk ke subpohon kosong. Jadi posisi yang kita isi tadinya kosong, dan simpul yang ditaruh di posisi kosong tidak punya anak. Satu fakta inilah yang membuat penyisipan murah: kita tidak pernah harus memindahkan simpul yang sudah ada, hanya menempel pada satu simpul.

Biaya. Penyusurannya adalah lintasan akar-ke-daun, jadi penyisipan berongkos $O(h)$, ditambah $O(1)$ untuk mengalokasikan dan menempelkan simpulnya.

Urutan itu penting, sangat penting. Sisipkan kunci 1,2,3,4,5 ke dalam pohon kosong dalam urutan itu dan setiap satunya pergi ke kanan, menghasilkan rantai bertinggi 4. Sisipkan kelima kunci yang sama dalam urutan 3,1,4,2,5 dan Anda mendapat pohon bertinggi 2. Kunci yang sama, algoritma yang sama, struktur yang sama sekali berbeda. Simpan pikiran itu sampai §6.13.

6.11 Penghapusan: Tiga Kasusnya

Penghapusan adalah satu-satunya operasi dalam bab ini yang punya analisis kasus yang sungguhan, dan ia ditata dengan cara yang baku: menurut berapa banyak anak yang dipunyai simpul yang hendak dilenyapkan. Ada tiga kemungkinan — tidak punya, satu, atau dua — dan ketiganya sungguh-sungguh persoalan yang berlainan.

6.11.1 Kasus 1 — simpulnya adalah simpul daun

Jika simpulnya tidak punya anak, tidak ada apa pun yang bergantung padanya. Lepaskan ia dari induknya lalu bebaskan.

Gambar 6.4 Kasus 1 penghapusan: simpul yang hendak dihapus, 20, adalah simpul daun. Ia cukup dibuang; sisa pohonnya tidak tersentuh.

Pada **Gambar 6.4** pohonnya adalah 50 di akar, dengan 30 dan 70 di bawahnya, serta simpul daun 20,40,60,80. Untuk menghapus 20:

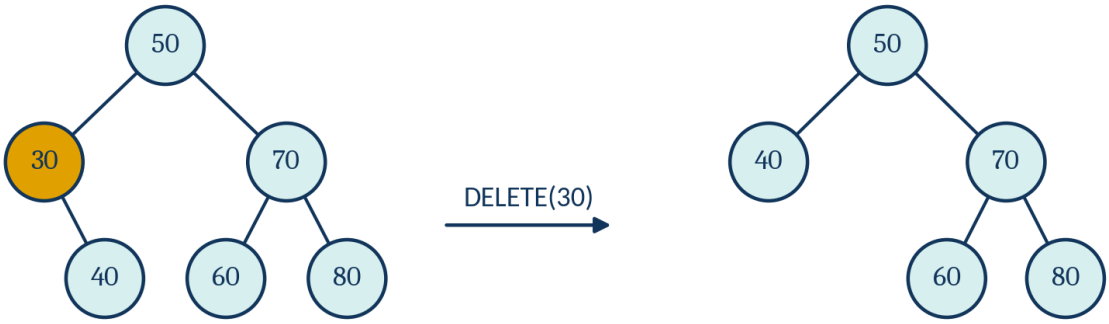
1. Cari 20: di 50, ke kiri; di 30, ke kiri; ketemu 20.
2. Simpul 20 tidak punya anak kiri dan tidak punya anak kanan, jadi ia simpul daun.
3. Jadikan anak kiri induknya, 30, kosong, lalu bebaskan simpulnya.

Sesudah itu penelusuran inordernya adalah 30,40,50,60,70,80 — daftar terurut dengan 20 dibuang, dan tidak ada hal lain yang terganggu.

6.11.2 Kasus 2 — simpulnya punya tepat satu anak

Jika simpulnya punya tepat satu anak, maka seluruh subpohon yang menggantung di bawahnya berada di satu sisi. Naikkan pangkat subpohon itu: jadikan ia anak dari induk simpul yang dihapus, pada posisi lama simpul yang dihapus itu.

Bedah ini dapat juga diungkapkan sebagai “salin anaknya ke simpul yang hendak dihapus, lalu hapus anaknya”, yang merupakan bedah yang sama dilihat dari ujung yang lain dan sama benarnya. Kita akan mengungkapkannya sebagai kenaikan pangkat, karena kenaikan pangkat itulah yang dikerjakan kode C++-nya.



Gambar 6.5 Kasus 2 penghapusan: simpul 30 hanya punya anak kanan, yaitu 40. Menghapus 30 menaikkan 40 ke tempatnya.

Melanjutkan dari pohon **Gambar 6.5** — yaitu pohon sebelumnya sesudah 20 dibuang — hapuslah 30:

1. Cari 30: di 50, ke kiri; ketemu 30.
2. Simpul 30 tidak punya anak kiri, tetapi punya anak kanan 40. Tepat satu anak.
3. Jadikan 40 anak kiri 50, menggantikan 30. Bebaskan simpul 30.

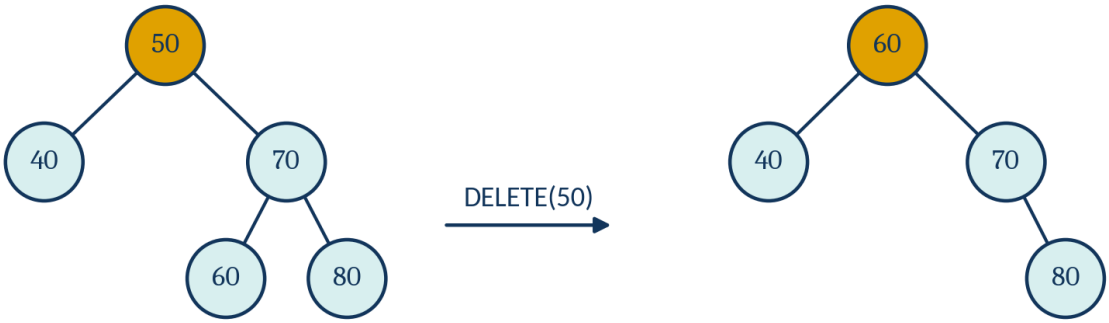
Mengapa sifatnya terjaga? Karena setiap kunci di subpohon yang dinaikkan pangkat itu sudah berada di sisi yang benar terhadap setiap leluhurnya. Subpohon yang berakar di 40 tadinya duduk di bawah 30, yang duduk di sebelah kiri 50; jadi setiap kunci di dalamnya sudah lebih kecil daripada 50. Memindahkannya naik satu aras tidak mengubah leluhur mana yang berada di atasnya, hanya seberapa jauh ia duduk di bawahnya.

Sesudah itu penelusuran inordernya adalah 40,50,60,70,80.

6.11.3 Kasus 3 — simpulnya punya dua anak

Inilah kasus yang sulit, dan kiatnya adalah menolak memecahkannya. Jika sebuah simpul punya dua anak, kita sama sekali tidak membuang simpul itu. Kita menimpa kuncinya dengan kunci yang secara sah boleh duduk di sana, lalu menghapus simpul *kunci itu*, yang dijamin merupakan kasus yang lebih mudah.

Kunci mana yang secara sah boleh duduk di sana? **Penerus inorder** — kunci terkecil di subpohon kanan. Resepnya, dinyatakan dengan persis: temukan penerus inordernya, yaitu min(subpohon kanan), salin ia ke simpul yang hendak dihapus, lalu hapus penerus inorder itu.



Gambar 6.6 Kasus 3 penghapusan: simpul 50 punya dua anak. Penerus inordernya adalah 60, kunci terkecil di subpohon kanannya. Nilai 60 disalin naik ke akar dan simpul 60 yang asli lalu dihapus, yang merupakan penghapusan kasus 1.

Melanjutkan dari pohon **Gambar 6.6** — pohon sebelumnya sesudah 30 dibuang, sehingga akar 50 punya anak kiri 40 dan anak kanan 70, dan 70 punya anak 60 dan 80 — hapuslah 50:

1. Cari 50: ia adalah akarnya. Ketemu seketika.
2. Simpul 50 punya anak kiri (40) dan anak kanan (70). Dua anak.
3. Temukan penerus inorder 50: melangkahlah satu kali ke kanan ke 70, lalu berjalanlah ke kiri sejauh mungkin. Anak kiri simpul 70 adalah 60; simpul 60 tidak punya anak kiri. Penerusnya adalah 60.
4. Salin kunci 60 ke dalam simpul akar, menimpa 50. Pohonnya sejenak memuat dua simpul berlabel 60.
5. Hapus simpul 60 yang *asli* dari subpohon kanan. Ia adalah simpul daun, jadi ini kasus 1: lepaskan ia dari induknya, 70.

Sesudah itu penelusuran inordernya adalah 40,60,70,80, persis seperti yang ditunjukkan pohon sebelah kanan **Gambar 6.6**.

Mengapa penerusnya adalah pengganti yang sah. Sebut simpul yang hendak dilenyapkan itu x dan penerusnya s. Setiap kunci di subpohon kiri x lebih kecil daripada kunci(x), dan kunci$(x) <$ kunci(s), jadi setiap kunci di subpohon kiri lebih kecil daripada kunci(s): sisi kirinya senang. Setiap kunci di subpohon kanan x lebih besar daripada kunci(x), dan kunci(s) adalah yang *terkecil* di antara mereka, jadi setiap kunci yang tersisa di subpohon kanan lebih besar daripada kunci(s): sisi kanannya senang. Menaruh kunci(s) di posisi x karena itu memenuhi sifat pohon pencarian biner di x, dan keadaan simpul lain tidak ada yang berubah.

Mengapa penghapusan rekursifnya lebih mudah. Penerus s adalah simpul paling kiri di subpohon kanan x, jadi menurut cara membangunnya s tidak punya anak kiri. Simpul tanpa anak kiri punya nol atau satu anak — kasus 1 atau kasus 2. Rekursinya karena itu berhenti seketika; ia tidak mungkin bertemu kasus 3 lagi.

Pendahulunya bekerja sama baiknya. *Pendahulu* inorder — max(subpohon kiri) — boleh dipakai sebagai gantinya, dengan alasan bayangan cerminnya. Ia tidak punya anak kanan, jadi ia juga kasus yang mudah. Sebagian implementasi berganti-ganti antara penerus dan pendahulu untuk menghindari kecondongan ke kiri yang cenderung ditimbulkan oleh kebiasaan selalu memilih penerus. Pada **Gambar 6.6**, memilih pendahulu akan menaikkan 40 alih-alih 60, memberikan pohon yang sama sahnya: 40 di akar dengan anak kanan 70 dan cucu 60,80.

Tabel 6.3 mengumpulkan ketiga kasusnya.

Tabel 6.3 Ketiga kasus penghapusan pada pohon pencarian biner, dan kerja yang dituntut masing-masingnya di luar pencarian awal.

Kasus	Anak simpul yang hendak dilenyapkan	Apa yang dilakukan	Biaya
1	tidak punya	lepaskan ia dari induknya lalu bebaskan	$O(1)$ sesudah pencariannya
2	tepat satu	naikkan seluruh subpohon anak itu ke tempatnya	$O(1)$ sesudah pencariannya
3	dua	salin kunci penerus inordernya ke dalamnya, lalu hapus simpul penerus itu (yang merupakan kasus 1 atau 2)	satu penyusuran $O(h)$ tambahan

Biaya. Menemukan simpulnya berongkos $O(h)$. Kasus 1 dan 2 lalu melakukan kerja tetap. Kasus 3 melakukan satu penyusuran lagi untuk menemukan penerusnya, yang juga $O(h)$. Jadi penghapusan berongkos $O(h)$ secara keseluruhan.

6.12 Pohon Pencarian Biner yang Lengkap dalam C++

Kode 6.4 adalah pohon pencarian biner yang utuh dan dapat dijalankan: sisip, cari, minimum, maksimum, tinggi, pendaftaran inorder, dan penghapusan dalam ketiga kasusnya. Fungsi `main`-nya memutar ulang ketiga contoh penghapusan di §6.11 secara berurutan, sehingga keluaran cetakannya bisa dicocokkan baris demi baris dengan **Gambar 6.4**, **Gambar 6.5**, dan **Gambar 6.6**.

Kode 6.4 Pohon pencarian biner yang lengkap. Kompilasi dengan `g++ -std=c++17 -O2 -o bst bst.cpp` lalu jalankan `./bst`.

```
// Pohon pencarian biner yang lengkap dalam C++: sisip, cari, minimum, maksimum,
// penerus, pendahulu, hapus (ketiga kasusnya), dan pendaftaran inorder.
// Fungsi main() memutar ulang ketiga contoh penghapusan di Subbab 6.11 secara berurutan.
#include <iostream>
#include <string>
#include <vector>

class BST {
public:
    BST() : root_(nullptr) {}
    ~BST() { destroy(root_); }

    void insert(int key)           { root_ = insertRec(root_, key); }
    bool contains(int key) const   { return find(root_, key) != nullptr; }
    void remove(int key)           { root_ = removeRec(root_, key); }

    int minimum() const { const Node* x = root_; while (x->left)  x = x->left;  return x-
>key; }
    int maximum() const { const Node* x = root_; while (x->right) x = x->right; return x-
>key; }

    int height() const { return heightRec(root_); }   // pohon kosong bertinggi -1

    std::vector<int> inorder() const {
        std::vector<int> out;
        inorderRec(root_, out);
        return out;
    }

    void print(const std::string& label) const {
        std::cout << label;
        std::vector<int> v = inorder();
        for (size_t i = 0; i < v.size(); ++i) {
            if (i) std::cout << ", ";
            std::cout << v[i];
        }
        std::cout << "   [height " << height() << "]\n";
    }

private:
    struct Node {
        int key;
        Node* left;
        Node* right;
        Node(int k) : key(k), left(nullptr), right(nullptr) {}
    };
    Node* root_;
```

Kode 6.4 (Lanjutan) *Pohon pencarian biner yang lengkap. Kompilasi dengan g++ -std=c++17 -O2 -o bst bst.cpp lalu jalankan ./bst.*

```
    static void destroy(Node* x) {
        if (!x) return;
        destroy(x->left);
        destroy(x->right);
        delete x;
    }

    static Node* insertRec(Node* x, int key) {
        if (!x) return new Node(key);
        if      (key < x->key) x->left  = insertRec(x->left,  key);
        else if (key > x->key) x->right = insertRec(x->right, key);
        return x;                                   // kunci sama: tidak melakukan apa pun
    }

    static const Node* find(const Node* x, int key) {
        while (x && x->key != key) x = (key < x->key) ? x->left : x->right;
        return x;
    }

    static int heightRec(const Node* x) {
        if (!x) return -1;
        int l = heightRec(x->left), r = heightRec(x->right);
        return 1 + (l > r ? l : r);
    }

    static void inorderRec(const Node* x, std::vector<int>& out) {
        if (!x) return;
        inorderRec(x->left, out);
        out.push_back(x->key);
        inorderRec(x->right, out);
    }

    static Node* minNode(Node* x) { while (x->left) x = x->left; return x; }

    static Node* removeRec(Node* x, int key) {
        if (!x) return nullptr;                     // kunci tidak ada: tidak ada yang perlu
dikerjakan
        if      (key < x->key) { x->left  = removeRec(x->left,  key); return x; }
        else if (key > x->key) { x->right = removeRec(x->right, key); return x; }
```

Kode 6.4 (Lanjutan) *Pohon pencarian biner yang lengkap. Kompilasi dengan g++ -std=c++17 -O2 -o bst bst.cpp lalu jalankan ./bst.*

```
        // x adalah simpul yang akan dihapus.
        if (!x->left && !x->right) {                // kasus 1: tanpa anak
            delete x;
            return nullptr;
        }
        if (!x->left) {                             // kasus 2a: hanya punya anak kanan
            Node* c = x->right;
            delete x;
            return c;
        }
        if (!x->right) {                            // kasus 2b: hanya punya anak kiri
            Node* c = x->left;
            delete x;
            return c;
        }
        Node* s = minNode(x->right);                // kasus 3: penerus inorder
        x->key = s->key;                            // salin kuncinya ke atas
        x->right = removeRec(x->right, s->key);     // lalu hapus ia dari subpohon kanan
        return x;
    }
};

int main() {
    BST t;
    for (int k : {50, 30, 70, 20, 40, 60, 80}) t.insert(k);
    t.print("start                 : ");

    std::cout << "contains(40) = " << (t.contains(40) ? "true" : "false")
              << ", contains(45) = " << (t.contains(45) ? "true" : "false") << "\n";
    std::cout << "minimum = " << t.minimum() << ", maximum = " << t.maximum() << "\n\n";

    t.remove(20);  t.print("case 1, delete 20     : ");
    t.remove(30);  t.print("case 2, delete 30     : ");
    t.remove(50);  t.print("case 3, delete 50     : ");
    return 0;
}
```

Programnya mencetak:

```
start                 : 20, 30, 40, 50, 60, 70, 80   [height 2]
contains(40) = true, contains(45) = false
minimum = 20, maximum = 80

case 1, delete 20     : 30, 40, 50, 60, 70, 80   [height 2]
case 2, delete 30     : 40, 50, 60, 70, 80   [height 2]
case 3, delete 50     : 40, 60, 70, 80   [height 2]
```

Tiga catatan tentang kodenya, karena masing-masingnya adalah tempat pemula kehilangan satu hari.

Fungsi-fungsinya mengembalikan akar baru subpohonnya. `insertRec` dan `removeRec` sama-sama menerima sebuah subpohon dan menyerahkan kembali sebuah subpohon, dan pemanggilnya menuliskan hasilnya ke penunjuk anak yang benar: `x->left = insertRec(x->left, key)`. Ungkapan ini melenyapkan kebutuhan akan penunjuk induk, dan melenyapkan satu keluarga kutu yang utuh, yaitu kutu yang membuat sebuah simpul berubah tetapi induknya masih menunjuk ke yang lama. Setiap fungsi pohon rekursif dalam bab ini memakainya.

Kasus 2 ditulis dua kali. `if (!x->left)` menangani simpul yang hanya punya anak kanan dan `if (!x->right)` menangani simpul yang hanya punya anak kiri. Keduanya bisa diciutkan menjadi

satu cabang, dan di kode AVL nanti kita memang melakukannya persis begitu. Ditulis terpisah di sini, keduanya lebih mudah dicocokkan dengan **Gambar 6.5**.

Kasus 3 menyalin sebuah kunci, bukan sebuah simpul. Sesudah `x->key = s->key`, objek simpul pada posisi x masih objek yang sama di memori; hanya bilangan bulat di dalamnyalah yang berubah. Kalau simpul itu juga membawa data satelit — nama pelanggan, berat — data itu pun harus ikut disalin, atau pohonnya akan memuat kunci milik penerus yang tertempel pada rekam pelanggan yang sudah dihapus. Pada kode produksi biasanya lebih aman menyambung ulang simpul penerusnya sendiri alih-alih menyalin isinya.

6.13 Ketika Pohonnya Rusak: h Bisa Mencapai n

Setiap biaya dalam bab ini sejauh ini adalah $O(h)$. Itu batas yang sungguhan tetapi bukan sebuah *janji*, karena sejauh ini tidak ada apa pun yang mencegah h menjadi besar.

Perhatikan penyisipan kunci 10,20,30,40,50 ke dalam pohon kosong dalam urutan itu.

1. 10 menjadi akarnya.
2. $20 > 10$, jadi ia pergi ke kanan. Pohonnya menjadi $10 \to 20$.
3. $30 > 10$, ke kanan; $30 > 20$, ke kanan. Pohonnya menjadi $10 \to 20 \to 30$.
4. 40 dengan cara yang sama berakhir di bawah 30.
5. 50 dengan cara yang sama berakhir di bawah 40.

Hasilnya adalah sebuah rantai: lima simpul, lima aras, tinggi 4. Setiap simpul kecuali yang terakhir punya tepat satu anak, dan anak itu selalu anak kanan. Bentuk ini disebut pohon **merosot** atau **miring**, dan ia persis sebuah senarai berantai yang mengenakan pakaian pohon. Mencarinya berongkos $\Theta(n)$. Inilah persis kegagalan yang patut ditakuti: sebagian besar operasi pohon pencarian biner memakan waktu yang sebanding dengan tinggi pohonnya, dan tinggi itu menjadi $n - 1$ untuk pohon yang miring.

Ini bukan kecelakaan langka. Masukan terurut adalah masukan yang paling lazim ada. Rekam datang dalam urutan tanggal, nomor faktur diterbitkan dalam urutan naik, baris log ditambahkan dalam urutan cap waktu, sebuah basis data mengekspor barisnya dalam urutan kunci primer. Setiap satu dari itu, kalau disuapkan langsung ke pohon pencarian biner biasa, menghasilkan sebuah rantai.

Kode 6.5 mengukur kerusakannya. Ia membangun dua pohon dari n kunci yang sama — sekali dalam urutan naik, sekali dalam keadaan teracak — lalu melaporkan tinggi masing-masingnya dan rata-rata cacah simpul yang harus ditengok oleh sebuah penyisipan.

Kode 6.5 Mengukur apa yang dilakukan urutan penyisipan terhadap sebuah pohon pencarian biner. Kompilasi dengan `g++ -std=c++17 -O2 -o degenerate degenerate.cpp`.

```
// Apa yang dilakukan urutan penyisipan terhadap tinggi pohon pencarian biner.
// Masukan terurut menghasilkan rantai n simpul; masukan teracak menghasilkan pohon pendek.
#include <iostream>
#include <iomanip>
#include <random>
#include <vector>
#include <algorithm>
#include <cmath>

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

// Penyisipan iteratif, dan ia mengembalikan berapa simpul yang harus ditengoknya.
int insertCounting(Node*& root, int key) {
    int probes = 0;
    if (!root) { root = new Node(key); return 1; }
    Node* x = root;
    for (;;) {
        ++probes;
        if (key < x->key) {
            if (!x->left) { x->left = new Node(key); return probes; }
            x = x->left;
        } else {
            if (!x->right) { x->right = new Node(key); return probes; }
            x = x->right;
        }
    }
}

int height(Node* x) {
    if (!x) return -1;
    int l = height(x->left), r = height(x->right);
    return 1 + (l > r ? l : r);
}

void destroy(Node* x) { if (!x) return; destroy(x->left); destroy(x->right); delete x; }
```

Kode 6.5 (Lanjutan) *Mengukur apa yang dilakukan urutan penyisipan terhadap sebuah pohon pencarian biner. Kompilasi dengan g++ -std=c++17 -O2 -o degenerate degenerate.cpp.*

```
// Bangun pohon dari urutan yang diberikan; laporkan tinggi dan biaya pencariannya.
void trial(const char* name, std::vector<int> order, int n) {
    Node* root = nullptr;
    long long totalProbes = 0;
    for (int k : order) totalProbes += insertCounting(root, k);
    int h = height(root);
    std::cout << std::left << std::setw(20) << name
              << " n = " << std::setw(7) << n
              << " height = " << std::setw(8) << h
              << " average probes per insert = "
              << std::fixed << std::setprecision(2)
              << (double)totalProbes / n << "\n";
    destroy(root);
}

int main() {
    std::mt19937 rng(12345);                  // benih tetap, agar hasil jalannya terulang
sama
    for (int n : {1000, 4000, 16000}) {
        std::vector<int> sorted(n);
        for (int i = 0; i < n; ++i) sorted[i] = i;
        std::vector<int> shuffled = sorted;
        std::shuffle(shuffled.begin(), shuffled.end(), rng);

        trial("sorted input",   sorted,   n);
        trial("shuffled input", shuffled, n);
        std::cout << "   lg(n) = " << std::fixed << std::setprecision(2)
                  << std::log2((double)n) << "\n\n";
    }
    return 0;
}
```

Programnya mencetak:

```
sorted input         n = 1000    height = 999      average probes per insert = 499.50
shuffled input       n = 1000    height = 21       average probes per insert = 10.55
   lg(n) = 9.97

sorted input         n = 4000    height = 3999     average probes per insert = 1999.50
shuffled input       n = 4000    height = 25       average probes per insert = 13.48
   lg(n) = 11.97

sorted input         n = 16000   height = 15999    average probes per insert = 7999.50
shuffled input       n = 16000   height = 32       average probes per insert = 16.03
   lg(n) = 13.97
```

Bacalah angka-angka itu pelan-pelan, karena merekalah pembenaran bagi seluruh paruh kedua bab ini.

Dengan masukan terurut tingginya selalu tepat $n - 1$: rantai yang sempurna, setiap kali. Rata-rata cacah sentuhan per penyisipan adalah $(n - 1)/2$, yang untuk $n = 16000$ bernilai 7999.5. Dengan masukan teracak, tingginya pada $n = 16000$ adalah 32, dan rata-rata cacah sentuhannya 16.03 — sementara $\lg 16000 \approx 13.97$. Masukan teracak karena itu berongkos kira-kira 16 sentuhan di tempat masukan terurut berongkos kira-kira 8000: faktor kira-kira **lima ratus**, pada data yang sama, dengan kode yang sama, hanya berbeda pada urutan kedatangannya.

> **Kasus acaknya baik-baik saja; kasus acak bukan persoalannya.** Dapat dibuktikan bahwa pohon pencarian biner yang dibangun dengan menyisipkan n kunci berbeda dalam urutan acak seragam punya tinggi harapan $\Theta(\lg n)$. Konstantanya lebih besar daripada satu: untuk n yang sangat besar tinggi harapannya mendekati $4.311 \ln n \approx 2.99 \lg n$, dan sebuah suku koreksi

membuatnya agak lebih rendah pada ukuran sedang. Itulah sebabnya tinggi terukur di atas adalah 21, 25, dan 32, bukan 10, 12, dan 14 seperti pada pohon yang seimbang sempurna. Persoalannya adalah data nyata hampir tidak pernah berada dalam urutan acak, dan Anda jarang menguasai urutannya. Struktur yang unjuk kerjanya bergantung pada keberuntungan Anda adalah struktur yang pada akhirnya akan gagal pada hari yang justru paling penting. Kita menginginkan sebuah *jaminan*, bukan sebuah harapan.

6.14 Pohon AVL: Keseimbangan sebagai Invarian

Pada 1962 dua matematikawan Soviet, **Georgy Adelson-Velsky** dan **Evgenii Landis**, menerbitkan struktur data pertama yang memberikan jaminan itu. Ia dinamai menurut inisial mereka: **pohon AVL**.

Gagasannya sederhana sampai mengecoh. Jangan biarkan pohonnya kehilangan bentuk sejak awal. Sesudah setiap penyisipan dan setiap penghapusan, periksa apakah ada simpul yang menjadi berat sebelah, dan kalau ada, luruskan ia seketika dengan penyusunan ulang setempat yang berongkos $O(1)$.

DEFINISI (faktor keseimbangan). Untuk sebuah simpul x, **faktor keseimbangan** (*balance factor*) adalah

$$\mathrm{bf}(x) = h(\mathrm{kanan}(x)) - h(\mathrm{kiri}(x)), \tag{6.3}$$

dengan $h(\cdot)$ adalah tinggi sebuah subpohon dan subpohon kosong bertinggi -1.

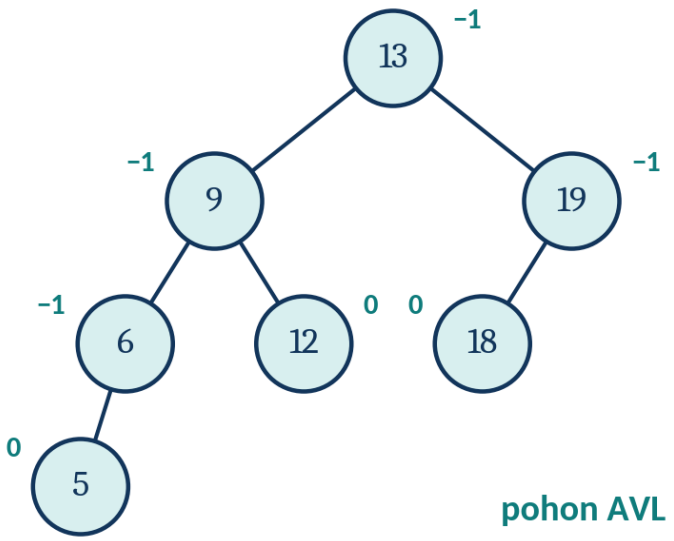
DEFINISI (pohon AVL). Sebuah **pohon AVL** adalah pohon pencarian biner yang memenuhi

$$\mathrm{bf}(x) \in \{-1, 0, +1\} \quad \text{untuk setiap simpul } x. \tag{6.4}$$

Setara dengan itu: pada setiap simpul, tinggi kedua subpohonnya berbeda paling banyak satu.

Dengan kata-kata, Persamaan (6.4) mengatakan: untuk tiap simpul, tinggi cabang kiri dan cabang kanan berselisih paling banyak 1. Kesepakatan tanda pada Persamaan (6.3) penting dan layak dihafalkan: faktor keseimbangan adalah tinggi cabang kanan dikurangi tinggi cabang kiri. Faktor keseimbangan negatif karena itu berarti **berat kiri**, yang positif berarti **berat kanan**, dan nol berarti kedua sisinya sama tinggi.

Berikut satu pohon dari masing-masing jenis. **Gambar 6.7** adalah pohon AVL; **Gambar 6.8** bukan.

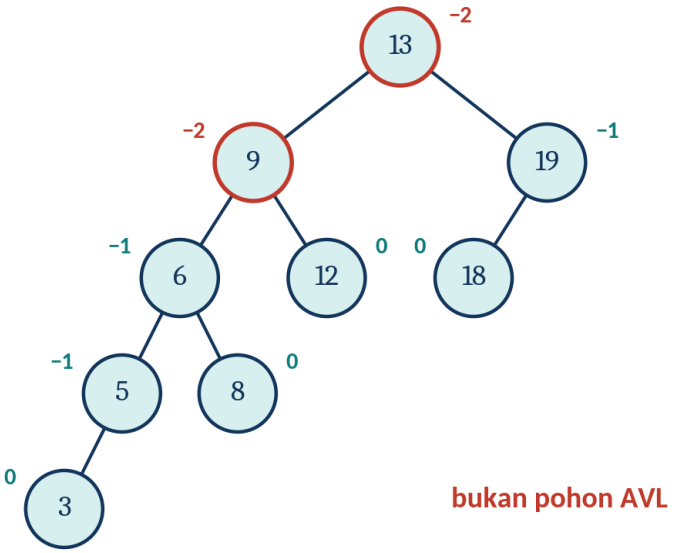


Gambar 6.7 Sebuah pohon AVL. Kedua subpohon tiap simpul berbeda tinggi paling banyak satu.

Periksa **Gambar 6.7** simpul demi simpul. Pohonnya adalah: akar 13, anak kiri 9, anak kanan 19; simpul 9 punya anak 6 dan 12; simpul 6 punya anak kiri 5; simpul 19 punya anak kiri 18.

- Simpul 5: simpul daun. Kedua subpohonnya kosong, tingginya -1 dan -1. bf $= 0$. ✓
- Simpul 12: simpul daun, bf $= 0$. ✓
- Simpul 18: simpul daun, bf $= 0$. ✓
- Simpul 6: subpohon kirinya adalah simpul tunggal 5, tinggi 0; subpohon kanannya kosong, tinggi -1. bf $= -1 - 0 = -1$. ✓
- Simpul 9: subpohon kirinya berakar di 6 dengan tinggi 1; subpohon kanannya adalah simpul tunggal 12, tinggi 0. bf $= 0 - 1 = -1$. ✓
- Simpul 19: subpohon kirinya adalah 18, tinggi 0; subpohon kanannya kosong, tinggi -1. bf $= -1 - 0 = -1$. ✓
- Simpul 13: subpohon kiri yang berakar di 9 bertinggi 2; subpohon kanan yang berakar di 19 bertinggi 1. bf $= 1 - 2 = -1$. ✓

Setiap faktor keseimbangannya ada di $\{-1, 0, +1\}$, jadi pohon itu adalah pohon AVL.



Gambar 6.8 Sebuah pohon pencarian biner yang **bukan** pohon AVL. Menambahkan simpul 8 dan 3 mendorong dua faktor keseimbangan keluar dari rentang: milik 9 dan 13, keduanya −2.

Gambar 6.8 adalah pohon yang sama dengan dua simpul tambahan: simpul 6 kini juga punya anak kanan 8, dan simpul 5 memperoleh anak kiri 3. Hitung ulang:

- Simpul 3: simpul daun, tinggi 0.
- Simpul 5: subpohon kirinya adalah 3, tinggi 0; subpohon kanannya kosong, tinggi -1. bf $= -1$. Masih baik-baik saja, dan simpul 5 kini bertinggi 1.
- Simpul 8: simpul daun, tinggi 0.
- Simpul 6: subpohon kirinya berakar di 5, tinggi 1; subpohon kanannya adalah 8, tinggi 0. bf $= 0 - 1 = -1$. Baik, dan simpul 6 kini bertinggi 2.
- Simpul 9: subpohon kirinya berakar di 6, tinggi 2; subpohon kanannya adalah 12, tinggi 0. bf $= 0 - 2 = \mathbf{-2}$. **Di luar rentang.**
- Simpul 13: subpohon kirinya berakar di 9, kini bertinggi 3; subpohon kanannya berakar di 19, tinggi 1. bf $= 1 - 3 = \mathbf{-2}$. **Juga di luar rentang.**

Simpul 9 dan 13 sama-sama melanggar invariannya, jadi pohonnya bukan pohon AVL — dan itu persis yang dikatakan label pada gambarnya, sekaligus alasan keduanya diberi garis tepi merah di sana. Perhatikan bahwa ia masih merupakan *pohon pencarian biner* yang sah sepenuhnya: penelusuran inordernya 3,5,6,8,9,12,13,18,19 terurut. Ke-AVL-an adalah syarat tambahan atas bentuk, yang dilapiskan di atas syarat pengurutan atas kunci.

> **Invariannya tentang tinggi, bukan tentang cacah.** Sebuah simpul yang subpohon kirinya memuat 1000 simpul dan subpohon kanannya memuat 3 simpul tetap bisa menjadi simpul AVL yang sah sepenuhnya, asalkan kedua subpohon itu berbeda tinggi paling banyak satu. Pohon AVL menyeimbangkan *kedalaman*, bukan *jumlah penduduk*. Itu sudah cukup, karena kedalamanlah yang dibayar oleh pencarian.

6.15 Mengapa Pohon AVL Selalu Pendek: Rekurensi Simpul-Minimum

Invariannya tampak sederhana — "tinggi berbeda paling banyak satu" — sehingga sungguh mengejutkan bahwa ia memaksa tingginya turun sampai $O(\lg n)$. Sekarang kita membuktikannya,

dan pembuktiannya adalah rekurensi yang persis sejenis dengan yang diajarkan Bab 3 untuk diselesaikan.

Kiatnya adalah membalikkan pertanyaannya. Alih-alih bertanya *"seberapa tinggi sebuah pohon dengan n simpul bisa jadi?"*, tanyakan *"seberapa sedikit simpul yang bisa dipunyai sebuah pohon bertinggi h?"* — karena pohon AVL paling rapuh pada tinggi tertentulah yang memberi kita batasnya.

> **DEFINISI.** Misalkan $N(h)$ adalah **cacah simpul minimum** dalam sembarang pohon AVL bertinggi h.

Kasus-kasus basisnya.

$N(0) = 1$. Pohon bertinggi 0 adalah satu simpul tunggal tanpa anak, dan satu simpul adalah yang paling sedikit yang mungkin.

$N(1) = 2$. Pohon bertinggi 1 harus punya sekurang-kurangnya satu anak yang menggantung di bawah akarnya, dan satu akar ditambah satu anak adalah pohon AVL yang sah (faktor keseimbangannya ± 1). Dua simpul.

Rekurensinya. Ambil sebuah pohon AVL bertinggi $h \geq 2$ dengan simpul sesedikit mungkin. Akarnya punya dua subpohon; sebut keduanya A dan B. Sekarang bernalarlah tentang mereka.

1. Pohonnya bertinggi h, yang berarti lintasan menurun terpanjangnya punya h sisi. Lintasan itu melewati salah satu dari kedua subpohonnya. Jadi sekurang-kurangnya satu subpohon bertinggi tepat $h - 1$. Katakanlah itu A: $h(A) = h - 1$.
2. Berapa $h(B)$ bisa jadi? Invarian AVL di akarnya berkata $|h(A) - h(B)| \leq 1$, jadi $h(B) \in \{h - 2, h - 1, h\}$. Ia tidak bisa h, sebab kalau begitu pohonnya akan lebih tinggi daripada h. Jadi $h(B)$ adalah $h - 1$ atau $h - 2$.
3. Kita menginginkan simpul se*sedikit* mungkin, dan subpohon yang lebih pendek memerlukan simpul lebih sedikit, jadi kita memilih $h(B) = h - 2$ — yang terkecil yang diizinkan invariannya.
4. A sendiri adalah pohon AVL bertinggi $h - 1$, dan untuk meminimumkan totalnya kita menjadikannya pohon terkecil semacam itu, dengan $N(h - 1)$ simpul. Begitu pula B adalah pohon AVL terkecil bertinggi $h - 2$, dengan $N(h - 2)$ simpul.
5. Tambahkan akarnya.

Itu memberikan, untuk setiap $h \geq 2$,

$$N(h) = N(h - 1) + N(h - 2) + 1, \qquad N(0) = 1, \quad N(1) = 2. \tag{6.5}$$

> **Beberapa nilai pertamanya, dihitung dengan tangan.**
>
> $N(2) = N(1) + N(0) + 1 = 2 + 1 + 1 = 4.$
>
> $N(3) = N(2) + N(1) + 1 = 4 + 2 + 1 = 7.$
>
> $N(4) = N(3) + N(2) + 1 = 7 + 4 + 1 = 12.$
>
> $N(5) = N(4) + N(3) + 1 = 12 + 7 + 1 = 20.$
>
> $N(6) = N(5) + N(4) + 1 = 20 + 12 + 1 = 33.$
>
> Jadi pohon AVL bertinggi 6 harus memuat sekurang-kurangnya 33 simpul. Dibalik: 32 **simpul tidak akan pernah bisa disusun menjadi pohon AVL bertinggi** 6**.** Mereka harus muat dalam tinggi 5 atau kurang.

Pohon-pohon yang mencapai nilai minimum ini disebut **pohon Fibonacci**, dan namanya adalah sebuah petunjuk.

6.15.1 Hubungannya dengan Fibonacci

Ingat kembali bilangan Fibonacci, yang didefinisikan oleh $F(0) = 0$, $F(1) = 1$, dan $F(m) = F(m-1) + F(m-2)$:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ 233,\ \ldots$$

Rekurensi Persamaan (6.5) adalah rekurensi Fibonacci dengan sebuah $+1$ yang mengganggu menempel padanya. Tambahan $+1$ semacam itu dilenyapkan dengan penggantian peubah. Tetapkan

$$M(h) = N(h) + 1.$$

Kemudian, dengan menyubstitusikan $N = M - 1$ ke dalam rekurensinya,

$$M(h) - 1 = (M(h-1) - 1) + (M(h-2) - 1) + 1,$$

dan konstanta-konstantanya runtuh:

$$M(h) = M(h-1) + M(h-2).$$

Itu persis rekurensi Fibonacci. Nilai basisnya adalah $M(0) = N(0) + 1 = 2$ dan $M(1) = N(1) + 1 = 3$. Menyusuri daftar Fibonacci, $2 = F(3)$ dan $3 = F(4)$. Dua suku berurutan cocok, dan kedua barisan menuruti rekurensi yang sama, jadi keduanya cocok selamanya:

$$N(h) + 1 = F(h+3). \tag{6.6}$$

Periksa Persamaan (6.6) terhadap hitungan tangan tadi: $N(4) + 1 = 13$ dan $F(7) = 13$ ✓; $N(6) + 1 = 34$ dan $F(9) = 34$ ✓.

Tabel 6.4 mendaftar angka-angkanya bersama-sama.

Tabel 6.4 Simpul paling sedikit yang bisa dimuat pohon AVL bertinggi h, dan bilangan Fibonacci yang sama dengannya begitu ditambah satu.

h	$N(h)$	$N(h)+1$	$F(h+3)$
0	1	2	2
1	2	3	3
2	4	5	5
3	7	8	8
4	12	13	13
5	20	21	21
6	33	34	34
7	54	55	55
8	88	89	89
9	143	144	144
10	232	233	233
11	376	377	377
12	609	610	610

6.15.2 Dari Fibonacci ke $1.44\lg n$

Bab 3 memberikan bentuk tertutup bilangan Fibonacci,

$$F(m) = \frac{\varphi^m - \hat{\varphi}^m}{\sqrt{5}}, \qquad \varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803, \qquad \hat{\varphi} = \frac{1-\sqrt{5}}{2} \approx -0.61803. \tag{6.7}$$

Pada Persamaan (6.7), karena $|\hat{\varphi}| < 1$, suku $\hat{\varphi}^m$ menyusut menuju nol, dan sesungguhnya $|\hat{\varphi}^m|/\sqrt{5} < 1/2$ untuk setiap $m \geq 0$. Jadi $F(m)$ selalu merupakan bilangan bulat terdekat dengan $\varphi^m/\sqrt{5}$, dan khususnya

$$F(m) + 1 \;\geq\; \frac{\varphi^m}{\sqrt{5}}. \tag{6.8}$$

Sekarang misalkan T adalah sembarang pohon AVL dengan n simpul dan tinggi h. Karena $N(h)$ adalah simpul *paling sedikit* yang bisa dipunyai pohon semacam itu,

$$n \;\geq\; N(h) \;=\; F(h+3) - 1.$$

Tambahkan 2 pada kedua ruas lalu terapkan Persamaan (6.8) dengan $m = h + 3$:

$$n + 2 \;\geq\; F(h+3) + 1 \;\geq\; \frac{\varphi^{h+3}}{\sqrt{5}}.$$

Ambil logaritma basis 2 kedua ruasnya — sah karena kedua ruasnya positif, dan lg naik:

$$\lg(n+2) \;\geq\; (h+3)\lg\varphi - \frac{1}{2}\lg 5.$$

Susun ulang untuk h, dengan membagi oleh konstanta positif $\lg\varphi$:

$$h + 3 \;\leq\; \frac{\lg(n+2) + 1/2\lg 5}{\lg\varphi}.$$

Masukkan angkanya: $\lg\varphi = \lg 1.61803 \approx 0.69424$, jadi $1/\lg\varphi \approx 1.44042$; dan $1/2\lg 5 \approx 1.16096$, jadi $1/2\lg 5 \,/\, \lg\varphi \approx 1.67231$. Karena itu

$$h \;\leq\; 1.44042\lg(n+2) + 1.67231 - 3,$$

yaitu,

$$\boxed{h \;\leq\; 1.4405\lg(n+2) - 1.3277} \tag{6.9}$$

Apa yang dikatakan batas itu dengan kata-kata. Sebuah pohon AVL tidak pernah lebih daripada kira-kira **44% lebih tinggi** daripada pohon biner yang seimbang sempurna dengan ukuran yang sama. Karena Persamaan (6.9) memberikan $h \leq 1.4405\lg(n+2) - 1.3277 = O(\lg n)$, setiap operasi yang berongkos $O(h)$ — pencarian, minimum, maksimum, penerus,

pendahulu, penyisipan, penghapusan — berongkos $O(\lg n)$ pada pohon AVL. Bukan rata-rata. Bukan biasanya. Selalu, pada setiap masukan, dalam urutan apa pun.

Catatan tentang konstanta yang akan Anda jumpai di tempat lain. Batas itu sangat sering dikutip sebagai $h < 1.4405\lg(n+2) - 0.3277$, tepat satu lebih besar daripada milik kita. Versi itu mencacah *simpul* pada lintasan terpanjang, bukan *sisi*, sehingga h-nya satu lebih besar daripada milik kita untuk pohon yang sama. Kedua pernyataan itu benar; keduanya hanya berbeda pada kesepakatan yang dipancangkan di §6.2. Selalu periksa yang mana yang sedang dipakai sebuah sumber sebelum membandingkan konstanta.

Kode 6.6 menghitung tabelnya, memeriksa $N(h) + 1 = F(h+3)$ secara numerik, memeriksa batasnya pada setiap tinggi, lalu menjawab pertanyaan praktisnya dari arah yang sebaliknya: diberikan n, seberapa tinggi pohonnya mungkin menjadi?

Kode 6.6 Rekurensi simpul-minimum, kesamaan Fibonacci, dan batas tingginya, semuanya diperiksa secara numerik. Kompilasi dengan `g++ -std=c++17 -O2 -o minnodes minnodes.cpp`.

```
// Pohon AVL terkecil untuk tiap tinggi, bilangan Fibonacci yang bersembunyi di dalamnya,
// dan batas h < 1.4405 * lg(n+2) - 1.3277 diperiksa terhadap angka yang sesungguhnya.
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>

int main() {
    const int H = 45;
    std::vector<long long> N(H + 1), F(H + 4);

    N[0] = 1;                        // satu simpul tunggal adalah pohon AVL bertinggi 0
    N[1] = 2;                        // akar dengan satu anak bertinggi 1
    for (int h = 2; h <= H; ++h) N[h] = N[h - 1] + N[h - 2] + 1;

    F[0] = 0; F[1] = 1;
    for (int i = 2; i < (int)F.size(); ++i) F[i] = F[i - 1] + F[i - 2];

    std::cout << " h        N(h)   N(h)+1   F(h+3)   bound 1.4405*lg(N(h)+2)-1.3277\n";
    std::cout << " ------------------------------------------------------------\n";
    for (int h = 0; h <= 12; ++h) {
        double bound = 1.4405 * std::log2((double)N[h] + 2.0) - 1.3277;
        std::cout << std::setw(2) << h
                  << std::setw(12) << N[h]
                  << std::setw(9)  << N[h] + 1
                  << std::setw(9)  << F[h + 3]
                  << std::setw(23) << std::fixed << std::setprecision(4) << bound
                  << (bound > h ? "   ok" : "   FAILS") << "\n";
    }

    std::cout << "\nHeight of the tallest AVL tree that can hold n nodes:\n";
    for (long long n : {1000LL, 1000000LL, 1000000000LL}) {
        int h = 0;
        while (h + 1 <= H && N[h + 1] <= n) ++h;
        std::cout << "  n = " << std::setw(11) << n
                  << "   tallest AVL height = " << std::setw(3) << h
                  << "   lg n = " << std::fixed << std::setprecision(2)
                  << std::log2((double)n)
                  << "   1.44*lg n = " << 1.4405 * std::log2((double)n) << "\n";
    }
    return 0;
}
```

Programnya mencetak:

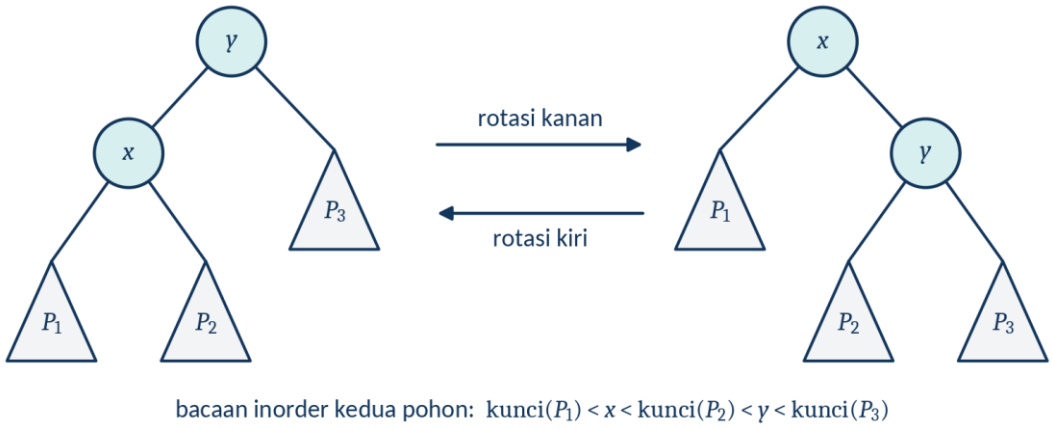
```
 h        N(h)   N(h)+1   F(h+3)   bound 1.4405*lg(N(h)+2)-1.3277
 ------------------------------------------------------------
 0          1        2        2                 0.9554   ok
 1          2        3        3                 1.5533   ok
 2          4        5        5                 2.3959   ok
 3          7        8        8                 3.2386   ok
 4         12       13       13                 4.1568   ok
 5         20       21       21                 5.0961   ok
 6         33       34       34                 6.0610   ok
 7         54       55       55                 7.0378   ok
 8         88       89       89                 8.0238   ok
 9        143      144      144                 9.0150   ok
10        232      233      233                10.0096   ok
11        376      377      377                11.0062   ok
12        609      610      610                12.0042   ok

Height of the tallest AVL tree that can hold n nodes:
  n =        1000   tallest AVL height =  13   lg n = 9.97   1.44*lg n = 14.36
  n =     1000000   tallest AVL height =  27   lg n = 19.93   1.44*lg n = 28.71
  n =  1000000000   tallest AVL height =  41   lg n = 29.90   1.44*lg n = 43.07
```

Tiga hal yang perlu dibawa pulang dari keluaran itu. Pertama, lajur $N(h) + 1$ dan lajur $F(h + 3)$ sama persis, dan itulah kesamaan yang kita buktikan tadi. Kedua, batasnya berlaku pada setiap tinggi dan menjadi makin *ketat* seiring bertumbuhnya h — pada $h = 12$ ia memberi 12.0042, yaitu selisih empat perseribu. Itu tanda sebuah batas yang pada hakikatnya persis, bukan taksiran berlebih yang murah hati. Ketiga, dan ini yang paling praktis: semiliar kunci dalam pohon AVL tidak akan pernah bisa lebih dalam daripada 41 aras. Satu pencarian menyentuh paling banyak 42 simpul. Bandingkan dengan pohon pencarian biner biasa di §6.13, yang pada masukan terurut akan sedalam semiliar aras.

6.16 Rotasi: Satu-satunya Perkakas

Sekarang kita tahu apa yang kita inginkan (keseimbangan) dan mengapa kita menginginkannya (tinggi $O(\lg n)$). Kita memerlukan cara *memulihkan* keseimbangan sesudah sebuah penyisipan atau penghapusan merusaknya, dan perbaikannya harus murah — $O(1)$ — atau kita akan membayar jaminannya dengan operasi yang lebih lambat.

Perkakas itu adalah **rotasi**, yang dilukiskan pada **Gambar 6.9**. Ia melayani penyisipan maupun penghapusan.



Gambar 6.9 Kedua rotasi. Kelima benda yang sama — simpul x dan y serta subpohon P_1, P_2, P_3 — muncul di kedua gambarnya. Rotasi kanan mengubah gambar kiri menjadi gambar kanan; rotasi kiri mengubah gambar kanan kembali menjadi gambar kiri. Keduanya balikan yang persis satu sama lain.

Lihat gambar sebelah kirinya. Simpul y berada di puncak; anak kirinya adalah x; subpohon kanannya adalah P_3. Simpul x punya subpohon P_1 di kiri dan P_2 di kanan.

Karena ini adalah pohon pencarian biner, sifat pohon pencarian biner di y dan di x bersama-sama memancangkan urutan segalanya:

$$\text{kunci}(P_1) \;<\; x \;<\; \text{kunci}(P_2) \;<\; y \;<\; \text{kunci}(P_3). \tag{6.10}$$

Bacalah Persamaan (6.10) dari kiri ke kanan: P_1, lalu kunci tunggal x, lalu P_2, lalu kunci tunggal y, lalu P_3. Itu persis penelusuran inorder pohon sebelah kiri.

Sekarang lihat gambar sebelah kanannya. Simpul x berada di puncak; subpohon kirinya adalah P_1; anak kanannya adalah y; dan y punya P_2 di kiri dan P_3 di kanan. Baca penelusuran inorder *gambar itu*: P_1, x, P_2, y, P_3.

Kedua penelusuran inorder itu sama persis. Itulah seluruh inti sebuah rotasi, dan ia layak dinyatakan sebagai sebuah teorema.

> **TEOREMA (rotasi mempertahankan sifat pohon pencarian biner).** Sebuah rotasi mengubah *bentuk* sebuah pohon tanpa mengubah *barisan inorder*-nya. Karena sebuah pohon biner merupakan pohon pencarian biner tepat ketika penelusuran inordernya naik (§6.6.1), dan sebuah rotasi meninggalkan penelusuran inordernya tanpa sentuhan, rotasi yang dilakukan pada pohon pencarian biner selalu menghasilkan pohon pencarian biner.

6.16.1 Rotasi kanan, lengkap dengan pembaruan penunjuk yang persis

Rotasi kanan terhadap y membawa gambar sebelah kiri **Gambar 6.9** ke gambar sebelah kanannya. Dengan kata-kata: x naik dan menjadi akar baru subpohon ini; y turun dan menjadi anak kanan x; dan subpohon P_2, yang tadinya subpohon kanan x, diserahkan untuk menjadi subpohon kiri y.

Hanya P_2 yang berganti induk. P_1 tetap bersama x; P_3 tetap bersama y. **Kode 6.7** menuliskan penugasan penunjuknya dalam urutan yang wajib dilakukan, di dalam sebuah program lengkap yang menyimpan tinggi di setiap simpul dan merotasi sebuah rantai kecil yang condong ke kiri.

Kode 6.7 Rotasi kanan beserta pembaruan penunjuknya yang persis, diterapkan pada rantai yang condong ke kiri $30 \to 20 \to 10$. Kompilasi dengan `g++ -std=c++17 -O2 -o rotright rotright.cpp`.

```
// Rotasi kanan beserta pembaruan penunjuknya yang persis, diterapkan pada rantai yang
condong ke kiri.
#include <iostream>
#include <string>
#include <algorithm>

struct Node {
    int   key;
    int   height;                  // daun bertinggi 0, pohon kosong -1
    Node* left;
    Node* right;
    Node(int k) : key(k), height(0), left(nullptr), right(nullptr) {}
};

int  h(const Node* x) { return x ? x->height : -1; }
void refresh(Node* x) { x->height = 1 + std::max(h(x->left), h(x->right)); }

Node* rotateRight(Node* y) {
    Node* x  = y->left;         // 1. ingat simpul yang akan naik
    Node* P2 = x->right;        // 2. ingat subpohon yang akan berganti induk
    x->right = y;               // 3. y menjadi anak kanan x
    y->left  = P2;              // 4. P2 menjadi anak kiri y
    refresh(y);                 // 5. y kini lebih rendah: hitung ulang tingginya LEBIH
DAHULU
    refresh(x);                 // 6. baru x, yang bergantung pada tinggi baru y
    return x;                   // 7. x adalah akar baru subpohon ini
}

std::string shape(const Node* x) {           // kunci[tinggi]{kiri,kanan}; "." berarti
kosong
    if (!x) return ".";
    std::string s = std::to_string(x->key) + "[" + std::to_string(x->height) + "]";
    if (x->left || x->right) s += "{" + shape(x->left) + "," + shape(x->right) + "}";
    return s;
}

int main() {
    Node* a = new Node(30); Node* b = new Node(20); Node* c = new Node(10);
    a->left = b; b->left = c;                   // rantai 30 -> 20 -> 10, condong ke kiri
    refresh(c); refresh(b); refresh(a);
    std::cout << "before: " << shape(a) << "\n";
    Node* root = rotateRight(a);
    std::cout << "after : " << shape(root) << "\n";
    delete a; delete b; delete c;
    return 0;
}
```

Setiap simpul dicetak sebagai `kunci[tinggi]`. Program itu mencetak:

```
before: 30[2]{20[1]{10[0],.},.}
after : 20[1]{10[0],30[0]}
```

Sebelum rotasi, rantainya bertinggi 2; sesudahnya 20 duduk di puncak dengan 10 dan 30 sebagai anak-anaknya, dan tingginya menjadi 1.

Langkah 5 dan langkah 6 di dalam `rotateRight` harus berada dalam urutan itu. Tinggi x dihitung dari tinggi anak-anaknya, dan salah satu anaknya sekarang adalah y; jadi tinggi y harus sudah benar ketika x disegarkan. Membalik kedua baris itu adalah kutu tunggal yang paling sering terjadi dalam pohon AVL yang ditulis tangan, dan ia menghasilkan pohon yang kelihatan benar tetapi diam-diam menyimpan tinggi yang salah.

Seluruh prosedurnya menyentuh sejumlah tetap penunjuk, jadi sebuah rotasi berongkos $\Theta(1)$ — tidak bergantung pada seberapa besar P_1, P_2, dan P_3. Perhatikan bahwa subpohonnya tidak pernah ditelusuri, hanya ditunjuk ulang.

6.16.2 Rotasi kiri

Rotasi kiri terhadap x adalah bayangan cerminnya: ia membawa gambar sebelah kanan **Gambar 6.9** kembali ke gambar sebelah kirinya. Kali ini y naik, x turun menjadi anak kiri y, dan P_2 — kali ini subpohon kiri y — diserahkan kepada x sebagai subpohon kanannya.

Kode 6.8 adalah program lengkap untuk arah ini; fungsi `main`-nya melakukan persis rotasi yang dikerjakan dengan tangan pada contoh berikut.

Kode 6.8 Rotasi kiri, diterapkan pada rantai merosot 10 → 20 → 30. Kompilasi dengan `g++ -std=c++17 -O2 -o rotleft rotleft.cpp`.

```
// Rotasi kiri, bayangan cerminnya, diterapkan pada rantai 10 -> 20 -> 30.
#include <iostream>
#include <string>
#include <algorithm>

struct Node {
    int   key;
    int   height;                  // daun bertinggi 0, pohon kosong -1
    Node* left;
    Node* right;
    Node(int k) : key(k), height(0), left(nullptr), right(nullptr) {}
};

int  h(const Node* x) { return x ? x->height : -1; }
void refresh(Node* x) { x->height = 1 + std::max(h(x->left), h(x->right)); }

Node* rotateLeft(Node* x) {
    Node* y  = x->right;        // simpul yang akan naik
    Node* P2 = y->left;         // subpohon yang akan berganti induk
    y->left  = x;
    x->right = P2;
    refresh(x);                 // x kini lebih rendah, jadi segarkan ia lebih dahulu
    refresh(y);
    return y;
}

std::string shape(const Node* x) {           // kunci[tinggi]{kiri,kanan}; "." berarti
kosong
    if (!x) return ".";
    std::string s = std::to_string(x->key) + "[" + std::to_string(x->height) + "]";
    if (x->left || x->right) s += "{" + shape(x->left) + "," + shape(x->right) + "}";
    return s;
}

void inorder(const Node* x) {
    if (!x) return;
    inorder(x->left); std::cout << x->key << " "; inorder(x->right);
}
```

Kode 6.8 (Lanjutan) *Rotasi kiri, diterapkan pada rantai merosot 10 \to 20 \to 30. Kompilasi dengan g++ -std=c++17 -O2 -o rotleft rotleft.cpp.*

```
int main() {
    Node* a = new Node(10); Node* b = new Node(20); Node* c = new Node(30);
    a->right = b; b->right = c;                  // rantai merosot 10 -> 20 -> 30
    refresh(c); refresh(b); refresh(a);
    std::cout << "before: " << shape(a) << "   inorder: "; inorder(a); std::cout << "\n";
    Node* root = rotateLeft(a);
    std::cout << "after : " << shape(root) << "   inorder: "; inorder(root); std::cout <<
"\n";
    delete a; delete b; delete c;
    return 0;
}
```

Program itu mencetak:

```
before: 10[2]{.,20[1]{.,30[0]}}   inorder: 10 20 30
after : 20[1]{10[0],30[0]}   inorder: 10 20 30
```

Sebuah rotasi dengan tangan. Ambil rantai tiga simpul $10 \to 20 \to 30$, dengan 10 sebagai akarnya, 20 sebagai anak kanannya, dan 30 sebagai anak kanan 20. Tingginya 2; ia pohon yang merosot; dan penelusuran inordernya adalah 10,20,30.

Lakukan rotasi kiri terhadap akarnya, $x = 10$. Maka $y = x$->right $= 20$, dan $P_2 = y$->left, yang kosong. Sekarang y->left $= x$ menjadikan 10 anak kiri 20; dan x->right $= P_2$ menjadikan subpohon kanan 10 kosong. Akar barunya adalah 20, dengan anak 10 dan 30.

Pohon barunya bertinggi 1, satu lebih kecil daripada sebelumnya, dan penelusuran inordernya tetap 10,20,30. Satu rotasi, tiga kali menulis penunjuk, dan pohonnya berubah dari merosot menjadi seimbang sempurna.

6.16.3 Apa yang dilakukan rotasi terhadap tinggi

Inilah fakta yang membuat seluruh rancangannya bekerja. Lihat **Gambar 6.9** sekali lagi. Pada gambar kiri, lintasan dari akar y turun ke dalam P_1 melewati y dan x: kedalaman 2 di atas P_1. Pada gambar kanan, lintasan dari akar x turun ke dalam P_1 hanya melewati x: kedalaman 1.

Jadi rotasi kanan **mengangkat P_1 satu aras** dan **menurunkan P_3 satu aras**, sementara P_2 tetap di tempatnya. Rotasi kiri melakukan kebalikannya. Sebuah rotasi karena itu memindahkan tinggi dari satu sisi subpohon ke sisi yang lain, satu aras pada satu waktu — dan itu persis pembetulan yang diperlukan ketika satu sisi menjadi satu aras kelewat tinggi.

6.17 Empat Kasus Rotasi

Sekarang kita menyuruh rotasi bekerja. Andaikan kita baru saja menyisipkan simpul baru ke dalam pohon AVL memakai penyisipan pohon pencarian biner biasa dari §6.10. Simpul barunya adalah simpul daun, jadi faktor keseimbangannya sendiri 0; tetapi subpohon setiap simpul pada lintasan dari akar turun ke simpul itu baru saja bertumbuh paling banyak satu aras, sehingga sebagian faktor keseimbangan itu mungkin bergeser keluar dari rentang.

Prosedurnya terdiri atas empat langkah.

PENYISIPAN AVL. Misalkan s adalah simpul yang sedang disisipkan.

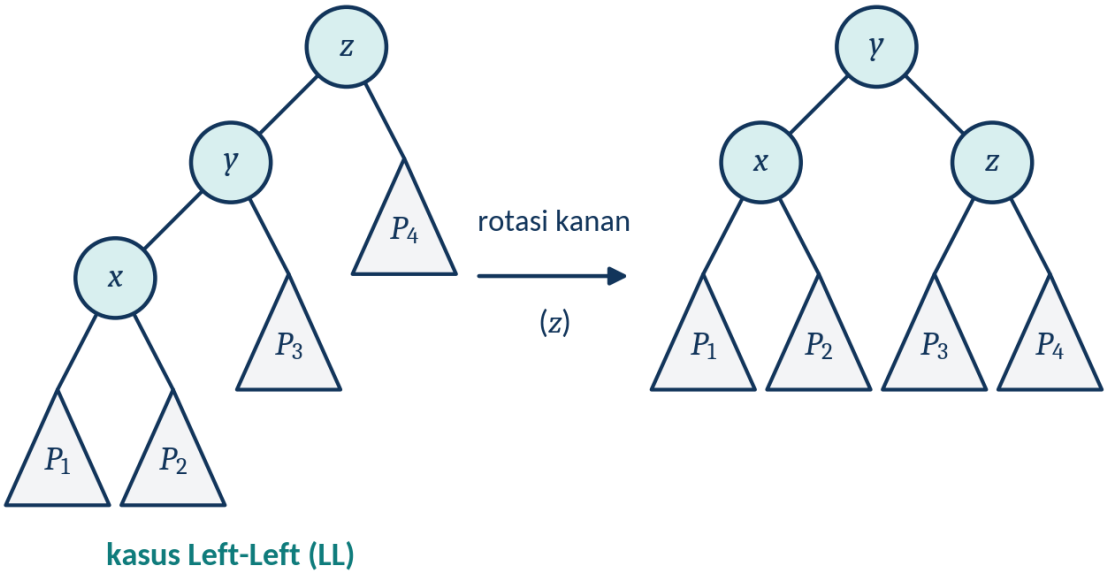
1. Lakukan penyisipan pohon pencarian biner baku atas s.
2. Mulai dari s, berjalanlah **ke atas** lalu temukan simpul pertama yang faktor keseimbangannya sudah meninggalkan $\{-1,0,+1\}$. Sebut simpul itu z — ia adalah simpul tak seimbang yang terdalam.

3. Misalkan y adalah anak z yang terletak pada lintasan dari s naik ke z, dan misalkan x adalah cucu z pada lintasan yang sama itu.
4. Seimbangkan ulang subpohon yang berakar di z dengan rotasi atau rotasi-rotasi yang dituntut kasusnya. Ada tepat empat kasus, karena x bisa duduk pada empat posisi relatif terhadap y dan z:
   - y adalah anak **kiri** z dan x adalah anak **kiri** y — kasus **Kiri-Kiri (LL)**;
   - y adalah anak **kiri** z dan x adalah anak **kanan** y — kasus **Kiri-Kanan (LR)**;
   - y adalah anak **kanan** z dan x adalah anak **kanan** y — kasus **Kanan-Kanan (RR)**;
   - y adalah anak **kanan** z dan x adalah anak **kiri** y — kasus **Kanan-Kiri (RL)**.

Namanya dibaca dari atas ke bawah: huruf pertama mengatakan ke arah mana y menggantung pada z, huruf kedua mengatakan ke arah mana x menggantung pada y. "Kiri-Kiri" berarti pohonnya condong ke kiri dua kali berturut-turut.

Pada keempat gambar yang menyusul, P_1, P_2, P_3, P_4 adalah subpohon, digambar sebagai segitiga karena isinya tidak penting — hanya tingginyalah yang penting. Andaikan sepanjang uraian ini tinggi-tinggi berikut, diukur *sesudah* penyisipannya. Kedua subpohon yang tergantung di bawah x bertinggi k dan $k-1$: yang bertinggi k adalah subpohon yang menerima simpul baru (sebelumnya ia bertinggi $k-1$), sedangkan saudaranya tidak berubah. Kedua subpohon lainnya bertinggi k. Jadi sebelum penyisipan setiap simpul seimbang; penyisipan itulah yang mengangkat x dari tinggi k ke $k+1$, dan satu tingkat tambahan itulah yang merambat naik sampai ke z.

6.17.1 Kasus LL — satu rotasi kanan



Gambar 6.10 Kasus Kiri-Kiri, yang juga muncul pada penghapusan. Satu rotasi kanan terhadap z memulihkan keseimbangannya; y menjadi akar baru subpohonnya, dengan x dan z sebagai anak-anaknya.

Keadaannya. Pada paruh kiri **Gambar 6.10**, z berada di puncak dengan subpohon kanan P_4; y adalah anak kiri z, dengan subpohon kanan P_3; x adalah anak kiri y, dengan subpohon P_1 dan P_2.

Tingginya. P_1 (yang menerima simpul baru), P_3, dan P_4 bertinggi k, dan P_2 bertinggi $k-1$; x bertinggi $k+1$; y bertinggi $\max(k+1, k)+1 = k+2$; z bertinggi $\max(k+2, k)+1 = k+3$. Karena itu

$$\mathrm{bf}(z) = h(P_4) - h(y) = k - (k+2) = -2,$$

yang di luar rentang: z berat kiri sebanyak dua. Dan

$$\mathrm{bf}(y) = h(P_3) - h(x) = k - (k+1) = -1,$$

yang negatif — berat kiri. Dua kecondongan ke kiri berturut-turut: inilah LL.

Perbaikannya. Satu rotasi kanan terhadap z. **Pseudokode 6.1** adalah kode §6.16.1 dengan peran yang diganti namanya, $y \mapsto z$ dan $x \mapsto y$:

Pseudokode 6.1 Perbaikan Kiri-Kiri: satu rotasi kanan terhadap z, dituliskan sebagai keempat penugasan penunjuknya.

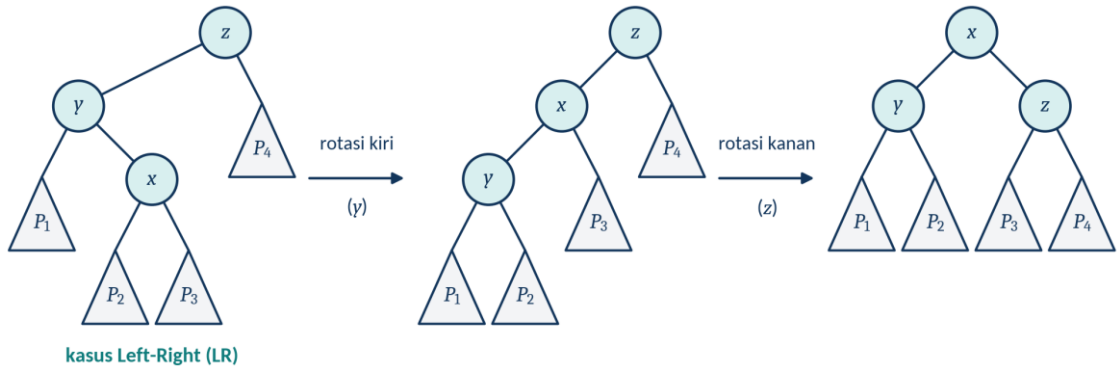
```
y        = z->left
P3       = y->right
y->right = z
z->left  = P3
refresh(z); refresh(y);          // z dahulu, karena ia kini lebih rendah
```

Hasilnya. Paruh kanan **Gambar 6.10**: y di puncak, dengan anak kiri x (yang memegang P_1, P_2) dan anak kanan z (yang memegang P_3, P_4).

Periksa tingginya sesudah itu. x tetap bertinggi $k+1$ dengan faktor keseimbangan -1. z kini punya anak P_3 dan P_4, keduanya bertinggi k, jadi $h(z) = k+1$ dan $\mathrm{bf}(z) = 0$. Dan y punya dua anak yang bertinggi $k+1$, jadi $h(y) = k+2$ dan $\mathrm{bf}(y) = 0$. Setiap faktor keseimbangannya sah.

Dan subpohonnya kembali ke tinggi sebelum penyisipan. Sebelum penyisipannya, subpohon yang berakar di z bertinggi $k+2$; penyisipannya mendorongnya ke $k+3$; sesudah rotasinya ia kembali $k+2$. Inilah pengamatan yang menentukan dari seluruh bagian ini, dan §6.19.2 akan menarik akibatnya.

6.17.2 Kasus LR — rotasi kiri lalu rotasi kanan



Gambar 6.11 Kasus Kiri-Kanan, yang juga muncul pada penghapusan. Dua rotasi diperlukan: pertama rotasi kiri terhadap y, yang mengubah persoalannya menjadi kasus LL, lalu rotasi kanan terhadap z.

Keadaannya. Pada sepertiga kiri **Gambar 6.11**, z berada di puncak dengan subpohon kanan P_4; y adalah anak kiri z, dengan subpohon *kiri* P_1; x adalah anak *kanan* y, dengan subpohon P_2 dan P_3.

Tingginya. P_1 dan P_4 bertinggi k; x bertinggi $k+1$ (anaknya yang lebih tinggi bertinggi k); y bertinggi $k+2$; z bertinggi $k+3$. Jadi

$$\mathrm{bf}(z) = k - (k+2) = -2 \quad \text{(berat kiri, di luar rentang)},$$

$$\mathrm{bf}(y) = h(x) - h(P_1) = (k+1) - k = +1 \quad \text{(berat kanan)}.$$

Condong ke kiri, lalu ke kanan: inilah LR.

Mengapa satu rotasi tidak cukup. Cobalah satu rotasi kanan tunggal terhadap z. Rotasi itu akan mengangkat y ke puncak, memberi y subpohon P_1 di kiri dan z di kanan, lalu menyerahkan x — yang merupakan subpohon kanan y — kepada z sebagai subpohon kirinya. Maka z akan punya x (tinggi $k+1$) di kiri dan P_4 (tinggi k) di kanan, sehingga $h(z) = k+2$; sementara anak kiri y, yaitu P_1, bertinggi k. Kita akan memperoleh $\mathrm{bf}(y) = (k+2) - k = +2$: masih rusak, hanya condong ke arah yang lain. Satu rotasi tunggal tidak bisa memperbaiki bentuk zig-zag.

Perbaikannya, dalam dua langkah.

Langkah 1: rotasi kiri terhadap y. Inilah gambar tengah **Gambar 6.11**. Simpul x naik menjadi anak kiri z; y turun menjadi anak kiri x; dan P_2, yang tadinya subpohon kiri x, diserahkan kepada y sebagai subpohon kanannya. Rantai $z \to y \to x$ tadinya zig-zag kiri-lalu-kanan; sekarang rantainya menjadi $z \to x \to y$, yang condong ke kiri dua kali. Persoalannya sudah diubah menjadi kasus LL.

Langkah 2: rotasi kanan terhadap z. Persis seperti di §6.17.1. Simpul x pindah ke puncak; y dan z menjadi kedua anaknya. **Pseudokode 6.2** melakukan kedua langkah itu.

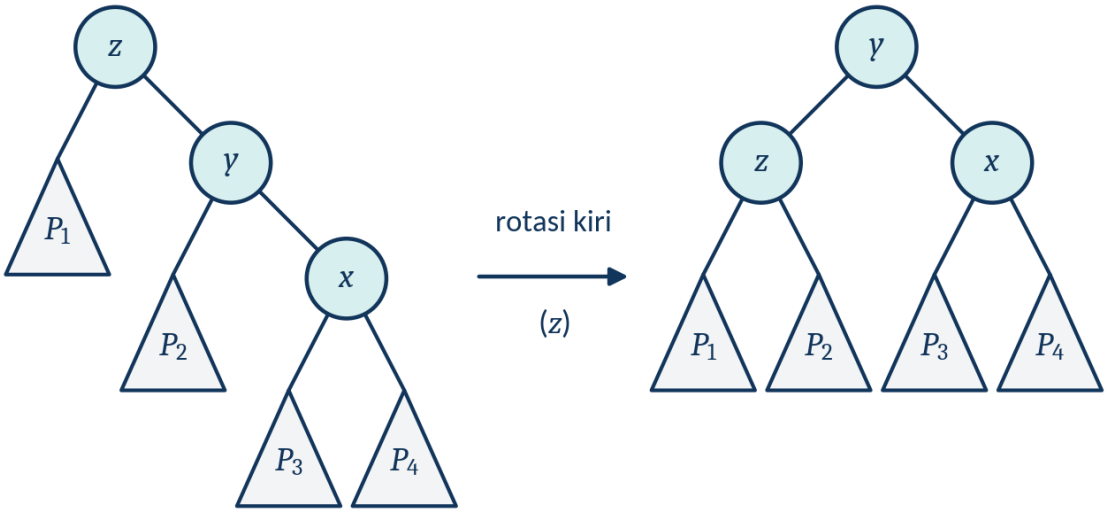
Pseudokode 6.2 Perbaikan Kiri-Kanan: rotasi kiri terhadap y mengubah zig-zagnya menjadi kasus Kiri-Kiri, lalu rotasi kanan terhadap z merampungkannya.

```
z->left = rotateLeft(z->left);   // langkah 1: mengubah LR menjadi LL
return rotateRight(z);           // langkah 2: perbaikan LL yang biasa
```

Hasilnya. Sepertiga kanan **Gambar 6.11**: x di puncak; anak kiri y memegang P_1 dan P_2; anak kanan z memegang P_3 dan P_4.

Periksa tingginya. y punya anak P_1 dan P_2; z punya anak P_3 dan P_4. Masing-masing dari keempat subpohon itu bertinggi k atau $k-1$ — salah satu dari P_2, P_3 bertumbuh karena penyisipannya, tetapi tak satu pun melampaui k. Jadi $h(y) = h(z) = k+1$ paling banyak, dan $h(x) = k+2$, dengan ketiga faktor keseimbangannya di $\{-1, 0, +1\}$. Subpohonnya kembali bertinggi $k+2$, yaitu tinggi sebelum penyisipannya.

6.17.3 Kasus RR — satu rotasi kiri



kasus Right-Right (RR)

Gambar 6.12 Kasus Kanan-Kanan, yang juga muncul pada penghapusan. Bayangan cermin yang persis dari **Gambar 6.10**: satu rotasi kiri terhadap z.

Keadaannya. Pada paruh kiri **Gambar 6.12**, z berada di puncak dengan subpohon kiri P_1; y adalah anak kanan z dengan subpohon kiri P_2; x adalah anak kanan y dengan subpohon P_3 dan P_4.

Tingginya. $h(x) = k + 1, h(y) = k + 2, h(z) = k + 3$, jadi

$$\begin{aligned} \mathrm{bf}(z) &= h(y) - h(P_1) \\ &= (k+2) - k = +2 \quad \text{(berat kanan, di luar rentang)}, \end{aligned}$$

$$\mathrm{bf}(y) = h(x) - h(P_2) = (k+1) - k = +1 \quad \text{(berat kanan)}.$$

Dua kecondongan ke kanan: RR.

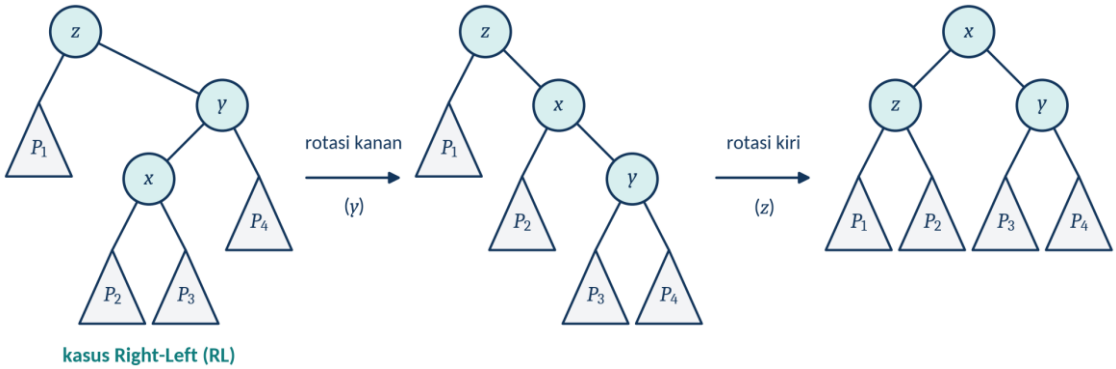
Perbaikannya. Satu rotasi kiri terhadap z, pada **Pseudokode 6.3**:

Pseudokode 6.3 Perbaikan Kanan-Kanan: satu rotasi kiri terhadap z, bayangan cermin **Pseudokode 6.1**.

```
y        = z->right
P2       = y->left
y->left  = z
z->right = P2
refresh(z); refresh(y);
```

Hasilnya. Paruh kanan **Gambar 6.12**: y di puncak, anak kiri z (memegang P_1, P_2), anak kanan x (memegang P_3, P_4). Simpul z dan y kini berfaktor keseimbangan 0, x mempertahankan ± 1 yang sudah dimilikinya, dan subpohonnya kembali bertinggi $k + 2$.

6.17.4 Kasus RL — rotasi kanan lalu rotasi kiri



Gambar 6.13 Kasus Kanan-Kiri, yang juga muncul pada penghapusan. Cermin dari **Gambar 6.11**: pertama rotasi kanan terhadap y, yang mengubah zig-zagnya menjadi kasus RR, lalu rotasi kiri terhadap z.

Keadaannya. Pada sepertiga kiri **Gambar 6.13**, z punya subpohon kiri P_1; y adalah anak kanan z dengan subpohon kanan P_4; x adalah anak *kiri* y, dengan subpohon P_2 dan P_3.

Tingginya. $\text{bf}(z) = +2$ dan $\text{bf}(y) = h(P_4) - h(x) = k - (k+1) = -1$. Condong ke kanan, lalu ke kiri: RL.

Perbaikannya, dalam dua langkah. Rotasi kanan terhadap y menarik x naik ke tempat y dan mendorong y turun ke kanan — gambar tengahnya — mengubah zig-zagnya menjadi kecondongan ganda ke kanan. Kemudian rotasi kiri terhadap z merampungkan pekerjaannya, seperti ditunjukkan **Pseudokode 6.4**.

Pseudokode 6.4 Perbaikan Kanan-Kiri: rotasi kanan terhadap y, lalu rotasi kiri terhadap z.

```
z->right = rotateRight(z->right);   // langkah 1: mengubah RL menjadi RR
return rotateLeft(z);               // langkah 2: perbaikan RR yang biasa
```

Hasilnya. Sepertiga kanan **Gambar 6.13**: x di puncak; anak kiri z memegang P_1 dan P_2; anak kanan y memegang P_3 dan P_4. Tingginya $k+2$ lagi.

6.17.5 Membedakan keempat kasus di dalam kode

Aturan keputusannya hanya memerlukan dua faktor keseimbangan: yang di z dan yang di y.

Tabel 6.5 Keempat kasus penyeimbangan ulang, dikenali lewat dua faktor keseimbangan, beserta rotasi yang dituntut masing-masingnya.

bf(z)	bf(y), dengan y adalah anak z yang lebih tinggi	Kasus	Rotasi	Gambar
< -1 (berat kiri)	negatif	Kiri-Kiri	rotasi kanan terhadap z	**Gambar 6.10**
< -1 (berat kiri)	positif	Kiri-Kanan	kiri terhadap y, lalu kanan terhadap z	**Gambar 6.11**
$> +1$ (berat kanan)	positif	Kanan-Kanan	rotasi kiri terhadap z	**Gambar 6.12**
$> +1$ (berat kanan)	negatif	Kanan-Kiri	kanan terhadap y, lalu kiri terhadap z	**Gambar 6.13**

Aturan yang sama dapat diungkapkan sebagai pembandingan tanda di sepanjang lintasan $z \to y \to x \to s$: untuk LL tandanya berjalan $(-)$ ke $(-)$, untuk LR berjalan $(-)$ ke $(+)$, untuk RR $(+)$ ke $(+)$, dan untuk RL $(+)$ ke $(-)$. Aturan yang sama, diucapkan dengan suara keras.

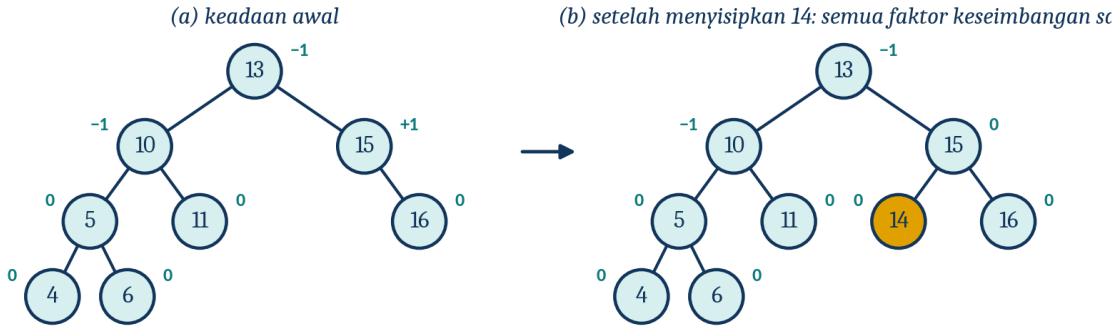
Bagaimana kalau $\mathrm{bf}(y) = 0$**?** Sesudah sebuah *penyisipan* hal ini tidak mungkin terjadi: subpohon yang berakar di y bertumbuh, jadi salah satu sisinya tegas lebih tinggi. Sesudah sebuah *penghapusan* hal ini bisa dan memang terjadi, dan kalau begitu rotasi tunggallah yang memulihkan invariannya. Kesepakatan yang berlaku umum — dan yang dipakai dalam kode §6.19 — adalah memperlakukan $\mathrm{bf}(y) = 0$ sebagai kasus rotasi tunggal, yaitu menuliskan `if (bf(z->left) <= 0) rotateRight(z);` alih-alih `< 0`. Salah di sini dan melakukan rotasi ganda dapat meninggalkan y dengan faktor keseimbangan ± 2: pohonnya masih benar sebagai pohon pencarian, tetapi bukan lagi pohon AVL.

6.18 Sebuah Contoh Penyisipan, Selangkah demi Selangkah

Gambar 6.14 sampai **Gambar 6.18** mengerjakan lima penyisipan, satu per gambar, dan bersama-sama kelimanya melatih setiap kasus dalam **Tabel 6.5**. Kita mereproduksi kelimanya di sini satu per satu, dengan faktor keseimbangan yang dihitung dengan tangan dan dikukuhkan oleh **Kode 6.9** di §6.19.

Sepanjang uraian ini, sebuah faktor keseimbangan dituliskan di samping simpulnya pada gambar-gambar ini, dengan warna hijau toska, memakai kesepakatan yang sama dengan kesepakatan kita: tinggi kanan dikurangi tinggi kiri.

6.18.1 Langkah 1 — sisipkan 14: tidak perlu penyeimbangan ulang



Gambar 6.14 Menyisipkan 14 ke dalam sebuah pohon AVL. Setiap faktor keseimbangannya tetap berada di dalam $\{-1, 0, +1\}$, jadi tidak ada rotasi yang dilakukan.

Pohon di sebelah kiri **Gambar 6.14** adalah: akar 13; anak kiri 10 dengan anak 5 dan 11; simpul 5 dengan anak 4 dan 6; anak kanan 15 dengan anak kanan 16.

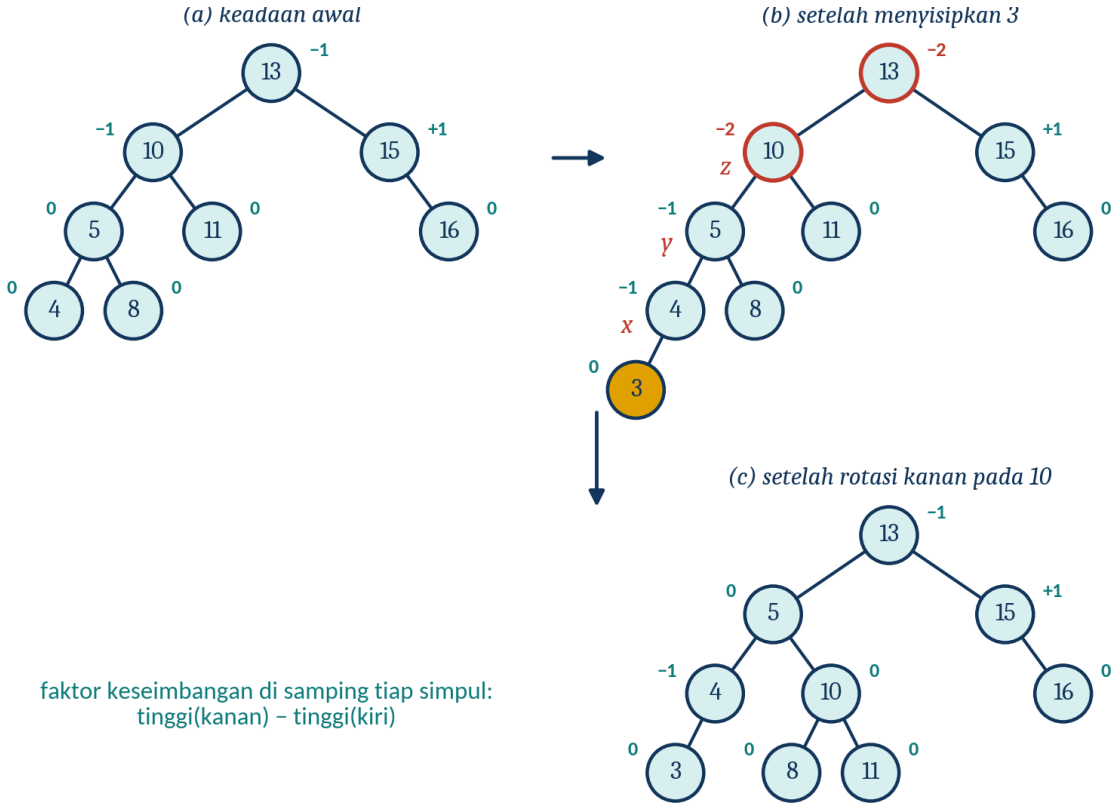
Sisipkan 14. Penyusurannya membandingkan 14 dengan 13 — lebih besar, ke kanan — lalu dengan 15 — lebih kecil, ke kiri — lalu menemukan slot kosong. Simpul 14 menjadi anak kiri 15.

Sekarang hitung ulang faktor keseimbangannya ke atas dari simpul barunya.

- Simpul 14 adalah simpul daun: $\mathrm{bf} = 0$, tinggi 0.
- Simpul 15 kini punya anak kiri 14 (tinggi 0) dan anak kanan 16 (tinggi 0). $\mathrm{bf}(15) = 0 - 0 = 0$, tinggi 1. Sebelum penyisipannya ia $+1$; penyisipannya justru *memperbaikinya*.
- Simpul 13 punya subpohon kiri yang berakar di 10, bertinggi 2, dan subpohon kanan yang berakar di 15, bertinggi 1. $\mathrm{bf}(13) = 1 - 2 = -1$, tinggi 3.

Tidak ada faktor keseimbangan yang meninggalkan rentang sahnya, jadi tidak ada yang perlu diperbaiki. Inilah kasus yang lazim: sebagian besar penyisipan ke dalam pohon AVL sama sekali tidak memerlukan rotasi.

6.18.2 Langkah 2 — sisipkan 3: kasus Kiri-Kiri



Gambar 6.15 Menyisipkan 3 menghasilkan faktor keseimbangan −2 di simpul 10. Kasusnya adalah Kiri-Kiri, dan satu rotasi kanan dengan 10 sebagai porosnya memperbaikinya.

Pohon awal pada **Gambar 6.15** adalah: akar 13; anak kiri 10 dengan anak 5 dan 11; simpul 5 dengan anak 4 dan 8; anak kanan 15 dengan anak kanan 16.

Sisipkan 3: di 13 ke kiri, di 10 ke kiri, di 5 ke kiri, di 4 ke kiri masuk ke slot kosong. Simpul 3 menjadi anak kiri 4.

Berjalanlah kembali ke atas sambil menghitung ulang.

- Simpul 3: simpul daun, tinggi 0, bf $= 0$.
- Simpul 4: anak kiri 3 (tinggi 0), kanan kosong (tinggi -1). bf(4) $= -1 - 0 = -1$, tinggi 1. Sah.
- Simpul 5: subpohon kiri berakar di 4 (tinggi 1), anak kanan 8 (tinggi 0). bf(5) $= 0 - 1 = -1$, tinggi 2. Sah.
- Simpul 10: subpohon kiri berakar di 5 (tinggi 2), anak kanan 11 (tinggi 0). bf(10) $= 0 - 2 = \mathbf{-2}$. **Tidak sah.** Berhenti di sini: $z = 10$.

Kenali ketiga simpulnya. $z = 10$. Lintasan dari simpul baru 3 naik ke z berjalan $3 \to 4 \to 5 \to 10$, jadi anak z pada lintasan itu adalah $y = 5$ dan cucunya adalah $x = 4$. Simpul y adalah anak *kiri* z; simpul x adalah anak *kiri* y. Kiri-Kiri.

Cek silang dengan **Tabel 6.5**: bf(z) $= -2 < -1$, dan bf(y) $=$ bf(5) $= -1$, negatif. LL terkukuhkan.

Terapkan satu rotasi kanan terhadap $z = 10$, persis seperti yang diresepkan **Gambar 6.10**. Di sini $x = 5$ menurut penamaan kode rotasinya (anak kiri porosnya), dan subpohon yang berganti induk adalah anak kanan 5, yaitu simpul 8.

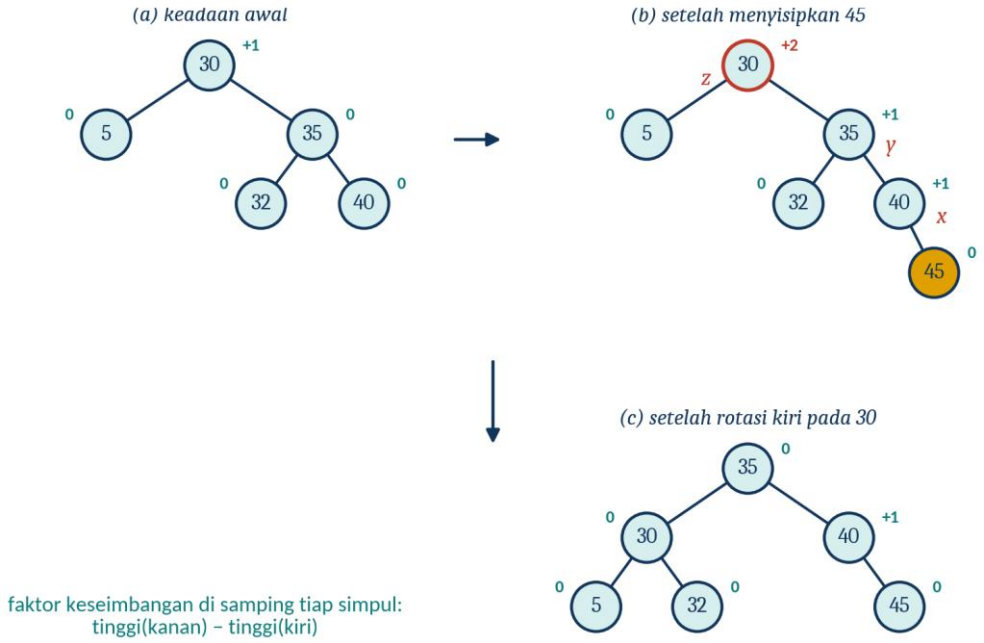
1. $x \leftarrow$ 10->left $= 5$.
2. $P_2 \leftarrow$ 5->right $= 8$.

3. 5->right ← 10.
4. 10->left ← 8.
5. Segarkan 10, lalu 5.
6. Akar subpohon yang baru, yaitu 5, ditempelkan di tempat 10 dulu berada: sebagai anak kiri 13.

Hasilnya adalah pohon di kanan bawah **Gambar 6.15**: akar 13; anak kiri 5 dengan anak 4 (memegang 3) dan 10 (memegang 8 dan 11); anak kanan 15 dengan anak kanan 16.

Periksa: $\text{bf}(4) = -1$; $\text{bf}(10) = 0$; $\text{bf}(5) = h(10) - h(4) = 1 - 1 = 0$; $\text{bf}(13) = h(15) - h(5) = 1 - 2 = -1$. Semuanya sah. Tinggi seluruh pohonnya 3, sama seperti sebelum penyisipannya — dan itulah sebabnya tidak ada rotasi lanjutan yang diperlukan lebih ke atas.

6.18.3 Langkah 3 — sisipkan 45: kasus Kanan-Kanan



Gambar 6.16 Menyisipkan 45 membuat simpul 30 berat kanan sebanyak dua. Kasusnya adalah Kanan-Kanan, dan satu rotasi kiri dengan 30 sebagai porosnya memperbaikinya.

Pohon awal pada **Gambar 6.16** adalah: akar 30, anak kiri 5, anak kanan 35 dengan anak 32 dan 40.

Sisipkan 45: di 30 ke kanan, di 35 ke kanan, di 40 ke kanan masuk ke slot kosong. Simpul 45 menjadi anak kanan 40.

Berjalan ke atas:

- Simpul 45: simpul daun, tinggi 0, $\text{bf} = 0$.
- Simpul 40: kiri kosong (tinggi -1), anak kanan 45 (tinggi 0). $\text{bf}(40) = 0 - (-1) = +1$, tinggi 1. Sah.
- Simpul 35: anak kiri 32 (tinggi 0), subpohon kanan berakar di 40 (tinggi 1). $\text{bf}(35) = 1 - 0 = +1$, tinggi 2. Sah.
- Simpul 30: anak kiri 5 (tinggi 0), subpohon kanan berakar di 35 (tinggi 2). $\text{bf}(30) = 2 - 0 = +\mathbf{2}$. **Tidak sah.** Jadi $z = 30$.

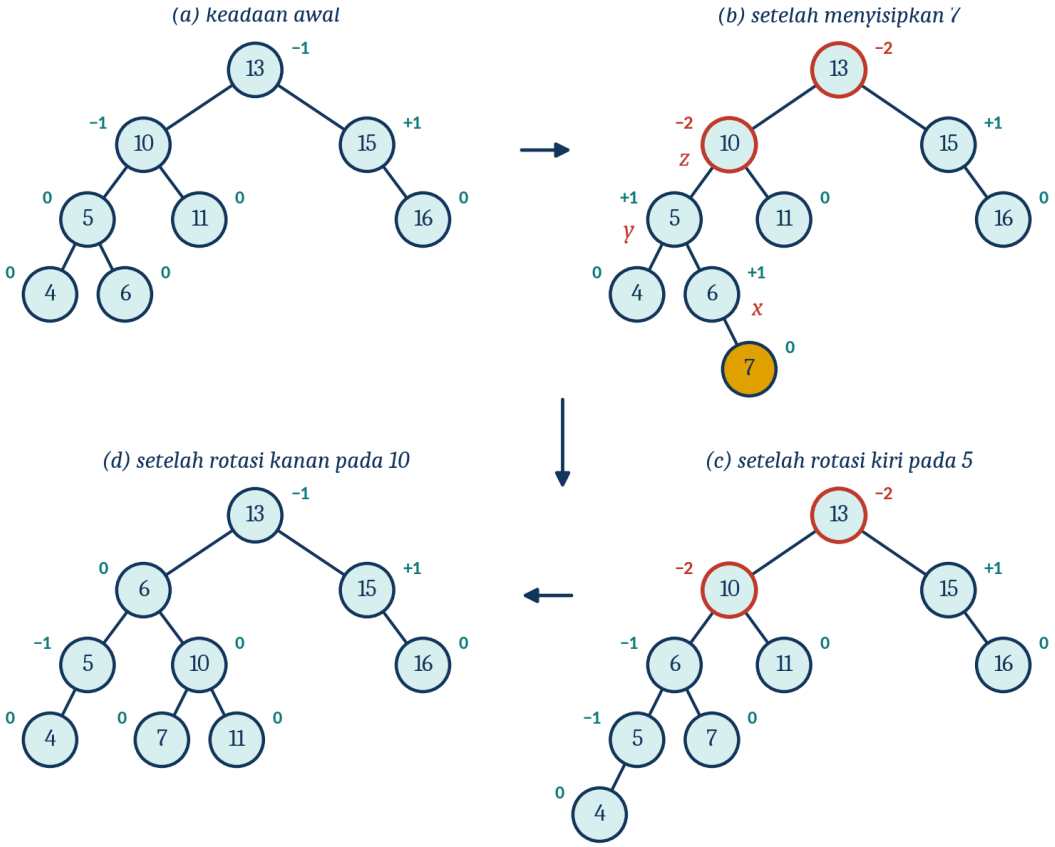
Lintasan dari 45 ke atas adalah $45 \to 40 \to 35 \to 30$, jadi $y = 35$ dan $x = 40$. Simpul y adalah anak kanan z; simpul x adalah anak kanan y. Kanan-Kanan — dan $\text{bf}(y) = +1$ positif, cocok dengan baris RR di **Tabel 6.5**.

Terapkan satu rotasi kiri terhadap $z = 30$:

1. $y \leftarrow$ 30->right $= 35$.
2. $P_2 \leftarrow$ 35->left $= 32$.
3. 35->left $\leftarrow 30$.
4. 30->right $\leftarrow 32$.
5. Segarkan 30, lalu 35.

Hasilnya adalah pohon di kanan bawah **Gambar 6.16**: akar 35; anak kiri 30 dengan anak 5 dan 32; anak kanan 40 dengan anak kanan 45. Faktor keseimbangannya: bf(30) $= 0$, bf(40)$= +1$, bf(35) $= h(40) - h(30) = 1 - 1 = 0$. Semuanya sah.

6.18.4 Langkah 4 — sisipkan 7: kasus Kiri-Kanan



Gambar 6.17 Menyisipkan 7 menciptakan zig-zag — ke kiri di 10, lalu ke kanan di 5. Dua rotasi diperlukan: kiri dengan 5 sebagai poros, lalu kanan dengan 10 sebagai poros.

Pohon awalnya sama dengan pohon di langkah 1: akar 13; anak kiri 10 dengan anak 5 dan 11; simpul 5 dengan anak 4 dan 6; anak kanan 15 dengan anak kanan 16.

Sisipkan 7: di 13 ke kiri, di 10 ke kiri, di 5 ke kanan, di 6 ke kanan masuk ke slot kosong. Simpul 7 menjadi anak kanan 6.

Berjalan ke atas:

- Simpul 7: simpul daun, tinggi 0.
- Simpul 6: anak kanan 7, kiri kosong. bf(6) $= 0 - (-1) = +1$, tinggi 1. Sah.
- Simpul 5: anak kiri 4 (tinggi 0), subpohon kanan berakar di 6 (tinggi 1). bf(5) $= 1 - 0 = +1$, tinggi 2. Sah.

- Simpul 10: subpohon kiri berakar di 5 (tinggi 2), anak kanan 11 (tinggi 0). $\text{bf}(10) = 0 - 2 = \mathbf{-2}$. **Tidak sah.** Jadi $z = 10$.

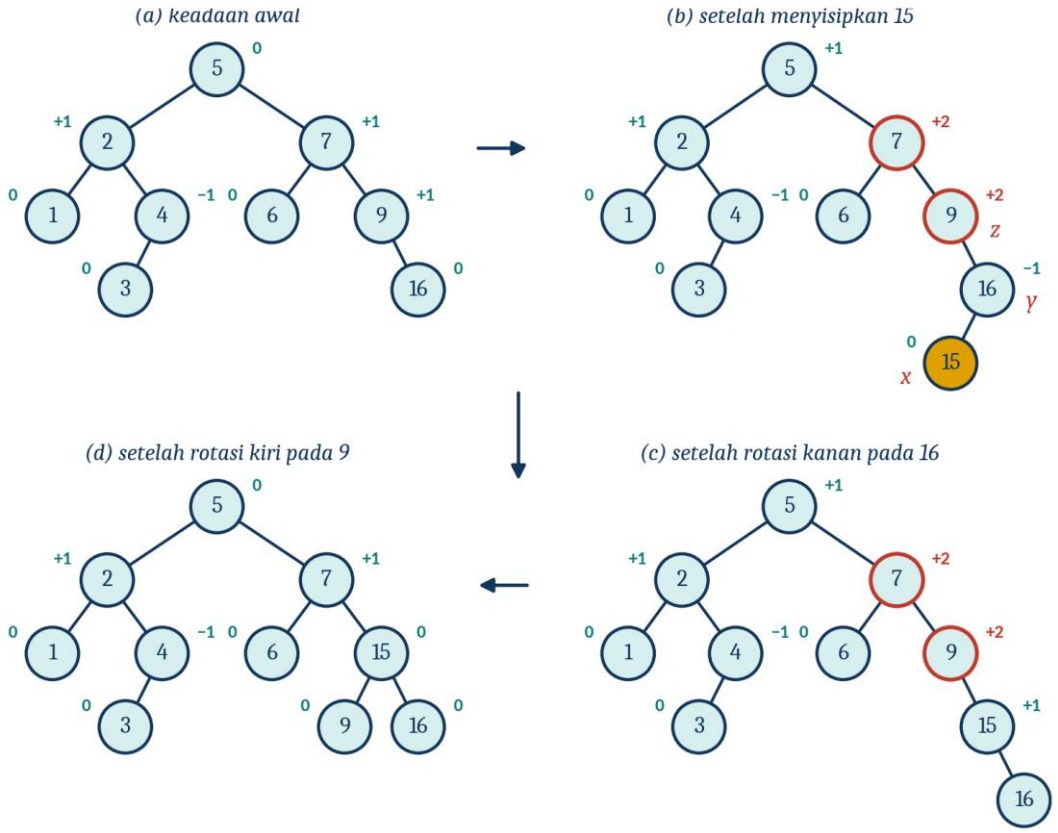
Lintasan dari 7 ke atas adalah $7 \to 6 \to 5 \to 10$, jadi $y = 5$ dan $x = 6$. Simpul y adalah anak *kiri* z; simpul x adalah anak *kanan* y. Kiri-Kanan — dan $\text{bf}(y) = +1$ positif, cocok dengan baris LR di **Tabel 6.5**.

Langkah 1: rotasi kiri terhadap $y = 5$. Simpul 6 naik ke tempat 5 sebagai anak kiri 10; simpul 5 menjadi anak kiri 6; subpohon kiri lama milik 6 — yang di sini kosong — pergi kepada 5 sebagai subpohon kanannya. Tulang punggung kiri 10 sekarang terbaca $10 \to 6 \to 5 \to 4$, yaitu kecondongan ganda ke kiri. Inilah panel (c) **Gambar 6.17**, yang di dalamnya simpul 10 masih menunjukkan -2 dan rantai di bawahnya kini 6 lalu 5 lalu 4.

Langkah 2: rotasi kanan terhadap $z = 10$. Simpul 6 pindah ke puncak subpohon ini; simpul 10 menjadi anak kanan 6; subpohon kanan lama milik 6, yaitu simpul tunggal 7, diserahkan kepada 10 sebagai subpohon kirinya.

Hasilnya adalah pohon di kiri bawah **Gambar 6.17**: akar 13; anak kiri 6 dengan anak 5 (memegang 4) dan 10 (memegang 7 dan 11); anak kanan 15 dengan anak kanan 16. Faktor keseimbangannya: $\text{bf}(5) = -1$, $\text{bf}(10) = 0$, $\text{bf}(6) = h(10) - h(5) = 1 - 1 = 0$, $\text{bf}(13) = 1 - 2 = -1$. Semuanya sah.

6.18.5 Langkah 5 — sisipkan 15: kasus Kanan-Kiri



Gambar 6.18 Menyisipkan 15 membuat simpul 9 berat kanan sebanyak dua sementara simpul 16 condong ke kiri. Kasusnya adalah Kanan-Kiri: rotasi kanan dengan 16 sebagai poros, lalu rotasi kiri dengan 9 sebagai poros.

Pohon awal pada **Gambar 6.18** adalah: akar 5; anak kiri 2 dengan anak 1 dan 4, dan simpul 4 punya anak kiri 3; anak kanan 7 dengan anak 6 dan 9, dan simpul 9 punya anak kanan 16.

Sisipkan 15: di 5 ke kanan, di 7 ke kanan, di 9 ke kanan, di 16 ke kiri masuk ke slot kosong. Simpul 15 menjadi anak kiri 16.

Berjalan ke atas:

- Simpul 15: simpul daun, tinggi 0.
- Simpul 16: anak kiri 15 (tinggi 0), kanan kosong. $\mathrm{bf}(16) = -1 - 0 = -1$, tinggi 1. Sah.
- Simpul 9: kiri kosong (tinggi -1), subpohon kanan berakar di 16 (tinggi 1). $\mathrm{bf}(9) = 1 - (-1) = +\mathbf{2}$. **Tidak sah.** Jadi $z = 9$.

Lintasan dari 15 ke atas adalah $15 \to 16 \to 9$, jadi $y = 16$ dan $x = 15$. Simpul y adalah anak *kanan* z; simpul x adalah anak *kiri* y. Kanan-Kiri — dan $\mathrm{bf}(y) = -1$ negatif, cocok dengan baris RL di **Tabel 6.5**.

Langkah 1: rotasi kanan terhadap $y = 16$. Simpul 15 naik menjadi anak kanan 9; simpul 16 menjadi anak kanan 15. Rantai di bawah 9 sekarang terbaca $9 \to 15 \to 16$, yaitu kecondongan ganda ke kanan.

Langkah 2: rotasi kiri terhadap $z = 9$. Simpul 15 naik ke tempat 9 sebagai anak kanan 7; simpul 9 menjadi anak kiri 15.

Hasilnya adalah pohon di kiri bawah **Gambar 6.18**: akar 5; anak kiri 2 tidak berubah; anak kanan 7 dengan anak kiri 6 dan anak kanan 15, dan 15 punya anak 9 dan 16. Faktor keseimbangannya: $\mathrm{bf}(9) = 0$, $\mathrm{bf}(16) = 0$, $\mathrm{bf}(15) = 0$, $\mathrm{bf}(7) = h(15) - h(6) = 1 - 0 = +1$, $\mathrm{bf}(5) = h(7) - h(2) = 2 - 2 = 0$. Semuanya sah.

Lima penyisipan, empat kasus, dan setiap satunya diperbaiki oleh paling banyak dua rotasi dan $O(1)$ operasi tulis penunjuk.

6.19 Pohon AVL yang Lengkap dalam C++

Kode 6.9 adalah keseluruhan strukturnya: tinggi yang disimpan di dalam simpul, kedua rotasi, penyeimbang ulang empat kasus, penyisipan, penghapusan, pencetak bentuk yang menampilkan setiap faktor keseimbangan, dan pemeriksaan mandiri yang memverifikasi invariannya di setiap simpul. Fungsi `main`-nya memutar ulang kelima langkah penyisipan §6.18 dan contoh penghapusan §6.20, sehingga setiap klaim yang dibuat dengan tangan dalam bab ini diperiksa oleh sebuah program.

Kode 6.9 Pohon AVL yang lengkap. Kompilasi dengan `g++ -std=c++17 -O2 -o avl avl.cpp` lalu jalankan `./avl`.

```
// Pohon AVL yang lengkap dalam C++: penyisipan dengan keempat kasus rotasi,
// penghapusan dengan penyeimbangan ulang sampai ke akar, dan pencetak yang
// menampilkan faktor keseimbangan tiap simpul. main() memutar ulang kelima
// contoh penyisipan di Subbab 6.18 dan contoh penghapusan di Subbab 6.20.
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

class AVL {
public:
    AVL() : root_(nullptr), rotations_(0) {}
    ~AVL() { destroy(root_); }

    void insert(int key) { root_ = insertRec(root_, key); }
    void remove(int key) { root_ = removeRec(root_, key); }

    int  rotations() const { return rotations_; }
    void resetRotations()  { rotations_ = 0; }

    int height() const { return h(root_); }

    // kunci(faktor keseimbangan){subpohon kiri, subpohon kanan}; "." adalah pohon kosong.
    std::string shape() const { return shapeRec(root_); }

    bool isAVL() const { return checkRec(root_); }

private:
    struct Node {
        int key;
        int height;                 // tinggi simpul daun 0, tinggi pohon kosong -1
        Node* left;
        Node* right;
        Node(int k) : key(k), height(0), left(nullptr), right(nullptr) {}
    };
    Node* root_;
    int   rotations_;

    static int h(const Node* x) { return x ? x->height : -1; }
    static int bf(const Node* x) { return x ? h(x->right) - h(x->left) : 0; }
    static void refresh(Node* x) { x->height = 1 + std::max(h(x->left), h(x->right)); }

    static void destroy(Node* x) { if (!x) return; destroy(x->left); destroy(x->right);
delete x; }
```

Kode 6.9 (Lanjutan) *Pohon AVL yang lengkap. Kompilasi dengan g++ -std=c++17 -O2 -o avl avl.cpp lalu jalankan ./avl.*

```
/*      y                 x
 *     / \               / \
 *    x   P3    ==>    P1   y
 *   / \                   / \
 *  P1  P2               P2   P3
 */
Node* rotateRight(Node* y) {
    Node* x  = y->left;
    Node* P2 = x->right;
    x->right = y;               // y menjadi anak kanan x
    y->left  = P2;              // subpohon tengah berganti induk
    refresh(y);                 // y kini lebih rendah, jadi hitung ulang ia dahulu
    refresh(x);
    ++rotations_;
    return x;                   // x adalah akar baru subpohon ini
}

/*    x                   y
 *   / \                 / \
 *  P1  y        ==>    x   P3
 *     / \             / \
 *    P2  P3          P1  P2
 */
Node* rotateLeft(Node* x) {
    Node* y  = x->right;
    Node* P2 = y->left;
    y->left  = x;
    x->right = P2;
    refresh(x);
    refresh(y);
    ++rotations_;
    return y;
}

Node* rebalance(Node* z) {
    refresh(z);
    int b = bf(z);
    if (b < -1) {                               // sisi kiri terlalu tinggi
        if (bf(z->left) <= 0) return rotateRight(z);          // Kiri-Kiri
        z->left = rotateLeft(z->left);                        // Kiri-Kanan
        return rotateRight(z);
    }
    if (b > 1) {                                // sisi kanan terlalu tinggi
        if (bf(z->right) >= 0) return rotateLeft(z);          // Kanan-Kanan
        z->right = rotateRight(z->right);                     // Kanan-Kiri
        return rotateLeft(z);
```

Kode 6.9 (Lanjutan) *Pohon AVL yang lengkap. Kompilasi dengan g++ -std=c++17 -O2 -o avl avl.cpp lalu jalankan ./avl.*

```
        }
        return z;                                   // sudah seimbang
    }

    Node* insertRec(Node* x, int key) {
        if (!x) return new Node(key);
        if      (key < x->key) x->left  = insertRec(x->left,  key);
        else if (key > x->key) x->right = insertRec(x->right, key);
        else return x;                              // kunci kembar: tidak berubah
        return rebalance(x);
    }

    static Node* minNode(Node* x) { while (x->left) x = x->left; return x; }

    Node* removeRec(Node* x, int key) {
        if (!x) return nullptr;
        if      (key < x->key) x->left  = removeRec(x->left,  key);
        else if (key > x->key) x->right = removeRec(x->right, key);
        else {
            if (!x->left || !x->right) {            // tanpa anak, atau tepat satu anak
                Node* c = x->left ? x->left : x->right;
                delete x;
                if (!c) return nullptr;
                x = c;
            } else {
                Node* s = minNode(x->right);        // dua anak: penerus inorder
                x->key   = s->key;
                x->right = removeRec(x->right, s->key);
            }
        }
        return rebalance(x);                        // seimbangkan ulang saat menaik
kembali
    }

    static std::string shapeRec(const Node* x) {
        if (!x) return ".";
        if (!x->left && !x->right)
            return std::to_string(x->key) + "(" + std::to_string(bf(x)) + ")";
        return std::to_string(x->key) + "(" + std::to_string(bf(x)) + "){"
             + shapeRec(x->left) + "," + shapeRec(x->right) + "}";
    }
```

Kode 6.9 (Lanjutan) *Pohon AVL yang lengkap. Kompilasi dengan g++ -std=c++17 -O2 -o avl avl.cpp lalu jalankan ./avl.*

```
    static bool checkRec(const Node* x) {
        if (!x) return true;
        if (bf(x) < -1 || bf(x) > 1) return false;
        return checkRec(x->left) && checkRec(x->right);
    }
};

void step(const char* title, const std::vector<int>& build, int newKey) {
    AVL t;
    for (int k : build) t.insert(k);
    std::cout << title << "\n";
    std::cout << "  before      : " << t.shape() << "\n";
    t.resetRotations();
    t.insert(newKey);
    std::cout << "  insert " << newKey << "    : " << t.shape() << "\n";
    std::cout << "  rotations   : " << t.rotations()
              << ",  height " << t.height()
              << ",  still AVL: " << (t.isAVL() ? "yes" : "no") << "\n\n";
}

int main() {
    step("Step 1  no rebalancing needed", {13,10,15,5,11,16,4,6}, 14);
    step("Step 2  Left-Left case",        {13,10,15,5,11,16,4,8},  3);
    step("Step 3  Right-Right case",      {30,5,35,32,40},        45);
    step("Step 4  Left-Right case",       {13,10,15,5,11,16,4,6},  7);
    step("Step 5  Right-Left case",       {5,2,7,1,4,6,9,3,16},   15);

    AVL d;
    for (int k : {44,17,62,32,50,78,48,54,88}) d.insert(k);
    std::cout << "Deletion example\n";
    std::cout << "  before      : " << d.shape() << "\n";
    d.resetRotations();
    d.remove(32);
    std::cout << "  delete 32   : " << d.shape() << "\n";
    std::cout << "  rotations   : " << d.rotations()
              << ",  height " << d.height()
              << ",  still AVL: " << (d.isAVL() ? "yes" : "no") << "\n";
    return 0;
}
```

Programnya mencetak:

```
Step 1  no rebalancing needed
  before      : 13(-1){10(-1){5(0){4(0),6(0)},11(0)},15(1){.,16(0)}}
  insert 14    : 13(-1){10(-1){5(0){4(0),6(0)},11(0)},15(0){14(0),16(0)}}
  rotations   : 0,  height 3,  still AVL: yes

Step 2  Left-Left case
  before      : 13(-1){10(-1){5(0){4(0),8(0)},11(0)},15(1){.,16(0)}}
  insert 3    : 13(-1){5(0){4(-1){3(0),.},10(0){8(0),11(0)}},15(1){.,16(0)}}
  rotations   : 1,  height 3,  still AVL: yes

Step 3  Right-Right case
  before      : 30(1){5(0),35(0){32(0),40(0)}}
  insert 45    : 35(0){30(0){5(0),32(0)},40(1){.,45(0)}}
  rotations   : 1,  height 2,  still AVL: yes

Step 4  Left-Right case
  before      : 13(-1){10(-1){5(0){4(0),6(0)},11(0)},15(1){.,16(0)}}
  insert 7    : 13(-1){6(0){5(-1){4(0),.},10(0){7(0),11(0)}},15(1){.,16(0)}}
  rotations   : 2,  height 3,  still AVL: yes

Step 5  Right-Left case
  before      : 5(0){2(1){1(0),4(-1){3(0),.}},7(1){6(0),9(1){.,16(0)}}}
  insert 15    : 5(0){2(1){1(0),4(-1){3(0),.}},7(1){6(0),15(0){9(0),16(0)}}}
  rotations   : 2,  height 3,  still AVL: yes

Deletion example
  before      : 44(1){17(1){.,32(0)},62(0){50(0){48(0),54(0)},78(1){.,88(0)}}}
  delete 32   : 62(-1){44(1){17(0),50(0){48(0),54(0)}},78(1){.,88(0)}}
  rotations   : 1,  height 3,  still AVL: yes
```

Bacalah untai bentuknya sambil menatap gambar-gambarnya. Hasil langkah 2, `13(-1){5(0){4(-1){3(0),.},10(0){8(0),11(0)}},15(1){.,16(0)}}`, persis adalah pohon di kanan bawah **Gambar 6.15**: akar 13, anak kiri 5 dengan anak 4 (memegang 3) dan 10 (memegang 8 dan 11), anak kanan 15 dengan anak kanan 16. Langkah 4 menunjukkan dua rotasi, sebagaimana dituntut kasus LR; begitu pula langkah 5 untuk RL; langkah 2 dan 3 masing-masing menunjukkan satu; langkah 1 tidak menunjukkan satu pun.

> **Satu-satunya perbedaan antara Kode 6.4 dan Kode 6.9** adalah sebuah medan tinggi di dalam simpulnya, dua fungsi pembantu rotasi, sebuah `rebalance` sepanjang sembilan baris, dan satu panggilan tambahan di ujung `insertRec` dan `removeRec`. Itulah seluruh harga sebuah jaminan kasus terburuk. Kode pencariannya sama sekali tidak disentuh — pohon AVL *adalah* pohon pencarian biner, dan mencarinya memakai empat baris yang sama dari §6.7.

6.19.1 Sebuah implementasi rujukan dalam Java

Gambar 6.19 dan **Gambar 6.20** menampilkan sebuah kelas Java, `AVLTree`, dan ia layak ditengok sekalipun dalam buku C++, karena ia mengambil satu keputusan rancangan yang juga diulangi kode kita.

```java
package AVLTree;

/**
 *
 * AVLTree class extended from Tree class.
 * Added a variable "height" for performing height-balanced efficiently
 */

public class AVLTree extends Tree {

    protected final boolean empty;
    protected final int value;
    protected final AVLTree left, right;
    protected final int height;

    /**
     * Creates a new tree whose root holds value and whose
     * left and right subtrees are left and right
     */
    public AVLTree(int value, AVLTree left, AVLTree right) {
        this.empty = false;
        this.value = value;
        this.left = left;
        this.right = right;
        this.height = 1 + Math.max(left.getHeight(), right.getHeight());
    }
```

Gambar 6.19 Kelas `AVLTree.java`, bagian pertama. Kelasnya memperluas kelas `Tree` biasa dan menambahkan satu medan — `height` — "for performing height-balanced efficiently". Konstruktornya menetapkan `height = 1 + Math.max(left.getHeight(), right.getHeight())`.

Dua hal dalam **Gambar 6.19** layak dikomentari. Pertama, komentar kelasnya sudah mengatakan segalanya: pohon AVL *adalah* sebuah pohon, dengan satu peubah tambahan. Itu hubungan yang sama dengan hubungan antara **Kode 6.9** dan **Kode 6.4** kita. Kedua, baris `this.height = 1 + Math.max(left.getHeight(), right.getHeight())` adalah, huruf demi huruf, fungsi `refresh` dalam **Kode 6.9**. Tinggi selalu satu lebih besar daripada anak yang lebih tinggi.

```
26
27      /**
28       * Creates an empty tree
29       */
30      public AVLTree() {
31          this.empty = true;
32          this.value = 0;
33          this.left = null;
34          this.right = null;
35          this.height = 0;
36      }
37
38      /**
39       * Creates a tree with a single node
40       */
41      public AVLTree(int value) {
42          this.empty = false;
43          this.value = value;
44          this.left = new AVLTree();
45          this.right = new AVLTree();
46          this.height = 1;
47      }
48
49      /**
50       * Gets the height
51       */
52      private int getHeight() {
53          return height;
54      }
55
56  }
```

Gambar 6.20 Kelas `AVLTree.java`, bagian kedua. Pohon kosong adalah sebuah objek dengan `height = 0`; pohon bersimpul tunggal punya `height = 1`.

Gambar 6.20 memakai kesepakatan tinggi *pencacahan simpul* yang sudah ditandai di §6.2: pohon kosong bertinggi 0 dan pohon bersimpul satu bertinggi 1. C++ kita memakai −1 dan 0. Faktor keseimbangannya, karena merupakan selisih, keluar sama persis dengan cara mana pun — dan itulah justru sebabnya kesepakatannya boleh dipilih bebas, asalkan dipilih sekali saja.

Implementasi Java-nya juga merepresentasikan pohon kosong sebagai objek nyata dengan bendera `empty`, alih-alih sebagai penunjuk nol. Cara itu menghindari pemeriksaan nol dengan harga mengalokasikan sebuah sentinel untuk setiap anak yang tidak ada. C++ kita memakai `nullptr` dan memeriksanya di dalam `h()`. Kedua rancangan itu baku; rancangan sentinel lebih lazim dalam bahasa yang penanganan nolnya tidak murah, dan Weiss [1] memakai simpul sentinel sejenis pada pohon merah-hitamnya.

6.20 Penghapusan pada Pohon AVL

Penghapusan pada pohon AVL dimulai persis seperti pada pohon pencarian biner biasa — ketiga kasus §6.11 — lalu menyeimbangkan ulang. Prosedurnya mengandung satu kerutan yang sangat penting.

> **PENGHAPUSAN AVL.** Misalkan s adalah simpul yang sedang dihapus.

1. Lakukan penghapusan pohon pencarian biner baku atas s.
2. Mulai dari posisi lama s, berjalanlah ke atas lalu temukan simpul tak seimbang yang pertama; sebut ia z.
3. Misalkan y adalah anak z **yang tingginya lebih besar**, dan misalkan x adalah anak y yang tingginya lebih besar. *(Catatan: ini aturan yang berbeda dari penyisipan, tempat y dan x ditentukan oleh lintasan yang ditempuh simpul barunya.)*

4. Seimbangkan ulang subpohon yang berakar di z dengan rotasi yang dituntut kasusnya — empat kasus yang sama, empat gambar yang sama: **Gambar 6.10**, **Gambar 6.11**, **Gambar 6.12**, **Gambar 6.13**.
5. **Teruskan.** Tidak seperti penyisipan, menyeimbangkan ulang z belum tentu memperbaiki seluruh pohonnya; teruslah naik sampai ke akar, menyeimbangkan ulang di mana pun perlu.

Langkah 3 dan langkah 5 adalah kedua perbedaannya, dan keduanya berasal dari satu fakta tunggal yang akan kita jelaskan sekarang.

6.20.1 Mengapa y dan x dipilih dengan cara yang berbeda

Pada penyisipan, lintasan simpul baru *adalah* penyebab ketakseimbangannya: subpohon yang bertumbuh adalah subpohon yang dimasuki simpul barunya, sehingga mengikuti lintasannya dengan sendirinya menemukan sisi yang tinggi. Pada penghapusan, ketakseimbangannya disebabkan oleh subpohon yang *menyusut* — dan subpohon yang menyusut adalah sisi yang pendek, bukan yang tinggi. Rotasi harus memindahkan tinggi dari sisi yang tinggi ke sisi yang pendek, jadi kita harus mengenali sisi yang tinggi itu secara langsung. Dari situlah "anak yang tingginya lebih besar".

Ketika kedua anak y sama tinggi, mana pun boleh dipilih sebagai x; pilihlah anak yang berada di sisi yang sama dengan y, supaya kasusnya menjadi rotasi tunggal, sebagaimana dijelaskan peringatan di §6.17.5. Contoh kerja di §6.20.3 menjumpai persis keadaan seri ini.

6.20.2 Mengapa penyisipan paling banyak perlu satu penyeimbangan ulang, sedangkan penghapusan bisa perlu lg$n$ kali

Kembalilah ke tinggi-tinggi yang dihitung di §6.17. Pada keempat kasus penyisipan, subpohon yang berakar di z bertinggi $k + 3$ sesudah penyisipannya dan bertinggi $k + 2$ sesudah rotasinya — dan $k + 2$ adalah tingginya *sebelum* penyisipan. Jadi dari sudut pandang induk z, dan setiap leluhur di atasnya, **tidak ada yang berubah**: subpohonnya persis setinggi dulu. Tidak ada faktor keseimbangan leluhur yang bisa bergeser, jadi tidak ada leluhur yang bisa menjadi tak seimbang. Satu operasi penyeimbangan ulang selalu cukup.

Penghapusan berbeda, dan bedanya sebesar satu aras. Ketika sebuah penghapusan memendekkan subpohon dari tinggi $k + 1$ menjadi k lalu sebuah rotasi dilakukan di z, subpohon yang diseimbangkan ulang itu bisa berakhir **lebih pendek** daripada sebelum penghapusannya. Pemendekan itu sendiri adalah perubahan tinggi, yang terlihat oleh induk z — yang karenanya bisa menjadi tak seimbang pada gilirannya. Perbaikannya karena itu merambat. Pada kasus terburuk ia merambat sampai ke akar, satu rotasi per aras, dan karena ada $O(\lg n)$ aras, penghapusan bisa melakukan $O(\lg n)$ rotasi.

Sekali lagi, dengan terus terang: menyeimbangkan ulang simpul z tidak menyeimbangkan ulang seluruh pohon AVL-nya; sesudah menyeimbangkan ulang z, leluhur z kemungkinan besar juga perlu diseimbangkan ulang, dan itulah sebabnya penelusurannya harus diteruskan sampai ke akar.

> **Kedua operasi itu tetap $O(\lg n)$.** Sebuah rotasi berongkos $O(1)$, dan jumlahnya paling banyak $h = O(\lg n)$, jadi penghapusan $O(\lg n)$ secara keseluruhan — biaya asimtotik yang sama dengan penyisipan. Bedanya ada pada konstantanya, dan pada satu rincian yang penting bagi struktur data konkuren: sebuah penyisipan menyentuh sejumlah simpul yang terbatas sesudah penyusurannya, sedangkan sebuah penghapusan bisa menyentuh seluruh lintasannya.

Kode 6.10 membuat perbedaan itu kasatmata. Ia membangun **pohon Fibonacci** — pohon AVL bersimpul minimum dari §6.15, yang merupakan pohon sah tertipis dan karena itu paling rapuh — lalu mengukur rotasi yang ditimbulkan oleh satu penyisipan dan oleh satu penghapusan.

Kode 6.10 Satu penyisipan tidak pernah berongkos lebih daripada satu penyeimbangan ulang; satu penghapusan bisa berongkos satu penyeimbangan ulang di setiap aras. Kompilasi dengan `g++ -std=c++17 -O2 -o cascade cascade.cpp`.

```
// Mengapa penyisipan AVL paling banyak perlu satu penyeimbangan ulang, sementara
// penghapusan AVL bisa memerlukan satu di setiap aras. Kasus terburuknya adalah pohon
// Fibonacci: pohon AVL tertipis pada tinggi tertentu. Menghapus kunci terbesarnya beruntun
sampai ke akar.
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <vector>
#include <cmath>

struct Node {
    int key, height;
    Node *left, *right;
    Node(int k) : key(k), height(0), left(nullptr), right(nullptr) {}
};

static int  h(Node* x)  { return x ? x->height : -1; }
static int  bf(Node* x) { return x ? h(x->right) - h(x->left) : 0; }
static void refresh(Node* x) { x->height = 1 + std::max(h(x->left), h(x->right)); }

int elementaryRotations = 0;   // rotasi tunggal yang dilakukan
int rebalanceEvents     = 0;   // simpul tempat sebuah kasus rotasi menyala

Node* rotateRight(Node* y) {
    Node* x = y->left; Node* P2 = x->right;
    x->right = y; y->left = P2; refresh(y); refresh(x);
    ++elementaryRotations; return x;
}
Node* rotateLeft(Node* x) {
    Node* y = x->right; Node* P2 = y->left;
    y->left = x; x->right = P2; refresh(x); refresh(y);
    ++elementaryRotations; return y;
}

Node* rebalance(Node* z) {
    refresh(z);
    if (bf(z) < -1) {
        ++rebalanceEvents;
        if (bf(z->left) <= 0) return rotateRight(z);
        z->left = rotateLeft(z->left);
        return rotateRight(z);
    }
    if (bf(z) > 1) {
        ++rebalanceEvents;
        if (bf(z->right) >= 0) return rotateLeft(z);
        z->right = rotateRight(z->right);
        return rotateLeft(z);
```

Kode 6.10 (Lanjutan) *Satu penyisipan tidak pernah berongkos lebih daripada satu penyeimbangan ulang; satu penghapusan bisa berongkos satu penyeimbangan ulang di setiap aras. Kompilasi dengan g++ -std=c++17 -O2 -o cascade cascade.cpp.*

```
    }
    return z;
}

Node* insertRec(Node* x, int key) {
    if (!x) return new Node(key);
    if      (key < x->key) x->left  = insertRec(x->left,  key);
    else if (key > x->key) x->right = insertRec(x->right, key);
    else return x;
    return rebalance(x);
}

Node* minNode(Node* x) { while (x->left) x = x->left; return x; }
Node* maxNode(Node* x) { while (x->right) x = x->right; return x; }

Node* removeRec(Node* x, int key) {
    if (!x) return nullptr;
    if      (key < x->key) x->left  = removeRec(x->left,  key);
    else if (key > x->key) x->right = removeRec(x->right, key);
    else {
        if (!x->left || !x->right) {
            Node* c = x->left ? x->left : x->right;
            delete x;
            if (!c) return nullptr;
            x = c;
        } else {
            Node* s = minNode(x->right);
            x->key = s->key;
            x->right = removeRec(x->right, s->key);
        }
    }
    return rebalance(x);
}

// Pohon Fibonacci bertinggi hh, berkunci 1,2,3,... secara inorder.
// Subpohon kirinya bertinggi hh-1 dan subpohon kanannya bertinggi hh-2:
// itu persis rekurensi N(h) = N(h-1) + N(h-2) + 1.
Node* fibTree(int hh, int& counter) {
    if (hh < 0) return nullptr;
    if (hh == 0) { Node* n = new Node(++counter); refresh(n); return n; }
    Node* L = fibTree(hh - 1, counter);
    Node* r = new Node(++counter);
    Node* R = fibTree(hh - 2, counter);
    r->left = L; r->right = R; refresh(r);
    return r;
}
```

Kode 6.10 (Lanjutan) *Satu penyisipan tidak pernah berongkos lebih daripada satu penyeimbangan ulang; satu penghapusan bisa berongkos satu penyeimbangan ulang di setiap aras. Kompilasi dengan g++ -std=c++17 -O2 -o cascade cascade.cpp.*

```
int size(Node* x) { return x ? 1 + size(x->left) + size(x->right) : 0; }
void destroy(Node* x) { if (!x) return; destroy(x->left); destroy(x->right); delete x; }

int main() {
    std::cout << "Insertion: rotations needed to add one key to a Fibonacci tree\n";
    std::cout << "  height   nodes   rebalance events   single rotations\n";
    for (int hh = 4; hh <= 16; hh += 3) {
        int c = 0;
        Node* t = fibTree(hh, c);
        int n = size(t);
        rebalanceEvents = elementaryRotations = 0;
        t = insertRec(t, 0);                      // sebuah kunci terkecil yang baru
        std::cout << std::setw(8) << hh << std::setw(8) << n
                  << std::setw(19) << rebalanceEvents
                  << std::setw(19) << elementaryRotations << "\n";
        destroy(t);
    }

    std::cout << "\nDeletion: rotations needed to remove the largest key of a Fibonacci
tree\n";
    std::cout << "  height   nodes   rebalance events   single rotations   lg(n)\n";
    for (int hh = 4; hh <= 16; hh += 3) {
        int c = 0;
        Node* t = fibTree(hh, c);
        int n = size(t);
        int largest = maxNode(t)->key;
        rebalanceEvents = elementaryRotations = 0;
        t = removeRec(t, largest);
        std::cout << std::setw(8) << hh << std::setw(8) << n
                  << std::setw(19) << rebalanceEvents
                  << std::setw(19) << elementaryRotations
                  << std::setw(9) << std::fixed << std::setprecision(1)
                  << std::log2((double)n) << "\n";
        destroy(t);
    }
    return 0;
}
```

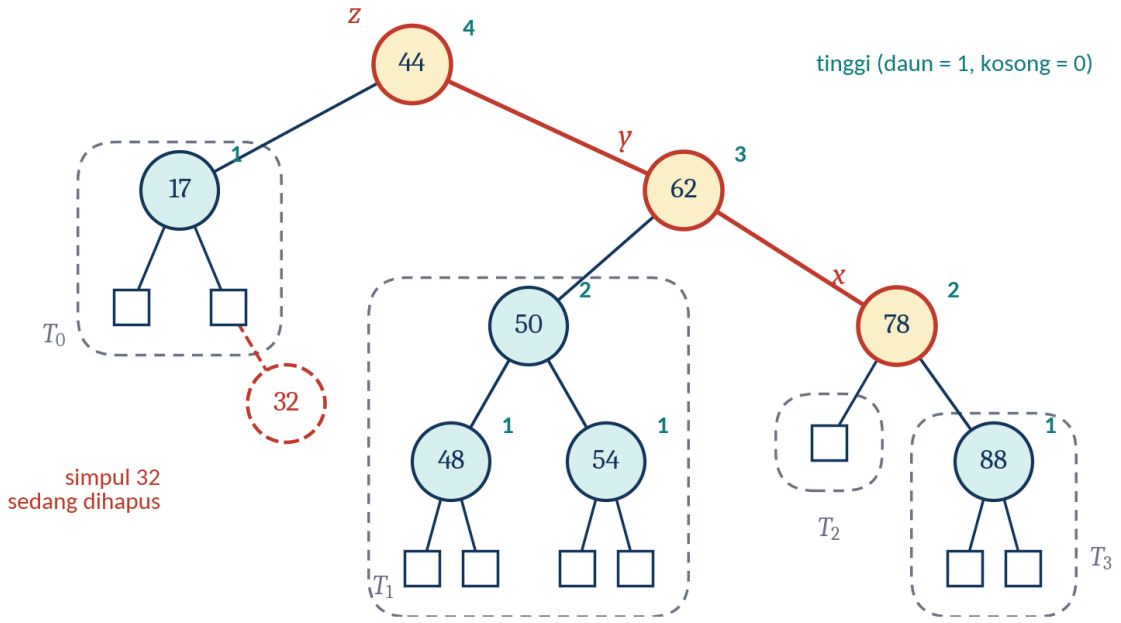
Programnya mencetak:

```
Insertion: rotations needed to add one key to a Fibonacci tree
  height   nodes   rebalance events   single rotations
       4      12                  1                  1
       7      54                  1                  1
      10     232                  1                  1
      13     986                  1                  1
      16    4180                  1                  1

Deletion: rotations needed to remove the largest key of a Fibonacci tree
  height   nodes   rebalance events   single rotations   lg(n)
       4      12                  2                  2      3.6
       7      54                  3                  3      5.8
      10     232                  5                  5      7.9
      13     986                  6                  6      9.9
      16    4180                  8                  8     12.0
```

Lajur penyisipannya adalah konstanta 1 betapapun tinggi pohonnya bertumbuh — persis yang diramalkan argumen tinggi §6.20.2. Lajur penghapusannya bertumbuh bersama tingginya: 2,3,5,6,8 rotasi untuk tinggi 4,7,10,13,16, yang kira-kira $h/2$, dan karena itu $\Theta(\lg n)$.

Perhatikan juga bahwa cacah simpul pada lajur pertama tiap tabelnya — 12,54,232,986,4180 — adalah nilai $N(4), N(7), N(10), N(13), N(16)$ dari rekurensi §6.15. Baris $h = 4$, 7, dan 10 bisa dicocokkan langsung dengan **Tabel 6.4**: 12, 54, dan 232. Programnya membangun pohonnya dari rekurensinya dan rekurensinya menghasilkan ukuran yang diramalkan tabelnya.

6.20.3 Sebuah contoh penghapusan, selangkah demi selangkah



Gambar 6.21 Menghapus dari pohon AVL: simpul 32 hendak dihapus. Tinggi dituliskan di samping simpulnya, dan leluhur tak seimbang yang pertama z, anaknya yang lebih tinggi y, dan anak y yang lebih tinggi x ditandai dengan warna merah. Kotak-kotak kosongnya adalah subpohon nol, dan T_0 sampai T_3 menamai keempat subpohon yang akan dipindahkan rotasinya.

Pohon **Gambar 6.21** adalah: akar 44; anak kiri 17, yang punya anak kanan 32; anak kanan 62, yang anak kirinya adalah 50 (dengan anak 48 dan 54) dan anak kanannya adalah 78 (dengan anak kanan 88).

Perhatikan kesepakatan tinggi pada gambarnya: simpul daun ditandai 1 dan subpohon nol ditandai 0. Itu kesepakatan pencacahan simpul dari §6.2, satu lebih besar daripada milik kita di mana-mana. Kita akan mengutip angka gambarnya sebagaimana gambarnya memberikannya dan angka program kita dalam kesepakatan kita sendiri; faktor keseimbangannya sepakat.

Langkah 1: hapus 32 **dengan aturan pohon pencarian biner biasa.** Simpul 32 adalah simpul daun, jadi ini kasus 1 dari §6.11: lepaskan ia dari induknya 17 lalu bebaskan. Simpul 17 menjadi simpul daun, dan dalam kesepakatan gambarnya tingginya turun dari 2 menjadi 1.

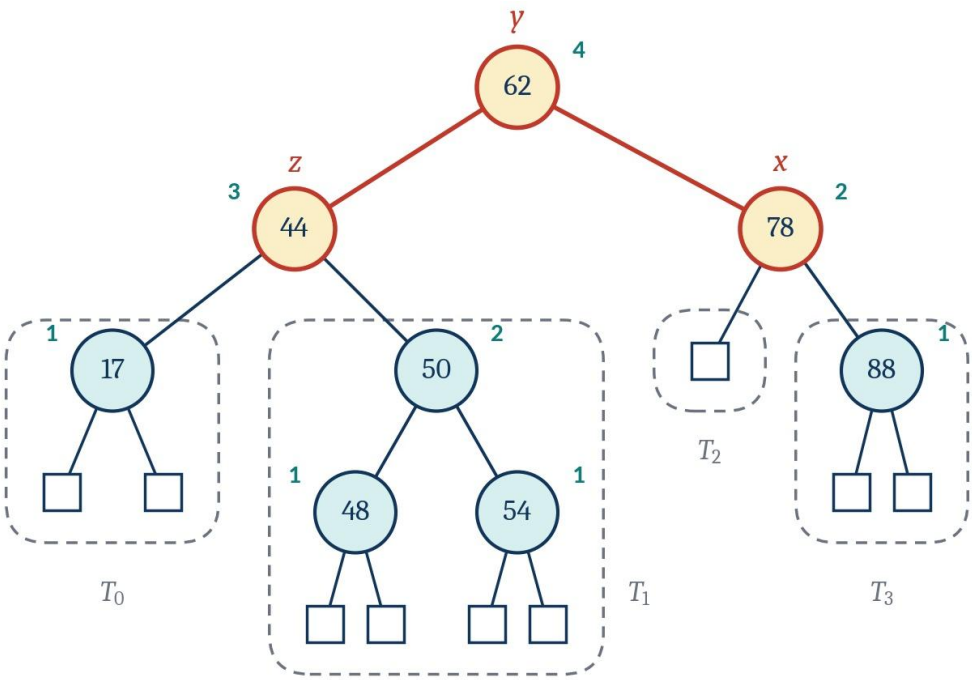
Langkah 2: berjalan ke atas dan temukan simpul tak seimbang yang pertama. Di simpul 17: kedua subpohonnya kosong, seimbang. Di simpul 44: subpohon kirinya kini hanya 17, bertinggi 1 dalam penomoran gambarnya, sementara subpohon kanannya yang berakar di 62 bertinggi 3. Selisihnya 2, jadi 44 tak seimbang: sesudah menghapus 32 kita naik dan mendapati bahwa simpul tak seimbang yang pertama adalah 44. Jadi $z = 44$.

Langkah 3: pilih y dan x menurut tinggi. Anak-anak $z = 44$ adalah 17 (tinggi 1) dan 62 (tinggi 3). Yang lebih tinggi adalah 62, jadi $y = 62$. Anak-anak $y = 62$ adalah 50 (tinggi 2) dan 78 (tinggi 2) — *seri*. Keduanya dapat menjadi x, sebab tingginya sama; kita memilih 78, anak yang berada di sisi yang sama dengan y, sehingga kasusnya menjadi rotasi tunggal. Jadi $x = 78$.

Langkah 4: kenali kasusnya. $y = 62$ adalah anak **kanan** $z = 44$, dan $x = 78$ adalah anak **kanan** y. Kanan-Kanan, jadi satu rotasi kiri yang akan dilakukan.

Langkah 5: lakukan satu rotasi kiri terhadap $z = 44$, persis seperti yang ditunjukkan **Gambar 6.12**.

1. $y \leftarrow$ 44->right = 62.
2. $P_2 \leftarrow$ 62->left = 50 — inilah subpohon yang diberi label T_1 pada gambarnya.
3. 62->left $\leftarrow$ 44.
4. 44->right $\leftarrow$ 50.
5. Segarkan 44, lalu 62.



Gambar 6.22 Pohonnya sesudah rotasi kiri. Simpul 62 adalah akar barunya; 44 turun menjadi anak kirinya dan mengambil subpohon T_1 (yang berakar di 50) sebagai anak kanannya; 78 tidak berubah sebagai anak kanan 62.

Gambar 6.22 adalah hasilnya. Simpul 62 kini menjadi akar dengan tinggi 4 dalam kesepakatan gambarnya; 44 adalah anak kirinya dengan tinggi 3, memegang 17 (T_0) di kiri dan 50 (T_1) di kanan; 78 adalah anak kanannya dengan tinggi 2, memegang subpohon nol T_2 di kiri dan 88 (T_3) di kanan.

Periksa invariannya di setiap simpul, dalam kesepakatan gambarnya:

- 17: dua anak nol, tinggi 0 dan 0. Seimbang.
- 48, 54, 88: simpul daun. Seimbang.
- 50: anak 48 dan 54, tinggi 1 dan 1. Seimbang.
- 78: anak T_2 (tinggi 0) dan 88 (tinggi 1). Selisih 1. Seimbang.
- 44: anak 17 (tinggi 1) dan 50 (tinggi 2). Selisih 1. Seimbang.
- 62: anak 44 (tinggi 3) dan 78 (tinggi 2). Selisih 1. Seimbang.

Setiap simpulnya sah, jadi satu rotasi sudah cukup di sini. Blok keluaran terakhir dari **Kode 6.9** mengukuhkan keseluruhannya:

```
Deletion example
  before      : 44(1){17(1){.,32(0)},62(0){50(0){48(0),54(0)},78(1){.,88(0)}}}
  delete 32   : 62(-1){44(1){17(0),50(0){48(0),54(0)}},78(1){.,88(0)}}
  rotations   : 1,  height 3,  still AVL: yes
```

Bacalah untai bentuk yang kedua sambil menatap **Gambar 6.22**: akar 62 dengan faktor keseimbangan -1; anak kiri 44 dengan faktor keseimbangan $+1$, memegang simpul daun 17 dan subpohon 50{48,54}; anak kanan 78 dengan faktor keseimbangan $+1$, tidak memegang apa-apa di kiri dan 88 di kanan. Itulah gambarnya, ditulis dalam satu baris.

6.21 Memilih: Pohon AVL, Pohon Merah-Hitam, atau Tabel Hash

Pohon AVL tidak selalu merupakan jawaban yang tepat. Dua struktur lain bersaing untuk pekerjaan yang sama, dan seorang profesional memilih di antara keduanya dengan sengaja.

6.21.1 Pohon merah-hitam

Sebuah **pohon merah-hitam** (*red-black tree*) adalah pohon pencarian biner penyeimbang-diri yang lain. Alih-alih menyimpan tinggi pada tiap simpul, ia menyimpan satu bit — sebuah warna, merah atau hitam — dan memelihara lima aturan [2], yang terpenting di antaranya: akarnya hitam; simpul merah tidak pernah punya anak merah; dan setiap lintasan dari sebuah simpul turun ke anak yang tidak ada melewati cacah simpul hitam yang sama.

Aturan terakhir itulah yang mengerjakan penyeimbangannya. Kalau setiap lintasan akar-ke-daun punya cacah simpul hitam yang sama, dan tidak ada dua merah yang bersebelahan, maka lintasan terpanjangnya paling banyak dua kali lintasan terpendeknya: merah dan hitam berselang-seling pada lintasan yang panjang, semuanya hitam pada yang pendek. Karena itu

$$h \leq 2\lg(n+1) \tag{6.11}$$

untuk pohon merah-hitam (Persamaan (6.11)), berhadapan dengan $h \leq 1.4405\lg(n+2) - 1.3277$ untuk pohon AVL (Persamaan (6.9)). **Pohon AVL tegas lebih pendek di antara keduanya.**

Kalau begitu, mengapa `std::map` dalam pustaka baku C++ adalah pohon merah-hitam dan bukan pohon AVL? Karena batas pada *tinggi* bukan satu-satunya batas yang penting. Pohon merah-hitam lebih longgar, dan longgar berarti ia perlu lebih jarang memperbaiki dirinya:

- sebuah **penyisipan** merah-hitam melakukan paling banyak 2 rotasi, lalu hanya pewarnaan ulang;
- sebuah **penghapusan** merah-hitam melakukan paling banyak 3 rotasi, lalu hanya pewarnaan ulang;
- sebuah **penyisipan** AVL melakukan paling banyak 2 rotasi dasar;
- sebuah **penghapusan** AVL bisa melakukan $\Theta(\lg n)$ rotasi, sebagaimana diperagakan **Kode 6.10**.

Jadi: pohon AVL memberikan pencarian yang lebih cepat, karena pohonnya lebih pendek; pohon merah-hitam memberikan pengubahan yang lebih cepat, karena perbaikannya terbatas. Aturan praktis yang dipakai setiap praktisi karena itu berbunyi:

> **Beban kerja yang berat membaca — banyak pencarian, sedikit perubahan — berpihak pada pohon AVL. Beban kerja yang berat menulis — penyisipan dan penghapusan terus-menerus — berpihak pada pohon merah-hitam.** Sebuah indeks basis data yang dibangun ulang tiap malam lalu dikuerikan sepanjang hari adalah persoalan berbentuk AVL;

> sebuah peta dalam memori yang terus-menerus diaduk adalah persoalan berbentuk merah-hitam.

6.21.2 Tabel hash

Sebuah **tabel hash** (*hash table*) menjawab pertanyaan yang sama sekali berbeda. Ia menghitung sebuah fungsi atas kuncinya, memakai hasilnya sebagai indeks larik, lalu melompat langsung ke sana. Biaya harapannya untuk sisip, cari, dan hapus adalah $O(1)$ — bukan $O(\lg n)$, melainkan $O(1)$, tidak bergantung pada n.

Itu keunggulan yang luar biasa dan Anda sebaiknya mengambilnya kapan pun Anda bisa. Tetapi ia dibeli dengan membuang urutan. Sebuah fungsi hash dengan sengaja menyerakkan kunci, jadi dalam tabel hash:

- tidak ada "kunci berikutnya sesudah k" yang bermakna — operasi penerusnya lenyap;
- tidak ada minimum dan tidak ada maksimum tanpa memindai setiap embernya;
- Anda tidak bisa mendaftar kuncinya dalam urutan terurut tanpa mengeluarkannya lalu mengurutkannya, dengan biaya $\Theta(n \lg n)$;
- Anda tidak bisa menjawab **kueri rentang** — "setiap kunci di antara a dan b" — tanpa memeriksa seluruh tabelnya;
- dan $O(1)$-nya adalah sebuah *harapan*, bukan sebuah jaminan: fungsi hash yang buruk atau masukan yang bersifat lawan bisa memerosotkan tabel hash menjadi $\Theta(n)$ per operasi, dan pengulangan hash menyebabkan penyisipan yang sesekali mahal.

Sebuah pohon seimbang mempertahankan semua kesanggupan itu, dengan $O(\lg n)$ per operasi, terjamin.

Kode 6.11 mengukur pertukarannya pada 200 000 kunci: sebuah pohon AVL, sebuah BST biasa, `std::map` (pohon merah-hitam), dan `std::unordered_map` (tabel hash).

Kode 6.11 Mengukur pohon AVL terhadap BST biasa, pohon merah-hitam, dan tabel hash. Kompilasi dengan `g++ -std=c++17 -O2 -o compare compare.cpp`.

```
// Perbandingan terukur: BST biasa, pohon AVL, std::map (pohon merah-hitam),
// dan std::unordered_map (tabel hash). Uji terakhirnya adalah kueri rentang,
// yang tak dapat dijawab tabel hash tanpa menengok setiap kunci.
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <chrono>
#include <map>
#include <unordered_map>
#include <random>
#include <vector>

struct Node {
    int key, height;
    Node *left, *right;
    Node(int k) : key(k), height(0), left(nullptr), right(nullptr) {}
};
static int  h(Node* x)  { return x ? x->height : -1; }
static int  bf(Node* x) { return x ? h(x->right) - h(x->left) : 0; }
static void refresh(Node* x) { x->height = 1 + std::max(h(x->left), h(x->right)); }
Node* rotR(Node* y) { Node* x=y->left;  Node* t=x->right; x->right=y; y->left =t;
refresh(y); refresh(x); return x; }
Node* rotL(Node* x) { Node* y=x->right; Node* t=y->left;  y->left =x; x->right=t;
refresh(x); refresh(y); return y; }
Node* balance(Node* z) {
    refresh(z);
    if (bf(z) < -1) { if (bf(z->left)  > 0) z->left  = rotL(z->left);  return rotR(z); }
    if (bf(z) >  1) { if (bf(z->right) < 0) z->right = rotR(z->right); return rotL(z); }
    return z;
}
Node* avlInsert(Node* x, int k) {
    if (!x) return new Node(k);
    if      (k < x->key) x->left  = avlInsert(x->left,  k);
    else if (k > x->key) x->right = avlInsert(x->right, k);
    else return x;
    return balance(x);
}
Node* bstInsert(Node* x, int k) {
    if (!x) return new Node(k);
    if      (k < x->key) x->left  = bstInsert(x->left,  k);
    else if (k > x->key) x->right = bstInsert(x->right, k);
    return x;
}
bool findIter(Node* x, int k) { while (x && x->key != k) x = (k < x->key) ? x->left : x-
>right; return x != nullptr; }
int  heightOf(Node* x) { return x ? 1 + std::max(heightOf(x->left), heightOf(x->right)) : -
1; }
void destroy(Node* x) { if (!x) return; destroy(x->left); destroy(x->right); delete x; }
// Jumlahkan kunci di [lo, hi] tanpa mengunjungi subpohon yang tak mungkin memuatnya.
long long rangeSum(Node* x, int lo, int hi) {
```

Kode 6.11 (Lanjutan) *Mengukur pohon AVL terhadap BST biasa, pohon merah-hitam, dan tabel hash. Kompilasi dengan g++ -std=c++17 -O2 -o compare compare.cpp.*

```
    if (!x) return 0;
    if (x->key < lo)  return rangeSum(x->right, lo, hi);
    if (x->key > hi)  return rangeSum(x->left,  lo, hi);
    return (long long)x->key + rangeSum(x->left, lo, hi) + rangeSum(x->right, lo, hi);
}

using Clock = std::chrono::steady_clock;
static double ms(Clock::time_point a, Clock::time_point b) {
    return std::chrono::duration<double, std::milli>(b - a).count();
}

int main() {
    const int n = 200000;
    std::vector<int> ascending(n);
    for (int i = 0; i < n; ++i) ascending[i] = i;
    std::vector<int> shuffled(ascending);          // sebuah salinan, lalu diacak di bawah
    std::mt19937 rng(2026);
    std::shuffle(shuffled.begin(), shuffled.end(), rng);

    std::cout << std::fixed << std::setprecision(2);

    // 1. Kunci menaik: BST biasa merosot, pohon AVL tidak.
    Node* a = nullptr;
    auto t0 = Clock::now();
    for (int k : ascending) a = avlInsert(a, k);
    auto t1 = Clock::now();
    std::cout << "AVL, ascending keys   : height " << std::setw(6) << heightOf(a)
              << "   insert " << std::setw(8) << ms(t0, t1) << " ms\n";

    Node* b = nullptr;
    t0 = Clock::now();
    for (int k : shuffled) b = bstInsert(b, k);
    t1 = Clock::now();
    std::cout << "BST, shuffled keys    : height " << std::setw(6) << heightOf(b)
              << "   insert " << std::setw(8) << ms(t0, t1) << " ms\n";

    std::map<int,int> rb;
    t0 = Clock::now();
    for (int k : ascending) rb[k] = k;
    t1 = Clock::now();
    std::cout << "std::map, ascending   : size   " << std::setw(6) << rb.size()
              << "   insert " << std::setw(8) << ms(t0, t1) << " ms\n";
```

Kode 6.11 (Lanjutan) *Mengukur pohon AVL terhadap BST biasa, pohon merah-hitam, dan tabel hash. Kompilasi dengan g++ -std=c++17 -O2 -o compare compare.cpp.*

```
    std::unordered_map<int,int> hm;
    t0 = Clock::now();
    for (int k : ascending) hm[k] = k;
    t1 = Clock::now();
    std::cout << "unordered_map, asc.   : size   " << std::setw(6) << hm.size()
              << "   insert " << std::setw(8) << ms(t0, t1) << " ms\n\n";

    // 2. Pencarian kunci-kunci yang semuanya ada.
    long long found = 0;
    t0 = Clock::now(); for (int k : shuffled) found += findIter(a, k);  t1 = Clock::now();
    std::cout << "AVL lookup            : " << std::setw(8) << ms(t0, t1) << " ms  (" <<
found << " hits)\n";
    found = 0;
    t0 = Clock::now(); for (int k : shuffled) found += (rb.find(k) != rb.end()); t1 =
Clock::now();
    std::cout << "std::map lookup       : " << std::setw(8) << ms(t0, t1) << " ms  (" <<
found << " hits)\n";
    found = 0;
    t0 = Clock::now(); for (int k : shuffled) found += (hm.find(k) != hm.end()); t1 =
Clock::now();
    std::cout << "unordered_map lookup  : " << std::setw(8) << ms(t0, t1) << " ms  (" <<
found << " hits)\n\n";

    // 3. Kueri rentang: jumlahkan setiap kunci antara 90000 dan 90999.
    long long s1 = 0, s2 = 0, s3 = 0;
    t0 = Clock::now(); s1 = rangeSum(a, 90000, 90999); t1 = Clock::now();
    std::cout << "AVL range sum         : " << std::setw(8) << ms(t0, t1) << " ms  (sum "
<< s1 << ")\n";
    t0 = Clock::now();
    for (auto it = rb.lower_bound(90000); it != rb.end() && it->first <= 90999; ++it) s2 +=
it->first;
    t1 = Clock::now();
    std::cout << "std::map range sum    : " << std::setw(8) << ms(t0, t1) << " ms  (sum "
<< s2 << ")\n";
    t0 = Clock::now();
    for (const auto& kv : hm) if (kv.first >= 90000 && kv.first <= 90999) s3 += kv.first;
    t1 = Clock::now();
    std::cout << "hash table range sum  : " << std::setw(8) << ms(t0, t1) << " ms  (sum "
<< s3 << ")\n";

    destroy(a); destroy(b);
    return 0;
}
```

Pada mesin yang dipakai menyiapkan bab ini, programnya mencetak:

```
AVL, ascending keys   : height     17   insert    38.18 ms
BST, shuffled keys    : height     41   insert   145.14 ms
std::map, ascending   : size   200000   insert    40.73 ms
unordered_map, asc.   : size   200000   insert    10.57 ms

AVL lookup            :    36.59 ms  (200000 hits)
std::map lookup       :   107.89 ms  (200000 hits)
unordered_map lookup  :     2.65 ms  (200000 hits)

AVL range sum         :     0.01 ms  (sum 90499500)
std::map range sum    :     0.01 ms  (sum 90499500)
hash table range sum  :     0.51 ms  (sum 90499500)
```

Waktunya berbeda-beda antarmesin, tetapi bentuk hasilnya tidak. Bacalah dalam tiga bagian.

Tinggi. Disuapi 200 000 kunci dalam urutan *naik* — kasus terburuk bagi BST biasa — pohon AVL-nya mencapai tinggi 17. Batas kita Persamaan (6.9) meramalkan paling banyak

$1.4405\lg(200002) - 1.3277 \approx 23.1$, jadi 17 berada nyaman di dalamnya, dan $\lg 200000 \approx 17.6$ mengatakan bahwa 17 pada hakikatnya optimal. BST biasanya, yang disuapi kunci *teracak* — kasus realistis terbaiknya — tetap mencapai tinggi 41, lebih daripada dua kali lebih dalam. BST biasa yang disuapi kunci menaik akan mencapai tinggi 199 999; jalannya dihilangkan semata-mata karena memakan waktu menit, bukan milidetik.

Pencarian. Tabel hashnya menang satu orde besaran: 2.65 ms berhadapan dengan 36.59 ms. Ketika yang Anda lakukan hanyalah mencari kunci tunggal, tabel hash adalah struktur yang tepat dan tidak ada pohon yang akan mengalahkannya.

Kueri rentang. Pohon AVL dan `std::map` menjawab "jumlahkan setiap kunci dari 90 000 sampai 90 999" dalam 0.01 ms, karena keduanya bisa menyusur turun ke awal rentangnya lalu menyusuri hanya 1000 kunci di dalamnya. Tabel hashnya memerlukan 0.51 ms — kira-kira 51 kali lebih lama — karena ia harus menengok seluruh 200 000 entrinya untuk mengetahui yang mana 1000 yang memenuhi syarat. Lebarkan tabelnya menjadi dua puluh juta kunci dan jurang itu menjadi faktor ribuan.

Tabel 6.6 adalah tabel keputusan yang perlu disimpan.

Tabel 6.6 Memilih kamus terurut. k adalah cacah kunci yang sungguh-sungguh dikembalikan kueri rentangnya.

Kebutuhan	BST biasa	Pohon AVL	Pohon merah-hitam	Tabel hash
Pencarian, kasus terburuk	$O(n)$	$O(\lg n)$	$O(\lg n)$	$O(n)$
Pencarian, lazimnya	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	$O(1)$
Sisip / hapus, kasus terburuk	$O(n)$	$O(\lg n)$	$O(\lg n)$	$O(n)$, teramortisasi $O(1)$
Rotasi per penyisipan	tidak ada	≤ 2	≤ 2	—
Rotasi per penghapusan	tidak ada	$O(\lg n)$	≤ 3	—
Tinggi yang terjamin	tidak ada	$\approx 1.44\lg n$	$\leq 2\lg n$	—
Kunci dalam urutan terurut	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n\lg n)$
Minimum / maksimum	$O(h)$	$O(\lg n)$	$O(\lg n)$	$\Theta(n)$
Penerus / pendahulu	$O(h)$	$O(\lg n)$	$O(\lg n)$	tidak didukung
Kueri rentang $[a, b]$	$O(h + k)$	$O(\lg n + k)$	$O(\lg n + k)$	$\Theta(n)$
Ruang tambahan per simpul	2 penunjuk	2 penunjuk + sebuah tinggi	2 penunjuk + 1 bit	larik ember + rantai
Pilihlah ketika	tidak pernah, di produksi	membaca mendominasi menulis	menulis sering terjadi	hanya pencarian titik yang diperlukan

6.22 Profitina dalam Praktik

Profitina adalah **platform keterlihatan AI (GEO)**: ia memindai situs web sebuah merek, menilai seberapa siap situs itu untuk *dikutip oleh mesin jawaban AI*, menjelaskan setiap sinyal yang diukurnya, lalu membantu memperbaiki apa yang ditemukannya. Ia berjalan di produksi pada `app.profitina.com`, di atas Python dan PostgreSQL, dengan inferensi model setempat melalui Ollama dan Qwen2.5. Semua isi bab ini muncul di dalamnya, pada empat tempat yang berbeda.

Batas jelajahnya adalah struktur terurut, bukan antrean. Sebuah pemindaian atas situs pelanggan menemukan URL lebih cepat daripada ia sanggup mengambilnya. URL yang tertunda itu tidak diproses menurut siapa datang dulu: mereka diproses menurut urutan prioritas yang memadukan kedalaman, taksiran kepentingan, dan jeda kesopanan. Struktur yang harus berulang kali menyerahkan "butir yang paling mendesak" sekaligus menerima butir baru kapan saja adalah persis persoalan kamus terurut dalam bab ini. Di tempat penjadwalnya tambahan lagi memerlukan *rentang* — "setiap URL yang pewaktu kesopanannya kedaluwarsa dalam tiga puluh detik berikutnya" — pohon pencarian seimbang membuktikan harganya, karena tumpukan tidak bisa menjawab itu dan tabel hash pun tidak.

Irama pemindaian ulangnya adalah kueri penerus. Setiap merek pada jenjang berbayar dipindai ulang secara otomatis pada irama per jenjang, dan penjadwalnya harus berulang kali bertanya: *pemindaian mana yang jatuh tempo berikutnya, dan kapan?* Itu adalah TREE-MINIMUM pada pohon yang berkunci waktu jatuh tempo. Ketika sebuah pemindaian selesai lalu dijadwalkan ulang, itu adalah satu penghapusan diikuti satu penyisipan. Ketika seorang operator bertanya "apa yang jatuh tempo antara 02.00 dan 04.00 malam ini?", itu adalah kueri rentang — $O(\lg n + k)$ pada pohon seimbang dan $\Theta(n)$ pada tabel hash. Inilah persis bentuk **Kode 6.12** di bawah.

Tren skor mingguannya adalah penelusuran inorder. Papan analitik historisnya menampilkan skor keterlihatan sebuah merek sepanjang waktu lalu mengekspornya sebagai CSV. Ekspornya adalah penelusuran inorder atas indeks berkunci waktu: $\Theta(n)$, dalam urutan terurut, tanpa langkah pengurutan. Bukti di §6.6.1 adalah alasan mengapa CSV-nya keluar dalam urutan tanggal tanpa ada seorang pun memanggil rutin pengurutan.

Kanonikalisasi URL adalah persoalan tabel hash, dengan sengaja. Menghilangkan kembaran `https://example.co.id/page`, `http://example.co.id/page/`, dan `https://www.example.co.id/page?utm_source=x` menjadi satu halaman kanonik hanya memerlukan *kesamaan* — tidak pernah "URL berikutnya sesudah yang ini". Itulah baris **Tabel 6.6** tempat tabel hash menang mutlak, dan sistem produksinya memang memakai tabel hash. Mengetahui kapan *tidak* boleh meraih pohon sama pentingnya dalam bab ini dengan mengetahui kapan boleh.

> **Dua aturan yang tidak kami tawar.** Pertama, platformnya menasihati; pelangganlah yang memutuskan. Skor keterlihatan adalah sebuah pengukuran yang disertai penjelasan, tidak pernah sebuah janji tentang bagaimana mesin jawaban akan berperilaku. Kedua, apa pun yang menyentuh data pribadi tunduk pada UU PDP 27/2022 — persetujuan tersurat, ekspor data, dan penghapusan akun yang tuntas. Penghapusan yang tuntas berarti setiap indeks yang menyimpan rujukan ke akun itu harus mendukung penghapusan, dan itulah satu alasan praktis mengapa struktur data di dalam sistem ini adalah struktur yang bisa menghapus dengan murah, bukan struktur yang hanya bisa ditambahi.

6.22.1 Studi kasus yang dikerjakan tuntas: Profitina Laundry

Studi kasus yang dapat dijalankan dalam buku ini selalu **Profitina Laundry**, sebuah usaha cuci kecil di sebuah kampung di Surabaya — bukan platform Profitina itu sendiri.

Usaha cuci itu menerima kantong cucian di meja depan sepanjang hari. Setiap kantong memperoleh sebuah nota, nama pelanggan, berat dalam kilogram, dan sebuah **waktu pengambilan yang dijanjikan**. Petugas di balik meja memerlukan empat hal, terus-menerus:

1. mengarsipkan kantong yang baru diterima di bawah menit yang dijanjikan untuknya;

2. menjawab "apa yang jatuh tempo berikutnya?" ketika giliran kerjanya sedang sepi dan mereka ingin mencuri langkah;
3. menjawab "apa saja yang dijanjikan antara pukul 10.00 dan 14.30?" ketika merencanakan muatan mesin untuk satu blok waktu siang;
4. membuang sebuah kantong dari papannya ketika pelanggannya mengambil.

Pertanyaan 2 adalah kueri **penerus**. Pertanyaan 3 adalah kueri **rentang**. Pertanyaan 1 dan 4 adalah penyisipan dan penghapusan. Tabel hash tidak menjawab satu pun dari ketiga yang pertama. Larik terurut menjawab 2 dan 3 tetapi $\Theta(n)$ untuk 1 dan 4. Pohon AVL yang berkunci menit janji menjawab keempatnya dalam $O(\lg n)$, terjamin, sekalipun notanya datang sepanjang hari dalam urutan waktu yang menaik — pola yang justru menghancurkan pohon pencarian biner biasa.

Kode 6.12 adalah papan pengambilan itu. Perhatikan dua rinciannya. Kuncinya adalah pasangan (menit, nota), sehingga dua kantong yang dijanjikan untuk menit yang sama tetap merupakan kunci yang berbeda dan pohonnya tidak pernah harus berurusan dengan kunci kembar. Dan `nextDueAfter` melakukan pencarian penerus sama sekali tanpa penunjuk induk, dengan membawa calon terbaik turun dari akarnya — teknik kedua yang disebut di §6.9.

Kode 6.12 Papan pengambilan Profitina Laundry: pohon AVL yang berkunci waktu pengambilan yang dijanjikan, mendukung kueri penerus, kueri rentang, dan penghapusan. Kompilasi dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp`.

```
// Profitina Laundry: papan pengambilan.
// Setiap pesanan diarsipkan di bawah menit yang dijanjikan untuknya. Petugas meja
// perlu empat hal: mengarsipkan pesanan baru, mencari pesanan jatuh tempo berikutnya
setelah "sekarang",
// mendaftar semua yang jatuh tempo dalam satu jendela waktu, dan menghapus pesanan begitu
diambil.
// Pohon AVL berkunci menit janji menjawab keempatnya dalam O(lg n) atau lebih baik.
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <cstdio>
#include <string>
#include <vector>

struct Order {
    int         minute;      // menit setelah tengah malam saat cucian dijanjikan
    std::string ticket;      // nomor nota yang tertulis di kantong
    std::string customer;
    int         kilos;
};

struct Node {
    Order data;
    int   height;
    Node* left;
    Node* right;
    Node(const Order& o) : data(o), height(0), left(nullptr), right(nullptr) {}
};

static int  h(Node* x)  { return x ? x->height : -1; }
static int  bf(Node* x) { return x ? h(x->right) - h(x->left) : 0; }
static void refresh(Node* x) { x->height = 1 + std::max(h(x->left), h(x->right)); }

Node* rotateRight(Node* y) { Node* x=y->left;  Node* t=x->right; x->right=y; y->left =t;
refresh(y); refresh(x); return x; }
Node* rotateLeft (Node* x) { Node* y=x->right; Node* t=y->left;  y->left =x; x->right=t;
refresh(x); refresh(y); return y; }

Node* rebalance(Node* z) {
    refresh(z);
    if (bf(z) < -1) { if (bf(z->left)  > 0) z->left  = rotateLeft(z->left);   return
rotateRight(z); }
    if (bf(z) >  1) { if (bf(z->right) < 0) z->right = rotateRight(z->right); return
rotateLeft(z);  }
    return z;
}
```

Kode 6.12 (Lanjutan) *Papan pengambilan Profitina Laundry: pohon AVL yang berkunci waktu pengambilan yang dijanjikan, mendukung kueri penerus, kueri rentang, dan penghapusan. Kompilasi dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
// Dua pesanan yang dijanjikan pada menit yang sama dipisahkan oleh nomor
// notanya, sehingga setiap kunci di pohon berbeda satu sama lain.
static int compareKey(int m1, const std::string& t1, int m2, const std::string& t2) {
    if (m1 != m2) return m1 < m2 ? -1 : 1;
    if (t1 == t2) return 0;
    return t1 < t2 ? -1 : 1;
}

Node* fileOrder(Node* x, const Order& o) {
    if (!x) return new Node(o);
    int c = compareKey(o.minute, o.ticket, x->data.minute, x->data.ticket);
    if (c < 0) x->left  = fileOrder(x->left,  o);
    else       x->right = fileOrder(x->right, o);
    return rebalance(x);
}

// Pesanan pertama yang dijanjikan benar-benar setelah "sekarang": pencarian penerus dari
akar.
const Order* nextDueAfter(Node* x, int now) {
    const Order* best = nullptr;
    while (x) {
        if (x->data.minute > now) { best = &x->data; x = x->left; }  // calon; coba yang
lebih awal
        else                      { x = x->right; }                  // terlalu awal;
menuju yang lebih lambat
    }
    return best;
}

void listWindow(Node* x, int lo, int hi, std::vector<const Order*>& out) {
    if (!x) return;
    if (x->data.minute >= lo) listWindow(x->left, lo, hi, out);   // lewati subpohon kiri
jika seluruhnya terlalu awal
    if (x->data.minute >= lo && x->data.minute <= hi) out.push_back(&x->data);
    if (x->data.minute <= hi) listWindow(x->right, lo, hi, out);  // lewati subpohon kanan
jika seluruhnya terlalu lambat
}

Node* minNode(Node* x) { while (x->left) x = x->left; return x; }

Node* collectOrder(Node* x, int minute, const std::string& ticket, bool& done) {
    if (!x) return nullptr;
    int c = compareKey(minute, ticket, x->data.minute, x->data.ticket);
    if      (c < 0) x->left  = collectOrder(x->left,  minute, ticket, done);
    else if (c > 0) x->right = collectOrder(x->right, minute, ticket, done);
    else {
        done = true;
        if (!x->left || !x->right) {                 // tanpa anak, atau tepat satu anak
            Node* c2 = x->left ? x->left : x->right;
            delete x;
            if (!c2) return nullptr;
```

Kode 6.12 (Lanjutan) *Papan pengambilan Profitina Laundry: pohon AVL yang berkunci waktu pengambilan yang dijanjikan, mendukung kueri penerus, kueri rentang, dan penghapusan. Kompilasi dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
            x = c2;
        } else {                                    // dua anak
            Node* s = minNode(x->right);            // penerus inorder
            Order moved = s->data;
            bool inner = false;
            x->right = collectOrder(x->right, moved.minute, moved.ticket, inner);
            x->data  = moved;
        }
    }
    return rebalance(x);
}

std::string clock24(int minute) {
    char buf[16];
    std::snprintf(buf, sizeof buf, "%02d:%02d", minute / 60, minute % 60);
    return buf;
}

void show(const Order& o) {
    std::cout << "  " << clock24(o.minute) << "  " << std::left << std::setw(7) << o.ticket
              << std::setw(10) << o.customer << std::right << std::setw(3) << o.kilos << "
kg\n";
}

int main() {
    Node* board = nullptr;
    std::vector<Order> intake = {
        {11*60+30, "PL-014", "Dewi",    4},
        { 9*60+15, "PL-002", "Bagus",   7},
        {16*60+00, "PL-031", "Sari",    3},
        {10*60+45, "PL-008", "Joko",    5},
        {13*60+20, "PL-019", "Rina",    9},
        { 8*60+30, "PL-001", "Agus",    2},
        {17*60+45, "PL-036", "Putu",    6},
        {14*60+10, "PL-023", "Mega",    8},
        {12*60+00, "PL-016", "Hendra", 11},
    };
    for (const Order& o : intake) board = fileOrder(board, o);
    std::cout << "orders on the board : " << intake.size()
              << ",  tree height : " << h(board) << "\n\n";

    int now = 12*60 + 30;
    std::cout << "It is " << clock24(now) << ". The next bag due is:\n";
    const Order* nxt = nextDueAfter(board, now);
    if (nxt) show(*nxt); else std::cout << "  nothing left today\n";
```

Kode 6.12 (Lanjutan) *Papan pengambilan Profitina Laundry: pohon AVL yang berkunci waktu pengambilan yang dijanjikan, mendukung kueri penerus, kueri rentang, dan penghapusan. Kompilasi dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
    std::cout << "\nEverything promised between 10:00 and 14:30:\n";
    std::vector<const Order*> win;
    listWindow(board, 10*60, 14*60+30, win);
    for (const Order* o : win) show(*o);

    std::cout << "\nRina collects PL-019 at 13:20.\n";
    bool done = false;
    board = collectOrder(board, 13*60+20, "PL-019", done);
    std::cout << "removed: " << (done ? "yes" : "no")
              << ",  tree height now : " << h(board) << "\n";

    std::cout << "\nThe window again, after the collection:\n";
    win.clear();
    listWindow(board, 10*60, 14*60+30, win);
    for (const Order* o : win) show(*o);
    return 0;
}
```

Programnya mencetak:

```
orders on the board : 9,  tree height : 3

It is 12:30. The next bag due is:
  13:20  PL-019 Rina        9 kg

Everything promised between 10:00 and 14:30:
  10:45  PL-008 Joko        5 kg
  11:30  PL-014 Dewi        4 kg
  12:00  PL-016 Hendra     11 kg
  13:20  PL-019 Rina        9 kg
  14:10  PL-023 Mega        8 kg

Rina collects PL-019 at 13:20.
removed: yes,  tree height now : 3

The window again, after the collection:
  10:45  PL-008 Joko        5 kg
  11:30  PL-014 Dewi        4 kg
  12:00  PL-016 Hendra     11 kg
  14:10  PL-023 Mega        8 kg
```

Sembilan pesanan duduk dalam pohon bertinggi 3. Batas kita mengizinkan paling banyak $1.4405\lg 11 - 1.3277 \approx 3.6$, jadi 3 adalah nilai sah tertinggi; dan karena pohon bertinggi 2 memuat paling banyak 7 simpul, 3 juga merupakan tinggi terpendek yang mungkin. Pohonnya persis sependek yang diizinkan oleh sembilan simpul. Daftar jendelanya keluar dalam urutan waktu tanpa langkah pengurutan apa pun — ia adalah penelusuran inorder yang dibatasi pada sebuah rentang. Dan kedua uji subpohon di dalam `listWindow` itulah yang membuat kuerinya $O(\lg n + k)$ alih-alih $\Theta(n)$: kalau waktu sebuah simpul sudah lebih awal daripada jendelanya, seluruh subpohon kirinya lebih awal lagi dan tidak pernah dikunjungi.

6.23 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. Memeriksa sifat pohon pencarian biner hanya terhadap anak langsungnya. Inilah kutu tunggal yang paling sering terjadi, dan ia adalah jebakan yang diperkenalkan di §6.3. Uji "anak kiri lebih kecil, anak kanan lebih besar" perlu tetapi jauh dari memadai. Uji yang benar membawa sebuah selang terbuka yang diizinkan turun menyusuri pohonnya: segala isi subpohon kiri x

harus terletak di (lo,kunci(x)) dan segala isi subpohon kanannya di (kunci(x),hi). **Kode 6.13** menunjukkan kedua uji itu pada sebuah pohon yang mengelabui uji yang pertama.

Kode 6.13 Cara yang salah dan cara yang benar untuk memverifikasi sifat pohon pencarian biner. Kompilasi dengan `g++ -std=c++17 -O2 -o isbst isbst.cpp`.

```
// Apakah pohon biner ini sebuah pohon pencarian biner?
// Uji lokal yang menggoda -- "anak kiri lebih kecil, anak kanan lebih besar" -- itu
// salah. Uji yang benar membawa rentang yang diizinkan turun menyusuri pohon.
#include <iostream>
#include <climits>

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k, Node* l = nullptr, Node* r = nullptr) : key(k), left(l), right(r) {}
};

// SALAH: hanya membandingkan sebuah simpul dengan kedua anaknya.
bool looksLocallyFine(Node* x) {
    if (!x) return true;
    if (x->left  && x->left->key  >= x->key) return false;
    if (x->right && x->right->key <= x->key) return false;
    return looksLocallyFine(x->left) && looksLocallyFine(x->right);
}

// BENAR: setiap kunci di subpohon kiri harus di bawah x, setiap kunci di
// subpohon kanan di atasnya, jadi turunkan selang terbuka (lo, hi) ke bawah.
bool isBST(Node* x, long long lo, long long hi) {
    if (!x) return true;
    if (x->key <= lo || x->key >= hi) return false;
    return isBST(x->left, lo, x->key) && isBST(x->right, x->key, hi);
}

int main() {
    /*        20
     *       /  \
     *      10   30
     *          /  \
     *         5    40      <-- 5 ada di subpohon KANAN milik 20: bukan BST
     */
    Node* bad = new Node(20, new Node(10), new Node(30, new Node(5), new Node(40)));

    /*        20
     *       /  \
     *      10   30
     *     /       \
     *    5         40
     */
    Node* good = new Node(20, new Node(10, new Node(5)), new Node(30, nullptr, new
Node(40)));
```

Kode 6.13 (Lanjutan) *Cara yang salah dan cara yang benar untuk memverifikasi sifat pohon pencarian biner. Kompilasi dengan g++ -std=c++17 -O2 -o isbst isbst.cpp.*

```
    std::cout << "bad  tree: local test says "
              << (looksLocallyFine(bad) ? "BST" : "not a BST")
              << ", correct test says "
              << (isBST(bad, LLONG_MIN, LLONG_MAX) ? "BST" : "not a BST") << "\n";
    std::cout << "good tree: local test says "
              << (looksLocallyFine(good) ? "BST" : "not a BST")
              << ", correct test says "
              << (isBST(good, LLONG_MIN, LLONG_MAX) ? "BST" : "not a BST") << "\n";
    return 0;
}
```

Programnya mencetak:

```
bad  tree: local test says BST, correct test says not a BST
good tree: local test says BST, correct test says BST
```

Baris pertamanya adalah seluruh pelajarannya: sebuah pohon yang lolos uji setempat dan gagal pada uji yang sesungguhnya. Selangnya diedarkan sebagai `long long` supaya sentinel `LLONG_MIN` dan `LLONG_MAX` bisa terletak tegas di luar jangkauan kunci `int` mana pun.

2. Menyegarkan tinggi dalam urutan yang salah sesudah rotasi. Dalam `rotateRight(y)`, simpul y berakhir *di bawah* x, jadi `refresh(y)` wajib berjalan sebelum `refresh(x)`. Balikkan kedua baris itu dan tinggi x akan dihitung dari nilai y yang sudah basi. Pohonnya tetap merupakan pohon pencarian yang benar, pemeriksaan invariannya bahkan mungkin lolos, namun tingginya melenceng — dan lencengan itu membuat penyeimbang ulangnya menyala di tempat yang keliru. Kutu ini tidak membuat program jatuh; ia hanya membuat pohonnya pelan-pelan makin buruk. Tambahkan sebuah asersi yang menghitung ulang tinggi dari nol lalu membandingkannya.

3. Mencampur kedua kesepakatan tinggi. Kalau `height(nullptr)` mengembalikan 0 dalam satu fungsi dan -1 dalam fungsi lain, setiap faktor keseimbangan di dekat simpul daun salah satu, dan pohonnya akan berotasi ketika seharusnya tidak, atau gagal berotasi ketika seharusnya iya. Definisikan satu fungsi pembantu `h()` dan jangan pernah membaca `x->height` secara langsung pada penunjuk yang mungkin nol.

4. Lupa bahwa penghapusan harus terus berjalan. Penyisipan boleh berhenti sesudah satu penyeimbangan ulang, tetapi penghapusan tidak boleh — §6.20.2 membuktikannya dan **Kode 6.10** mengukurnya. Kode yang hanya menyeimbangkan ulang leluhur tak seimbang yang pertama lalu kembali akan menghasilkan pohon yang melanggar invariannya. Perumusan rekursif **Kode 6.9** melakukannya dengan benar secara cuma-cuma, karena `rebalance` dipanggil pada *setiap* simpul dalam perjalanan naik kembali.

5. Mengacaukan sifat pohon pencarian biner dengan sifat tumpukan. Dibahas di §6.4 dan dalam **Tabel 6.1**. Kalau Anda mendapati diri Anda menulis "kunci terbesar ada di akar" tentang sebuah pohon pencarian, berhentilah: kunci terbesar sebuah pohon pencarian ada di ujung kanan, dan akarnya ada di suatu tempat di tengah.

6. Mengira sebuah BST seimbang karena biasanya memang begitu. Pohon pencarian biner biasa bersifat $\Theta(\lg n)$ pada masukan acak dan $\Theta(n)$ pada masukan terurut, dan data nyata jauh lebih sering terurut daripada acak. **Kode 6.5** mengukur faktor lima ratus. Kalau kodenya tidak boleh lambat sama sekali, pakailah pohon seimbang; kalau ia hanya perlu biasanya cepat, Anda sedang berjudi dengan data yang tidak Anda kuasai.

7. Menyalin kunci tetapi tidak data satelitnya. Pada kasus 3 penghapusan, `x->key = s->key` memindahkan kuncinya ke atas. Kalau simpulnya juga memegang rekam pelanggan, rekam itu harus ikut pindah. Melupakannya menghasilkan pohon yang kuncinya benar sementara muatannya milik orang lain — kutu yang tidak akan pernah ditemukan pemeriksaan invarian mana pun.

8. Menghapus sebuah simpul selagi ia masih terjangkau. Pada kasus satu anak, naikkan anaknya *sebelum* memanggil `delete`, dan jangan pernah mendereferensi penunjuknya sesudah itu. Ungkapan dalam **Kode 6.4** dan **Kode 6.9** — simpan anaknya, hapus simpulnya, kembalikan anaknya — aman; menyusun ulang urutannya tidak.

9. Memperlakukan kunci kembar sebagai renungan belakangan. Bab ini melarang kunci kembar, dan kedua implementasinya diam-diam mengabaikan kunci yang berulang. Kalau aplikasi Anda memerlukan kunci kembar, putuskanlah dengan sengaja: simpan sebuah cacah di dalam simpulnya, atau perluas kuncinya dengan pemutus seri sebagaimana dilakukan **Kode 6.12** dengan nomor notanya. Yang tidak boleh Anda lakukan adalah membiarkan kunci yang sama melayang ke kedua sisi, sebab kemudian pencariannya tidak lagi bisa tahu sisi mana yang harus diambil.

10. Memakai pohon seimbang di tempat tabel hash sudah cukup. Baris terakhir **Tabel 6.6** bekerja ke dua arah. Kalau operasi Anda hanya "sisipkan", "cari", dan "hapus" sebuah kunci tunggal, tabel hash lebih sederhana dan kira-kira sepuluh kali lebih cepat, sebagaimana diukur **Kode 6.11**. Pohon baru membuktikan biaya tambahannya ketika urutan itu penting.

6.24 Ringkasan Bab

Sebuah **pohon pencarian biner** adalah pohon biner yang setiap kunci di subpohon kiri sebuah simpulnya lebih kecil daripada simpul itu, dan setiap kunci di subpohon kanannya lebih besar — *setiap* kunci, bukan sekadar anak-anaknya. Satu aturan itu, yang dinyatakan dengan tepat di §6.3, membuat tiap pembandingan membuang satu sisi penuh dari koleksinya.

Sifat tumpukan adalah aturan yang berbeda dengan tujuan yang berbeda, dan §6.4 membandingkan keduanya baris demi baris. Tumpukan menemukan nilai ekstremnya dengan cepat; pohon pencarian menemukan apa pun dengan cepat.

Ada tiga **penelusuran**. Penelusuran inorder mencetak kuncinya dalam urutan terurut, dan §6.6.1 membuktikannya dengan induksi atas cacah simpul. Penelusuran preorder membangun kembali bentuk persis sebuah pohon; penelusuran postorder memusnahkannya dengan aman.

Tujuh operasi — **SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, INSERT, dan DELETE** — masing-masing menyusuri satu lintasan, dan masing-masing karena itu berongkos $O(h)$. Penghapusan punya tiga kasus, yang dibedakan oleh cacah anaknya: lepaskan simpul daunnya; naikkan satu-satunya anaknya; atau, untuk dua anak, salin kunci penerus inordernya ke atas lalu hapus penerusnya, yang dijamin merupakan salah satu dari kedua kasus yang mudah.

Seluruh strukturnya hidup atau mati oleh h, dan §6.13 menunjukkan bahwa h bisa mencapai $n - 1$ pada masukan terurut, dengan biaya terukur sebesar faktor kira-kira lima ratus pada $n = 16\,000$.

Sebuah **pohon AVL** [3] memperbaiki ini dengan memelihara sebuah invarian: faktor keseimbangan $\mathrm{bf}(x) = h(\text{kanan}) - h(\text{kiri})$ tetap berada di $\{-1, 0, +1\}$ pada setiap simpul.

Menuliskan $N(h)$ untuk simpul paling sedikit yang bisa dimuat pohon AVL bertinggi h memberikan

$$N(h) = N(h-1) + N(h-2) + 1, \qquad N(0) = 1,\ N(1) = 2,$$

yang penyelesaiannya adalah $N(h) + 1 = F(h+3)$ — sebuah bilangan Fibonacci — dan yang akibatnya adalah

$$h \leq 1.4405\lg(n+2) - 1.3277 = O(\lg n).$$

Sebuah pohon AVL karena itu tidak pernah lebih daripada kira-kira 44% lebih tinggi daripada pohon yang seimbang sempurna [4].

Invariannya dipulihkan oleh **rotasi**, yang mengubah bentuk sebuah pohon sambil membiarkan penelusuran inordernya tidak tersentuh, dan karena itu selalu mengubah pohon pencarian biner menjadi pohon pencarian biner. Ada tepat **empat kasus** — Kiri-Kiri, Kiri-Kanan, Kanan-Kanan, Kanan-Kiri — yang dikenali lewat dua faktor keseimbangan dan diperbaiki oleh satu atau dua rotasi, yang diringkas dalam **Tabel 6.5**.

Penyisipan melakukan paling banyak satu penyeimbangan ulang, karena rotasinya memulihkan subpohonnya ke tinggi sebelum penyisipan sehingga tidak ada leluhur yang menyadarinya. **Penghapusan bisa melakukan satu di setiap aras**, karena sebuah rotasi sesudah penghapusan bisa meninggalkan subpohonnya lebih pendek daripada sebelumnya, dan pemendekan itu merambat. Keduanya tetap $O(\lg n)$.

Tabel 6.7 adalah ringkasan yang perlu dihafalkan.

Tabel 6.7 Biaya kasus terburuk untuk pohon pencarian biner biasa dan untuk pohon AVL dengan n simpul.

Operasi	BST biasa, kasus terburuk	Pohon AVL, kasus terburuk	Mengapa
SEARCH	$\Theta(n)$	$\Theta(\lg n)$	satu lintasan akar-ke-daun
MINIMUM, MAXIMUM	$\Theta(n)$	$\Theta(\lg n)$	satu lintasan, selalu ke arah yang sama
SUCCESSOR, PREDECESSOR	$\Theta(n)$	$\Theta(\lg n)$	satu lintasan turun, atau satu lintasan naik
INSERT	$\Theta(n)$	$\Theta(\lg n)$, ≤ 2 rotasi	susur turun, tempel, seimbangkan ulang sekali
DELETE	$\Theta(n)$	$\Theta(\lg n)$, $O(\lg n)$ rotasi	susur turun, sambung, seimbangkan ulang sampai ke atas
Penelusuran inorder	$\Theta(n)$	$\Theta(n)$	tiap simpul dikunjungi sekali
Kueri rentang yang mengembalikan k kunci	$\Theta(n)$	$\Theta(\lg n + k)$	susur turun sekali, lalu susuri rentangnya
Bangun dari n kunci	$\Theta(n^2)$	$\Theta(n \lg n)$	n penyisipan
Tinggi	sampai $n-1$	$\leq 1.4405\lg(n+2) - 1.3277$	rekurensi simpul-minimum

Akhirnya, §6.21 menempatkan pohon AVL di antara para pesaingnya. Berhadapan dengan **pohon merah-hitam** ia lebih pendek sehingga mencari lebih cepat, tetapi lebih sering menyeimbangkan ulang pada penghapusan, dan itulah sebabnya `std::map` milik pustaka baku C++ bersifat merah-hitam [5] dan sebabnya beban kerja yang berat membaca lebih menyukai AVL [6]. Berhadapan dengan **tabel hash** ia kira-kira sepuluh kali lebih lambat untuk pencarian kunci tunggal dan jauh

tak terbandingkan lebih baik untuk apa pun yang melibatkan urutan: minimum, penerus, pendaftaran terurut, kueri rentang.

6.24.1 Menengok ke depan

Bab berikutnya meninggalkan struktur data dan kembali ke perancangan algoritma, berpindah dari *divide and conquer* (bagi dan kuasai) — yang Anda temui bersama Merge Sort di Bab 3 — ke **pemrograman dinamis**. Jembatannya adalah wajah yang sudah kita kenal: bilangan Fibonacci yang mengukur pohon AVL tertipis di §6.15 juga merupakan contoh pertama yang baku untuk menunjukkan mengapa rekursi naif bersifat eksponensial dan mengapa mengingat jawaban membuatnya linear.

6.25 Soal Latihan

6.1 Gambarlah pohon pencarian biner yang dihasilkan oleh penyisipan kunci 15,6,18,3,7,17,20,2,4,13,9 ke dalam pohon kosong dalam urutan itu. Lalu berikan penelusuran inorder, preorder, dan postordernya.

6.2 Untuk pohon Soal 6.1, berikan hasil SEARCH(13) sebagai sebuah lintasan, SEARCH(8) sebagai sebuah lintasan, MINIMUM, MAXIMUM, SUCCESSOR(13), SUCCESSOR(7), dan PREDECESSOR(6).

6.3 Sebuah pohon biner punya penelusuran preorder 50,30,20,40,70,60,80 dan penelusuran inorder 20,30,40,50,60,70,80. Rekonstruksilah pohonnya. Lalu jelaskan mengapa penelusuran inordernya saja tidak akan cukup.

6.4 Buktikan bahwa jika sebuah simpul dalam pohon pencarian biner punya dua anak, maka penerus inordernya tidak punya anak kiri dan pendahulu inordernya tidak punya anak kanan.

6.5 Berikan sebuah barisan n kunci berbeda yang penyisipannya ke dalam pohon pencarian biner kosong menghasilkan pohon bertinggi $n - 1$, dan sebuah barisan lain dari n kunci yang sama yang menghasilkan tinggi seminimum mungkin. Kerjakan secara nyata untuk $n = 7$ dengan kunci $1, \ldots, 7$.

6.6 Hapuslah kunci 15 dari pohon Soal 6.1, memakai penerus inordernya. Lalu mulailah lagi dari pohon aslinya dan hapuslah 15 memakai pendahulu inordernya. Apakah kedua hasilnya pohon yang sama?

6.7 Hitunglah $N(h)$ untuk $h = 0,1, \ldots, 9$ dari rekurensi $N(h) = N(h-1) + N(h-2) + 1$. Berapa tinggi terbesar yang bisa dipunyai pohon AVL dengan 100 simpul? Dengan 1000?

6.8 Verifikasilah kesamaan $N(h) + 1 = F(h+3)$ untuk $h = 7$ dan $h = 8$ dengan menghitung kedua ruasnya.

6.9 Sisipkan kunci 10,20,30,40,50,25 satu per satu ke dalam pohon AVL kosong. Sesudah tiap penyisipan, nyatakan setiap faktor keseimbangannya, sebutkan kasusnya kalau sebuah rotasi diperlukan, lalu gambar pohonnya.

6.10 Pohon dalam **Gambar 6.8** bukan pohon AVL. Kenalilah z, y, dan x, sebutkan kasusnya, lakukan rotasi atau rotasi-rotasinya, lalu gambar hasilnya lengkap dengan semua faktor keseimbangannya.

6.11 Jelaskan, dengan tinggi-tingginya, mengapa satu rotasi kanan tunggal tidak bisa memperbaiki ketakseimbangan Kiri-Kanan. Berikan faktor keseimbangan yang dihasilkan kalau seseorang mencobanya.

6.12 Buktikan bahwa sebuah rotasi memakan waktu $\Theta(1)$ tanpa peduli ukuran subpohonnya, dan bahwa ia meninggalkan penelusuran inordernya tidak berubah.

6.13 Sebuah pohon AVL bertinggi h. Berapa cacah simpul terkecil dan terbesar yang boleh dimuatnya? Berikan keduanya sebagai rumus lalu hitung nilainya untuk $h = 5$.

6.14 Jelaskan mengapa sebuah *penyisipan* AVL tidak pernah memerlukan lebih daripada satu operasi penyeimbangan ulang, tetapi sebuah *penghapusan* AVL bisa memerlukan satu di setiap aras. Jawaban Anda wajib berbicara tentang tinggi subpohon yang diseimbangkan ulang.

6.15 Anda harus menyimpan 10 juta rekam dan mendukung: (a) pencarian menurut ID rekam; (b) pendaftaran setiap rekam yang cap waktunya jatuh dalam satu jam tertentu; (c) pencarian rekam dengan cap waktu paling awal. Struktur mana atau struktur-struktur mana yang akan Anda pakai, dan mengapa? Bela setiap pilihan Anda terhadap **Tabel 6.6**.

6.16 Tulislah fungsi C++ `int kthSmallest(Node* root, int k)` yang mengembalikan kunci terkecil ke-k dalam sebuah pohon pencarian biner, terindeks mulai 1, dalam waktu $O(h + k)$ tanpa membangun vektor berisi semua kuncinya.

6.17 Ubahlah **Kode 6.9** sehingga setiap simpul juga menyimpan cacah simpul di dalam subpohonnya. Jelaskan apa yang harus berubah dalam rotasinya, dan nyatakan operasi baru apa yang menjadi mungkin dalam $O(\lg n)$ karenanya.

6.18 Buktikan bahwa dalam pohon pencarian biner mana pun, penelusuran inorder mengunjungi kuncinya dalam urutan naik, lalu pakailah itu untuk membuktikan bahwa penerus inorder sebuah simpul x yang punya subpohon kanan adalah minimum subpohon itu.

6.26 Penyelesaian Soal Latihan

6.1 Sisipkan satu kunci pada satu waktu, selalu menyusur turun lewat pembandingan.

15 menjadi akarnya. $6 < 15$, jadi ia anak kiri. $18 > 15$, anak kanan. $3 < 15$, ke kiri ke 6; $3 < 6$, anak kiri 6. $7 < 15$, ke kiri ke 6; $7 > 6$, anak kanan 6. $17 > 15$, ke kanan ke 18; $17 < 18$, anak kiri 18. $20 > 15$, ke kanan ke 18; $20 > 18$, anak kanan 18. 2: ke kiri ke 6, ke kiri ke 3, anak kiri 3. 4: ke kiri ke 6, ke kiri ke 3, $4 > 3$, anak kanan 3. 13: ke kiri ke 6, ke kanan ke 7, $13 > 7$, anak kanan 7. 9: ke kiri ke 6, ke kanan ke 7, ke kanan ke 13, $9 < 13$, anak kiri 13.

Menuliskan bentuknya sebagai `kunci{kiri,kanan}`:

```
15{6{3{2,4},7{.,13{9,.}}},18{17,20}}
```

Tingginya 4 (lintasan $15 \to 6 \to 7 \to 13 \to 9$). Ketiga penelusurannya adalah

- inorder: 2,3,4,6,7,9,13,15,17,18,20 — terurut, sebagaimana dijanjikan §6.6.1;
- preorder: 15,6,3,2,4,7,13,9,18,17,20;
- postorder: 2,4,3,9,13,7,6,17,20,18,15.

6.2

SEARCH(13): $15 \to 6 \to 7 \to 13$. Ketemu, empat simpul dikunjungi.

SEARCH(8): $8 < 15$ ke kiri; $8 > 6$ ke kanan; $8 > 7$ ke kanan; $8 < 13$ ke kiri; $8 < 9$ ke kiri, dan anak kiri 9 kosong. Lintasan $15 \to 6 \to 7 \to 13 \to 9$, gagal. Andaikan kita sedang menyisipkan 8, ia akan menjadi anak kiri 9.

MINIMUM: $15 \to 6 \to 3 \to 2$, jawabannya 2.

MAXIMUM: $15 \to 18 \to 20$, jawabannya 20.

SUCCESSOR(13): simpul 13 tidak punya anak kanan, jadi memanjatlah. 13 adalah anak kanan 7 — terus memanjat. 7 adalah anak kanan 6 — terus memanjat. 6 adalah anak *kiri* 15 — berhenti. Jawabannya 15, dan itu memang yang mengikuti 13 dalam penelusuran inordernya.

SUCCESSOR(7): simpul 7 punya subpohon kanan, yang berakar di 13. Minimumnya ditemukan dengan berjalan ke kiri dari 13, memberikan 9. Jawabannya 9.

PREDECESSOR(6): simpul 6 punya subpohon kiri, yang berakar di 3. Maksimumnya ditemukan dengan berjalan ke kanan dari 3, memberikan 4. Jawabannya 4.

6.3 Kunci pertama sebuah penelusuran preorder selalu akarnya, jadi akarnya adalah 50. Temukan 50 dalam penelusuran inordernya: ia duduk di posisi 4, dengan 20,30,40 sebelumnya dan 60,70,80 sesudahnya. Segala yang sebelumnya adalah subpohon kirinya, segala yang sesudahnya adalah subpohon kanannya — itulah arti "inorder".

Subpohon kirinya karena itu memuat {20,30,40} dan subpohon kanannya memuat {60,70,80}. Ambil tiga kunci berikutnya dari penelusuran preordernya, 30,20,40: inilah penelusuran preorder subpohon kirinya, jadi akarnya 30, subpohon kirinya {20} dan kanannya {40}. Ketiga kunci preorder yang tersisa, 70,60,80, memberikan subpohon kanannya: akar 70, kiri {60}, kanan {80}.

Pohon yang direkonstruksi persis adalah **Gambar 6.1**.

Penelusuran inordernya saja tidak berguna untuk rekonstruksi *justru karena* teorema di §6.6.1: setiap pohon pencarian biner atas himpunan kunci yang sama punya penelusuran inorder yang sama, yaitu daftar terurutnya. Cacah bentuk pohon biner yang berbeda atas n kunci adalah bilangan Catalan $C_n = \frac{1}{n+1}\binom{2n}{n}$, yang untuk $n = 7$ bernilai 429. Penelusuran inordernya tidak bisa membedakan 429 pohon itu; penelusuran preordernya menyebut akarnya di setiap aras sehingga memancangkan bentuknya.

6.4 Misalkan x punya dua anak dan misalkan s adalah penerus inordernya. Menurut kasus 1 di §6.9, s adalah minimum subpohon kanan x, dan TREE-MINIMUM berhenti tepat ketika simpul yang dipijaknya tidak punya anak kiri. Karena itu s tidak punya anak kiri.

Secara simetris, pendahulu inorder p adalah maksimum subpohon kiri x, dan TREE-MAXIMUM berhenti tepat ketika simpulnya tidak punya anak kanan, jadi p tidak punya anak kanan. ■

Inilah fakta yang membuat kasus 3 penghapusan berhenti: simpul tempat kita berekursi punya paling banyak satu anak, jadi rekursinya tidak mungkin bertemu kasus 3 untuk kedua kalinya.

6.5 Sembarang barisan terurut menghasilkan sebuah rantai. Dengan $n = 7$ dan kunci $1, \ldots, 7$, menyisipkan 1,2,3,4,5,6,7 mengirim setiap kunci ke kanan, memberikan pohon bertinggi 6. Begitu pula kebalikannya, 7,6,5,4,3,2,1, yang mengirim setiap kunci ke kiri.

Untuk tinggi minimum, sisipkan mediannya dulu lalu berekursi: 4, lalu median tiap paruhnya, 2 dan 6, lalu keempat kunci yang tersisa 1,3,5,7. Urutan 4,2,6,1,3,5,7 menghasilkan pohon sempurna bertinggi 2, yang merupakan minimum yang diizinkan oleh Persamaan (6.1): $h \geq \lceil \lg(n+1) \rceil - 1 = \lceil \lg 8 \rceil - 1 = 2$.

6.6 Dengan **penerus**: simpul 15 adalah akarnya dan punya dua anak. Subpohon kanannya berakar di 18; minimum subpohon itu adalah 17. Salin 17 ke akarnya lalu hapus simpul 17 yang asli, yang merupakan simpul daun. Hasilnya adalah

```
17{6{3{2,4},7{.,13{9,.}}},18{.,20}}
```

Dengan **pendahulu**: subpohon kiri 15 berakar di 6; maksimumnya ditemukan dengan berjalan ke kanan dari 6 — ke 7, lalu ke 13 — memberikan 13. Simpul 13 punya anak kiri 9 dan tidak punya

anak kanan. Salin 13 ke akarnya, lalu hapus simpul 13 yang asli, yang merupakan kasus 2: naikkan satu-satunya anaknya, 9. Hasilnya adalah

```
13{6{3{2,4},7{.,9}},18{17,20}}
```

Kedua hasilnya **bukan** pohon yang sama. Keduanya pohon pencarian biner yang sah yang memuat sepuluh kunci yang sama, dan keduanya punya penelusuran inorder yang sama, 2,3,4,6,7,9,13,17,18,20 — tetapi bentuknya berbeda. Penghapusan tidak ditentukan secara tunggal; pilihan penerus atau pendahulu adalah pilihan bebas, dan implementasi yang selalu memilih penerus cenderung membuat sisi kiri pohonnya berangsur-angsur makin berat.

6.7 Dari $N(0) = 1$ dan $N(1) = 2$:

$$N(2) = 4,\ N(3) = 7,\ N(4) = 12,\ N(5) = 20,\ N(6) = 33,\ N(7) = 54,\ N(8) = 88,\ N(9) = 143.$$

Untuk 100 simpul: kita memerlukan h terbesar dengan $N(h) \leq 100$. Karena $N(8) = 88 \leq 100$ dan $N(9) = 143 > 100$, jawabannya $h = 8$.

Untuk 1000 simpul: melanjutkan, $N(10) = 232$, $N(11) = 376$, $N(12) = 609$, $N(13) = 986$, $N(14) = 1596$. Karena $N(13) = 986 \leq 1000 < 1596$, jawabannya $h = 13$ — yang merupakan nilai yang dicetak **Kode 6.6**.

6.8 $N(7) = 54$, jadi $N(7) + 1 = 55$. Bilangan Fibonaccinya adalah $F(8) = 21, F(9) = 34, F(10) = 55$, jadi $F(7 + 3) = F(10) = 55$. ✓

$N(8) = 88$, jadi $N(8) + 1 = 89$, dan $F(11) = F(10) + F(9) = 55 + 34 = 89$. ✓

6.9 Memakai $\text{bf} = h(\text{kanan}) - h(\text{kiri})$ sepanjang pengerjaan, dan menuliskan bentuknya sebagai `kunci(bf){kiri,kanan}`:

Sisipkan 10. `10(0)`. Tinggi 0. Tanpa rotasi.

Sisipkan 20. Pergi ke kanan 10. `10(1){.,20(0)}`. Tinggi 1. Faktor keseimbangan $+1$ sah; tanpa rotasi.

Sisipkan 30. Ke kanan 10, ke kanan 20. Sekarang $\text{bf}(20) = +1$ dan $\text{bf}(10) = h(20) - h(\text{kosong}) = 1 - (-1) = +2$: tak seimbang. $z = 10$, $y = 20$ (anak kanan), $x = 30$ (anak kanan y) — **Kanan-Kanan**. Satu rotasi kiri terhadap 10 memberikan `20(0){10(0),30(0)}`, tinggi 1.

Sisipkan 40. Ke kanan 20, ke kanan 30. $\text{bf}(30) = +1$, $\text{bf}(20) = h(\text{subpohon-30}) - h(10) = 1 - 0 = +1$. Keduanya sah. Pohonnya: `20(1){10(0),30(1){.,40(0)}}`, tinggi 2. Tanpa rotasi.

Sisipkan 50. Ke kanan 20, ke kanan 30, ke kanan 40. Sekarang $\text{bf}(40) = +1$, $\text{bf}(30) = h(40) - h(\text{kosong}) = 1 - (-1) = +2$: tak seimbang. $z = 30$, $y = 40$, $x = 50$ — **Kanan-Kanan**. Satu rotasi kiri terhadap 30 memberikan `20(1){10(0),40(0){30(0),50(0)}}`, tinggi 2.

Sisipkan 25. Lintasannya: $25 > 20$ ke kanan ke 40; $25 < 40$ ke kiri ke 30; $25 < 30$, tempelkan sebagai anak kiri 30. Sekarang $\text{bf}(30) = h(\text{kosong}) - h(25) = -1 - 0 = -1$ dan $h(30) = 1$; $\text{bf}(40) = h(50) - h(30) = 0 - 1 = -1$ dan $h(40) = 2$; $\text{bf}(20) = h(40) - h(10) = 2 - 0 = +\mathbf{2}$: tak seimbang. $z = 20$; $y = 40$, anak kanannya; $x = 30$, anak *kiri* y — **Kanan-Kiri**. Mula-mula rotasi kanan terhadap $y = 40$, lalu rotasi kiri terhadap $z = 20$. Hasilnya adalah

```
30(0){20(0){10(0),25(0)},40(1){.,50(0)}}
```

tinggi 2, dan setiap faktor keseimbangannya sah. Enam penyisipan, tiga penyeimbangan ulang, dan pohonnya tidak pernah melampaui tinggi 2 — sedangkan pohon pencarian biner biasa yang disuapi 10,20,30,40,50 sudah akan menjadi rantai bertinggi 4.

6.10 Pohon **Gambar 6.8** adalah: akar 13; anak kiri 9 dengan anak 6 dan 12; simpul 6 dengan anak 5 dan 8; simpul 5 dengan anak kiri 3; anak kanan 19 dengan anak kiri 18.

Hitung ke atas dari 3: $h(3) = 0$; $h(5) = 1$ dengan $\mathrm{bf}(5) = -1$; $h(8) = 0$; $h(6) = 2$ dengan $\mathrm{bf}(6) = 0 - 1 = -1$; $h(12) = 0$; dan $\mathrm{bf}(9) = h(12) - h(6) = 0 - 2 = \mathbf{-2}$.

Jadi $z = 9$, simpul tak seimbang yang terdalam. Anak z pada lintasan dari 3 adalah $y = 6$, dan cucunya adalah $x = 5$. Simpul y adalah anak *kiri* z dan simpul x adalah anak *kiri* y: inilah kasus **Kiri-Kiri**, yang dikukuhkan oleh $\mathrm{bf}(y) = -1 < 0$ dalam **Tabel 6.5**.

Satu rotasi kanan terhadap $z = 9$: simpul 6 naik, 9 menjadi anak kanan 6, dan anak kanan lama milik 6, yaitu 8, menjadi anak kiri 9. Tempelkan 6 di tempat 9 tadi berada, sebagai anak kiri 13. Hasilnya, lengkap dengan faktor keseimbangannya, adalah

```
13(-1){6(0){5(-1){3(0),.},9(0){8(0),12(0)}},19(-1){18(0),.}}
```

Perhatikan bahwa simpul 13 *juga* berada di luar rentang sebelum rotasinya ($\mathrm{bf}(13) = 1 - 3 = -2$), dan bahwa memperbaiki $z = 9$ dengan sendirinya memperbaikinya juga — sebab rotasinya memulihkan subpohonnya ke tinggi sebelum penyisipan, persis seperti yang dijelaskan §6.17.1 dan §6.20.2.

6.11 Ambil susunan Kiri-Kanan **Gambar 6.11**, dengan P_1 dan P_4 bertinggi k, simpul x bertinggi $k + 1$, simpul y bertinggi $k + 2$, dan simpul z bertinggi $k + 3$.

Lakukan satu rotasi kanan tunggal terhadap z. Simpul y pindah ke puncak; P_1 — subpohon kiri lamanya, tinggi k — tetap menjadi subpohon kirinya; z menjadi anak kanannya; dan x, yang tadinya subpohon kanan y, diserahkan kepada z sebagai subpohon kiri z.

Sekarang z punya anak x (tinggi $k + 1$) dan P_4 (tinggi k), jadi $h(z) = k + 2$. Dan y punya anak P_1 (tinggi k) dan z (tinggi $k + 2$), jadi

$$\mathrm{bf}(y) = (k + 2) - k = +2.$$

Masih di luar rentang, hanya condong ke arah yang lain. Satu rotasi tunggal hanya bisa memindahkan satu aras tinggi melintasi akar subpohonnya; sebuah zig-zag menyembunyikan kelebihan tingginya satu aras lebih dalam, jadi ia harus lebih dulu diluruskan menjadi zig-zig oleh rotasi dalamnya. Itulah sebabnya LR dan RL selalu berongkos dua rotasi.

6.12 *Waktu konstan.* Periksalah `rotateRight` di §6.16.1. Ia membaca dua penunjuk, menulis dua penunjuk, memanggil `refresh` dua kali — yang masing-masing membaca dua tinggi dan menulis satu — lalu kembali. Itu sejumlah tetap operasi mesin, yang tak satu pun bergantung pada cacah simpul di P_1, P_2, atau P_3: subpohonnya dipindahkan dengan mengubah penunjuk yang menamainya, tidak pernah dengan mengunjunginya. Karena itu $\Theta(1)$.

Inorder terjaga. Sebelum rotasinya, subpohonnya adalah y dengan anak kiri x (subpohon P_1, P_2) dan subpohon kanan P_3. Penelusuran inordernya, menurut definisi, adalah penelusuran subpohon x, lalu y, lalu P_3; dan penelusuran subpohon x adalah P_1, lalu x, lalu P_2. Jadi seluruh penelusurannya adalah

$$P_1,\ x,\ P_2,\ y,\ P_3.$$

Sesudah rotasinya, subpohonnya adalah x dengan subpohon kiri P_1 dan anak kanan y (subpohon P_2, P_3). Penelusuran inordernya adalah P_1, lalu x, lalu penelusuran subpohon y, yaitu P_2, lalu y, lalu P_3 — memberikan

$$P_1,\ x,\ P_2,\ y,\ P_3,$$

barisan yang sama. Karena sebuah pohon biner merupakan pohon pencarian biner tepat ketika penelusuran inordernya naik, dan penelusurannya tidak berubah, pohon yang dirotasi tetap merupakan pohon pencarian biner. ■

6.13 Cacah simpul **terkecil**nya adalah $N(h)$, yang diberikan oleh rekurensi §6.15: $N(h) = N(h-1) + N(h-2) + 1$, setara dengan $N(h) = F(h+3) - 1$. Yang **terbesar** dicapai oleh pohon biner sempurna, yang mengisi penuh setiap arasnya: $1 + 2 + \cdots + 2^h = 2^{h+1} - 1$, dan pohon sempurna punya setiap faktor keseimbangan bernilai 0, jadi ia pasti pohon AVL.

Untuk $h = 5$: yang terkecil adalah $N(5) = 20$ dan yang terbesar adalah $2^6 - 1 = 63$. Jadi pohon AVL bertinggi 5 memuat antara 20 dan 63 simpul, keduanya termasuk.

6.14 Sesudah sebuah penyisipan yang membuat z tak seimbang, subpohon yang berakar di z bertinggi $k + 3$ sedangkan tingginya sebelum penyisipan adalah $k + 2$ — lihat hitungan tinggi di §6.17.1 sampai §6.17.4. Rotasinya mengembalikan subpohon itu ke tinggi $k + 2$. Karena tinggi subpohonnya kini sama persis dengan tingginya sebelum penyisipan, setiap leluhur z melihat seorang anak yang tingginya tidak berubah, sehingga tidak ada faktor keseimbangan leluhur yang bisa bergeser. Satu penyeimbangan ulang sudah cukup.

Sesudah sebuah penghapusan, hitungannya berbeda satu. Penghapusannya memendekkan suatu subpohon; rotasi di z kemudian menghasilkan subpohon yang mungkin *satu lebih pendek* daripada sebelum penghapusannya. Pemendekan itu adalah perubahan tinggi yang sungguhan sebagaimana dilihat induk z, dan ia bisa mendorong faktor keseimbangan induknya ke ± 2. Hal yang sama bisa terjadi di kakeknya, dan seterusnya. Pada kasus terburuk perbaikannya merambat sampai ke akar, satu penyeimbangan ulang per aras, yang berarti $O(\lg n)$ kali. **Kode 6.10** mengukur persis ini pada pohon Fibonacci: konstanta 1 untuk penyisipan pada setiap tinggi, dan 2,3,5,6,8 untuk penghapusan pada tinggi 4,7,10,13,16.

6.15

(a) *Pencarian menurut ID rekam* adalah pencarian titik murni: diberi sebuah kunci yang persis, kembalikan rekamnya. Tidak ada pengurutan yang terlibat. Pakailah **tabel hash**, harapan $O(1)$. Menurut baris terakhir **Tabel 6.6**, inilah kasus tempat biaya tambahan sebuah pohon tidak membeli apa pun.

(b) *Setiap rekam yang cap waktunya jatuh dalam satu jam tertentu* adalah kueri rentang. Tabel hash harus memindai seluruh 10 juta rekamnya; pohon seimbang yang berkunci cap waktu menyusur turun sekali lalu menyusuri hanya rekam yang cocok, $O(\lg n + k)$. Pakailah **pohon AVL atau pohon merah-hitam yang berkunci cap waktu**. Apakah AVL atau merah-hitam bergantung pada laju operasi tulisnya: kalau 10 juta rekam itu dimuat sekali lalu sebagian besar hanya dibaca, tinggi AVL yang lebih pendek menang; kalau rekam datang dan dihapus terus-menerus, biaya penghapusan merah-hitam yang terbatas yang menang.

(c) *Rekam dengan cap waktu paling awal* adalah TREE-MINIMUM pada pohon berkunci cap waktu yang sama: $O(\lg n)$, dan gratis, karena indeksnya sudah ada.

Jawaban yang jujur karena itu adalah **dua struktur di atas data yang sama**: sebuah indeks hash atas ID rekam, dan sebuah indeks terurut atas cap waktu. Inilah persis yang dilakukan sebuah basis data relasional ketika Anda mendeklarasikan indeks hash dan indeks B-tree pada tabel yang sama, dan inilah sebabnya sistem nyata memelihara beberapa indeks alih-alih satu.

6.16 Penelusuran inorder mengunjungi kunci dalam urutan naik, jadi kunci ke-k yang dikeluarkannya adalah kunci terkecil ke-k. Kiatnya adalah menghentikan penelusurannya begitu cacahnya mencapai k, yang membuat ongkosnya $O(h+k)$ alih-alih $\Theta(n)$: paling banyak h simpul dilalui dalam perjalanan turun ke kunci terkecilnya, dan paling banyak k simpul dikeluarkan sesudah itu.

Kode 6.14 Kunci terkecil ke-k dari sebuah pohon pencarian biner dalam waktu $O(h+k)$. Kompilasi dengan `g++ -std=c++17 -O2 -o kth kth.cpp`.

```
// Penyelesaian Soal 6.16: kunci terkecil ke-k pada pohon pencarian biner,
// ditemukan lewat penelusuran inorder yang berhenti begitu sudah mencacah k kunci.
#include <iostream>
#include <stdexcept>

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

Node* insertKey(Node* x, int k) {
    if (!x) return new Node(k);
    if      (k < x->key) x->left  = insertKey(x->left,  k);
    else if (k > x->key) x->right = insertKey(x->right, k);
    return x;
}

// Menelusuri inorder tetapi mengembalikan true begitu kunci ke-k ditemukan, sehingga
// rekursinya terurai tanpa menyentuh sisa pohonnya.
bool kthHelper(Node* x, int& remaining, int& answer) {
    if (!x) return false;
    if (kthHelper(x->left, remaining, answer)) return true;   // ditemukan di sebelah kiri
    if (--remaining == 0) { answer = x->key; return true; }   // simpul ini adalah yang ke-
k
    return kthHelper(x->right, remaining, answer);            // kalau tidak, tengok ke
kanan
}

int kthSmallest(Node* root, int k) {
    if (k < 1) throw std::out_of_range("k must be at least 1");
    int remaining = k, answer = 0;
    if (!kthHelper(root, remaining, answer))
        throw std::out_of_range("the tree has fewer than k keys");
    return answer;
}

int main() {
    Node* t = nullptr;
    for (int k : {15, 6, 18, 3, 7, 17, 20, 2, 4, 13, 9}) t = insertKey(t, k);

    for (int k = 1; k <= 11; ++k)
        std::cout << "k = " << k << "  ->  " << kthSmallest(t, k) << "\n";
```

Kode 6.14 (Lanjutan) *Kunci terkecil ke-k dari sebuah pohon pencarian biner dalam waktu O(h+k). Kompilasi dengan g++ -std=c++17 -O2 -o kth kth.cpp.*

```
    try { kthSmallest(t, 12); }
    catch (const std::out_of_range& e) { std::cout << "k = 12 ->  " << e.what() << "\n"; }
    return 0;
}
```

Dijalankan pada pohon Soal 6.1, **Kode 6.14** mencetak:

```
k = 1  ->  2
k = 2  ->  3
k = 3  ->  4
k = 4  ->  6
k = 5  ->  7
k = 6  ->  9
k = 7  ->  13
k = 8  ->  15
k = 9  ->  17
k = 10  ->  18
k = 11  ->  20
k = 12 ->  the tree has fewer than k keys
```

yang merupakan penelusuran inorder Soal 6.1, unsur demi unsur.

6.17 Tambahkan sebuah medan `int size;` pada `Node`, yang memuat cacah simpul di subpohon yang berakar di sana, lalu peliharalah ia persis sebagaimana tingginya dipelihara: satu fungsi bantu baru `sz`, yang mengembalikan 0 untuk pohon kosong, dan satu baris baru di dalam `refresh`. **Kode 6.15** menampilkan hasil selengkapnya.

Kode 6.15 Pohon statistik terurut: pohon AVL **Kode 6.9** dengan ukuran subpohon di setiap simpul, yang mendukung `select` dan `rank` dalam $O(\lg n)$. Kompilasi dengan `g++ -std=c++17 -O2 -o ostree ostree.cpp`.

```
// Pohon AVL yang simpulnya juga menyimpan ukuran subpohonnya: pohon statistik terurut.
// select(k) menemukan kunci terkecil ke-k dan rank(key) mencacah kunci yang lebih kecil,
// keduanya dengan satu penyusuran turun, jadi keduanya berongkos O(lg n).
#include <iostream>
#include <algorithm>

struct Node {
    int key, height, size;             // size = cacah simpul di subpohon ini
    Node* left;
    Node* right;
    Node(int k) : key(k), height(0), size(1), left(nullptr), right(nullptr) {}
};

static int h (const Node* x) { return x ? x->height : -1; }
static int sz(const Node* x) { return x ? x->size   :  0; }
static int bf(const Node* x) { return h(x->right) - h(x->left); }
static void refresh(Node* x) {
    x->height = 1 + std::max(h(x->left), h(x->right));
    x->size   = 1 + sz(x->left) + sz(x->right);     // baris yang baru
}

static Node* rotateRight(Node* y) {                 // tidak berubah dari kode pohon AVL
    Node* x = y->left; y->left = x->right; x->right = y;
    refresh(y); refresh(x); return x;
}
static Node* rotateLeft(Node* x) {
    Node* y = x->right; x->right = y->left; y->left = x;
    refresh(x); refresh(y); return y;
}
static Node* rebalance(Node* z) {
    refresh(z);
    if (bf(z) < -1) {
        if (bf(z->left) > 0) z->left = rotateLeft(z->left);
        return rotateRight(z);
    }
    if (bf(z) > 1) {
        if (bf(z->right) < 0) z->right = rotateRight(z->right);
        return rotateLeft(z);
    }
    return z;
}
static Node* insert(Node* x, int key) {
    if (!x) return new Node(key);
    if      (key < x->key) x->left  = insert(x->left,  key);
    else if (key > x->key) x->right = insert(x->right, key);
    return rebalance(x);
```

Kode 6.15 (Lanjutan) *Pohon statistik terurut: pohon AVL Kode 6.9 dengan ukuran subpohon di setiap simpul, yang mendukung select dan rank dalam O(\lg n). Kompilasi dengan g++ -std=c++17 -O2 -o ostree ostree.cpp.*

```
}

// Kunci terkecil ke-k, berindeks mulai 1. Di tiap simpul, L = ukuran subpohon kiri.
static int select(const Node* x, int k) {
    while (x) {
        int L = sz(x->left);
        if (k <= L)          x = x->left;          // jawabannya ada di kiri
        else if (k == L + 1) return x->key;        // tepat L kunci yang lebih kecil
        else { k -= L + 1;   x = x->right; }       // lewati subpohon kiri dan x
    }
    return -1;                                      // k di luar rentang
}

// Berapa banyak kunci di dalam pohon yang lebih kecil daripada key.
static int rank(const Node* x, int key) {
    int r = 0;
    while (x) {
        if (key <= x->key) x = x->left;
        else { r += sz(x->left) + 1; x = x->right; }
    }
    return r;
}

static void destroy(Node* x) { if (x) { destroy(x->left); destroy(x->right); delete x; } }

int main() {
    Node* root = nullptr;
    for (int k : {15, 6, 18, 3, 7, 17, 20, 2, 4, 13, 9}) root = insert(root, k);
    std::cout << "n = " << sz(root) << ", height = " << h(root) << "\n";
    std::cout << "select(k) for k = 1..11:";
    for (int k = 1; k <= sz(root); ++k) std::cout << " " << select(root, k);
    std::cout << "\n";
    for (int key : {2, 9, 15, 21})
        std::cout << "rank(" << key << ") = " << rank(root, key) << "\n";
    destroy(root);
    return 0;
}
```

Dijalankan pada kunci-kunci Soal 6.1, program itu mencetak:

```
n = 11, height = 3
select(k) for k = 1..11: 2 3 4 6 7 9 13 15 17 18 20
rank(2) = 0
rank(9) = 5
rank(15) = 7
rank(21) = 11
```

Karena kedua rotasinya sudah memanggil `refresh` pada kedua simpul yang anaknya berubah — dan dalam urutan dari bawah ke atas yang benar — **tidak ada perubahan lain pada rotasinya yang diperlukan**. Itulah hasil dari mengisolasi pembaruannya dalam satu fungsi pembantu. Rekursi penyisipan dan penghapusannya juga tidak memerlukan perubahan, karena setiap simpul pada lintasan naik kembali diserahkan kepada `rebalance`, yang dimulai dengan memanggil `refresh`.

Operasi baru yang menjadi mungkin karenanya adalah **statistik terurut dalam** $O(\lg n)$: kunci terkecil ke-k kini bisa ditemukan dengan menyusur turun sekali, tanpa penelusuran $O(k)$ milik **Kode 6.14**; fungsi `select` pada **Kode 6.15** adalah penyusuran itu. Pada sebuah simpul x, misalkan $L = \text{sz}(x\text{->left})$. Jika $k \leq L$, berekursilah ke kiri; jika $k = L + 1$, jawabannya adalah kunci(x); selain itu berekursilah ke kanan sambil mencari yang terkecil ke-$(k - L - 1)$. Operasi kebalikannya, **peringkat** — "berapa banyak kunci yang lebih kecil daripada k?" — datang cuma-

cuma lewat penyusuran yang sama, seperti ditunjukkan `rank`: setiap kali penyusurannya berbelok ke kanan, simpul itu beserta seluruh subpohon kirinya lebih kecil, dan cacah mereka ditambahkan. Pohon yang membawa ukuran subpohon disebut *order-statistic tree* (pohon statistik terurut), dan ia adalah contoh kerja baku untuk augmentasi pohon [7].

6.18 Paruh pertamanya adalah Teorema 6.6.1, yang dibuktikan di sana dengan induksi atas cacah simpulnya: penelusuran inorder pohon kosong naik secara remeh, dan untuk pohon berakar x penelusurannya adalah (penelusuran L), kunci(x), (penelusuran R), yang ketiga potongnya naik menurut hipotesis induksi dan yang kedua sambungannya naik menurut sifat pohon pencarian biner di x.

Untuk paruh keduanya, misalkan x punya subpohon kanan R yang tidak kosong, dan misalkan m adalah minimum R. Setiap kunci R melampaui kunci(x), jadi setiap kunci R adalah calon bagi "kunci terkecil yang lebih besar daripada kunci(x)", dan kunci(m) adalah yang terkecil di antara mereka. Yang tersisa adalah menyingkirkan calon di luar R. Misalkan u adalah sembarang simpul di luar subpohon yang berakar di x dengan kunci$(u) >$ kunci(x). Susurlah turun dari akarnya menuju x: pada leluhur terakhir a tempat lintasannya membelok ke kiri masuk ke subpohon yang memuat x, setiap kunci di subpohon kanan a — dan di sanalah u pasti terletak — melampaui kunci(a), sedangkan kunci(a) melampaui setiap kunci di subpohon yang berakar di x, sehingga melampaui kunci(m). Jadi kunci$(u) >$ kunci(m), dan tidak ada simpul luar yang bisa mengalahkan m. Karena itu penerus x adalah m, minimum subpohon kanannya. ■

Daftar Pustaka

[1] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, 4th ed. Boston, MA, USA: Pearson, 2014.

[2] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[3] G. M. Adelson-Velsky and E. M. Landis, "An algorithm for the organization of information," *Doklady Akademii Nauk SSSR*, vol. 146, no. 2, pp. 263–266, 1962. (English translation: *Soviet Mathematics Doklady*, vol. 3, pp. 1259–1263, 1962.)

[4] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[5] ISO/IEC 14882:2024, *Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2024.

[6] B. Pfaff, "Performance analysis of BSTs in system software," in *Proc. ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems*, New York, NY, USA, 2004, pp. 410–411.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

Bab 7
Dari Divide and Conquer ke Dynamic Programming

Di akhir bab ini Anda akan sanggup menatap sebuah algoritma rekursif lalu mengatakan, dengan bukti di tangan, apakah algoritma itu sedang menghabiskan umurnya untuk menghitung ulang jawaban yang sudah pernah ia temukan. Anda akan sanggup mengubah algoritma semacam itu menjadi algoritma yang polinomial, bukan eksponensial, lewat salah satu dari dua perbaikan baku — mengingat jawaban sambil berjalan, atau membangunnya dari yang terkecil lebih dulu — dan Anda akan tahu mana di antara keduanya yang pantas diambil, beserta alasannya. Anda akan sanggup menyatakan apa arti *optimal substructure* (substruktur optimal), dan membuktikan bahwa sebuah persoalan tertentu memilikinya. Anda akan sanggup mengambil sederetan matriks, mengisi tabel *dynamic programming*-nya dengan tangan, dan membaca cara terbaik mengurung hasil kalinya langsung dari tabel itu. Anda akan sanggup menghitung *longest common subsequence* dari dua untai, merekonstruksi subbarisannya sendiri dari sebuah tabel panah, dan memangkas memori yang diperlukannya dari $\Theta(mn)$ menjadi $\Theta(\min(m,n))$ ketika subbarisannya sendiri tidak Anda butuhkan. Dan Anda akan sanggup menyatakan, dalam satu kalimat yang bisa Anda pertahankan di hadapan siapa pun yang meragukannya, beda antara persoalan yang menuntut *divide and conquer* dan persoalan yang menuntut *dynamic programming*.

7.1 Dari Mana Kita Melanjutkan

Bab 2 dan 3 menyerahkan kepada kita sebuah cara menghitung dan sebuah cara berargumen. Bab 3 secara khusus memberi kita *recurrence* (rekurensi): persamaan yang di dalamnya sebuah fungsi didefinisikan lewat dirinya sendiri pada masukan yang lebih kecil.

Ingat kembali notasi yang kita tetapkan di sana, sebab bab ini bersandar padanya terus-menerus. Jika sebuah algoritma pada masukan berukuran n memecah masukan itu menjadi a potongan, masing-masing berukuran n/b, menyelesaikan tiap potongan dengan memanggil dirinya sendiri, dan menghabiskan waktu $f(n)$ untuk segala sesuatu yang lain — pemecahan sebelumnya dan penggabungan sesudahnya — maka waktu jalannya $T(n)$ memenuhi Persamaan (7.1):

$$T(n) = a\,T(n/b) + f(n), \qquad a \geq 1,\ b > 1. \tag{7.1}$$

Kita menyebutnya **rekurensi divide-and-conquer umum**, dan kita mempelajari tiga cara menyelesaikannya: *substitution method* (metode substitusi — tebak sebuah batas, lalu buktikan dengan induksi), *recursion-tree method* (metode pohon rekursi — gambar pohon panggilannya, jumlahkan biaya tiap aras, lalu jumlahkan arasnya), dan *Master Theorem* (Teorema Master — resep yang membaca jawabannya dengan membandingkan $f(n)$ terhadap $n^{\log_b a}$).

Kita juga berjumpa, di §3.8.9, dengan bilangan Fibonacci, dan menghitungnya dengan rumus tertutup. Dalam bab ini bilangan itu kembali sebagai penjahat di babak pembuka — lalu sebagai pahlawan dalam babak perbaikan.

Dua patah kosakata dari Bab 3 pantas dinyatakan ulang sebelum kita melangkah, sebab seluruh isi bab ini adalah pernyataan tentang keduanya.

DEFINISI. Sebuah **submasalah** (*subproblem*) dari sebuah instans persoalan adalah instans yang lebih kecil dari persoalan yang *sama*, yang dihasilkan oleh algoritma dalam perjalanannya menyelesaikan persoalan asli. Menghitung F_5 adalah submasalah dari menghitung F_7; mengurutkan $A[1..4]$ adalah submasalah dari mengurutkan $A[1..8]$.

DEFINISI. Pohon rekursi (*recursion tree*) sebuah panggilan rekursif adalah pohon yang akarnya adalah panggilan asli, yang anak-anaknya adalah panggilan-panggilan yang dibuat langsung oleh panggilan itu, dan seterusnya ke bawah sampai panggilan-panggilan yang tidak membuat panggilan lagi (kasus basis, digambar sebagai daun). Setiap simpul diberi label submasalah yang ia selesaikan.

Segala sesuatu dalam bab ini mengalir dari satu pengamatan tentang pohon tersebut: **kadang-kadang label yang sama muncul pada lebih dari satu simpul.** Ketika hal itu tidak terjadi, *divide and conquer* adalah perkakas yang tepat. Ketika hal itu terjadi — dan terjadi dengan frekuensi yang membawa bencana — *dynamic programming* adalah perkakas yang tepat. Sisa bab ini dihabiskan untuk membuat kalimat itu menjadi tepat, membuktikannya, mengukurnya, dan menerapkannya pada tiga persoalan secara tuntas.

7.1.1 Apa yang masih kurang

Bab 3 mengandaikan, tanpa pernah mengatakannya, bahwa potongan-potongan yang dihasilkan sebuah algoritma *divide and conquer* adalah *potongan yang berbeda-beda*. Merge Sort memecah $A[1..8]$ menjadi $A[1..4]$ dan $A[5..8]$; kedua paruh itu tidak berbagi satu elemen pun, dan mengurutkan yang satu tidak memberi tahu apa pun tentang yang lain. Di bawah andaian itu, pohon rekursinya tidak punya label yang berulang, setiap simpul mengerjakan kerja baru yang jujur, dan hitungan Teorema Master persis benar.

Andaian itu tidak selalu benar. Ketika ia gagal, Teorema Master tetap menyelesaikan rekurensinya dengan benar — ia hanya menyelesaikannya menjadi jawaban seperti $\Theta(1.618^n)$, yang merupakan gambaran tepat tentang sebuah bencana. Bab ini adalah tentang menyadari bencana itu dan mencegahnya.

7.2 Divide and Conquer sebagai Disiplin Perancangan

7.2.1 Tiga langkahnya

Sebelum kita mematahkan *divide and conquer*, mari kita nyatakan ia sekali lagi dengan bersih. **Gambar 7.1** memuat pernyataan lengkapnya.

Gambar 7.1 Metode *divide and conquer* untuk perancangan algoritma: bagilah persoalan menjadi submasalah yang **saling lepas**, kuasai submasalah itu dengan rekursi, lalu gabungkan solusinya menjadi solusi persoalan asli.

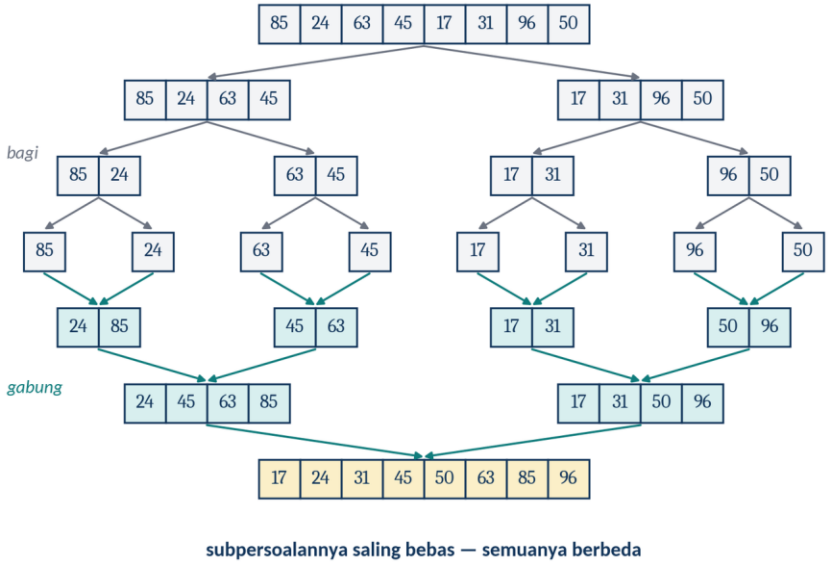
Tiga patah kata, dan masing-masing adalah instruksi yang sungguhan:

1. **Divide (bagi).** Jika masukannya terlalu besar untuk ditangani sekaligus, potonglah menjadi dua submasalah atau lebih yang merupakan instans lebih kecil dari persoalan yang sama. Kata kuncinya adalah *disjoint* (saling lepas): potongan-potongan itu tidak tumpang tindih.
2. **Conquer (kuasai).** Selesaikan tiap submasalah dengan memanggil prosedur yang sama atasnya. Jika sebuah submasalah sudah cukup kecil — biasanya berukuran 0 atau 1 — selesaikan langsung, tanpa rekursi. Kasus langsung itulah **kasus basis**, dan tanpa satu pun kasus basis rekursinya tak pernah berhenti.
3. **Combine (gabungkan).** Ambil solusi submasalah-submasalahnya dan leburkan menjadi solusi persoalan asli.

> **DENGAN BAHASA SEDERHANA.** Anda punya segunung cucian. Anda tidak bisa mencucinya sekaligus, jadi Anda membaginya menjadi dua tumpukan (*divide*), mencuci tiap tumpukan dengan cara yang sama seperti kalau Anda mencuci semuanya (*conquer*), lalu menyatukan dua tumpukan bersih itu menjadi satu gunung yang bersih (*combine*). Kedua tumpukan itu *saling lepas*: tidak ada satu kemeja pun yang berada di keduanya. Mencuci tumpukan pertama tidak mengajari Anda apa pun tentang tumpukan kedua, dan Anda tidak akan pernah terpikir untuk mencuci kemeja yang sama dua kali.

7.2.2 Merge Sort, sekali lagi dilihat sebagai bentuk

Contoh bakunya adalah Merge Sort, dan **Gambar 7.2** menggambarkannya.



Gambar 7.2 Merge Sort sebagai *divide and conquer*. Larik dibelah dua, tiap paruh diurutkan secara rekursif, lalu kedua paruh terurut itu digabungkan. Kalimat yang penting tertulis di bawah gambarnya: **submasalahnya saling bebas, semuanya berbeda.**

Bacalah kalimat pada **Gambar 7.2** itu pelan-pelan, sebab itulah engsel seluruh bab ini: *submasalahnya saling bebas, semuanya berbeda*. Ketika Merge Sort diminta mengurutkan larik berisi enam belas elemen, ia meminta dua larik berisi delapan elemen. Kedua larik itu tidak pernah berimpit. Masing-masing meminta dua larik berisi empat elemen; keempat larik itu pun tidak pernah berimpit. Tidak ada label yang terulang di mana pun dalam pohon itu. Setiap simpul mengerjakan kerja yang tidak dikerjakan simpul lain mana pun.

Itulah sebabnya rekurensi $T(n) = 2T(n/2) + \Theta(n)$ menceritakan seluruh kebenaran tentang Merge Sort, dan itulah sebabnya jawaban $\Theta(n\lg n)$ adalah algoritma yang cepat, bukan yang lambat.

7.2.3 Kapan divide and conquer adalah perkakas yang tepat

Divide and conquer membayar ongkosnya sendiri tepat ketika tiga syarat berlaku bersamaan:

- persoalannya sungguh-sungguh terbelah — ada cara alami untuk memotong masukan menjadi instans lebih kecil dari persoalan yang sama;
- potongan-potongannya **saling lepas**, sehingga tidak ada kerja yang terduplikasi di antara mereka;
- langkah **combine**-nya murah dibandingkan kerja di bawahnya, sehingga aras-aras pohon rekursinya tidak semuanya berbiaya lebih mahal daripada aras terakhir.

Merge Sort memenuhi ketiganya. Begitu pula pencarian biner (satu potongan alih-alih dua, biaya penggabungan nol, $T(n) = T(n/2) + \Theta(1) = \Theta(\lg n)$). Begitu pula perkalian matriks Strassen ($T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{\lg 7})$). Begitu pula Quicksort, pada kasus rata-rata.

7.2.4 Kapan divide and conquer memboroskan kerja

Sekarang andaikan syarat kedua gagal. Andaikan potongan-potongannya *tumpang tindih* — bahwa submasalah "urutkan $A[1..4]$" dan submasalah "urutkan $A[3..6]$" sama-sama muncul, berbagi dua elemen; atau, lebih buruk lagi, bahwa submasalah yang persis sama muncul dua kali, tiga kali, seribu kali.

Implementasi rekursif yang polos tidak punya ingatan. Ia tidak tahu bahwa ia pernah melihat sebuah submasalah sebelumnya. Mintalah jawabannya dua kali dan ia akan menghitung jawaban itu dua kali, dari nol, sampai ke daun-daunnya. Mintalah sejuta kali dan ia akan mengerjakan sejuta perhitungan penuh.

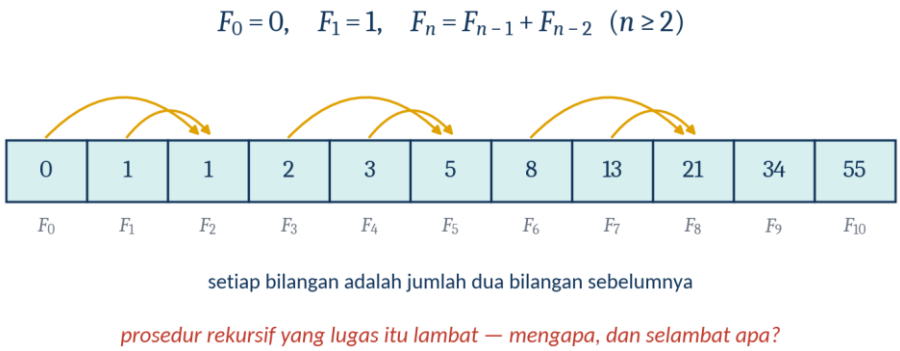
> **JEBAKANNYA.** Rekursi itu tidak lambat. *Perhitungan ulang*-lah yang lambat. Prosedur rekursif yang submasalahnya tidak pernah berulang secepat apa pun; prosedur rekursif yang submasalahnya berulang adalah mesin untuk mengerjakan kerja yang sama berkali-kali, dan biayanya bisa eksponensial bahkan ketika banyaknya pertanyaan yang sungguh-sungguh berbeda yang perlu ia jawab hanyalah segelintir.

Seberapa buruk jadinya? Contoh menarik yang terkecil — tiga baris kode, tanpa kepintaran apa pun, diajarkan kepada pemula di mana-mana — sudah cukup buruk untuk tidak terpakai. Contoh itu adalah Fibonacci, dan ke sanalah kita melangkah berikutnya.

7.3 Fibonacci: Persoalan Terkecil yang Mematahkan Divide and Conquer

7.3.1 Definisinya

Gambar 7.3 menyatakan definisinya.



Gambar 7.3 Bilangan Fibonacci, beserta peringatan yang menyusul persis sesudahnya: *prosedur rekursif yang lurus-lurus saja itu lambat*. Mengapa, dan selambat apa, adalah urusan empat subbab berikutnya.

Dalam lambang, dan dengan memperkenalkan tiap lambang sambil jalan: kita menulis F_n (dibaca "F sub n") untuk bilangan Fibonacci ke-n, dengan n sebuah bilangan bulat 0,1,2,3, … yang disebut **indeks**. Definisinya adalah Persamaan (7.2):

$$F_0 = 0, \qquad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2} \text{ untuk } n \geq 2. \tag{7.2}$$

Dua baris pertama adalah **kasus basis**: keduanya menyatakan terus terang berapa F_0 dan F_1, tanpa perhitungan apa pun. Baris ketiga adalah **kasus rekursif**: ia mengatakan bahwa mulai dari bilangan ketiga dan seterusnya, tiap bilangan adalah jumlah dua bilangan sebelumnya.

Mari kita hasilkan barisannya dengan tangan, satu langkah satu baris, supaya tidak ada yang diterima begitu saja:

- $F_0 = 0$ (diberikan)
- $F_1 = 1$ (diberikan)
- $F_2 = F_1 + F_0 = 1 + 0 = 1$
- $F_3 = F_2 + F_1 = 1 + 1 = 2$
- $F_4 = F_3 + F_2 = 2 + 1 = 3$
- $F_5 = F_4 + F_3 = 3 + 2 = 5$
- $F_6 = F_5 + F_4 = 5 + 3 = 8$
- $F_7 = F_6 + F_5 = 8 + 5 = 13$
- $F_8 = F_7 + F_6 = 13 + 8 = 21$
- $F_9 = F_8 + F_7 = 21 + 13 = 34$
- $F_{10} = F_9 + F_8 = 34 + 21 = 55$

yaitu barisan 0,1,1,2,3,5,8,13,21,34,55, … yang tercetak pada **Gambar 7.3**.

7.3.2 Rekursi yang lurus-lurus saja

Definisi itu berpindah menjadi kode dengan begitu langsungnya sampai **Pseudokode 7.1** nyaris merupakan penyalinan:

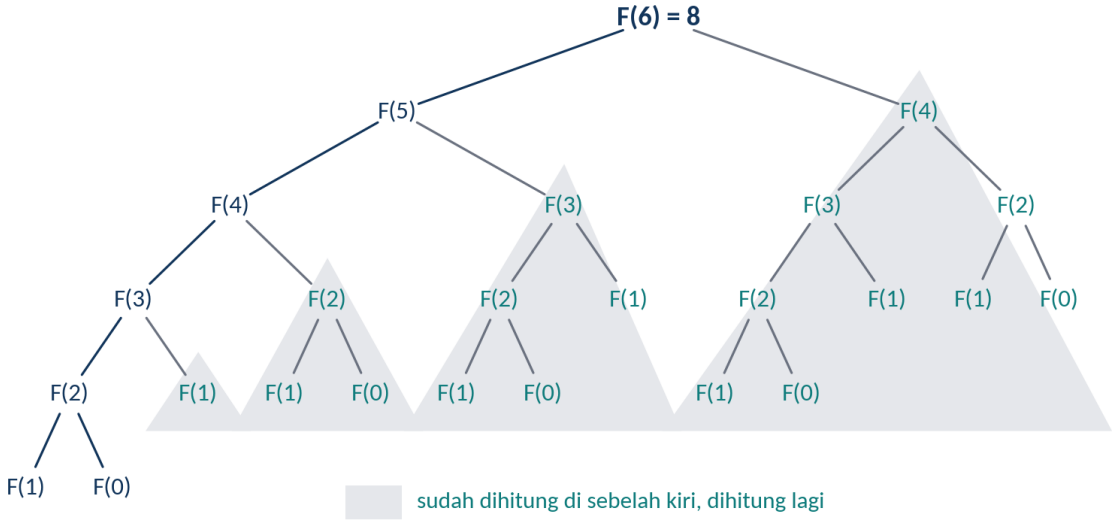
Pseudokode 7.1 Rekursi yang lurus-lurus saja untuk F_n, disalin langsung dari definisinya.

```
long long fib(int n) {
    if (n <= 1) return n;            // kasus basis: F(0)=0 dan F(1)=1
    return fib(n - 1) + fib(n - 2);  // kasus rekursif
}
```

Ia benar. Ia juga, sebagaimana diperingatkan **Gambar 7.3**, lambat — dan bukan "agak lambat", melainkan lambat dengan cara yang membuat sebuah program tidak terpakai. Untuk melihat alasannya, kita gambar pohon rekursinya.

7.3.3 Pohon rekursinya

Gambar 7.4 menggambar pohon rekursi untuk F_6.



Kita terus-menerus menghitung nilai yang sama berulang-ulang.

Gambar 7.4 Pohon rekursi `fib(6)` yang lurus-lurus saja. Segitiga kelabu menandai subpohon yang sudah pernah dihitung di suatu tempat di sebelah kirinya dan kini sedang dihitung ulang dari awal. Seluruh paruh kanan pohon itu — seluruh subpohon $F(4)$ — adalah duplikat subpohon yang sudah selesai dikerjakan di sebelah kiri.

Telusurilah. Akarnya meminta $F(6)$. Untuk menjawabnya, ia harus meminta $F(5)$ dan $F(4)$. Panggilan $F(5)$ harus meminta $F(4)$ dan $F(3)$ — dan di situlah duplikasi pertama muncul: $F(4)$ kini akan dihitung dua kali, sekali di bawah $F(5)$ dan sekali lagi langsung di bawah akar. Tak satu pun dari kedua panggilan itu tahu tentang yang lain. Masing-masing akan turun sampai ke daun.

Turun satu aras lagi dan keadaannya makin buruk. $F(3)$ muncul tiga kali. $F(2)$ muncul lima kali. $F(1)$ muncul delapan kali. Komentar yang tercetak di bawah gambarnya adalah seluruh diagnosisnya dalam satu kalimat: **kita terus-menerus menghitung nilai yang sama berulang-ulang.**

7.3.4 Menghitung panggilannya dengan tepat

"Berulang-ulang" adalah sebuah perasaan. Mari kita ubah menjadi sebuah bilangan.

Misalkan $C(n)$ adalah banyaknya seluruh pemanggilan `fib` yang dibuat ketika Anda memanggil `fib(n)` — termasuk panggilan terluar itu sendiri. Dari kodenya:

- Jika $n \leq 1$ fungsinya langsung kembali, tanpa membuat panggilan lagi. Jadi $C(0) = 1$ dan $C(1) = 1$.
- Selain itu panggilan itu membuat satu pemanggilan atas dirinya sendiri, ditambah semua yang dikerjakan `fib(n-1)`, ditambah semua yang dikerjakan `fib(n-2)`. Jadi $C(n) = C(n-1) + C(n-2) + 1$.

Itu sendiri adalah sebuah rekurensi, dan ia punya bentuk tertutup yang indah.

KLAIM. $C(n) = 2F_{n+1} - 1$ untuk setiap $n \geq 0$.

Bukti, dengan induksi kuat atas n.

Kasus basis. Untuk $n = 0$: $2F_1 - 1 = 2(1) - 1 = 1 = C(0)$. ✓ Untuk $n = 1$: $2F_2 - 1 = 2(1) - 1 = 1 = C(1)$. ✓

Langkah induktif. Ambil $n \geq 2$ dan andaikan klaimnya berlaku untuk semua indeks yang lebih kecil. Maka

$$C(n) = C(n-1) + C(n-2) + 1 = (2F_n - 1) + (2F_{n-1} - 1) + 1.$$

Kumpulkan sukunya: ketiga tetapan $-1, -1, +1$ berjumlah -1, dan kedua suku Fibonacci berlipat dua itu dapat difaktorkan:

$$C(n) = 2(F_n + F_{n-1}) - 1 = 2F_{n+1} - 1,$$

dengan langkah terakhir hanyalah rekurensi Fibonacci yang dibaca maju. ■

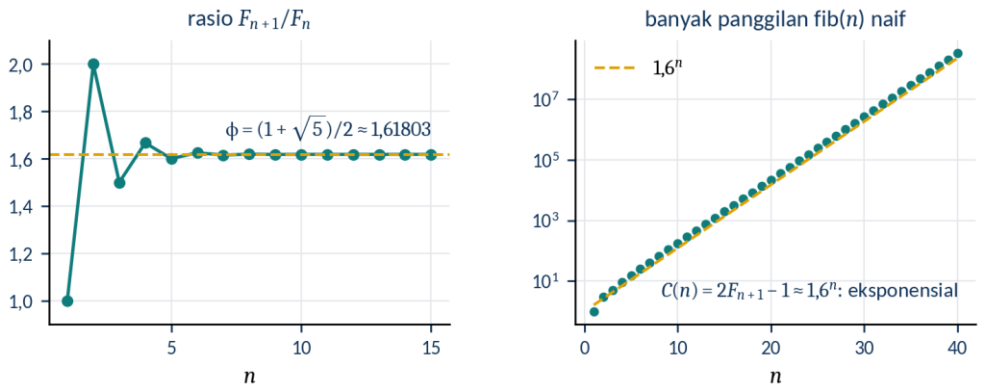
Dua hitungan pendamping jatuh dari gambar yang sama, dan kita akan memerlukan keduanya:

- **Daun** pohon itu persis adalah panggilan-panggilan dengan $n \leq 1$. Misalkan $L(n)$ adalah banyaknya. Maka $L(0) = L(1) = 1$ dan $L(n) = L(n-1) + L(n-2)$ — rekurensi Fibonacci itu sendiri, dengan indeks yang digeser — sehingga $L(n) = F_{n+1}$.
- **Simpul dalam** adalah panggilan-panggilan yang melakukan sebuah penjumlahan. Banyaknya $C(n) - L(n) = (2F_{n+1} - 1) - F_{n+1} = F_{n+1} - 1$. Itulah jawaban atas pertanyaan "ada berapa penjumlahan?".

Untuk $n = 6$ hitungan-hitungan ini memberi $C(6) = 2F_7 - 1 = 2(13) - 1 = 25$ panggilan, $L(6) = F_7 = 13$ daun, dan $F_7 - 1 = 12$ penjumlahan. Hitunglah simpul pada **Gambar 7.4** kalau Anda mau; ada dua puluh lima.

7.3.5 Mengapa biayanya $\Theta(\phi^n)$

Kita sekarang punya $C(n) = 2F_{n+1} - 1$, sehingga waktu jalan prosedur naif itu adalah $\Theta(F_n)$. Itu baru menolong kalau kita tahu seberapa cepat F_n sendiri tumbuh. **Gambar 7.5** memberikan jawabannya secara garis besar.



Gambar 7.5 Argumennya secara garis besar. Kiri: bilangan Fibonacci yang berurutan mendekati rasio emas $\phi \approx 1.61803$, sehingga $F_n \approx 1.6^n$. Kanan: banyaknya panggilan $2F_{n+1} - 1$ prosedur naif, pada skala logaritmik, sejajar dengan 1.6^n — pohonnya punya kira-kira 1.6^n daun, dan waktu jalannya eksponensial.

Mari kita turunkan itu dengan benar, bukan sekadar menerimanya. Perkenalkan lambang ϕ (huruf Yunani *phi*), yang didefinisikan sebagai

$$\phi = \frac{1+\sqrt{5}}{2} = 1.6180339887\ldots$$

dan pasangannya ψ (*psi*), yang didefinisikan sebagai

$$\psi = \frac{1-\sqrt{5}}{2} = -0.6180339887\ldots$$

Keduanya adalah akar persamaan kuadrat $x^2 = x + 1$; Anda dapat memeriksanya dengan substitusi. Fakta tunggal itulah yang mengikat keduanya pada Fibonacci, sebab rekurensi Fibonacci $F_n = F_{n-1} + F_{n-2}$ berbentuk persis sama: jika Anda mencoba solusi berbentuk $F_n = x^n$, Anda memperoleh $x^n = x^{n-1} + x^{n-2}$, dan membagi seluruhnya dengan x^{n-2} memberi $x^2 = x + 1$.

Karena ϕ^n maupun ψ^n sama-sama menyelesaikan rekurensinya, maka kombinasi apa pun $A\phi^n + B\psi^n$ juga menyelesaikannya. Memilih A dan B agar kasus basisnya keluar dengan benar ($F_0 = 0$, $F_1 = 1$) memberi $A = 1/\sqrt{5}$ dan $B = -1/\sqrt{5}$, dan karenanya lahirlah **rumus Binet**, Persamaan (7.3):

$$F_n = \frac{\phi^n - \psi^n}{\sqrt{5}}. \tag{7.3}$$

Sekarang pengamatan kunci tentang Persamaan (7.3). $|\psi| = 0.618\ldots < 1$, sehingga ψ^n menyusut menuju nol seiring n membesar; bahkan $\left|\psi^n/\sqrt{5}\right| < 1/2$ untuk setiap $n \geq 0$. Karena itu

$$F_n = \frac{\phi^n}{\sqrt{5}} + \varepsilon_n, \qquad |\varepsilon_n| < 1/2,$$

yang mengatakan bahwa F_n adalah bilangan bulat terdekat dengan $\phi^n/\sqrt{5}$. Dalam bahasa asimtotik, $F_n = \Theta(\phi^n)$. Knuth [1, §1.2.8] mengembangkan bilangan Fibonacci dan rasio emas panjang lebar, termasuk sejarah rumus Binet.

Menyatukan potongan-potongannya memberikan Persamaan (7.4):

$$C(n) = 2F_{n+1} - 1 = \Theta(\phi^n) = \Theta(1.618^n). \tag{7.4}$$

> **VONISNYA.** Rekursi yang lurus-lurus saja itu berjalan dalam waktu $\Theta(\phi^n)$ — eksponensial. Menambah satu pada n melipatkan kerjanya kira-kira 1.618 kali; menambah dua melipatkannya $\phi^2 = 2.618$ kali. Ini bukan persoalan faktor tetapan yang bisa diperbaiki komputer yang lebih cepat. Komputer seribu kali lebih cepat hanya membelikan Anda kira-kira empat belas nilai n tambahan.

7.3.6 Berapa banyak kerja yang sesungguhnya terulang

Inilah bagian yang seharusnya membuat Anda marah atas nama algoritmanya. Untuk menghitung F_n hanya ada $n + 1$ pertanyaan berbeda yang perlu dijawab: berapa F_0, berapa F_1, …, berapa F_n. Hanya itu. Namun prosedur naif itu mengajukan $2F_{n+1} - 1$ pertanyaan.

Berapa kali tiap nilai dihitung ulang? Dengan membaca **Gambar 7.4**, dan dikonfirmasi oleh program §7.3.7, banyaknya panggilan ke $F(k)$ di dalam `fib(n)` diberikan oleh Persamaan (7.5):

$$\text{panggilan ke } F(k) \;=\; F_{n-k+1} \quad (k \geq 1), \qquad \text{panggilan ke } F(0) \;=\; F_{n-1}. \tag{7.5}$$

Untuk $n = 6$ rumus itu meramalkan $F(6)$ sekali, $F(5)$ sekali, $F(4)$ dua kali, $F(3)$ tiga kali, $F(2)$ lima kali, $F(1)$ delapan kali, dan $F(0)$ lima kali — perhatikan bahwa banyaknya kemunculan itu sendiri adalah bilangan-bilangan Fibonacci, dibaca mundur. Dua puluh lima panggilan untuk menjawab tujuh pertanyaan berbeda.

7.3.7 Mengukurnya

Bilangan di atas kertas adalah satu hal; program yang tidak mau selesai adalah hal lain. **Kode 7.1** memasang alat ukur pada rekursi naif itu: ia mencacah setiap panggilan, mencatat berapa kali tiap $F(k)$ diminta, dan mengukur waktu seluruhnya.

Kode 7.1 Rekursi Fibonacci naif, lengkap dengan alat ukurnya. Simpan sebagai `fib_naive.cpp`, kompilasi dengan `g++ -std=c++17 -O2 -o fib_naive fib_naive.cpp`, lalu jalankan `./fib_naive`.

```
// fib_naive.cpp - rekursi yang lurus-lurus saja, dengan alat ukur terpasang.
// Mencacah tiap panggilan, mencacah seberapa sering tiap F(k) dihitung ulang, dan mengukur
waktunya.
#include <iostream>
#include <iomanip>
#include <vector>
#include <chrono>
using namespace std;
using namespace std::chrono;

long long calls = 0;                // jumlah seluruh pemanggilan fib()
vector<long long> perLevel;         // perLevel[k] = berapa kali F(k) dipanggil

long long fib(int n) {
    ++calls;
    if (n < (int)perLevel.size()) ++perLevel[n];
    if (n <= 1) return n;           // kasus basis F(0)=0, F(1)=1
    return fib(n - 1) + fib(n - 2);
}

int main() {
    // Bagian 1: profil panggilan fib(6), pohon yang digambar di Subbab 7.3.3.
    perLevel.assign(7, 0);
    calls = 0;
    long long v = fib(6);
    cout << "fib(6) = " << v << ", total calls = " << calls << "\n";
    cout << "calls to F(k), k = 0..6:";
    for (int k = 0; k <= 6; ++k) cout << " F(" << k << ")=" << perLevel[k];
    cout << "\n\n";

    // Bagian 2: pertumbuhan cacah panggilan dan waktu jam dindingnya.
    cout << left << setw(5) << "n" << setw(14) << "F(n)"
         << setw(14) << "calls" << setw(12) << "ms"
         << setw(10) << "calls/prev" << "\n";
    long long prevCalls = 0;
    for (int n = 20; n <= 42; n += 2) {
        perLevel.assign(0, 0);
        calls = 0;
        auto t0 = steady_clock::now();
        long long f = fib(n);
        auto t1 = steady_clock::now();
        double ms = duration<double, milli>(t1 - t0).count();
        cout << left << setw(5) << n << setw(14) << f << setw(14) << calls
             << setw(12) << fixed << setprecision(2) << ms;
        if (prevCalls) cout << setw(10) << setprecision(3) << (double)calls / prevCalls;
        cout << "\n";
        prevCalls = calls;
```

Kode 7.1 (Lanjutan) *Rekursi Fibonacci naif, lengkap dengan alat ukurnya. Simpan sebagai fib_naive.cpp, kompilasi dengan g++ -std=c++17 -O2 -o fib_naive fib_naive.cpp, lalu jalankan ./fib_naive.*

```
    }
    return 0;
}
```

Dikompilasi dan dijalankan, ia mencetak:

```
fib(6) = 8, total calls = 25
calls to F(k), k = 0..6: F(0)=5 F(1)=8 F(2)=5 F(3)=3 F(4)=2 F(5)=1 F(6)=1

n    F(n)          calls         ms          calls/prev
20   6765          21891         0.05
22   17711         57313         0.12        2.618
24   46368         150049        0.34        2.618
26   121393        392835        0.86        2.618
28   317811        1028457       2.21        2.618
30   832040        2692537       5.83        2.618
32   2178309       7049155       16.27       2.618
34   5702887       18454929      40.88       2.618
36   14930352      48315633      108.97      2.618
38   39088169      126491971     280.85      2.618
40   102334155     331160281     735.49      2.618
42   267914296     866988873     1943.29     2.618
```

Setiap bilangan yang kita ramalkan dengan tangan ada di sana. Profil panggilan `fib(6)` persis 25 panggilan yang terbagi sebagai 1,1,2,3,5,8,5. Dan lihatlah kolom terakhir: nisbah cacah panggilan yang berurutan, berjarak dua langkah, adalah 2.618 sampai tiga angka di belakang koma — yaitu $\phi^2 = 2.6180339\ldots$, persis seperti yang dituntut teorinya.

Akibat praktisnya ada di kolom waktu. Menghitung F_{42} — sebuah bilangan yang muat dengan nyaman dalam bilangan bulat 32 bit, sebuah bilangan yang bisa Anda tuliskan di selembar tisu dalam lima detik — memakan hampir dua detik dan delapan ratus enam puluh enam juta panggilan fungsi. Diekstrapolasi dengan faktor ϕ per langkah, F_{90} akan memakan waktu kira-kira $1.94 \times \phi^{48} \approx 2 \times 10^{10}$ detik bagi program ini — kurang lebih tujuh abad.

> **JANGAN SALAH BACA.** Persoalannya bukan bahwa algoritmanya rekursif, dan bukan pula bahwa bilangan Fibonacci itu sulit. Persoalannya adalah bahwa pohon rekursinya memuat $2F_{n+1} - 1$ simpul yang hanya membawa $n + 1$ label berbeda. Segala sesuatu yang lain dalam bab ini adalah teknik untuk meruntuhkan simpul-simpul duplikat itu menjadi satu.

7.4 Empat Perbaikan untuk Fibonacci

Resepnya ada pada **Gambar 7.6**, dan di dalamnya terkandung, dimampatkan menjadi satu gambar, seluruh gagasan *dynamic programming*.

Gambar 7.6 Perbaikannya, secara ringkas: hitunglah F_n dalam waktu linear dengan *mengingat solusi submasalah yang sudah terselesaikan* — *dynamic programming* — membangun solusinya secara *bottom-up*, menukar ruang dengan waktu. Pasangan yang disorot itulah pengamatan yang akan menjadi §7.4.3: hanya dua nilai yang perlu diingat pada saat mana pun.

Tekniknya punya nama dan punya penemu: Richard Bellman menciptakan istilah *dynamic programming* pada tahun 1950-an [2], memilih kata "programming" dalam maknanya yang lebih tua, yaitu *menyusun rencana urutan tindakan*, bukan dalam makna menulis kode.

Perhatikan frasa **tukar ruang dengan waktu**. Itulah tawar-menawar yang sedang disepakati. Kita akan membelanjakan memori yang sebelumnya tidak kita belanjakan, dan sebagai gantinya kita berhenti membayar kerja yang sudah kita kerjakan. Dalam kasus Fibonacci tawar-menawar itu luar biasa menguntungkan: kita membelanjakan $\Theta(n)$ kata memori (dan kemudian, dengan sedikit pemikiran, hanya dua kata), dan kita beranjak dari waktu $\Theta(\phi^n)$ ke $\Theta(n)$.

Kita akan mengerjakan perbaikannya dengan empat cara berbeda. Dua yang pertama adalah dua bentuk kanonik *dynamic programming*, dan segala sesuatu selanjutnya dalam bab ini adalah salah satu di antara keduanya. Yang ketiga adalah optimasi atas yang kedua. Yang keempat adalah bonus yang meninggalkan *dynamic programming* sama sekali, dan disertakan karena ia indah, serta karena ia menunjukkan bahwa jawaban berwaktu linear pun masih dapat dikalahkan.

7.4.1 Perbaikan 1: memoization — ingat apa yang sudah pernah Anda jawab

> **DEFINISI.** *Memoization* (memoisasi) adalah teknik menyimpan tabel jawaban yang sudah terhitung, memeriksanya di awal setiap panggilan rekursif, dan mengembalikan jawaban tersimpan alih-alih berekursi bila jawabannya ditemukan. *Struktur* rekursinya tidak berubah; yang dihilangkan hanyalah *pengulangannya*. (Katanya adalah "memoization", dari *memo*, catatan untuk diri sendiri — bukan "memorization", menghafal.)

Perubahan pada kodenya hanya tiga baris, seperti tampak pada **Pseudokode 7.2**:

Pseudokode 7.2 Fibonacci dengan *memoization*: rekursi yang sama, yang memeriksa dan mengisi sebuah tabel jawaban.

```
vector<long long> memo;                    // memo[k] = F(k), atau -1 bila belum diketahui
long long fibMemo(int n) {
    if (memo[n] != -1) return memo[n];      // <-- sudah terselesaikan: jangan berekursi
    long long r = (n <= 1) ? n : fibMemo(n - 1) + fibMemo(n - 2);
    memo[n] = r;                            // <-- catat sebelum kembali
    return r;
}
```

Mengapa ini linear? Karena sebuah argumen yang harus Anda latih sampai keluar dengan sendirinya.

Setiap entri tabel `memo[k]` ditulis persis sekali, yaitu pertama kali $F(k)$ sungguh-sungguh dihitung. Setiap permintaan $F(k)$ berikutnya mengambil cabang pertama dan kembali dalam waktu tetap. Ada $n+1$ entri. Masing-masing dari $n+1$ perhitungan sungguhan itu mengerjakan kerja $O(1)$ miliknya sendiri (satu penjumlahan, satu penyimpanan), sebab dua panggilan rekursif yang ia buat *dibebankan pada entri masing-masing*, bukan pada entri ini. Karena itu totalnya adalah $\Theta(n)$ perhitungan sungguhan ditambah $\Theta(n)$ penelusuran tabel, yaitu waktu $\Theta(n)$ dan ruang $\Theta(n)$.

> **TRIK PEMBUKUANNYA.** Ketika menganalisis rekursi ber-*memoization*, jangan coba mengikuti panggilannya. Sebaliknya, hitunglah entri tabelnya lalu kalikan dengan biaya mengisi satu entri, dengan *mengabaikan* panggilan rekursif di dalamnya (panggilan itu dibayar oleh entrinya sendiri). Di sini: $n+1$ entri $\times$ $O(1)$ masing-masing $=\Theta(n)$. Aturan ini berlaku untuk setiap algoritma ber-*memoization* dalam bab ini.

Pada pohon rekursi **Gambar 7.4**, *memoization* berarti melenyapkan setiap segitiga kelabu: pertama kali sebuah subpohon dimasuki ia dihitung, dan setiap kedatangan berikutnya pada label yang sama berhenti di akar subpohon itu dan membaca jawabannya.

7.4.2 Perbaikan 2: bottom-up — bangun jawaban kecil lebih dulu

Memoization adalah rekursi yang sudah belajar untuk tidak mengulang dirinya. Metode **bottom-up** (bawah-atas) meninggalkan rekursi sama sekali. Ia menyadari bahwa jika kita toh akan menghitung seluruh $F_0, F_1, \ldots, F_n$, sebaiknya kita menghitungnya dalam urutan $F_0, F_1, F_2, \ldots$, dan dalam urutan itu setiap nilai yang kita butuhkan sudah duduk di dalam tabel ketika kita mengulurkan tangan untuk mengambilnya. **Pseudokode 7.3** mengerjakan persis hal itu.

Pseudokode 7.3 Fibonacci secara *bottom-up*: tabelnya diisi dari indeks terkecil ke atas.

```
long long fibTable(int n) {
    vector<long long> F(n + 1);
    F[0] = 0; if (n >= 1) F[1] = 1;
    for (int i = 2; i <= n; ++i) F[i] = F[i-1] + F[i-2];
    return F[n];
}
```

Satu gelung, $n - 1$ iterasi, satu penjumlahan tiap iterasi: waktu $\Theta(n)$, ruang $\Theta(n)$, tanpa rekursi, tanpa tumpukan.

Urutannyalah seluruh seninya. Kita harus menata gelungnya sedemikian rupa sehingga ketika algoritmanya menghitung sebuah entri, segala sesuatu yang menjadi sandaran entri itu sudah terhitung. Di sini hal itu mudah — F_i bersandar pada F_{i-1} dan F_{i-2}, keduanya berindeks lebih kecil, jadi menaikkan i sudah memadai. Di §7.6 urutan yang benar akan jauh kurang kentara, dan menentukannya dengan benar akan menjadi inti algoritmanya.

7.4.3 Perbaikan 3: ruang $\Theta(1)$

Lihat sekali lagi dua sel yang disorot pada **Gambar 7.6**: *dalam kasus ini, hanya dua nilai yang perlu diingat pada saat mana pun.*

Itulah penghematan besar kedua dari *dynamic programming*, dan ia pantas diberi nama. **Pseudokode 7.4** mempraktikkannya.

> **PENGHEMATAN RUANG.** Jika rekurensinya hanya pernah menjangkau mundur sejauh jarak terbatas, Anda tidak memerlukan seluruh tabelnya — hanya jendela geser berisi entri-entri yang masih terjangkau. Segala yang lebih tua boleh dilupakan. F_i menjangkau mundur persis dua tempat, jadi jendela selebar dua sudah cukup.

Pseudokode 7.4 Fibonacci dalam ruang tetap: jendela geser berisi dua nilai menggantikan tabelnya.

```
long long fibIter(int n) {
    if (n <= 1) return n;
    long long prev = 0, cur = 1;            // prev = F(i-2), cur = F(i-1)
    for (int i = 2; i <= n; ++i) {
        long long next = prev + cur;         // F(i)
        prev = cur;                          // geser jendelanya satu langkah ke kanan
        cur  = next;
    }
    return cur;
}
```

Waktu $\Theta(n)$, ruang $\Theta(1)$. Telusurilah untuk $n = 6$, satu baris satu iterasi, supaya gesernya terlihat:

Tabel 7.1 Jendela geser `fibIter` sewaktu menghitung F_6. Sesudah iterasi terakhir `cur` memuat $8 = F_6$, dan tidak ada satu larik pun yang pernah dialokasikan.

i	`prev` sebelum	`cur` sebelum	`next` = prev+cur	`prev` sesudah	`cur` sesudah
2	0	1	1	1	1
3	1	1	2	1	2
4	1	2	3	2	3
5	2	3	5	3	5
6	3	5	8	5	8

Trik yang sama, diterapkan pada tabel dua dimensi alih-alih satu dimensi, persis itulah yang memberi versi beruang $\Theta(\min(m,n))$ dari *longest common subsequence* di §7.9.10. Pelajarilah di sini, tempat ia masih mudah.

7.4.4 Perbaikan 4 (bonus): pangkat matriks, $O(\lg n)$

Ketiga perbaikan di atas semuanya $\Theta(n)$, dan untuk waktu yang lama orang menganggap sampai di situ saja ceritanya: Anda tidak bisa menghitung suku ke-n sebuah barisan dalam kurang dari n langkah. Ternyata bisa.

Mulailah dari pengamatan bahwa satu langkah rekurensi Fibonacci adalah sebuah pemetaan linear. Tulis keadaannya sebagai kolom berisi dua nilai berurutan. Maka

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}.$$

Periksa hitungan klaim itu: baris atas hasil kalinya adalah $1 \cdot F_n + 1 \cdot F_{n-1} = F_{n+1}$ ✓, dan baris bawahnya adalah $1 \cdot F_n + 0 \cdot F_{n-1} = F_n$ ✓.

Sebut matriks itu $Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. Menerapkan pemetaannya n kali dari keadaan awal $(F_1, F_0)^\top = (1,0)^\top$ memberi

$$Q^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}, \tag{7.6}$$

yaitu Persamaan (7.6), sebuah fakta yang dibuktikan dengan induksi yang mudah ($n = 1$ adalah Q itu sendiri; langkahnya adalah satu perkalian matriks dan tiga kali pemakaian $F_k = F_{k-1} + F_{k-2}$). Jadi F_n adalah entri kanan atas Q^n.

Sekarang, secepat apa kita bisa memangkatkan sebuah matriks sampai pangkat n? Bukan dengan mengalikan n kali — itu $\Theta(n)$ lagi. Melainkan dengan **binary exponentiation** (eksponensiasi biner), yang juga disebut *repeated squaring* (pengkuadratan berulang):

$$Q^n = \begin{cases} I & n = 0, \\ (Q^{n/2})^2 & n \text{ genap}, \\ Q \cdot Q^{n-1} & n \text{ ganjil}. \end{cases}$$

Setiap pemaruhan n berbiaya satu perkalian matriks 2×2 untuk pengkuadratannya, ditambah satu lagi bila n ganjil — paling banyak dua, dan masing-masing terdiri atas delapan perkalian skalar dan empat penjumlahan, sebuah tetapan. Banyaknya pemaruhan sebelum n mencapai nol adalah $\lfloor \lg n \rfloor + 1$. Karena itu seluruhnya $O(\lg n)$ operasi aritmetika.

CATATAN KECILNYA. $O(\lg n)$ menghitung *operasi aritmetika*, dengan memperlakukan tiap penjumlahan dan perkalian sebagai berbiaya satu satuan — model RAM dari Bab 2. F_n punya kira-kira $0.209n$ digit desimal (kira-kira $0.694n$ bit), jadi untuk n yang sangat besar bilangannya sendiri berhenti muat dalam satu kata mesin dan tiap operasi berhenti berwaktu tetap. Program di bawah menghindari hal ini dengan bekerja dalam modulo 1 000 000 007, sebuah konvensi lazim yang menjaga setiap nilai tetap berada di dalam bilangan bulat 64 bit. Jika Anda memerlukan nilai eksak $F_{1\,000\,000}$, keempat metode itu harus memakai bilangan bulat besar, dan perbandingannya pun berubah.

7.4.5 Keempatnya, diukur

Kode 7.2 mengimplementasikan keempat perbaikan itu dan mengukur waktunya pada masukan yang sama.

Kode 7.2 Fibonacci ber-*memoization*, *bottom-up*, beruang tetap, dan berpangkat matriks, keempatnya diadu waktunya satu sama lain. Kompilasi dengan `g++ -std=c++17 -O2 -o fib_fast fib_fast.cpp`.

```
// fib_fast.cpp - empat cara menghitung F(n) yang tidak eksponensial.
// Nilainya diambil modulo 1000000007 supaya tetap berada di dalam bilangan bulat 64 bit
// untuk n besar; waktu jalannya tidak terpengaruh oleh pilihan itu.
#include <iostream>
#include <iomanip>
#include <vector>
#include <array>
#include <chrono>
using namespace std;
using namespace std::chrono;

const unsigned long long MOD = 1000000007ULL;
const int MEMO_LIMIT = 250000;   // terukur: rekursi yang lebih dalam meluapkan tumpukan 8
MiB

// (1) Top-down dengan memoization.
vector<long long> memo;
long long fibMemo(int n) {
    if (memo[n] != -1) return memo[n];           // sudah terselesaikan: telusuri saja,
jangan berekursi
    long long r = (n <= 1) ? n : (fibMemo(n - 1) + fibMemo(n - 2)) % MOD;
    memo[n] = r;                                 // catat jawabannya sebelum
mengembalikannya
    return r;
}

// (2) Bottom-up atas tabel penuh.
long long fibTable(int n) {
    vector<long long> F(n + 1);
    F[0] = 0; if (n >= 1) F[1] = 1;
    for (int i = 2; i <= n; ++i) F[i] = (F[i-1] + F[i-2]) % MOD;
    return F[n];
}

// (3) Bottom-up yang hanya menyimpan dua nilai: ruang Theta(1).
long long fibIter(int n) {
    if (n <= 1) return n;
    long long prev = 0, cur = 1;                 // prev = F(i-2), cur = F(i-1)
    for (int i = 2; i <= n; ++i) {
        long long next = (prev + cur) % MOD;
        prev = cur; cur = next;
    }
    return cur;
}
```

Kode 7.2 (Lanjutan) *Fibonacci ber-memoization, bottom-up, beruang tetap, dan berpangkat matriks, keempatnya diadu waktunya satu sama lain. Kompilasi dengan g++ -std=c++17 -O2 -o fib_fast fib_fast.cpp.*

```
// (4) Pangkat matriks: [[1,1],[1,0]]^n memuat F(n) pada posisi (0,1).
typedef array<array<unsigned long long,2>,2> Mat;
Mat mul(const Mat& A, const Mat& B) {
    Mat C{};
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j) {
            unsigned long long s = 0;
            for (int k = 0; k < 2; ++k) s = (s + A[i][k] * B[k][j]) % MOD;
            C[i][j] = s;
        }
    return C;
}
long long fibMatrix(int n) {
    Mat R{{{{1,0}},{{0,1}}}};                    // identitas
    Mat B{{{{1,1}},{{1,0}}}};
    int e = n;
    while (e > 0) {                              // eksponensiasi biner: kira-kira lg n
pengkuadratan
        if (e & 1) R = mul(R, B);
        B = mul(B, B);
        e >>= 1;
    }
    return (long long)R[0][1];
}

template <class F> double timeIt(F f, int n, int reps, long long& out) {
    auto t0 = steady_clock::now();
    for (int r = 0; r < reps; ++r) out = f(n);
    auto t1 = steady_clock::now();
    return duration<double, micro>(t1 - t0).count() / reps;
}

int main() {
    cout << "F(n) mod 1000000007, and microseconds per call\n\n";
    cout << left << setw(10) << "n" << setw(14) << "F(n)"
         << setw(13) << "memo us" << setw(13) << "table us"
         << setw(13) << "iter us" << setw(13) << "matrix us" << "\n";
    for (int n : {40, 1000, 10000, 100000, 250000, 1000000}) {
        long long a = -1, b, c, d;
        double tm = -1;
        if (n <= MEMO_LIMIT) { memo.assign(n + 1, -1);
            tm = timeIt([](int k){ return fibMemo(k); }, n, 1, a); }
        double tt = timeIt(fibTable,  n, 5, b);
        double ti = timeIt(fibIter,   n, 5, c);
        double tx = timeIt(fibMatrix, n, 5, d);
        if (!(b == c && c == d) || (a != -1 && a != b)) { cout << "MISMATCH\n"; return 1; }
        cout << left << setw(10) << n << setw(14) << b;
```

Kode 7.2 (Lanjutan) *Fibonacci ber-memoization, bottom-up, beruang tetap, dan berpangkat matriks, keempatnya diadu waktunya satu sama lain. Kompilasi dengan g++ -std=c++17 -O2 -o fib_fast fib_fast.cpp.*

```
        if (tm >= 0) cout << setw(13) << fixed << setprecision(2) << tm;
        else         cout << setw(13) << "overflow";
        cout << setw(13) << fixed << setprecision(2) << tt
             << setw(13) << ti << setw(13) << tx << "\n";
    }
    cout << "\nAll methods that ran agreed on every n above.\n";
    cout << "F(0..10) exactly: 0";
    long long p = 0, q = 1;
    for (int i = 1; i <= 10; ++i) { cout << ", " << q; long long t = p + q; p = q; q = t; }
    cout << "\n";
    return 0;
}
```

Keluarannya:

```
F(n) mod 1000000007, and microseconds per call

n         F(n)          memo us      table us     iter us      matrix us
40        102334155     0.83         0.53         0.10         0.03
1000      517691607     18.34        4.04         3.19         0.05
10000     271496360     177.21       45.30        32.21        0.07
100000    911435502     2140.96      523.36       325.41       0.09
250000    608672132     4561.73      1259.70      821.79       0.11
1000000   918091266     overflow     5041.49      3378.51      0.13

All methods that ran agreed on every n above.
F(0..10) exactly: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

Ada enam hal dalam tabel itu yang pantas dikomentari, dan bersama-sama keenamnya adalah isi praktis bab ini.

Pertama, bandingkan dengan **Kode 7.1**. Rekursi naif memerlukan 735 490 mikrodetik untuk $n = 40$. Versi ber-*memoization* memerlukan 0.83. Itu percepatan kira-kira 886 000 kali, hanya dengan menambahkan dua baris kode.

Kedua, ketiga metode $\Theta(n)$ itu sungguh-sungguh linear. Kalikan n dengan sepuluh, dari 10 000 ke 100 000, dan `table` beranjak dari 45.30 ke 523.36 mikrodetik — faktor 11.6, dekat dengan faktor 10 yang diramalkan kelinearan, dengan kelebihannya dijelaskan oleh lalu lintas memori larik yang sepuluh kali lebih besar.

Ketiga, *memoization* secara konsisten paling lambat di antara ketiga yang linear — 2141 mikrodetik melawan 523 milik tabel, empat kali lebih lambat pada $n = 100\,000$. Kerja yang dilakukan sama. Bedanya adalah biaya tambahan: satu panggilan fungsi, satu bingkai tumpukan, satu pembandingan terhadap *sentinel* dan satu pengembalian, per entri, alih-alih satu iterasi gelung per entri.

Keempat, iterasi beruang tetap mengalahkan tabel penuh (325 melawan 523 mikrodetik pada $n = 100\,000$) meskipun keduanya $\Theta(n)$ dan mengerjakan penjumlahan yang sama. Ia tak pernah mengalokasikan apa pun, tak pernah menyentuh memori di luar tiga register, dan tak pernah mengalami *cache miss*. Asimtotika mengatakan keduanya setara; mesin mengatakan sebaliknya. Kedua pernyataan itu benar.

Kelima, kolom *memoization* berbunyi `overflow` pada $n = 1\,000\,000$, dan itu bukan salah ketik. Rekursinya mencapai kedalaman n. Dengan tumpukan bawaan 8 MiB pada mesin yang dipakai di sini, ia selamat pada $n = 250\,000$ dan hancur dengan *segmentation fault* pada $n = 300\,000$. Rekursi *top-down* punya biaya sumber daya tersembunyi yang tidak dimiliki iterasi *bottom-up*.

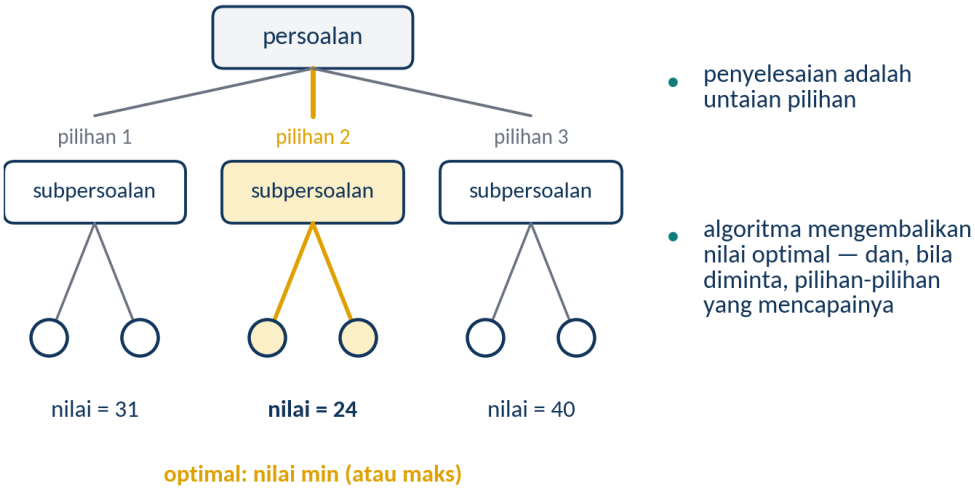
Keenam, dan paling mencolok: metode matriks memakan 0.13 mikrodetik untuk $n = 1\,000\,000$ — kira-kira dua puluh enam ribu kali lebih cepat daripada versi iteratif. Kolomnya nyaris tidak bergerak sama sekali sementara n tumbuh empat orde besaran, dari 0.03 ke 0.13 mikrodetik, sebab kerjanya sebanding dengan $\lg n$, dan lg dari satu juta hanyalah dua puluh.

Tabel 7.2 Lima cara menghitung F_n dari bab ini, lengkap dengan biaya masing-masing. Baris pertama adalah yang harus dihindari; baris terakhir adalah yang menang ketika n besar.

Metode	Waktu	Ruang	Catatan
Rekursi naif	$\Theta(\phi^n)$	tumpukan $\Theta(n)$	Tak terpakai melewati $n \approx 45$
Memoization (*top-down*)	$\Theta(n)$	$\Theta(n)$ + tumpukan $\Theta(n)$	Penulisan ulang termudah; terbatas tumpukan
Tabel *bottom-up*	$\Theta(n)$	$\Theta(n)$	Tanpa rekursi, tanpa risiko tumpukan
Jendela geser	$\Theta(n)$	$\Theta(1)$	Tak bisa memulihkan F_i antara
Pangkat matriks	$\Theta(\lg n)$	$\Theta(1)$	Perlu wawasan aljabar linear

7.5 Persoalan Optimasi

Fibonacci hanyalah pemanasan. Ia memang punya submasalah yang berulang, tetapi ia bukan persoalan *optimasi*: tidak ada yang bisa dipilih, sebab F_n hanya punya satu nilai. Persoalan-persoalan yang menjadi alasan ditemukannya *dynamic programming* semuanya melibatkan pemilihan. **Gambar 7.7** membingkainya.



Gambar 7.7 Persoalan optimasi secara garis besar: ada banyak solusi, satu di antaranya optimal, sebuah solusi adalah *rangkaian pilihan*, dan algoritmanya menghitung nilai optimal serta — bila kita memintanya — solusi optimalnya sendiri.

7.5.1 Apa yang membuat sebuah persoalan menjadi persoalan optimasi

> **DEFINISI. Persoalan optimasi** adalah persoalan yang tiap masukannya menerima banyak **solusi layak** (*feasible solution*), tiap solusi layak membawa sebuah **nilai** numerik (sebuah biaya, sebuah panjang, sebuah laba), dan tugasnya adalah menemukan solusi layak yang nilainya terkecil (**minimisasi**) atau terbesar (**maksimisasi**). Solusi semacam itu disebut **sebuah solusi optimal** — "sebuah", bukan "solusi optimalnya", sebab beberapa solusi berbeda bisa saja seri pada nilai optimal yang sama.

Ada tiga hal dalam kerangka itu yang pantas ditekankan.

Sebuah solusi memperlihatkan struktur. Ia bukan sebuah atom. Ia dibangun.

Ia terdiri atas rangkaian pilihan yang telah diambil. Jika kita bisa menyebutkan nama pilihan-pilihannya, dan jika tiap pilihan mereduksi persoalan menjadi satu atau lebih persoalan sejenis yang lebih kecil, maka solusinya adalah sebuah *lintasan* menembus ruang submasalah — dan lintasan adalah sesuatu yang dapat kita cari secara sistematis.

Algoritmanya menghitung nilai optimal ditambah, bila diperlukan, solusi optimalnya. Ini dua keluaran yang berbeda dan dua jumlah kerja yang berbeda. Nilainya adalah sebuah bilangan; solusinya adalah objek yang mencapai bilangan itu. *Dynamic programming* secara alami menghasilkan nilainya, dan menghasilkan solusinya hanya bila kita tambahan mencatat, pada tiap langkah, pilihan *yang mana* yang mencapai optimum. Catatan tambahan itulah tabel c pada §7.6.9 dan tabel b pada §7.9.8.

7.5.2 Optimal substructure, dinyatakan dengan tepat

Satu-satunya sifat yang membuat *dynamic programming* mungkin adalah **optimal substructure** (substruktur optimal). Sifat itu mudah dinyatakan dengan longgar dan mudah disalahpahami, jadi inilah ia dengan hati-hati.

> **DEFINISI.** Sebuah persoalan memperlihatkan **optimal substructure** bila solusi optimal persoalan itu memuat di dalamnya solusi optimal bagi submasalah-submasalah penyusunnya. Secara formal: jika S adalah solusi optimal bagi instans P, dan S terurai menjadi solusi $S_1, \ldots, S_r$ bagi submasalah $P_1, \ldots, P_r$, maka tiap S_i haruslah sendirinya merupakan solusi optimal bagi P_i.

Perhatikan dengan cermat apa yang diklaim dan apa yang tidak diklaim.

- Yang **diklaim** adalah bahwa jika Anda membongkar sebuah solusi optimal, potongan-potongannya optimal bagi submasalahnya masing-masing.
- Yang **tidak** diklaim adalah bahwa merekatkan potongan-potongan optimal menghasilkan keseluruhan yang optimal. Hal itu salah secara umum: potongan optimal bagi penguraian yang *keliru* akan merekat menjadi sampah. Inilah sebabnya algoritma-algoritma dalam bab ini mencoba *setiap* penguraian lalu mengambil yang terbaik.

Optimal substructure adalah izin yang melahirkan rekurensinya. Jika Anda tahu solusi optimalnya terbelah pada suatu titik k, dan Anda tahu kedua sisinya optimal bagi paruhnya masing-masing, maka nilai optimalnya adalah $\min_k\{\text{kiri-terbaik}(k) + \text{kanan-terbaik}(k) + \text{biaya membelah di } k\}$ — dan setiap suku di ruas kanan adalah instans lebih kecil dari pertanyaan yang sama.

7.5.3 Argumen potong-dan-tempel

Bagaimana Anda *membuktikan* sebuah persoalan punya *optimal substructure*? Hampir selalu dengan argumen tiga baris yang sama, yang dikenal dengan nama **cut and paste** (potong dan tempel).

> **BUKTI POTONG-DAN-TEMPEL.** Andaikan S adalah solusi optimal bagi P, yang terurai menjadi $S_1, \ldots, S_r$ bagi submasalah $P_1, \ldots, P_r$. Andaikan, demi kontradiksi, ada suatu S_i yang *tidak* optimal bagi P_i — artinya ada solusi yang lebih baik S_i' bagi P_i. **Potong** S_i keluar dari S lalu **tempel** S_i' di tempatnya. Hasilnya tetap solusi layak bagi P (inilah langkah yang harus Anda periksa), dan nilainya betul-betul lebih baik daripada S. Itu bertentangan dengan keoptimalan S. Karena itu setiap S_i memang sudah optimal sejak semula. ■

Langkah yang dicetak tebal itulah yang paling sering dilewati, dan melewatinya adalah awal lahirnya pembuktian yang keliru. Menempelkan potongan yang lebih baik harus meninggalkan Anda dengan sesuatu yang *sah*. Dalam perkalian deretan matriks hal itu terpenuhi, sebab pengelompokan tanda kurung mana pun atas $A_i \cdots A_k$ dapat duduk di dalam pengelompokan mana pun atas keseluruhannya. Pada persoalan yang potongan-potongannya saling berinteraksi — yang memakai paruh kiri lebih murah berarti melarang sesuatu yang Anda butuhkan di kanan — penempelannya tidak sah, argumennya runtuh, dan *dynamic programming* tidak berlaku.

7.5.4 Sebuah persoalan *tanpa* optimal substructure

Supaya definisinya menggigit, inilah sebuah persoalan yang tidak punya sifat itu.

Dalam graf berarah, **lintasan sederhana terpanjang** (*longest simple path*) dari u ke v adalah lintasan terpanjang yang tidak mengunjungi satu simpul pun dua kali. Apakah ia punya *optimal substructure*? Andaikan lintasan sederhana terpanjang dari u ke v melewati w. Apakah potongan dari u ke w lalu merupakan lintasan sederhana u-ke-w yang terpanjang? Graf empat simpul berikut menjawab tidak. Ambil simpul q, r, s, t yang tersusun sebagai persegi $q - r - t - s - q$, dengan setiap sisinya hadir dalam *kedua* arah: $q \to r$ dan $r \to q$, $r \to t$ dan $t \to r$, $t \to s$ dan $s \to t$, $s \to q$ dan $q \to s$.

- *Persoalan keseluruhannya.* Lintasan sederhana dari q ke t dapat memutari persegi itu dari sisi mana pun: $q \to r \to t$ atau $q \to s \to t$. Masing-masing berpanjang 2, dan tidak ada lintasan sederhana yang lebih panjang, sebab melanjutkan melewati t tidak ada gunanya dan r serta s tidak bertetangga. Jadi $q \to r \to t$ adalah sebuah lintasan sederhana terpanjang dari q ke t.
- *Potongan pertamanya.* Di dalam jawaban itu, potongan dari q ke r adalah sisi tunggal $q \to r$, berpanjang 1. Padahal lintasan sederhana terpanjang dari q ke r adalah $q \to s \to t \to r$, berpanjang 3. Potongan itu **tidak** optimal bagi submasalahnya sendiri.
- *Penempelannya.* Bisakah potongan yang lebih panjang itu ditempelkan? Ambil lintasan sederhana terpanjang $q \to s \to t \to r$ lalu sambung dengan lintasan sederhana terpanjang dari r ke t, yang dengan penalaran yang sama adalah $r \to q \to s \to t$. Hasilnya, $q \to s \to t \to r \to q \to s \to t$, mengunjungi q, s, dan t dua kali: ia sama sekali bukan lintasan sederhana.

Kedua submasalah itu tidak saling bebas — keduanya memperebutkan simpul yang sama — sehingga penempelannya tidak sah, dan potongan-potongan yang optimal tidak tersusun menjadi keseluruhan yang optimal.

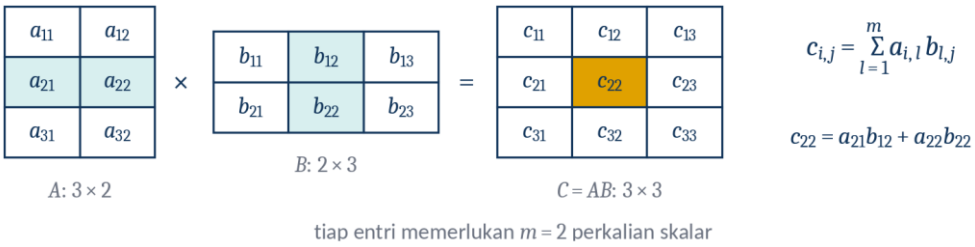
> **PELAJARANNYA.** Sebelum menulis satu baris pun kode *dynamic programming*, *buktikanlah* adanya *optimal substructure* dengan argumen potong-dan-tempel, dan periksalah bahwa penempelannya menghasilkan solusi yang sah. Lintasan terpendek punya *optimal substructure*; lintasan sederhana terpanjang tidak punya, dan berapa pun banyaknya tabel yang Anda isi tidak akan menyelamatkannya. (Lintasan sederhana terpanjang nyatanya NP-hard.)

7.6 Mengalikan Sederetan Matriks

Inilah persoalan utama yang dikerjakan dalam bab ini, dan inilah contoh pertama *dynamic programming* terbaik yang pernah dirancang orang, sebab yang dioptimasi bukanlah hitungannya melainkan *urutan mengerjakan hitungan itu*.

7.6.1 Biaya satu hasil kali matriks

Pertama, aturan dasarnya. **Matriks** adalah kisi bilangan berbentuk persegi panjang. Kita menyebut sebuah matriks berukuran $n \times m$ ("n kali m") bila ia punya n baris dan m kolom. **Gambar 7.8** menunjukkan cara mengalikan dua matriks.



Gambar 7.8 Mengalikan matriks $n \times m$ bernama A dengan matriks $m \times k$ bernama B, di sini dengan $n = 3$, $m = 2$, $k = 3$. Baris A yang diarsir dan kolom B yang diarsir bergabung menghasilkan satu entri terarsir c_{22} pada hasilnya, lewat rumus $c_{i,j} = \sum_{l=1}^{m} a_{i,l} \cdot b_{l,j}$.

Bacalah **Gambar 7.8** dengan cermat. Untuk mengalikan A $(n \times m)$ dengan B $(m \times k)$:

- Kedua matriks harus **conformable** (sepadan): banyaknya kolom A harus sama dengan banyaknya baris B. Keduanya m di sini. Jika berbeda, hasil kalinya tidak ada.
- Hasilnya C berukuran $n \times k$: sebanyak baris A, sebanyak kolom B.
- Entri $c_{i,j}$ pada hasilnya diperoleh dari baris i matriks A dan kolom j matriks B:

$$c_{i,j} = \sum_{l=1}^{m} a_{i,l} \cdot b_{l,j}. \tag{7.7}$$

Lambang $\sum$ (huruf besar Yunani sigma) berarti "jumlahkan". Persamaan Persamaan (7.7) berbunyi: biarkan l berjalan dari 1 sampai m; untuk tiap l, kalikan entri ke-l pada baris i matriks A dengan entri ke-l pada kolom j matriks B; jumlahkan seluruh m hasil kali itu.

Sekarang hitung perkaliannya. Masing-masing dari $n \times k$ entri C berbiaya m perkalian skalar. Karena itu:

> **ATURAN BIAYANYA.** Mengalikan matriks $n \times m$ dengan matriks $m \times k$ berbiaya $n \cdot m \cdot k$ perkalian skalar. Satu rumus inilah seluruh model biaya untuk sisa §7.6. Hafalkan.

Sebuah instans konkret: matriks 10×20 dikali matriks 20×3 berbiaya $10 \times 20 \times 3 = 600$ perkalian dan menghasilkan hasil berukuran 10×3.

7.6.2 Keasosiatifan, dan mengapa pengelompokannya penting

Perkalian matriks bersifat **asosiatif**: $(AB)C = A(BC)$. Kedua ruas memuat bilangan yang sama pada posisi yang sama. Yang *tidak* sama pada keduanya adalah banyaknya kerja.

Untuk melihat mengapa tidak sama, tinjaulah tiga matriks

- A: 10×100
- B: 100×5
- C: 5×50

Menghitung $(AB)C$:

1. AB berbiaya $10 \times 100 \times 5 = 5\,000$, menghasilkan matriks 10×5.
2. Mengalikan hasil itu dengan C berbiaya $10 \times 5 \times 50 = 2\,500$.
3. Total: $5\,000 + 2\,500 = 7\,500$.

Menghitung $A(BC)$:

1. BC berbiaya $100 \times 5 \times 50 = 25\,000$, menghasilkan matriks 100×50.
2. Mengalikan A dengan hasil itu berbiaya $10 \times 100 \times 50 = 50\,000$.

3. Total: $25\,000 + 50\,000 = 75\,000$.

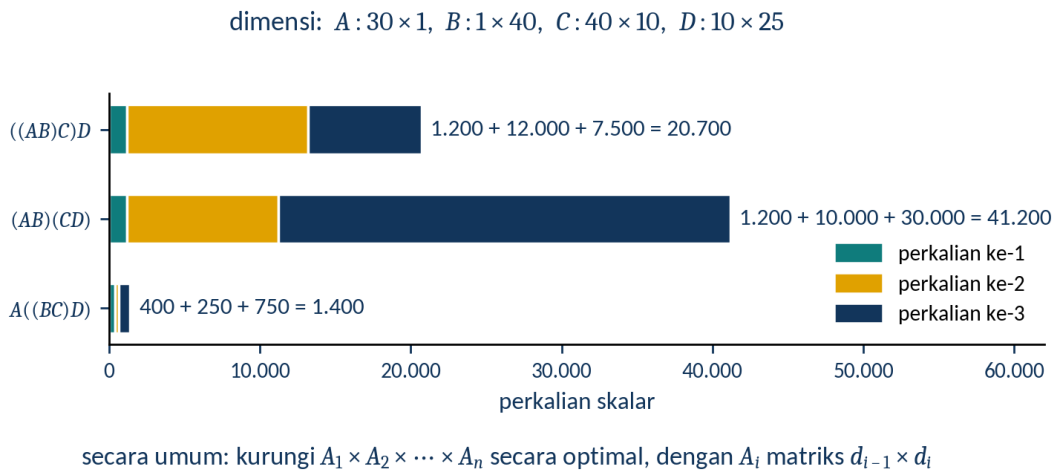
Jawaban yang sama. Kerja sepuluh kali lipat. Tidak ada yang dihitung keliru pada versi kedua; ia semata-mata membangun matriks antara yang raksasa, yang tak pernah diciptakan versi pertama.

> **DEFINISI.** *Parenthesization* (pengelompokan tanda kurung) atas hasil kali $A_1 A_2 \cdots A_n$ adalah cara menyisipkan tanda kurung sedemikian rupa sehingga setiap perkalian terjadi antara persis dua matriks. Sebuah hasil kali disebut **terkurung penuh** bila ia berupa matriks tunggal, atau berupa hasil kali berkurung dari dua hasil kali yang terkurung penuh. **Persoalan perkalian deretan matriks** (*matrix-chain multiplication problem*) berbunyi: diberikan ukuran $A_1, \ldots, A_n$, carilah pengelompokan tanda kurung yang meminimumkan total banyaknya perkalian skalar.

Kita tidak sedang diminta mengalikan apa pun. Kita sedang diminta menghasilkan sebuah *rencana*. Keluarannya adalah sehimpunan tanda kurung.

7.6.3 Sebuah contoh empat matriks, dihitung harganya

Gambar 7.9 adalah sebuah instans kecil, dan ia pantas dikerjakan sampai digit terakhir.



Gambar 7.9 Sebuah contoh empat matriks. Untuk A (30×1), B (1×40), C (40×10), D (10×25), tiga di antara lima pengurungan yang mungkin berbiaya masing-masing 20 700, 41 200, dan 1 400; setiap batang memecah totalnya menjadi tiga perkaliannya. Baris terakhirnya menetapkan notasi yang dipakai mulai dari sini: kita mengelompokkan tanda kurung secara optimal atas $A_1 \times A_2 \times \cdots \times A_n$, dengan A_i berupa matriks $d_{i-1} \times d_i$.

Konvensi ukuran pada baris terakhir **Gambar 7.9** adalah kepingan notasi terpenting bagi seluruh subbab ini, jadi mari kita bongkar. Alih-alih mendaftar kedua ukuran tiap matriks secara terpisah — yang akan mengulang setiap bilangan, sebab matriks yang berurutan harus sepadan — kita mendaftar satu barisan tunggal

$$d = \langle d_0, d_1, d_2, \ldots, d_n \rangle$$

berisi $n + 1$ bilangan, lalu menyatakan bahwa A_i berukuran $d_{i-1} \times d_i$. Kesepadanannya lalu terjadi dengan sendirinya: kolom A_i dan baris A_{i+1} sama-sama d_i.

> **NOTASI.** Buku ini menulis barisan ukurannya sebagai d dan tabel biayanya sebagai M. Cormen *dkk.* [3] menulis barisan yang sama sebagai p dan tabel yang sama sebagai m, sehingga rekurensinya berbunyi $m[i,j] = \min_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$. **Keduanya adalah hal yang sama**, huruf demi huruf: $d \equiv p$ dan $M \equiv m$. Kita memakai d dan M di

sepanjang buku ini agar teksnya sejalan dengan gambar-gambarnya, dan kita mengutip bentuk p-dan-m sekali saja, di sini, supaya Anda mengenalinya di CLRS dan di tempat lain.

Untuk contoh pada **Gambar 7.9**, A berukuran 30×1, B berukuran 1×40, C berukuran 40×10, D berukuran 10×25, sehingga

$$d = \langle 30,\ 1,\ 40,\ 10,\ 25 \rangle, \qquad n = 4.$$

Sekarang hitung harga ketiga pengurungan pada gambar itu, satu perkalian satu baris.

Pengurungan 1: $((AB)C)D$**.**

1. AB: $(30 \times 1)(1 \times 40)$ berbiaya $30 \times 1 \times 40 = 1\,200$; hasilnya 30×40.
2. $(AB)C$: $(30 \times 40)(40 \times 10)$ berbiaya $30 \times 40 \times 10 = 12\,000$; hasilnya 30×10.
3. $((AB)C)D$: $(30 \times 10)(10 \times 25)$ berbiaya $30 \times 10 \times 25 = 7\,500$; hasilnya 30×25.
4. **Total:** $1\,200 + 12\,000 + 7\,500 = 20\,700$**.**

Pengurungan 2: $(AB)(CD)$**.**

1. AB: berbiaya $30 \times 1 \times 40 = 1\,200$; hasilnya 30×40.
2. CD: $(40 \times 10)(10 \times 25)$ berbiaya $40 \times 10 \times 25 = 10\,000$; hasilnya 40×25.
3. $(AB)(CD)$: $(30 \times 40)(40 \times 25)$ berbiaya $30 \times 40 \times 25 = 30\,000$; hasilnya 30×25.
4. **Total:** $1\,200 + 10\,000 + 30\,000 = 41\,200$**.**

Pengurungan 3: $A((BC)D)$**.**

1. BC: $(1 \times 40)(40 \times 10)$ berbiaya $1 \times 40 \times 10 = 400$; hasilnya 1×10.
2. $(BC)D$: $(1 \times 10)(10 \times 25)$ berbiaya $1 \times 10 \times 25 = 250$; hasilnya 1×25.
3. $A((BC)D)$: $(30 \times 1)(1 \times 25)$ berbiaya $30 \times 1 \times 25 = 750$; hasilnya 30×25.
4. **Total:** $400 + 250 + 750 = 1\,400$**.**

Setiap angka cocok dengan **Gambar 7.9**. Dan lihatlah rentangnya: yang terburuk di antara ketiganya berbiaya 41 200 perkalian, yang terbaik berbiaya 1 400. Itu faktor **dua puluh sembilan**, untuk empat matriks kecil, tanpa perubahan apa pun pada jawaban yang dihasilkan.

MENGAPA YANG KETIGA MENANG. B berukuran 1×40 — satu baris. Mengalikan apa pun dengan matriks satu baris itu murah, sebab n pada $n \cdot m \cdot k$ bernilai 1. Pengurungan 3 memakai B sejak awal dan menjaga setiap matriks antara tetap setinggi satu baris. Pengurungan 1 dan 2 membangun matriks antara yang gemuk, 30×40 dan 40×25, lalu membayar untuk mengalikannya. Pelajarannya berlaku umum: pengelompokan tanda kurung yang baik membuat ukuran yang *kecil* yang mengerjakan kerjanya, dan tak pernah mewujudkan matriks antara besar yang tidak ia perlukan.

7.6.4 Ada berapa banyak parenthesization?

Algoritma yang paling kentara adalah: coba semuanya. Sebelum kita menerimanya, sebaiknya kita hitung dulu ada berapa banyak.

Misalkan $P(n)$ adalah banyaknya cara mengurungkan penuh sebuah hasil kali n matriks. Sebuah pengurungan atas $A_1 \cdots A_n$ punya satu perkalian terluar, yang membelah deretannya antara A_k dan A_{k+1} untuk suatu k dengan $1 \le k \le n-1$. Bagian kiri dapat dikurungkan dalam $P(k)$ cara dan bagian kanan dalam $P(n-k)$ cara, saling bebas. Jadi kita memperoleh rekurensi Persamaan (7.8):

$$P(1) = 1, \qquad P(n) = \sum_{k=1}^{n-1} P(k)\,P(n-k) \quad (n \geq 2). \tag{7.8}$$

Mari kita hitung dengan tangan untuk beberapa nilai pertama:

- $P(1) = 1$ — matriks tunggal, tidak ada yang bisa dikurung.
- $P(2) = P(1)P(1) = 1$ — hanya (A_1A_2).
- $P(3) = P(1)P(2) + P(2)P(1) = 1 + 1 = 2$ — yaitu $(A_1(A_2A_3))$ dan $((A_1A_2)A_3)$.
- $P(4) = P(1)P(3) + P(2)P(2) + P(3)P(1) = 2 + 1 + 2 = 5$.
- $P(5) = P(1)P(4) + P(2)P(3) + P(3)P(2) + P(4)P(1) = 5 + 2 + 2 + 5 = 14$.

Inilah **bilangan Catalan**. Dalam bentuk tertutup, $P(n+1) = C_n$, dengan C_n diberikan oleh Persamaan (7.9):

$$C_n = \frac{1}{n+1}\binom{2n}{n}, \tag{7.9}$$

dan Stanley [4] mengatalogkan lebih dari dua ratus keluarga objek berbeda yang dicacah bilangan-bilangan itu. Pertumbuhannya diketahui sebagai

$$C_n = \Omega\left(\frac{4^n}{n^{3/2}}\right),$$

yang eksponensial dalam n. Jadi pencacahan *brute force* tidak punya harapan. **Kode 7.3** mencacah seluruhnya untuk deretan empat matriks — lima pengurungan, didaftar dan dihitung harganya semuanya — dan akan memakan waktu lebih lama daripada umur alam semesta pada $n = 60$.

Kode 7.3 Pencacahan *brute force* atas setiap pengelompokan tanda kurung, dipakai sebagai kebenaran acuan untuk memeriksa hasil program dinamisnya. Kompilasi dengan `g++ -std=c++17 -O2 -o brute brute.cpp`.

```
// brute.cpp - cacah SETIAP pengelompokan tanda kurung sebuah deretan dan hitung harganya,
// supaya jawaban dynamic programming-nya dapat diperiksa terhadap kebenarannya.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <climits>
using namespace std;

vector<long long> d;

// Setiap cara membelah A_i..A_j, dengan biayanya dan pengelompokan yang dicetak.
vector<pair<long long,string>> allWays(int i, int j) {
    if (i == j) return {{0, "A" + to_string(i)}};
    vector<pair<long long,string>> out;
    for (int k = i; k < j; ++k)
        for (auto& L : allWays(i, k))
            for (auto& R : allWays(k + 1, j))
                out.push_back({L.first + R.first + d[i-1]*d[k]*d[j],
                               "(" + L.second + R.second + ")"});
    return out;
}

int main() {
    d = {10, 20, 3, 5, 30};
    int n = (int)d.size() - 1;
    auto ways = allWays(1, n);
    cout << "d = [10, 20, 3, 5, 30]: " << ways.size() << " parenthesizations\n\n";
    cout << left << setw(22) << "grouping" << right << setw(10) << "cost" << "\n";
    long long best = LLONG_MAX, worst = 0; string bestS, worstS;
    for (auto& w : ways) {
        cout << left << setw(22) << w.second << right << setw(10) << w.first << "\n";
        if (w.first < best)  { best  = w.first; bestS  = w.second; }
        if (w.first > worst) { worst = w.first; worstS = w.second; }
    }
    cout << "\ncheapest : " << bestS  << " at " << best  << "\n";
    cout << "dearest  : " << worstS << " at " << worst << "\n";
    cout << "ratio    : " << fixed << setprecision(2) << (double)worst / best << "
times\n";
    return 0;
}
```

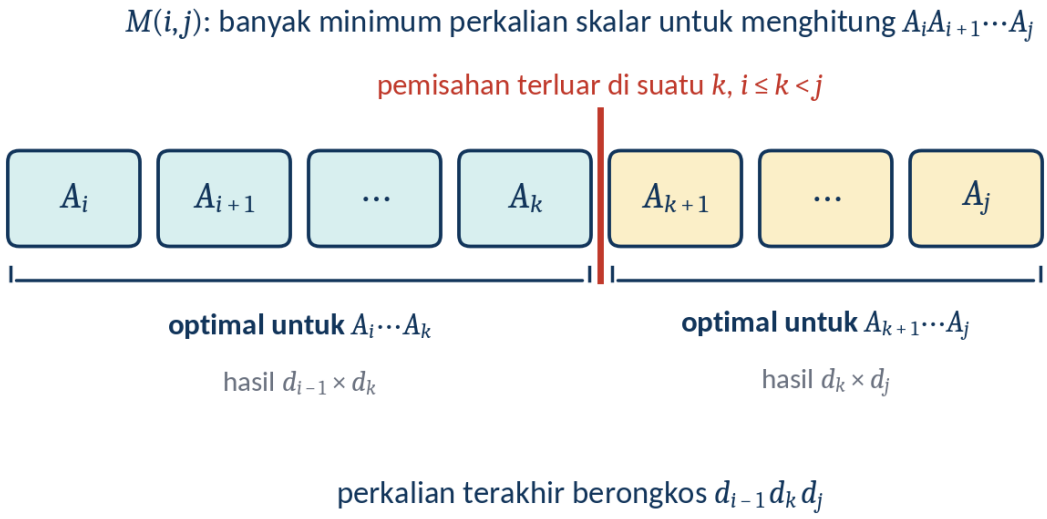
Pada deretan yang sebentar lagi kita selesaikan dengan tangan, $d = \langle 10,20,3,5,30 \rangle$, ia mencetak:

```
d = [10, 20, 3, 5, 30]: 5 parenthesizations

grouping                   cost
(A1(A2(A3A4)))             8250
(A1((A2A3)A4))             9300
((A1A2)(A3A4))             1950
((A1(A2A3))A4)             2800
(((A1A2)A3)A4)             2250

cheapest : ((A1A2)(A3A4)) at 1950
dearest  : (A1((A2A3)A4)) at 9300
ratio    : 4.77 times
```

Lima pengurungan, persis seperti diramalkan $P(4) = 5$, dan yang terbaik 4.77 kali lebih murah daripada yang terburuk. Simpanlah bilangan 1 950 dan pengelompokan $((A_1A_2)(A_3A_4))$ dalam ingatan: di §7.6.8 kita akan menurunkan keduanya dengan tangan, tanpa mencacah apa pun.

7.6.5 Submasalahnya, dan rekurensi yang diturunkan

Gambar 7.10 adalah langkah yang mengubah persoalannya menjadi rekurensi, dan ia melakukannya dengan dua pengamatan.



Gambar 7.10 Dua pengamatan kuncinya. Pertama, kurung terluar pasti membelah deretannya pada *suatu* posisi k dengan $i \le k < j$. Kedua, berapa pun nilai k itu, pengelompokan tanda kurung di kedua sisinya haruslah sendirinya optimal — *optimal substructure*, dan itulah pembenaran bagi rekurensinya.

Ambil keduanya satu per satu.

Mendefinisikan submasalahnya. Mari kita tuliskan: *misalkan $M(i,j)$ adalah banyaknya perkalian minimum yang diperlukan untuk menghitung* $\prod_{k=i}^{j} A_k$, yaitu hasil kali $A_iA_{i+1}\cdots A_j$. Inilah keputusan pemodelan terpenting dari seluruh algoritmanya. Submasalah kita adalah **subderetan yang bersinambung**, ditandai oleh indeks pertamanya i dan indeks terakhirnya j, dengan $1 \le i \le j \le n$. Persoalan aslinya adalah $M(1,n)$.

Mengapa harus bersinambung? Karena pengelompokan tanda kurung hanya pernah bisa mengurung matriks yang *bersebelahan*; tidak ada cara mengalikan A_1 dengan A_3 tanpa A_2 di antara keduanya. Struktur persoalannya sendirilah yang menyerahkan ruang submasalahnya kepada kita.

Pengamatan satu: belahan terluar. Setiap pengurungan penuh atas $A_i \cdots A_j$ (dengan $i < j$) berakhir dengan satu perkalian terakhir, atas dua matriks yang sudah terhitung. Kedua matriks itu adalah $A_i \cdots A_k$ dan $A_{k+1} \cdots A_j$ untuk suatu k dengan $i \le k < j$. Tidak ada kemungkinan lain: perkalian terakhir pasti menggabungkan sebuah awalan dengan akhiran pasangannya.

Berapa biaya perkalian terakhir itu? Faktor kirinya $A_i \cdots A_k$ berukuran $d_{i-1} \times d_k$ (baris A_i, kolom A_k). Faktor kanannya $A_{k+1} \cdots A_j$ berukuran $d_k \times d_j$. Menurut aturan biaya §7.6.1, mengalikan keduanya berbiaya

$$d_{i-1} \cdot d_k \cdot d_j.$$

Perhatikan bahwa biaya ini bersandar **hanya** pada i, j, dan k — sama sekali tidak pada bagaimana kedua paruhnya dikurung di dalamnya. Kebebasan itulah yang membuat rekurensinya bekerja.

Pengamatan dua: kedua paruhnya harus optimal. Inilah *optimal substructure*, dan inilah bukti potong-dan-tempelnya.

> **KLAIM.** Jika sebuah pengurungan optimal atas $A_i \cdots A_j$ membelah pada k, maka pengurungan yang ia pakai untuk $A_i \cdots A_k$ optimal bagi subderetan itu, demikian pula untuk $A_{k+1} \cdots A_j$.
>
> **Bukti.** Andaikan pengurungan optimal S atas $A_i \cdots A_j$ membelah pada k, dan bagian kirinya memakai pengurungan S_L berbiaya c_L. Andaikan ada pengurungan lain $S_L{}'$ atas $A_i \cdots A_k$ yang

berbiaya $c_L' < c_L$. **Potong** S_L keluar dari S lalu **tempel** S_L' ke dalamnya. Hasilnya tetap pengurungan yang sah atas $A_i \cdots A_j$ — pengurungan subderetan kiri dapat disubstitusikan dengan bebas, sebab ukuran faktor kirinya, $d_{i-1} \times d_k$, tidak bersandar pada bagaimana ia dikurung. Biayanya $c_L' + c_R + d_{i-1}d_kd_j < c_L + c_R + d_{i-1}d_kd_j$, yaitu biaya S. Itu bertentangan dengan keoptimalan S. Karena itu S_L' semacam itu tidak ada, dan S_L memang optimal. Argumen yang sama berlaku bagi bagian kanannya. ■

Menyatukan keduanya. Kita tidak tahu k mana yang dipakai optimumnya. Jadi kita coba semuanya lalu simpan yang terbaik. Itulah rekurensinya, dan **Gambar 7.11** menyatakannya.

$$M(i,i) = 0$$

$$M(i,j) = \min_{i \le k < j} \{ M(i,k) + M(k+1,j) + d_{i-1}\, d_k\, d_j \}$$

Gambar 7.11 Rekurensi deretan matriks. Kasus basisnya adalah $M(i,i) = 0$; kasus umumnya meminimumkan, atas setiap titik belah k, biaya subderetan kiri ditambah biaya subderetan kanan ditambah $d_{i-1}d_kd_j$ untuk perkalian terakhirnya.

Dalam tipografi kita, dan dengan kasus basis yang dieja terang-terangan, rekurensi itu adalah Persamaan (7.10):

$$M(i,j) = \begin{cases} 0 & \text{jika } i = j, \\ \min\limits_{i \le k < j} \{ M(i,k) + M(k+1,j) + d_{i-1}\, d_k\, d_j \} & \text{jika } i < j. \end{cases} \tag{7.10}$$

Bacalah baris keduanya dengan kata-kata, sebab inilah kalimat yang harus dapat Anda keluarkan dari ingatan: *cara termurah mengalikan subderetan dari i sampai j ditemukan dengan mencoba setiap tempat k yang mungkin menjadi tempat perkalian terakhir terjadi, dan untuk tiap tempat itu menjumlahkan tiga hal — biaya termurah segala sesuatu di kiri belahan, biaya termurah segala sesuatu di kanan belahan, dan biaya perkalian terakhir itu sendiri — lalu mengambil total yang terkecil.*

Kasus basisnya mengatakan bahwa matriks tunggal sama sekali tidak memerlukan perkalian, dan itu benar begitu dilihat: $M(i,i) = 0$.

TIGA JEBAKAN INDEKS. (1) Jangkauannya $i \le k < j$, **bukan** $i \le k \le j$; mengambil $k = j$ akan menaruh semua matriksnya di kiri dan tidak satu pun di kanan. (2) Subderetan kanannya mulai di $k + 1$, bukan di k; A_k milik bagian kiri. (3) Triple ukurannya adalah d_{i-1}, d_k, d_j — perhatikan $i - 1$-nya, dan itulah sebabnya barisan d punya $n + 1$ entri dan diindeks mulai dari 0. Salah pada salah satu di antaranya menghasilkan program yang jalan, mencetak sebuah bilangan, dan diam-diam keliru.

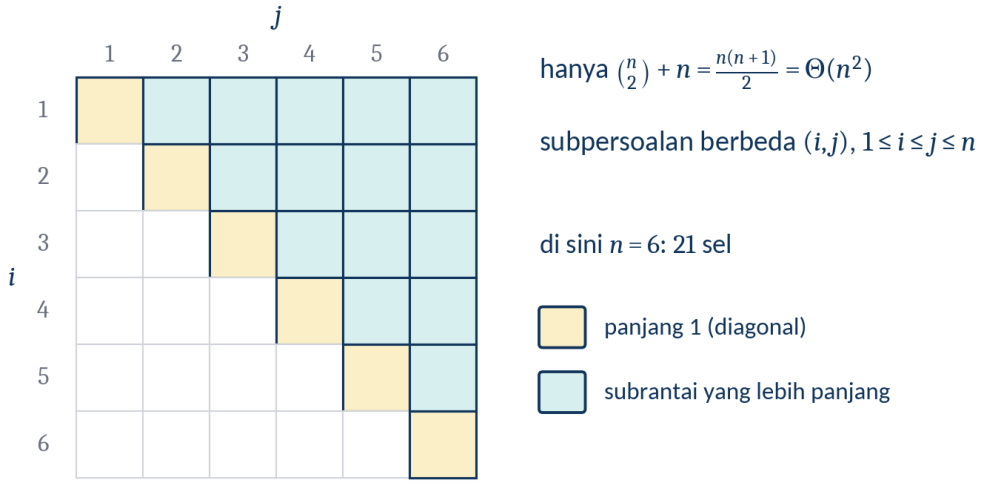
7.6.6 Menghitung submasalahnya

Mengapa *dynamic programming* akan menolong di sini? Persis karena alasan yang sama seperti pada Fibonacci: submasalah yang sungguh berbeda jauh lebih sedikit daripada yang dikunjungi rekursinya.

Sebuah submasalah adalah pasangan (i, j) dengan $1 \le i \le j \le n$. Ada berapa banyak pasangan semacam itu? Yang dengan $i < j$ banyaknya $\binom{n}{2}$ — pilih dua indeks berbeda dari n lalu jadikan yang lebih kecil sebagai i — dan yang dengan $i = j$ banyaknya n. Jadi totalnya, Persamaan (7.11), adalah

$$\binom{n}{2} + n = \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2} = \Theta(n^2), \tag{7.11}$$

yang persis merupakan hitungan pada **Gambar 7.12**. Untuk $n = 60$ itu berarti 1 830 pertanyaan berbeda, melawan $P(60) \approx 4.1 \times 10^{32}$ pengelompokan tanda kurung yang akan diperiksa pencarian *brute force*. **Gambar 7.12** menegaskan pokok yang mengikutinya.



Gambar 7.12 Hitungan submasalahnya: hanya ada $\binom{n}{2} + n = \Theta(n^2)$ submasalah (i, j) yang berbeda dengan $1 \le i \le j \le n$. Segitiganya menggambarkan submasalah itu untuk $n = 6$: 21 sel pada dan di atas diagonal. Jurang antara $\Theta(n^2)$ pertanyaan dan kunjungan rekursif yang banyaknya eksponensial itulah celah yang dimanfaatkan *dynamic programming*.

7.6.7 Algoritmanya

Rekurensinya belum menjadi algoritma, sebab ia belum mengatakan dalam urutan apa entri-entrinya harus dihitung. **Pseudokode 7.5** memberikan jawabannya: **Matrix-Chain-Order**.

Pseudokode 7.5 Matrix-Chain-Order dalam pseudokode. Baris 1–2 mengisi diagonalnya dengan nol. Baris 3 adalah yang paling penting: gelung terluarnya berjalan atas **panjang** deretan l, bukan atas i atau j, supaya setiap subderetan terselesaikan sebelum subderetan lebih panjang mana pun yang bersandar padanya. Baris 7–11 mencoba setiap belahan k dan menyimpan baik biaya terbaik di M maupun belahan pemenangnya di c.

```
MATRIX-CHAIN-ORDER(d[0..n]):
 1.  for i ← 1 to n
 2.      M[i,i] ← 0
 3.  for l ← 2 to n                          // l = panjang subrantai
 4.      for i ← 1 to n − l + 1
 5.          j ← i + l − 1
 6.          M[i,j] ← ∞
 7.          for k ← i to j − 1
 8.              q ← M[i,k] + M[k+1,j] + d[i−1]·d[k]·d[j]
 9.              if q < M[i,j] then
10.                  M[i,j] ← q
11.                  c[i,j] ← k
12.  return M, c
```

Baris 3 adalah inti persoalannya dan pantas diberi satu paragraf tersendiri.

> **MENGAPA MENGGELUNG ATAS PANJANG.** Entri $M[i,j]$ bersandar pada $M[i,k]$ dan $M[k+1,j]$ untuk berbagai k yang terletak betul-betul di antaranya. Keduanya adalah subderetan yang *lebih pendek*: $M[i,k]$ berpanjang $k-i+1 \leq j-i$, dan $M[k+1,j]$ berpanjang $j-k \leq j-i$, sedangkan $M[i,j]$ berpanjang $j-i+1$. Jadi jika kita memproses subderetan dalam urutan panjang yang membesar, segala yang diperlukan sebuah sel sudah ada di dalam tabel ketika kita tiba di sel itu. Menggelung atas i atau j dalam urutan alaminya **tidak** punya sifat itu dan menghasilkan sampah.

Baris demi baris, inilah yang dikerjakan pseudokodenya:

- **Baris 1–2.** Untuk tiap i, tetapkan $M[i,i] \leftarrow 0$. Inilah kasus basisnya: deretan berpanjang 1 itu gratis.
- **Baris 3.** `for l ← 2 to n` — tinjau deretan berpanjang 2,3, …, n dalam urutan itu.
- **Baris 4.** `for i ← 1 to n-l+1` — indeks awal sebuah deretan berpanjang l bisa berada di mana saja dari 1 sampai $n-l+1$, dan di luar itu deretannya akan melewati ujung.
- **Baris 5.** `j ← i+l-1` — indeks akhirnya mengikuti dari awal dan panjangnya.
- **Baris 6.** `M[i,j] ← ∞` — inisialisasi minimum berjalannya ke "tak hingga" supaya calon pertama selalu mengalahkannya. Dalam C++ kita memakai sebuah tetapan yang sangat besar.
- **Baris 7–8.** Untuk tiap belahan k dari i sampai $j-1$, hitung biaya calonnya $q = M[i,k] + M[k+1,j] + d_{i-1}d_kd_j$.
- **Baris 9–11.** Jika q mengalahkan yang terbaik sejauh ini, catat baik biaya terbaik yang baru di $M[i,j]$ maupun titik belah pemenangnya di $c[i,j]$.
- **Baris 12.** Kembalikan kedua tabelnya.

Tabel c murni pembukuan — ia tidak berbiaya apa pun untuk dipelihara dan ia satu-satunya alasan mengapa kita nanti dapat merekonstruksi pengurungan yang sebenarnya, bukan sekadar harganya.

7.6.8 Tabel yang diisi dengan tangan, sel demi sel

Mari kita jalankan algoritmanya pada $d = [10,20,3,5,30]$, tanpa ada yang disembunyikan.

Deretannya berisi $n = 4$ matriks:

- A_1 berukuran $d_0 \times d_1 = 10 \times 20$
- A_2 berukuran $d_1 \times d_2 = 20 \times 3$
- A_3 berukuran $d_2 \times d_3 = 3 \times 5$
- A_4 berukuran $d_3 \times d_4 = 5 \times 30$

Langkah 0: diagonalnya. $M[1,1] = M[2,2] = M[3,3] = M[4,4] = 0$. Matriks tunggal tidak berbiaya apa pun.

Langkah 1: deretan berpanjang $l = 2$. Indeks awalnya i berjalan dari 1 sampai $n-l+1 = 3$. Untuk $l = 2$ hanya ada satu belahan yang mungkin, $k = i$, jadi tidak perlu ada minimisasi.

Sel $M[1,2]$ ($i = 1, j = 2$, hanya $k = 1$):

$$M[1,1] + M[2,2] + d_0d_1d_2 = 0 + 0 + 10 \times 20 \times 3 = 600.$$

Jadi $M[1,2] = 600$ dan $c[1,2] = 1$.

Sel $M[2,3]$ ($i = 2, j = 3$, hanya $k = 2$):

$$M[2,2] + M[3,3] + d_1 d_2 d_3 = 0 + 0 + 20 \times 3 \times 5 = 300.$$

Jadi $M[2,3] = 300$ dan $c[2,3] = 2$.

Sel $M[3,4]$ ($i = 3, j = 4$, hanya $k = 3$):

$$M[3,3] + M[4,4] + d_2 d_3 d_4 = 0 + 0 + 3 \times 5 \times 30 = 450.$$

Jadi $M[3,4] = 450$ dan $c[3,4] = 3$.

Langkah 2: deretan berpanjang $l = 3$. Sekarang i berjalan dari 1 sampai $n - l + 1 = 2$, dan tiap sel punya dua calon belahan.

Sel $M[1,3]$ ($i = 1, j = 3$; k boleh 1 atau 2):

- $k = 1$: $M[1,1] + M[2,3] + d_0 d_1 d_3 = 0 + 300 + 10 \times 20 \times 5 = 0 + 300 + 1\,000 = 1\,300$.
- $k = 2$: $M[1,2] + M[3,3] + d_0 d_2 d_3 = 600 + 0 + 10 \times 3 \times 5 = 600 + 0 + 150 = 750$.
- Minimumnya 750, dicapai pada $k = 2$. Jadi $M[1,3] = 750$, $c[1,3] = 2$.

Berhentilah sejenak pada perbandingan itu. Membelah pada $k = 1$ berarti menghitung $A_2 A_3$ lebih dulu (murah, 300) lalu mengalikan A_1 yang jangkung dengan hasilnya — tetapi perkalian terakhirnya $10 \times 20 \times 5 = 1\,000$, dan itulah yang mendominasi. Membelah pada $k = 2$ berarti membayar 600 di muka untuk $A_1 A_2$, tetapi itu meruntuhkan 20 menjadi 3, sehingga perkalian terakhirnya hanya berbiaya 150. Membayar lebih banyak di awal demi menyusutkan sebuah ukuran adalah tema yang berulang dalam persoalan ini.

Sel $M[2,4]$ ($i = 2, j = 4$; k boleh 2 atau 3):

- $k = 2$: $M[2,2] + M[3,4] + d_1 d_2 d_4 = 0 + 450 + 20 \times 3 \times 30 = 0 + 450 + 1\,800 = 2\,250$.
- $k = 3$: $M[2,3] + M[4,4] + d_1 d_3 d_4 = 300 + 0 + 20 \times 5 \times 30 = 300 + 0 + 3\,000 = 3\,300$.
- Minimumnya 2 250, dicapai pada $k = 2$. Jadi $M[2,4] = 2\,250$, $c[2,4] = 2$.

Langkah 3: seluruh deretannya, $l = 4$. Hanya $i = 1$, sehingga $j = 4$, dengan tiga calon belahan.

- $k = 1$: $M[1,1] + M[2,4] + d_0 d_1 d_4 = 0 + 2\,250 + 10 \times 20 \times 30 = 0 + 2\,250 + 6\,000 = 8\,250$.
- $k = 2$: $M[1,2] + M[3,4] + d_0 d_2 d_4 = 600 + 450 + 10 \times 3 \times 30 = 600 + 450 + 900 = 1\,950$.
- $k = 3$: $M[1,3] + M[4,4] + d_0 d_3 d_4 = 750 + 0 + 10 \times 5 \times 30 = 750 + 0 + 1\,500 = 2\,250$.
- Minimumnya 1 950, dicapai pada $k = 2$. Jadi $M[1,4] = 1\,950$, $c[1,4] = 2$.

Itulah jawabannya: 1 950 **perkalian skalar**, persis bilangan yang ditemukan **Kode 7.3** dengan mencacah kelima pengurungannya. Tabel yang sudah selesai adalah **Tabel 7.3** dan **Tabel 7.4**.

Tabel 7.3 Tabel M yang sudah selesai untuk $d = \langle 10,20,3,5,30 \rangle$. Hanya segitiga atasnya yang bermakna, sebab sebuah subderetan haruslah punya $i \leq j$. Jawaban persoalan aslinya ada di pojok kanan atas, $M[1,4] = 1950$.

	j=1	j=2	j=3	j=4
$i = 1$	0	600	750	**1950**
$i = 2$	—	0	300	2250
$i = 3$	—	—	0	450
$i = 4$	—	—	—	0

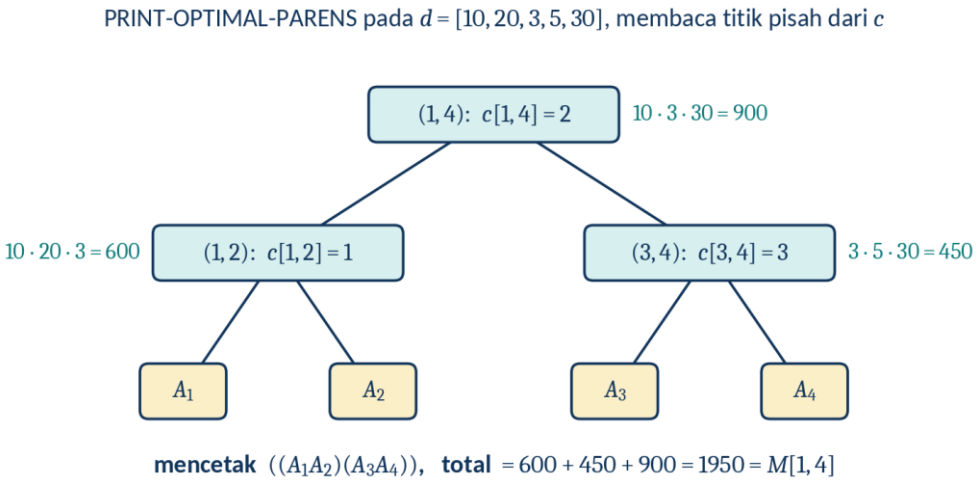
Tabel 7.4 Tabel c yang sudah selesai: $c[i,j]$ mencatat titik belah k yang mencapai minimum pada $M[i,j]$. Ia satu-satunya catatan tentang *bagaimana* optimumnya dicapai, dan §7.6.9 membaca pengurungannya dari sana.

	j=1	j=2	j=3	j=4
$i = 1$	—	1	2	**2**
$i = 2$	—	—	2	2
$i = 3$	—	—	—	3
$i = 4$	—	—	—	—

Perhatikan urutan pengisian selnya: diagonalnya lebih dulu, lalu diagonal tepat di atasnya, lalu berikutnya lagi, berakhir pada satu sel pojok. Tabelnya diisi **diagonal demi diagonal**, bergerak ke atas dan ke kanan — dan persis itulah yang dihasilkan penggelungan atas l.

7.6.9 Tabel c dan rekonstruksi solusinya

Gambar 7.13 menyatakan apa yang diberikan kedua tabel itu kepada kita.



Gambar 7.13 Sesudah algoritmanya berjalan, $M[1,n]$ memuat nilai optimalnya dan c memuat pembagian optimalnya; sebuah prosedur rekursif sederhana **Print-Optimal-Parens**(c,i,j) menyusuri c dan mencetak pengurungannya. Pohonnya memperlihatkan penyusuran itu pada contoh yang baru saja kita selesaikan dengan tangan, $d = [10,20,3,5,30]$, dengan ongkos tiap perkalian di samping panggilannya.

Print-Optimal-Parens hanya tiga baris, seperti tampak pada **Pseudokode 7.6**:

Pseudokode 7.6 Print-Optimal-Parens: membaca pengurungan optimal dari tabel c.

```
string printOptimalParens(int i, int j) {
    if (i == j) return "A" + to_string(i);          // matriks tunggal dicetak sebagai
dirinya sendiri
    int k = c[i][j];                                 // di mana optimumnya membelah deretan
ini
    return "(" + printOptimalParens(i, k)
               + printOptimalParens(k + 1, j) + ")";
}
```

Mari kita jalankan dengan tangan pada **Tabel 7.4**, satu panggilan satu baris.

1. `parens(1,4)`: $i \neq j$, dan $c[1,4] = 2$. Jadi jawabannya `"(" + parens(1,2) + parens(3,4) + ")"`.
2. `parens(1,2)`: $i \neq j$, dan $c[1,2] = 1$. Jadi jawabannya `"(" + parens(1,1) + parens(2,2) + ")"`.
3. `parens(1,1)`: $i = j$, mengembalikan `"A1"`.

4. `parens(2,2)`: $i = j$, mengembalikan `"A2"`.
5. Kembali ke langkah 2: `"(A1A2)"`.
6. `parens(3,4)`: $i \neq j$, dan $c[3,4] = 3$. Jadi jawabannya `"(" + parens(3,3) + parens(4,4) + ")"`.
7. `parens(3,3)` mengembalikan `"A3"`; `parens(4,4)` mengembalikan `"A4"`.
8. Kembali ke langkah 6: `"(A3A4)"`.
9. Kembali ke langkah 1: `"((A1A2)(A3A4))"`.

Pengurungan optimalnya adalah $((A_1A_2)(A_3A_4))$. Mari kita periksa harganya secara mandiri, hanya dari aturan biayanya:

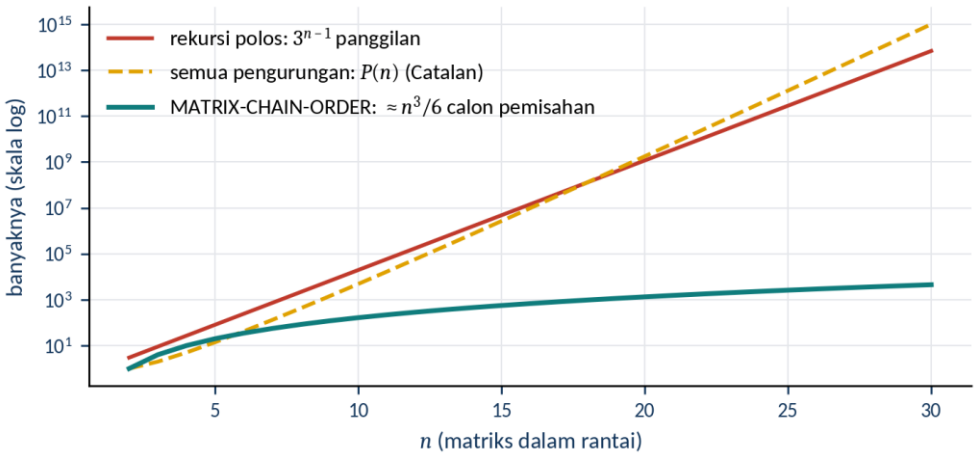
1. A_1A_2: $(10 \times 20)(20 \times 3)$ berbiaya $10 \times 20 \times 3 = 600$; hasilnya 10×3.
2. A_3A_4: $(3 \times 5)(5 \times 30)$ berbiaya $3 \times 5 \times 30 = 450$; hasilnya 3×30.
3. Hasil kali terakhirnya: $(10 \times 3)(3 \times 30)$ berbiaya $10 \times 3 \times 30 = 900$; hasilnya 10×30.
4. Total: $600 + 450 + 900 = 1\,950$. ✓

Ia sesuai dengan $M[1,4]$ dan dengan pencacahan *brute force*-nya. Tiga jalan mandiri menuju bilangan yang sama.

> **MENGAPA c HARUS DISIMPAN.** Tabel M sendirian hanya memberi tahu Anda bahwa optimumnya *berbiaya* 1 950; ia tidak memberi tahu cara mencapainya. Anda memang bisa memulihkan titik belahnya dengan memeriksa ulang M (untuk tiap k, periksa apakah $M[i,k] + M[k+1,j] + d_{i-1}d_kd_j$ sama dengan $M[i,j]$), dengan biaya $\Theta(n)$ per langkah rekonstruksi. Mencatat c sambil jalan berbiaya tambahan memori $\Theta(n^2)$ dan membuat rekonstruksinya $\Theta(n)$ seluruhnya. Inilah pola umum **langkah 4** dari resep DP di §7.8: simpanlah pilihannya, bukan hanya nilainya.

7.6.10 Waktu $\Theta(n^3)$ dan ruang $\Theta(n^2)$

Gambar 7.14 memberikan vonis tentang biayanya.



Gambar 7.14 Waktu jalan **Matrix-Chain-Order**: dengan mudah terlihat $O(n^3)$, dan sesungguhnya juga $\Omega(n^3)$, sehingga $\Theta(n^3)$. Grafiknya, pada skala logaritmik, menyandingkan banyaknya calon pemisahan itu dengan 3^{n-1} panggilan rekursi polos dan $P(n)$ pengurungan pencarian menyeluruh: *dari waktu eksponensial ke polinomial*, seluruh pokok bab ini dalam lima kata.

Mari kita buktikan kedua batasnya, bukan sekadar menyatakannya.

Batas atasnya. Algoritmanya punya tiga gelung bersarang. Gelung luar atas l berjalan paling banyak n kali. Gelung tengah atas i berjalan paling banyak n kali. Gelung dalam atas k berjalan

paling banyak n kali. Tiap iterasi terdalam mengerjakan kerja $O(1)$ — dua penelusuran tabel, dua perkalian, dua penjumlahan, satu pembandingan, dan mungkin dua penyimpanan. Karena itu $O(n \cdot n \cdot n \cdot 1) = O(n^3)$.

Batas bawahnya. Hitunglah dengan tepat. Untuk l tertentu, gelung tengahnya berjalan $n - l + 1$ kali, dan untuk masing-masingnya gelung dalamnya berjalan $j - i = l - 1$ kali. Jadi banyaknya iterasi terdalam secara tepat adalah

$$\sum_{l=2}^{n} (n - l + 1)\,(l - 1).$$

Substitusikan $t = l - 1$, sehingga t berjalan dari 1 sampai $n - 1$ dan $n - l + 1 = n - t$:

$$\sum_{t=1}^{n-1} (n - t)\ t \ = \ n \sum_{t=1}^{n-1} t \ - \ \sum_{t=1}^{n-1} t^2 \ = \ n \cdot \frac{(n-1)n}{2} \ - \ \frac{(n-1)n(2n-1)}{6}.$$

Menaruh kedua suku di atas penyebut 6 memberikan Persamaan (7.12):

$$\begin{aligned} &= \frac{3n^2(n-1) - (n-1)n(2n-1)}{6} = \frac{n(n-1)[3n - (2n-1)]}{6} \\ &= \frac{n(n-1)(n+1)}{6} = \frac{n^3 - n}{6}. \end{aligned} \tag{7.12}$$

Jadi pernyataan terdalamnya dieksekusi persis $(n^3 - n)/6$ kali — sebuah rumus bersih yang pantas diingat. Untuk $n = 4$ itu berarti $(64 - 4)/6 = 10$ iterasi, dan memang penelusuran §7.6.8 menilai persis sepuluh calon belahan: $1 + 1 + 1$ untuk $l = 2$, $2 + 2$ untuk $l = 3$, dan 3 untuk $l = 4$. Karena $(n^3 - n)/6 = \Theta(n^3)$, waktu jalannya adalah $\Theta(n^3)$, bukan sekadar $O(n^3)$.

Ruangnya. Dua tabel berukuran $n \times n$, jadi $\Theta(n^2)$ kata. Sebagaimana tampak pada **Pseudokode 7.5**, hanya segitiga atasnya yang pernah dipakai — kira-kira $n^2/2$ sel masing-masing — sehingga implementasi yang cermat dapat memaruh tetapannya, tetapi orde pertumbuhannya tetap $\Theta(n^2)$.

> **INTI BERITANYA.** *Brute force* memeriksa $P(n) = \Omega(4^n/n^{3/2})$ pengelompokan tanda kurung. *Dynamic programming* memeriksa $(n^3 - n)/6$ calon belahan. Pada $n = 25$ itu berarti 1 289 904 147 324 melawan 2 600 — faktor kira-kira setengah miliar. Inilah yang dibeli ungkapan "dari waktu eksponensial ke polinomial".

7.6.11 Program lengkapnya

Kode 7.4 mengimplementasikan **Matrix-Chain-Order** dan **Print-Optimal-Parens** persis sebagaimana diberikan **Pseudokode 7.5**, sambil mencetak setiap calon belahan saat ia dinilai, supaya jejak mesinnya dapat dibandingkan baris demi baris dengan penelusuran tangan di §7.6.8.

Kode 7.4 **Matrix-Chain-Order** dan **Print-Optimal-Parens** dalam C++, lengkap dengan jejak penuh setiap calon belahan. Kompilasi dengan `g++ -std=c++17 -O2 -o mcm mcm.cpp` lalu jalankan `./mcm`.

```
// mcm.cpp - Matrix-Chain-Order persis seperti dinyatakan di Subbab 7.6.7, dengan jejak
lengkap.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <climits>
using namespace std;

int n;                        // banyaknya matriks
vector<long long> d;          // d[0..n]: A_i berukuran d[i-1] x d[i]
vector<vector<long long>> M; // M[i][j] = biaya termurah untuk A_i..A_j
vector<vector<int>> c;        // c[i][j] = k yang mencapainya

void matrixChainOrder(bool trace) {
    M.assign(n + 1, vector<long long>(n + 1, 0));
    c.assign(n + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= n; ++i) M[i][i] = 0;             // satu matriks: tidak ada yang
dikalikan
    for (int l = 2; l <= n; ++l)                          // l = panjang deretan
        for (int i = 1; i <= n - l + 1; ++i) {
            int j = i + l - 1;
            M[i][j] = LLONG_MAX;
            for (int k = i; k <= j - 1; ++k) {
                long long q = M[i][k] + M[k+1][j] + d[i-1]*d[k]*d[j];
                if (trace)
                    cout << "  l=" << l << "  M[" << i << "," << j << "]  k=" << k
                         << ":  M[" << i << "," << k << "]=" << M[i][k]
                         << " + M[" << k+1 << "," << j << "]=" << M[k+1][j]
                         << " + " << d[i-1] << "*" << d[k] << "*" << d[j]
                         << "=" << d[i-1]*d[k]*d[j] << "  ->  " << q << "\n";
                if (q < M[i][j]) { M[i][j] = q; c[i][j] = k; }
            }
            if (trace) cout << "  => M[" << i << "," << j << "] = " << M[i][j]
                            << ", c[" << i << "," << j << "] = " << c[i][j] << "\n\n";
        }
}

string printOptimalParens(int i, int j) {                 // Print-Optimal-Parens dari
Subbab 7.6.9
    if (i == j) return "A" + to_string(i);
    return "(" + printOptimalParens(i, c[i][j])
               + printOptimalParens(c[i][j] + 1, j) + ")";
}
```

Kode 7.4 (Lanjutan) *Matrix-Chain-Order dan Print-Optimal-Parens dalam C++, lengkap dengan jejak penuh setiap calon belahan. Kompilasi dengan g++ -std=c++17 -O2 -o mcm mcm.cpp lalu jalankan ./mcm.*

```
void showTables() {
    cout << "  M table (blank below the diagonal):\n      ";
    for (int j = 1; j <= n; ++j) cout << setw(8) << ("j=" + to_string(j));
    cout << "\n";
    for (int i = 1; i <= n; ++i) {
        cout << "  i=" << i << " ";
        for (int j = 1; j <= n; ++j) cout << setw(8) << (j < i ? string(".") :
to_string(M[i][j]));
        cout << "\n";
    }
    cout << "\n  c table (only j > i is meaningful):\n      ";
    for (int j = 1; j <= n; ++j) cout << setw(8) << ("j=" + to_string(j));
    cout << "\n";
    for (int i = 1; i <= n; ++i) {
        cout << "  i=" << i << " ";
        for (int j = 1; j <= n; ++j) cout << setw(8) << (j <= i ? string("-") :
to_string(c[i][j]));
        cout << "\n";
    }
}

int main() {
    // ---- Contoh di Subbab 7.6.8: d = [10, 20, 3, 5, 30] ----
    d = {10, 20, 3, 5, 30};
    n = (int)d.size() - 1;
    cout << "CHAIN 1: d = [10, 20, 3, 5, 30]  (A1 10x20, A2 20x3, A3 3x5, A4 5x30)\n\n";
    matrixChainOrder(true);
    showTables();
    cout << "\n  optimal cost      = " << M[1][n] << "\n";
    cout << "  optimal grouping  = " << printOptimalParens(1, n) << "\n";

    // ---- Contoh yang lebih awal di Subbab 7.6.3 ----
    d = {30, 1, 40, 10, 25};
    n = (int)d.size() - 1;
    cout << "\n\nCHAIN 2: d = [30, 1, 40, 10, 25]  (A 30x1, B 1x40, C 40x10, D 10x25)\n";
    matrixChainOrder(false);
    showTables();
    cout << "\n  optimal cost      = " << M[1][n] << "\n";
    cout << "  optimal grouping  = " << printOptimalParens(1, n) << "\n";
    return 0;
}
```

Keluarannya, selengkapnya:

```
CHAIN 1: d = [10, 20, 3, 5, 30]  (A1 10x20, A2 20x3, A3 3x5, A4 5x30)

  l=2  M[1,2]  k=1:  M[1,1]=0 + M[2,2]=0 + 10*20*3=600  ->  600
  => M[1,2] = 600, c[1,2] = 1

  l=2  M[2,3]  k=2:  M[2,2]=0 + M[3,3]=0 + 20*3*5=300  ->  300
  => M[2,3] = 300, c[2,3] = 2

  l=2  M[3,4]  k=3:  M[3,3]=0 + M[4,4]=0 + 3*5*30=450  ->  450
  => M[3,4] = 450, c[3,4] = 3

  l=3  M[1,3]  k=1:  M[1,1]=0 + M[2,3]=300 + 10*20*5=1000  ->  1300
  l=3  M[1,3]  k=2:  M[1,2]=600 + M[3,3]=0 + 10*3*5=150  ->  750
  => M[1,3] = 750, c[1,3] = 2

  l=3  M[2,4]  k=2:  M[2,2]=0 + M[3,4]=450 + 20*3*30=1800  ->  2250
  l=3  M[2,4]  k=3:  M[2,3]=300 + M[4,4]=0 + 20*5*30=3000  ->  3300
  => M[2,4] = 2250, c[2,4] = 2

  l=4  M[1,4]  k=1:  M[1,1]=0 + M[2,4]=2250 + 10*20*30=6000  ->  8250
  l=4  M[1,4]  k=2:  M[1,2]=600 + M[3,4]=450 + 10*3*30=900  ->  1950
  l=4  M[1,4]  k=3:  M[1,3]=750 + M[4,4]=0 + 10*5*30=1500  ->  2250
  => M[1,4] = 1950, c[1,4] = 2

  M table (blank below the diagonal):
           j=1     j=2     j=3     j=4
  i=1        0     600     750    1950
  i=2        .       0     300    2250
  i=3        .       .       0     450
  i=4        .       .       .       0

  c table (only j > i is meaningful):
           j=1     j=2     j=3     j=4
  i=1        -       1       2       2
  i=2        -       -       2       2
  i=3        -       -       -       3
  i=4        -       -       -       -

  optimal cost      = 1950
  optimal grouping  = ((A1A2)(A3A4))


CHAIN 2: d = [30, 1, 40, 10, 25]  (A 30x1, B 1x40, C 40x10, D 10x25)
  M table (blank below the diagonal):
           j=1     j=2     j=3     j=4
  i=1        0    1200     700    1400
  i=2        .       0     400     650
  i=3        .       .       0   10000
  i=4        .       .       .       0

  c table (only j > i is meaningful):
           j=1     j=2     j=3     j=4
  i=1        -       1       1       1
  i=2        -       -       2       3
  i=3        -       -       -       3
  i=4        -       -       -       -

  optimal cost      = 1400
  optimal grouping  = (A1((A2A3)A4))
```

Bandingkan sepuluh baris jejak mesin itu dengan sepuluh calon belahan yang dihitung dengan tangan di §7.6.8: 600; 300; 450; 1 300 dan 750; 2 250 dan 3 300; 8 250, 1 950, dan 2 250. Semuanya cocok. Kedua tabelnya cocok dengan **Tabel 7.3** dan **Tabel 7.4** sel demi sel, dan rekonstruksinya cocok dengan penyusuran tangan di §7.6.9.

Deretan 2 adalah bonus: ia adalah contoh dari **Gambar 7.9**, dan programnya mengukuhkan bahwa pengurungan ketiga yang harganya dihitung dengan tangan di §7.6.3, $A((BC)D)$ seharga 1 400, bukan sekadar lebih murah daripada dua lainnya yang ditampilkan — ia adalah optimum global atas kelimanya.

7.7 Memoization: Mempertahankan Rekursinya

7.7.1 Lookup-Chain

Pengisian *bottom-up* itu efisien, tetapi ia punya ongkosnya sendiri: Anda harus menemukan urutan penghitungan yang benar lalu mengkodekannya ke dalam gelung-gelungnya. Untuk deretan matriks urutan itu adalah "menurut panjang yang membesar", yang memerlukan satu paragraf pembenaran. Untuk rekurensi yang lebih rumit, urutan itu bisa sungguh-sungguh sulit dilihat.

Memoization menghindari persoalannya sama sekali. Rekursinya menemukan urutan yang benar dengan sendirinya — ia sekadar meminta apa yang ia butuhkan, dan apa pun yang ia minta akan terhitung sebelum jawabannya kembali. **Pseudokode 7.7** memberikan versi itu.

Pseudokode 7.7 Lookup-Chain: rekurensi deretan matriks yang ditulis sebagai rekursi, dengan baris 1 memotong langsung setiap submasalah yang sudah terselesaikan. Seluruh entri M diinisialisasi ke ∞ sebelumnya, dan ∞ sekaligus berperan sebagai penanda "belum terhitung".

```
LOOKUP-CHAIN(d, i, j):
 1.  if M[i,j] < ∞ then
 2.      return M[i,j]                          // sudah pernah diselesaikan: pakai lagi
 3.  if i = j then
 4.      M[i,j] ← 0
 5.  else for k ← i to j − 1
 6.      q ← LOOKUP-CHAIN(d, i, k) + LOOKUP-CHAIN(d, k+1, j) + d[i−1]·d[k]·d[j]
 7.      if q < M[i,j] then
 8.          M[i,j] ← q
 9.  return M[i,j]
```

Baris 1, yang diulang pada **Pseudokode 7.8**, adalah seluruh beda antara program ini dan bencana eksponensial:

Pseudokode 7.8 Baris pertama **Lookup-Chain**: kembalikan jawaban tersimpan alih-alih menghitungnya ulang.

```
1  if M[i,j] < ∞ then
2      return M[i,j]
```

Jika entrinya sudah ditulis, kembalikan. Kalau tidak, jatuh terus ke bawah, hitung dengan cara yang lambat, simpan, lalu kembalikan. Nilai *sentinel* ∞ melayani dua tujuan sekaligus: ia berarti "belum terhitung" dan ia juga kalah dalam setiap pembandingan, sehingga tidak diperlukan penanganan khusus ketika minimumnya diambil pertama kali.

> **SENTINEL HARUS MUSTAHIL.** Memakai ∞ sebagai "belum terhitung" bekerja di sini hanya karena tidak ada biaya sungguhan yang pernah bernilai ∞. Jika Anda memilih *sentinel* yang mungkin disamai jawaban sungguhan — katakanlah 0, padahal jawaban sungguhan bernilai 0 itu mungkin — maka tabel memonya akan diam-diam menghitung ulang entri-entri itu selamanya, dan algoritma "ber-*memoization*" Anda tetap eksponensial. Larik boolean terpisah `computed[i][j]` selalu aman dan hanya berbiaya satu bit per sel.

Waktu jalan **Lookup-Chain** tetap $\Theta(n^3)$ yang sama, lewat trik pembukuan §7.4.1: ada $\Theta(n^2)$ entri tabel, mengisi satu entri berbiaya $O(n)$ (gelung atas k, dengan panggilan rekursifnya dibebankan pada entrinya masing-masing), sehingga totalnya $\Theta(n^2) \cdot O(n) = O(n^3)$.

7.7.2 Diukur: rekursi polos, ber-memoization, bottom-up

Kode 7.5 mendudukkan ketiganya pada pijakan yang sama — rekurensi yang sama, deretan acak yang sama, mesin yang sama — dan sekaligus menghitung cacah Catalan dari §7.6.4.

Kode 7.5 Rekurensi deretan matriks yang sama diimplementasikan tiga cara — rekursi polos, rekursi ber-*memoization*, tabel *bottom-up* — saling diperiksa satu sama lain dan diukur waktunya. Kompilasi dengan `g++ -std=c++17 -O2 -o mcm_compare mcm_compare.cpp`.

```
// mcm_compare.cpp - tiga implementasi rekurensi yang sama, diukur waktunya berdampingan,
// ditambah cacah banyaknya pengelompokan tanda kurung (bilangan Catalan).
#include <iostream>
#include <iomanip>
#include <vector>
#include <chrono>
#include <climits>
#include <random>
using namespace std;
using namespace std::chrono;

int n;
vector<long long> d;
long long calls;

// (a) Rekurensinya diterjemahkan langsung menjadi rekursi. Tanpa tabel.
long long plain(int i, int j) {
    ++calls;
    if (i == j) return 0;
    long long best = LLONG_MAX;
    for (int k = i; k < j; ++k)
        best = min(best, plain(i, k) + plain(k + 1, j) + d[i-1]*d[k]*d[j]);
    return best;
}

// (b) Rekursi yang sama dengan tabel jawaban yang diingat: Lookup-Chain.
vector<vector<long long>> Mm;
long long lookupChain(int i, int j) {
    ++calls;
    if (Mm[i][j] != LLONG_MAX) return Mm[i][j];   // sudah pernah diselesaikan: kembalikan
nilai tersimpannya
    if (i == j) return Mm[i][j] = 0;
    long long best = LLONG_MAX;
    for (int k = i; k < j; ++k)
        best = min(best, lookupChain(i,k) + lookupChain(k+1,j) + d[i-1]*d[k]*d[j]);
    return Mm[i][j] = best;
}
```

Kode 7.5 (Lanjutan) *Rekurensi deretan matriks yang sama diimplementasikan tiga cara — rekursi polos, rekursi ber-memoization, tabel bottom-up — saling diperiksa satu sama lain dan diukur waktunya. Kompilasi dengan g++ -std=c++17 -O2 -o mcm_compare mcm_compare.cpp.*

```
// (c) Bottom-up: Matrix-Chain-Order.
long long bottomUp() {
    vector<vector<long long>> M(n+2, vector<long long>(n+2, 0));
    for (int l = 2; l <= n; ++l)
        for (int i = 1; i <= n - l + 1; ++i) {
            int j = i + l - 1; M[i][j] = LLONG_MAX;
            for (int k = i; k < j; ++k)
                M[i][j] = min(M[i][j], M[i][k] + M[k+1][j] + d[i-1]*d[k]*d[j]);
        }
    return M[1][n];
}

int main() {
    // Ada berapa cara deretan n matriks dapat dikurungkan? P(1)=1,
    // P(n) = jumlah atas k dari P(k)P(n-k): bilangan Catalan.
    cout << "n :  parenthesizations P(n)\n";
    vector<double> P(26, 0); P[1] = 1;
    for (int m = 2; m <= 25; ++m) for (int k = 1; k < m; ++k) P[m] += P[k]*P[m-k];
    for (int m : {1,2,3,4,5,6,10,15,20,25})
        cout << setw(3) << m << " :  " << fixed << setprecision(0) << P[m] << "\n";

    cout << "\nSame answer, three implementations, on random chains:\n\n";
    cout << left << setw(5) << "n" << setw(12) << "answer"
         << setw(14) << "plain calls" << setw(12) << "plain ms"
         << setw(14) << "memo calls" << setw(12) << "memo ms"
         << setw(12) << "bottom ms" << "\n";
    mt19937 rng(12345);
    for (int sz : {4, 8, 12, 15, 17, 60, 200}) {
        n = sz;
        d.assign(n + 1, 0);
        for (int i = 0; i <= n; ++i) d[i] = 1 + rng() % 50;

        long long ansB; double tb;
        { auto t0 = steady_clock::now(); ansB = bottomUp();
          tb = duration<double,milli>(steady_clock::now()-t0).count(); }

        long long ansM; double tm; long long memoCalls;
        { Mm.assign(n+2, vector<long long>(n+2, LLONG_MAX)); calls = 0;
          auto t0 = steady_clock::now(); ansM = lookupChain(1, n);
          tm = duration<double,milli>(steady_clock::now()-t0).count(); memoCalls = calls; }
```

Kode 7.5 (Lanjutan) *Rekurensi deretan matriks yang sama diimplementasikan tiga cara — rekursi polos, rekursi ber-memoization, tabel bottom-up — saling diperiksa satu sama lain dan diukur waktunya. Kompilasi dengan g++ -std=c++17 -O2 -o mcm_compare mcm_compare.cpp.*

```
        cout << left << setw(5) << n << setw(12) << ansB;
        if (n <= 17) {                              // melewati ini versi polosnya tidak
punya harapan
            calls = 0;
            auto t0 = steady_clock::now(); long long ansP = plain(1, n);
            double tp = duration<double,milli>(steady_clock::now()-t0).count();
            if (ansP != ansB) { cout << "MISMATCH\n"; return 1; }
            cout << setw(14) << calls << setw(12) << fixed << setprecision(3) << tp;
        } else cout << setw(14) << "(not run)" << setw(12) << "-";
        if (ansM != ansB) { cout << "MISMATCH\n"; return 1; }
        cout << setw(14) << memoCalls << setw(12) << fixed << setprecision(3) << tm
             << setw(12) << tb << "\n";
    }
    return 0;
}
```

Keluarannya:

```
n :  parenthesizations P(n)
  1 :  1
  2 :  1
  3 :  2
  4 :  5
  5 :  14
  6 :  42
 10 :  4862
 15 :  2674440
 20 :  1767263190
 25 :  1289904147324

Same answer, three implementations, on random chains:

n    answer      plain calls   plain ms    memo calls    memo ms     bottom ms
4    15596       27            0.000       21            0.000       0.003
8    3839        2187          0.006       169           0.001       0.001
12   16508       177147        0.501       573           0.002       0.003
15   62862       4782969       12.033      1121          0.004       0.004
17   14523       43046721      109.037     1633          0.005       0.006
60   44364       (not run)     -           71981         0.208       0.076
200  143061      (not run)     -           2666601       6.834       1.601
```

Tabel pertamanya mengukuhkan §7.6.4 dengan tepat: $P(4) = 5$, $P(5) = 14$, $P(25) = 1\,289\,904\,147\,324$.

Tabel kedua pantas dipelajari kolom demi kolom.

Cacah panggilan rekursi polosnya persis 3^{n-1}. Periksa: $3^3 = 27$ untuk $n = 4$; $3^7 = 2\,187$ untuk $n = 8$; $3^{11} = 177\,147$ untuk $n = 12$; $3^{16} = 43\,046\,721$ untuk $n = 17$. Itu bukan kebetulan. Tuliskan $C(L)$ untuk banyaknya panggilan pada subderetan berpanjang L; rekurensi panggilan $C(i,j) = 1 + \sum_{k=i}^{j-1}(C(i,k) + C(k+1,j))$ menjadi $C(L) = 1 + 2\sum_{t=1}^{L-1} C\,(t)$, yang memberikan $C(1) = 1$, $C(2) = 3$, $C(3) = 9$ dan, dengan induksi, $C(L) = 3^{L-1}$. Untuk setiap ukuran dalam tabel ini, angka itu bahkan melebihi banyaknya pengelompokan tanda kurung — 43 juta panggilan melawan $P(17) = 35\,357\,670$ pengurungan — sebab rekursi polosnya menurunkan ulang subderetan bersama sekali untuk tiap belahan yang melingkupinya. (Di atas kira-kira $n = 20$, bilangan Catalan, yang tumbuh seperti 4^n, menyalip 3^{n-1}; keduanya sama-sama tanpa harapan.)

Cacah panggilan versi ber-*memoization* adalah polinomial. 1 633 panggilan pada $n = 17$ melawan 43 juta. Pada $n = 200$, *memoization* membuat 2 666 601 panggilan — dan $2\,666\,601 = (200^3 - 200)/3 + 1$, persis seperti yang diramalkan pembukuannya: masing-masing dari $(n^3 - n)/6$ triple (i,j,k) membuat dua panggilan, satu untuk tiap paruh, dan panggilan terluar menambah satu.

Waktunya: pada $n = 17$, versi polosnya memakan 109.037 ms, versi ber-*memoization* memakan 0.005 ms. *Memoization* kira-kira dua puluh dua ribu kali lebih cepat, hanya dari satu baris kode tambahan.

Dan imbal-tukarnya, yang merupakan pokok subbab ini. Pada $n = 60$ versi ber-*memoization* memakan 0.208 ms sedangkan *bottom-up* memakan 0.076 ms; pada $n = 200$ jaraknya 6.834 ms melawan 1.601 ms. *Bottom-up* secara konsisten kira-kira empat kali lebih cepat. Keduanya $\Theta(n^3)$. Bedanya seluruhnya terletak pada faktor tetapan: biaya panggilan fungsi, bingkai tumpukan, dan pola akses memori yang melompat-lompat di dalam tabel alih-alih menyapunya.

7.7.3 Imbal-tukarnya, dinyatakan

Tabel 7.5 *Memoization* melawan *bottom-up*. Keduanya memberi waktu jalan asimtotik yang sama; segala yang berbeda ada di dalam tabel ini.

	***Memoization* (*top-down*)**	**Tabel *bottom-up***
Urutan penghitungan	Ditemukan sendiri oleh rekursinya	Anda harus menentukannya dan mengkodekannya ke dalam gelung
Bentuk kode	Rekurensinya, nyaris disalin begitu saja	Gelung yang bisa jadi tidak menyerupai rekurensinya
Submasalah yang diselesaikan	Hanya yang benar-benar terjangkau	Semuanya, terjangkau atau tidak
Faktor tetapan	Lebih tinggi (panggilan, bingkai, akses tersebar)	Lebih rendah (gelung rapat, akses menyapu)
Tumpukan	Kedalamannya sebanding dengan kedalaman rekursi; bisa meluap	Tidak ada
Penghematan ruang	Sulit — seluruh tabelnya harus bertahan	Mudah — baris/diagonal lama boleh dibuang

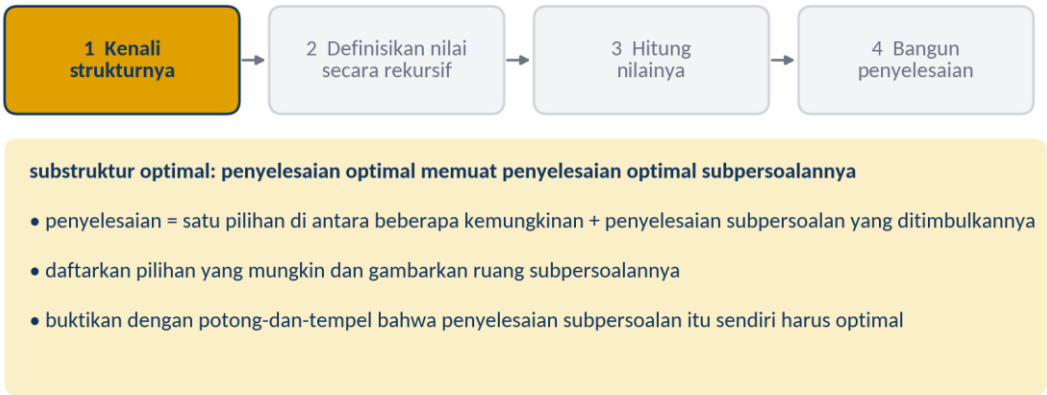
MANA YANG DIPAKAI. Pakailah **memoization** ketika rekurensinya canggung untuk diurutkan, ketika hanya sebagian kecil submasalahnya yang benar-benar akan diperlukan, atau ketika Anda sedang menulis versi pertama dan ingin ia jelas-jelas benar. Pakailah **bottom-up** ketika setiap submasalahnya toh akan diselesaikan, ketika faktor tetapannya penting, ketika rekursinya akan terlalu dalam bagi tumpukan, atau ketika Anda ingin memanfaatkan penghematan ruang seperti pada §7.4.3 dan §7.9.10. Kalau ragu, tulislah versi ber-*memoization* lebih dulu, periksa, lalu ubah menjadi *bottom-up* dan periksa keduanya sepakat — dan persis itulah yang dikerjakan **Kode 7.5**.

7.8 Empat Langkah Merancang Algoritma Dynamic Programming

Kita sekarang sudah mengerjakan hal yang sama dua kali, untuk Fibonacci dan untuk deretan matriks. Kini saatnya menyarikan metode umumnya, dalam empat langkah, sebab metode itu adalah resep yang dapat Anda terapkan pada persoalan yang belum pernah Anda lihat. Kleinberg dan Tardos [5, bab 6] memberikan penyajian pelengkap atas resep yang sama, yang dibangun di sekitar himpunan contoh yang berbeda.

7.8.1 Langkah 1: cirikan struktur solusi optimalnya

Gambar 7.15 adalah langkah pertamanya.



Gambar 7.15 Langkah 1 resepnya: tunjukkan adanya *optimal substructure*. Sebuah solusi adalah satu pilihan dari sejumlah kemungkinan ditambah solusi bagi submasalah yang diciptakan pilihan itu; lalu tunjukkan, dengan potong-dan-tempel, bahwa solusi submasalah itu sendiri haruslah optimal.

Langkah itu terpecah menjadi tiga pertanyaan yang harus Anda jawab berurutan.

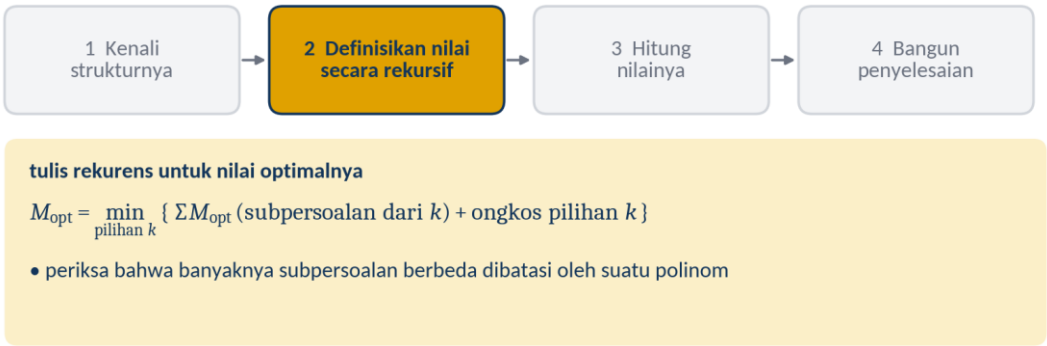
Apa saja pilihan yang mungkin? Pada aras teratas, keputusan apa yang diambil algoritmanya? Untuk deretan matriks, pilihannya adalah *di mana kurung terluar ditaruh* — titik belah k. Untuk *longest common subsequence* di §7.9, pilihannya nanti adalah *apa yang harus dilakukan terhadap karakter terakhir tiap untai*. Menyebut nama pilihannya adalah tindakan kreatifnya; segala sesudah itu bersifat mekanis.

Submasalah apa yang diciptakan sebuah pilihan? Sesudah pilihan itu diambil, instans lebih kecil apa dari persoalan yang sama yang masih tersisa? Inilah yang mendefinisikan **ruang submasalah**, dan Anda harus mencirikannya dengan tepat, sebab tabel yang akan Anda bangun punya satu sel per submasalah. Deretan matriks: subderetan bersinambung (i, j), jadi $\Theta(n^2)$ sel. LCS: pasangan prefiks (i, j), jadi $\Theta(mn)$ sel.

Apakah solusi bagiannya niscaya optimal? Buktikan, dengan argumen potong-dan-tempel §7.5.3. Jangan lewati langkah ini. Jika jawabannya tidak, *dynamic programming* tidak berlaku dan Anda harus mencari metode lain.

7.8.2 Langkah 2: definisikan nilai solusi optimalnya secara rekursif

Gambar 7.16 adalah langkah keduanya.



Gambar 7.16 Langkah 2: tuliskan rekurensinya. Bentuk umumnya adalah $M_{opt} = \min_{\text{pilihan } k}\{(\text{jumlah } M_{opt} \text{ atas submasalah yang dihasilkan } k) + (\text{biaya memilih } k)\}$ — lalu tunjukkan bahwa banyaknya instans submasalah yang berbeda terbatasi oleh sebuah polinomial.

Templat pada gambar itu pantas dituliskan selengkapnya, sebab setiap program dinamis dalam buku ini adalah instansnya:

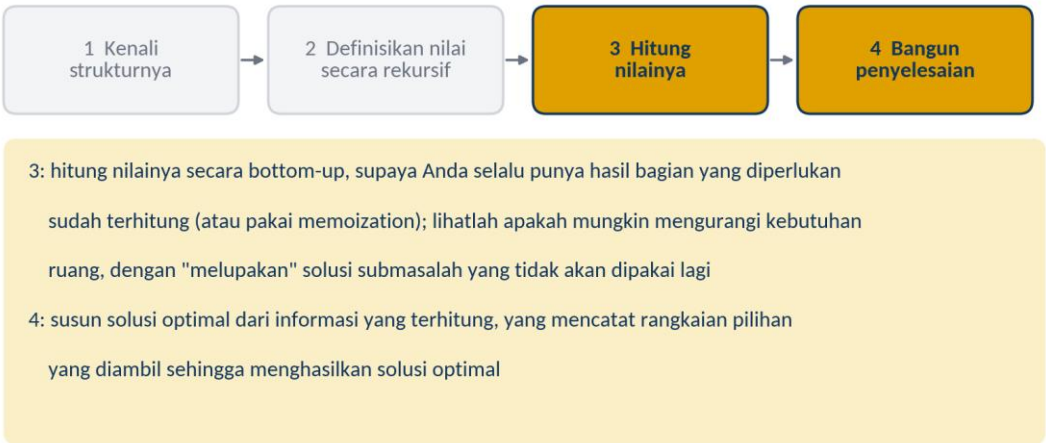
$$M_{\text{opt}}(\text{persoalan}) = \min_{\text{semua pilihan } k} \left\{ \sum_{\text{submasalah } s \text{ yang diciptakan } k} M_{\text{opt}}(s) + \text{biaya}(k) \right\}. \tag{7.13}$$

Periksa rekurensi deretan matriks Persamaan (7.10) terhadap templat Persamaan (7.13): pilihannya adalah titik belah k; submasalah yang diciptakan pilihan k adalah (i, k) dan $(k + 1, j)$; biaya mengambil pilihan itu adalah $d_{i-1}d_k d_j$. Ia pas benar.

Paruh kedua **Gambar 7.16** adalah pemeriksaan kewarasan yang tidak boleh dilewati: **tunjukkan bahwa banyaknya instans submasalah yang berbeda terbatasi oleh sebuah polinomial.** Jika tidak — jika submasalah Anda ternyata, katakanlah, seluruh 2^n himpunan bagian dari masukannya — maka tabelnya punya sel sebanyak eksponensial dan mengisinya tidak lebih baik daripada *brute force*. Banyaknya submasalah yang polinomial persis merupakan syarat yang disebut **overlapping subproblems** (submasalah yang tumpang tindih, §7.10), dan bersama *optimal substructure* ia merupakan salah satu dari dua ciri khas persoalan *dynamic programming*.

7.8.3 Langkah 3 dan 4: hitung nilainya, lalu susun solusinya

Gambar 7.17 memberikan dua langkah terakhirnya sekaligus.



Gambar 7.17 Langkah 3 dan 4: hitung nilai optimalnya secara *bottom-up* supaya hasil bagian yang diperlukan selalu sudah siap (atau pakai *memoization*), carilah peluang mengurangi ruangnya dengan melupakan submasalah yang tidak akan dipakai lagi, dan akhirnya susunlah sebuah solusi optimal dari rangkaian pilihan yang tercatat.

Langkah 3: hitung nilai sebuah solusi optimal, secara *bottom-up*. Frasa yang perlu diingat adalah *supaya Anda selalu punya hasil bagian yang diperlukan sudah terhitung*. Itulah seluruh syarat atas urutan gelung Anda: ketika Anda menghitung sebuah sel, segala sesuatu yang menjadi sandarannya harus sudah selesai. Jika memilih urutan itu menyakitkan, kalimat dalam kurung *(atau pakai memoization)* adalah pintu darurat Anda.

Gambar 7.17 lalu menambahkan optimasi ruangnya: *lihatlah apakah mungkin mengurangi kebutuhan ruang, dengan "melupakan" solusi submasalah yang tidak akan dipakai lagi.* Kita sudah melakukannya untuk Fibonacci di §7.4.3, beranjak dari $\Theta(n)$ ke $\Theta(1)$, dan kita akan melakukannya untuk LCS di §7.9.10, beranjak dari $\Theta(mn)$ ke $\Theta(\min(m, n))$.

Langkah 4: susun sebuah solusi optimal dari informasi yang terhitung. Kata-kata kuncinya adalah *yang mencatat rangkaian pilihan yang diambil sehingga menghasilkan solusi optimal*. Inilah

tabel c pada §7.6.9 dan tabel b pada §7.9.8. Langkah 4 bersifat opsional — lewati saja jika Anda hanya memerlukan nilainya — dan melewatinya sering kali justru yang memungkinkan penghematan ruang langkah 3, sebab tabel rekonstruksi tidak bisa dilupakan sambil jalan.

7.8.4 Resepnya, dalam satu halaman

EMPAT LANGKAHNYA. 1. Cirikan struktur sebuah solusi optimal. Sebut nama pilihannya; sebut nama submasalah yang diciptakan pilihan itu; buktikan dengan potong-dan-tempel bahwa solusi bagiannya haruslah optimal. **2. Definisikan secara rekursif** nilai sebuah solusi optimal: $M_{\text{opt}} = \min_k \{\sum M_{\text{opt}}(\text{submasalah}) + \text{biaya}(k)\}$, lengkap dengan kasus basisnya. Periksa bahwa cacah submasalahnya polinomial. **3. Hitung** nilainya, secara *bottom-up* dalam urutan yang menghormati kebergantungannya (atau secara *top-down* dengan *memoization*). Pertimbangkan untuk melupakan apa yang tidak lagi Anda perlukan. **4. Susun** sebuah solusi optimal dari tabel pilihan yang tercatat — bila Anda memerlukan solusinya dan bukan sekadar nilainya.

KEGAGALAN YANG PALING SERING. Para pemula melompat langsung ke langkah 3 — mereka mulai menggambar tabel sebelum menyebut nama submasalahnya. Tabel adalah hal *terakhir* yang dipikirkan, bukan yang pertama. Jika Anda belum sanggup menyelesaikan kalimat "submasalahnya adalah: diberikan …, carilah … yang terbaik", Anda belum siap menulis satu baris kode pun.

7.9 Longest Common Subsequence

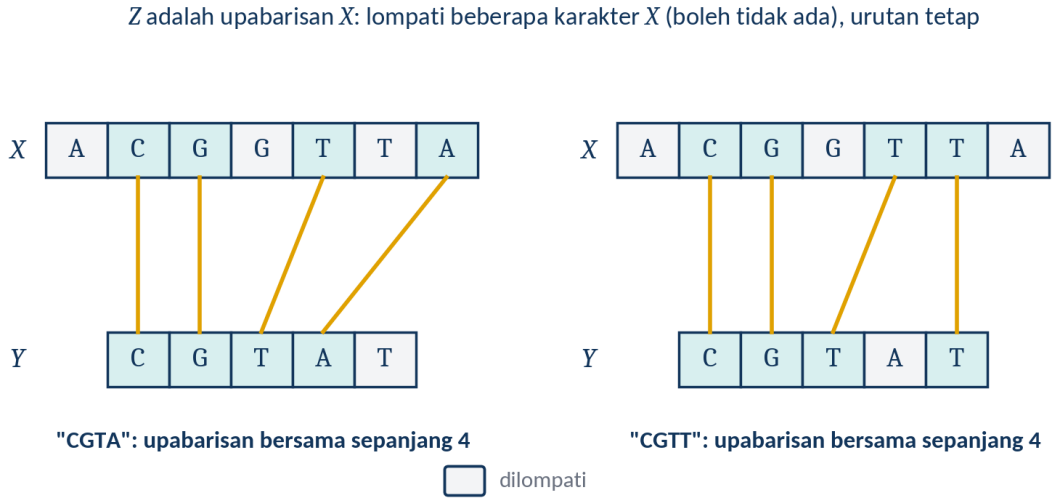
7.9.1 Mengapa orang peduli

Dua potong teks diberikan, dan kita menginginkan sebuah bilangan yang menyatakan seberapa mirip keduanya. Ada tiga kegunaan yang menonjol: membandingkan barisan DNA dalam kajian tentang bagaimana spesies berevolusi; pemeriksaan ejaan; dan, secara umum, pertanyaan apa pun yang berbentuk "berapa banyak kesamaan antara dua untai ini?". Sistem kendali versi menjawab `diff` persis dengan perhitungan ini; pendeteksi plagiarisme, kotak pencarian yang toleran salah ketik, dan studi kasus Profitina Laundry di §7.11 semuanya bersandar padanya.

Salah satu ukuran kemiripan yang baku adalah panjang **longest common subsequence** (subbarisan bersama terpanjang), dan ia adalah program dinamis buku teks — yang kedua dari dua contoh besar yang dikerjakan dalam bab ini.

7.9.2 Subsequence, didefinisikan

Gambar 7.18 memberikan definisinya.



Gambar 7.18 Definisinya: Z adalah *subsequence* dari X bila Z dapat dibangkitkan dengan melewatkan sejumlah (boleh nol) karakter X. Untuk $X = $ "ACGGTTA" dan $Y = $ "CGTAT", LCS(X, Y) adalah "CGTA" atau "CGTT". Menyelesaikan persoalannya berarti menemukan *lewatan-lewatannya*.

DEFINISI. Misalkan $X = x_1 x_2 \cdots x_m$ adalah sebuah untai. Untai $Z = z_1 z_2 \cdots z_r$ disebut **subsequence** (subbarisan) dari X bila ada indeks $i_1 < i_2 < \cdots < i_r$ dengan $z_t = x_{i_t}$ untuk setiap t. Dengan kata lain: Z diperoleh dari X dengan mencoret sejumlah (boleh nol) karakter dan mempertahankan sisanya **dalam urutan aslinya**.

SUBSEQUENCE BUKAN SUBSTRING. Substring (subuntai) haruslah bersinambung; **subsequence** tidak harus. Dalam "ACGGTTA", untai "CGG" adalah *substring* sekaligus *subsequence*, sedangkan "AGT" adalah *subsequence* (posisi 1, 3, 5) tetapi bukan *substring*. Mengacaukan keduanya adalah kesalahan paling sering pada topik ini.

DEFINISI. Z disebut **common subsequence** (subbarisan bersama) dari X dan Y bila ia *subsequence* dari keduanya. Sebuah **longest common subsequence** dari X dan Y, ditulis LCS(X, Y), adalah subbarisan bersama yang panjangnya terbesar. Perhatikan "sebuah", bukan "sang": bisa saja ada beberapa, dan semuanya berpanjang sama.

Kerjakan contoh pada **Gambar 7.18** itu dengan tangan. Dengan $X = $ "ACGGTTA" dan $Y = $ "CGTAT":

- Apakah "CGT" bersama? Di X ia ada pada posisi 2, 3, 5 (C, G, T) — menaik ✓. Di Y ia ada pada posisi 1, 2, 3 — menaik ✓. Ya, berpanjang 3.
- Apakah "CGTA" bersama? Di X: posisi 2, 3, 5, 7 ✓. Di Y: posisi 1, 2, 3, 4 ✓. Ya, berpanjang 4.
- Apakah "CGTT" bersama? Di X: posisi 2, 3, 5, 6 ✓. Di Y: posisi 1, 2, 3, 5 ✓. Ya, berpanjang 4.
- Apakah "CGTAT" bersama? Ia adalah seluruh Y. Di X kita memerlukan C, G, T, A, T pada posisi yang menaik: C di 2, G di 3, T di 5, A di 7, lalu satu T lagi sesudah posisi 7 — tidak ada. Tidak.

Jadi *longest common subsequence*-nya berpanjang 4, dan baik "CGTA" maupun "CGTT" mencapainya, persis seperti yang dikatakan **Gambar 7.18**.

7.9.3 Brute force, dan mengapa ia gagal

Algoritma yang paling kentara: daftar setiap *subsequence* dari X, uji masing-masing terhadap Y, simpan yang terpanjang yang lolos.

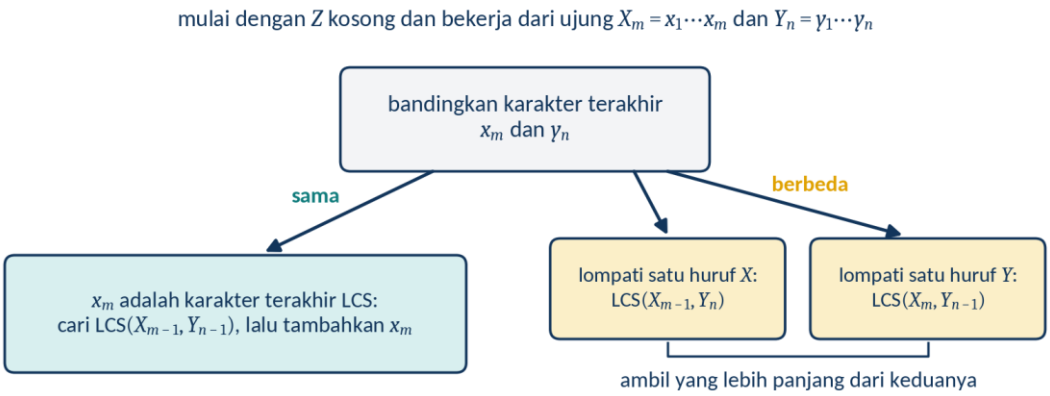
Ada berapa banyak *subsequence* dimiliki untai berpanjang m? Masing-masing dari m karakternya dipertahankan atau dicoret, secara saling bebas, jadi ada 2^m pilihan (sebagiannya menghasilkan

untai kembar, tetapi paling banyak 2^m). Memeriksa apakah sebuah calon berpanjang sampai m merupakan *subsequence* dari Y memakan $O(n)$ lewat satu sapuan. Totalnya: $O(n\,2^m)$.

Untuk dua untai berpanjang 20, itu kira-kira sejuta pemeriksaan — masih bisa bertahan. Untuk dua barisan DNA berpanjang 1000, itu 2^{1000}, sebuah bilangan berdigit 302. *Brute force* bukan pilihan, dan kita memerlukan program dinamisnya.

7.9.4 Optimal substructure, dibuktikan

Gambar 7.19 memberikan penurunannya, dan ia adalah seluruh algoritmanya dalam satu gambar.



Gambar 7.19 *Optimal substructure* milik LCS. Bekerja dari *ujung* X_m dan Y_n: jika karakter terakhirnya sama, karakter itu bergabung ke dalam jawabannya dan kita berekursi pada kedua untai yang masing-masing dipendekkan satu; jika berbeda, kita harus melewatkan satu huruf dari X atau satu huruf dari Y, dan kita memutuskan yang mana dengan membandingkan kedua panjang LCS yang dihasilkan. Pembenarannya adalah argumen potong-dan-tempel.

Pertama, notasinya. Tulis X_i untuk **prefiks** (awalan) $x_1 x_2 \cdots x_i$ dari X — yaitu i karakter pertamanya — dengan X_0 adalah untai kosong. Demikian pula Y_j. Submasalahnya nanti adalah semua pasangan prefiks.

Sekarang kedua kasusnya, beserta buktinya.

> **TEOREMA (optimal substructure milik LCS).** Misalkan $X = x_1 \cdots x_m$ dan $Y = y_1 \cdots y_n$, dan misalkan $Z = z_1 \cdots z_r$ adalah *sembarang longest common subsequence* dari X dan Y.
>
> **(1)** Jika $x_m = y_n$, maka $z_r = x_m = y_n$, dan Z_{r-1} adalah LCS dari X_{m-1} dan Y_{n-1}. **(2)** Jika $x_m \neq y_n$ dan $z_r \neq x_m$, maka Z adalah LCS dari X_{m-1} dan Y_n. **(3)** Jika $x_m \neq y_n$ dan $z_r \neq y_n$, maka Z adalah LCS dari X_m dan Y_{n-1}.

Bukti (1). Andaikan $x_m = y_n = ch$.

Pertama, $z_r = ch$. Andaikan tidak. Maka tambahkan ch ke ujung Z, menghasilkan untai berpanjang $r + 1$. Apakah itu subbarisan bersama? Di X, karakter-karakter Z menempati posisi yang semuanya paling banyak m, dan karena $z_r \neq x_m$ semuanya paling banyak $m - 1$; jadi menambahkan x_m pada posisi m menjaga indeksnya tetap menaik ✓. Argumen yang sama berlaku di Y ✓. Jadi kita punya subbarisan bersama berpanjang $r + 1$, yang bertentangan dengan Z sebagai yang terpanjang. Karena itu $z_r = ch$.

Kedua, Z_{r-1} *adalah LCS dari* X_{m-1} *dan* Y_{n-1}. Ia jelas merupakan subbarisan bersama keduanya: hapus karakter terakhir dari Z dan dari kedua untainya. Andaikan ia bukan yang *terpanjang* —

andaikan ada subbarisan bersama W dari X_{m-1} dan Y_{n-1} yang berpanjang lebih dari $r-1$. **Potong** Z_{r-1} keluar lalu **tempel** W ke dalamnya: W yang diikuti ch adalah subbarisan bersama dari X dan Y (sah, sebab indeks W semuanya $<m$ di X dan $<n$ di Y) yang berpanjang lebih dari r. Kontradiksi. ■

Bukti (2). Andaikan $x_m \neq y_n$ dan $z_r \neq x_m$. Maka tidak ada karakter Z yang memakai posisi m dari X (hanya yang terakhir yang bisa, sebab indeksnya menaik), sehingga Z adalah subbarisan bersama dari X_{m-1} dan Y. Jika ada subbarisan bersama W dari X_{m-1} dan Y yang lebih panjang, maka W juga akan menjadi subbarisan bersama dari X dan Y — setiap subbarisan dari sebuah prefiks adalah subbarisan dari keseluruhannya — dan lebih panjang daripada Z. Kontradiksi. ■

Bukti (3) adalah (2) dengan peran X dan Y dipertukarkan. ■

> **MENGAPA TEOREMA ITU LENGKAP.** Jika $x_m \neq y_n$, maka z_r tidak mungkin sama dengan x_m dan y_n sekaligus, sehingga setidaknya salah satu dari kasus (2) dan (3) berlaku. Bersama kasus (1) itu mencakup setiap kemungkinan, jadi rekurensi di bawah ini tuntas. Ketika (2) dan (3) berlaku bersamaan kita boleh mengambil yang mana pun, dan itulah sebabnya seri boleh diputus sesuka hati dan sebabnya bisa ada beberapa LCS.

7.9.5 Rekurensinya

Mengikuti langkah 2 resepnya, definisikan tabelnya:

> **DEFINISI.** Misalkan $c[i,j]$ adalah panjang sebuah *longest common subsequence* dari prefiks X_i dan Y_j. Jawaban persoalan aslinya adalah $c[m,n]$.

Gambar 7.20 memberikan rekurensinya.

$$c[i,j] = \begin{cases} 0 & \text{jika } i = 0 \text{ atau } j = 0 \\ c[i-1,j-1]+1 & \text{jika } i,j > 0 \text{ dan } x_i = y_j \\ \max(c[i,j-1],\ c[i-1,j]) & \text{jika } i,j > 0 \text{ dan } x_i \neq y_j \end{cases}$$

Gambar 7.20 Rekurensi LCS: nol bila salah satu prefiksnya kosong; satu lebih banyak daripada tetangga diagonalnya ketika kedua karakternya cocok; selain itu yang lebih besar di antara tetangga di atas dan tetangga di kirinya.

Dituliskan selengkapnya, rekurensi itu adalah Persamaan (7.14):

$$c[i,j] = \begin{cases} 0 & \text{jika } i = 0 \text{ atau } j = 0, \\ c[i-1,j-1]+1 & \text{jika } i,j > 0 \text{ dan } x_i = y_j, \\ \max\{c[i,j-1],\ c[i-1,j]\} & \text{jika } i,j > 0 \text{ dan } x_i \neq y_j. \end{cases} \tag{7.14}$$

Tiap barisnya, dengan kata-kata:

- **Baris 1 (kasus basis).** Jika salah satu untainya kosong, satu-satunya subbarisan bersamanya adalah untai kosong, berpanjang 0.
- **Baris 2 (cocok).** Jika karakter terakhirnya sama, bagian (1) teoremanya mengatakan karakter itu ada di dalam jawabannya, dan sisa jawabannya adalah LCS kedua prefiks yang dipendekkan. Karena itu satu lebih banyak daripada tetangga *diagonal*-nya.

- **Baris 3 (tak cocok).** Jika keduanya berbeda, bagian (2) dan (3) teoremanya mengatakan jawabannya adalah yang lebih baik di antara dua kemungkinan: buang karakter terakhir X (memberi tetangga *di atas*, $c[i-1, j]$), atau buang karakter terakhir Y (memberi tetangga *di kiri*, $c[i, j-1]$). Ambil yang lebih besar.

> **DUA KONVENSI PENGINDEKSAN SEKALIGUS.** Tabelnya diindeks mulai dari 0 ($c[0, j]$ dan $c[i, 0]$ adalah baris dan kolom prefiks kosong) sedangkan untainya diindeks mulai dari 1 (x_1 adalah karakter pertama). Dalam C++ sebuah `string` diindeks mulai dari 0, sehingga karakter yang dibandingkan pada posisi tabel (i, j) adalah `X[i-1]` melawan `Y[j-1]`. Setiap kutu salah-satu pada kode LCS bersarang di baris itu.

Ada berapa submasalah? Pasangan (i, j) dengan $0 \le i \le m$ dan $0 \le j \le n$: persis $(m+1)(n+1) = \Theta(mn)$. Polinomial ✓ — pemeriksaan kewarasan langkah 2 lolos.

7.9.6 Algoritmanya

Pseudokode 7.9 memberikan **LCS-Length**.

Pseudokode 7.9 LCS-Length dalam pseudokode. Baris 1–4 mengisi baris nol dan kolom nolnya. Baris 5–15 menyapu tabelnya baris demi baris, mengisi $c[i, j]$ menurut rekurensinya dan mencatat di $b[i, j]$ kasus mana di antara ketiganya yang dipakai: `"copy"`, `"skipx"`, atau `"skipy"`.

```
LCS-LENGTH(X, Y, m, n):
 1.  for i ← 1 to m
 2.      c[i,0] ← 0
 3.  for j ← 0 to n
 4.      c[0,j] ← 0
 5.  for i ← 1 to m
 6.      for j ← 1 to n
 7.          if x_i = y_j then
 8.              c[i,j] ← c[i-1,j-1] + 1
 9.              b[i,j] ← "copy"
10.          else if c[i-1,j] ≥ c[i,j-1] then
11.              c[i,j] ← c[i-1,j]
12.              b[i,j] ← "skipx"
13.          else
14.              c[i,j] ← c[i,j-1]
15.              b[i,j] ← "skipy"
16.  return c, b
```

Tabel kedua, b, adalah langkah 4 resepnya. Pada tiap selnya ia mencatat *pilihan* yang menghasilkan nilainya:

- `"copy"` — karakternya cocok; nilainya datang dari diagonal.
- `"skipx"` — keduanya tidak cocok dan nilainya datang dari atas, $c[i-1, j]$; kita melewatkan sebuah karakter dari X.
- `"skipy"` — keduanya tidak cocok dan nilainya datang dari kiri, $c[i, j-1]$; kita melewatkan sebuah karakter dari Y.

Kita menggambar ketiganya sebagai panah — ↖ untuk *copy*, ↑ untuk *skip-x*, ← untuk *skip-y* — sebab rekonstruksinya secara harfiah adalah mengikuti panah secara mundur.

Perhatikan baris 10 pseudokodenya: `else if c[i-1,j] ≥ c[i,j-1]`. Pembandingannya memakai $\ge$, bukan $>$, sehingga seri jatuh ke `skipx`. Aturan konsisten apa pun benar; yang ini adalah aturan pada **Pseudokode 7.9**, dan kita mengikutinya supaya tabel di bawah dapat direproduksi.

Urutan gelungnya di sini adalah yang sederhana: baris demi baris, kiri ke kanan. Ia bekerja sebab $c[i,j]$ hanya bersandar pada $c[i-1,j-1]$, $c[i-1,j]$, dan $c[i,j-1]$ — semuanya di baris sebelumnya, atau lebih awal di baris ini.

7.9.7 Tabel yang diisi dengan tangan

Gambar 7.21 memperlihatkan tujuan kita: tabel yang sudah terisi untuk $X = $ "ACGGTTA" dan $Y = $ "CGTAT". Inilah jalannya yang mengisi tabel itu, satu sel satu baris.

		0	1 C	2 G	3 T	4 A	5 T
0		0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1
2	C	0	↖ 1	← 1	← 1	↑ 1	↑ 1
3	G	0	↑ 1	↖ 2	← 2	← 2	← 2
4	G	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 2
5	T	0	↑ 1	↑ 2	↖ 3	← 3	↖ 3
6	T	0	↑ 1	↑ 2	↖ 3	↑ 3	↖ 4
7	A	0	↑ 1	↑ 2	↑ 3	↖ 4	↑ 4

c[7,5] = 4

penelusuran balik mengej "CGTT"

hanya panjangnya? cukup dua baris: baris $i-1$ dan baris i

Gambar 7.21 Tabel yang sudah terisi untuk $X = $ "ACGGTTA" dan $Y = $ "CGTAT": tiap sel memuat $c[i,j]$ beserta panahnya $b[i,j]$, dan sel yang diarsir adalah penelusuran balik yang mengeja "CGTT". Garis di sampingnya mengajukan pertanyaan §7.9.10: seberapa banyak ruang yang diperlukan jika kita tidak merekonstruksi subbarisannya sendiri?

Jadi $X = x_1x_2x_3x_4x_5x_6x_7 = $ A, C, G, G, T, T, A dengan $m = 7$, dan $Y = y_1y_2y_3y_4y_5 = $ C, G, T, A, T dengan $n = 5$.

Baris 0 dan kolom 0 seluruhnya nol, menurut kasus basisnya.

Baris $i = 1$ ($x_1 = $ A).

- $c[1,1]$: $y_1 = $ C, dan A $\neq$ C. Bandingkan atas $c[0,1] = 0$ dengan kiri $c[1,0] = 0$. Seri, jadi $\geq$ mengambil yang atas: $c[1,1] = 0$, $b = \uparrow$.
- $c[1,2]$: $y_2 = $ G, A $\neq$ G. Atas $c[0,2] = 0$, kiri $c[1,1] = 0$. Seri: $c[1,2] = 0$, $b = \uparrow$.
- $c[1,3]$: $y_3 = $ T, A $\neq$ T. Atas 0, kiri 0. Seri: $c[1,3] = 0$, $b = \uparrow$.
- $c[1,4]$: $y_4 = $ A, dan A $=$ A — cocok. Diagonal $c[0,3] = 0$, jadi $c[1,4] = 0 + 1 = 1$, $b = \nwarrow$.
- $c[1,5]$: $y_5 = $ T, A $\neq$ T. Atas $c[0,5] = 0$, kiri $c[1,4] = 1$. Karena $0 \geq 1$ salah, ambil yang kiri: $c[1,5] = 1$, $b = \leftarrow$.

Baris $i = 2$ ($x_2 = $ C).

- $c[2,1]$: $y_1 = $ C — cocok. Diagonal $c[1,0] = 0$, jadi $c[2,1] = 1$, $b = \nwarrow$.
- $c[2,2]$: G, tidak cocok. Atas $c[1,2] = 0$, kiri $c[2,1] = 1$. $0 \geq 1$ salah: $c[2,2] = 1$, $b = \leftarrow$.
- $c[2,3]$: T, tidak cocok. Atas $c[1,3] = 0$, kiri $c[2,2] = 1$. $c[2,3] = 1$, $b = \leftarrow$.
- $c[2,4]$: A, tidak cocok (C $\neq$ A). Atas $c[1,4] = 1$, kiri $c[2,3] = 1$. $1 \geq 1$ benar: $c[2,4] = 1$, $b = \uparrow$.
- $c[2,5]$: T, tidak cocok. Atas $c[1,5] = 1$, kiri $c[2,4] = 1$. $1 \geq 1$: $c[2,5] = 1$, $b = \uparrow$.

Baris $i = 3$ ($x_3 = $ G).

- $c[3,1]$: C, tidak cocok. Atas $c[2,1] = 1$, kiri $c[3,0] = 0$. $1 \geq 0$: $c[3,1] = 1$, $b = \uparrow$.
- $c[3,2]$: G — cocok. Diagonal $c[2,1] = 1$, jadi $c[3,2] = 2$, $b = \nwarrow$.
- $c[3,3]$: T, tidak cocok. Atas $c[2,3] = 1$, kiri $c[3,2] = 2$. $1 \geq 2$ salah: $c[3,3] = 2$, $b = \leftarrow$.
- $c[3,4]$: A, tidak cocok. Atas $c[2,4] = 1$, kiri $c[3,3] = 2$: $c[3,4] = 2$, $b = \leftarrow$.
- $c[3,5]$: T, tidak cocok. Atas $c[2,5] = 1$, kiri $c[3,4] = 2$: $c[3,5] = 2$, $b = \leftarrow$.

Baris $i = 4$ ($x_4 = $ G).

- $c[4,1]$: C, tidak cocok. Atas $c[3,1] = 1$, kiri 0. $c[4,1] = 1$, $b = \uparrow$.
- $c[4,2]$: G — cocok. Diagonal $c[3,1] = 1$: $c[4,2] = 2$, $b = \nwarrow$.
- $c[4,3]$: T, tidak cocok. Atas $c[3,3] = 2$, kiri $c[4,2] = 2$. $2 \geq 2$: $c[4,3] = 2$, $b = \uparrow$.
- $c[4,4]$: A, tidak cocok. Atas $c[3,4] = 2$, kiri $c[4,3] = 2$: $c[4,4] = 2$, $b = \uparrow$.
- $c[4,5]$: T, tidak cocok. Atas $c[3,5] = 2$, kiri $c[4,4] = 2$: $c[4,5] = 2$, $b = \uparrow$.

Baris $i = 5$ ($x_5 = $ T).

- $c[5,1]$: C, tidak cocok. Atas $c[4,1] = 1$, kiri 0: $c[5,1] = 1$, $b = \uparrow$.
- $c[5,2]$: G, tidak cocok. Atas $c[4,2] = 2$, kiri $c[5,1] = 1$. $2 \geq 1$: $c[5,2] = 2$, $b = \uparrow$.
- $c[5,3]$: T — cocok. Diagonal $c[4,2] = 2$: $c[5,3] = 3$, $b = \nwarrow$.
- $c[5,4]$: A, tidak cocok. Atas $c[4,4] = 2$, kiri $c[5,3] = 3$. $2 \geq 3$ salah: $c[5,4] = 3$, $b = \leftarrow$.
- $c[5,5]$: T — cocok. Diagonal $c[4,4] = 2$: $c[5,5] = 3$, $b = \nwarrow$.

Baris $i = 6$ ($x_6 = $ T).

- $c[6,1]$: C, tidak cocok. Atas $c[5,1] = 1$, kiri 0: $c[6,1] = 1$, $b = \uparrow$.
- $c[6,2]$: G, tidak cocok. Atas $c[5,2] = 2$, kiri $c[6,1] = 1$: $c[6,2] = 2$, $b = \uparrow$.
- $c[6,3]$: T — cocok. Diagonal $c[5,2] = 2$: $c[6,3] = 3$, $b = \nwarrow$.
- $c[6,4]$: A, tidak cocok. Atas $c[5,4] = 3$, kiri $c[6,3] = 3$. $3 \geq 3$: $c[6,4] = 3$, $b = \uparrow$.
- $c[6,5]$: T — cocok. Diagonal $c[5,4] = 3$: $c[6,5] = 4$, $b = \nwarrow$.

Baris $i = 7$ ($x_7 = $ A).

- $c[7,1]$: C, tidak cocok. Atas $c[6,1] = 1$, kiri 0: $c[7,1] = 1$, $b = \uparrow$.
- $c[7,2]$: G, tidak cocok. Atas $c[6,2] = 2$, kiri $c[7,1] = 1$: $c[7,2] = 2$, $b = \uparrow$.
- $c[7,3]$: T, tidak cocok. Atas $c[6,3] = 3$, kiri $c[7,2] = 2$. $3 \geq 2$: $c[7,3] = 3$, $b = \uparrow$.
- $c[7,4]$: A — cocok. Diagonal $c[6,3] = 3$: $c[7,4] = 4$, $b = \nwarrow$.
- $c[7,5]$: T, tidak cocok. Atas $c[6,5] = 4$, kiri $c[7,4] = 4$. $4 \geq 4$: $c[7,5] = 4$, $b = \uparrow$.

Tabel yang sudah selesai adalah **Tabel 7.6**, dan jawabannya duduk di pojok kanan bawah: $c[7,5] = 4$.

Tabel 7.6 Tabel c untuk $X = $ "ACGGTTA" dan $Y = $ "CGTAT", diisi sel demi sel di §7.9.7. Setiap entri adalah panjang sebuah LCS dari kedua prefiks yang melabeli baris dan kolomnya. Jawabannya, $c[7,5] = 4$, adalah sel yang dicetak tebal.

	j=0	j=1 (C)	j=2 (G)	j=3 (T)	j=4 (A)	j=5 (T)
$i = 0$	0	0	0	0	0	0
$i = 1$ **(A)**	0	0	0	0	1	1
$i = 2$ **(C)**	0	1	1	1	1	1
$i = 3$ **(G)**	0	1	2	2	2	2
$i = 4$ **(G)**	0	1	2	2	2	2
$i = 5$ **(T)**	0	1	2	3	3	3
$i = 6$ **(T)**	0	1	2	3	3	4
$i = 7$ **(A)**	0	1	2	3	4	**4**

Dua pemeriksaan kewarasan yang dapat Anda terapkan pada tabel LCS mana pun. Pertama, entrinya tak pernah mengecil saat Anda bergerak ke kanan atau ke bawah — menambah sebuah karakter tak pernah bisa memendekkan subbarisan bersama terbaik. Kedua, dua entri yang bersebelahan mendatar atau menegak berselisih paling banyak 1. Pindailah **Tabel 7.6**: keduanya berlaku di mana-mana.

7.9.8 Tabel panahnya

Tabel 7.7 mencatat, untuk tiap sel, kasus rekurensi mana yang dipakai.

Tabel 7.7 Tabel panah b untuk instans yang sama. ↖ adalah `"copy"` (karakternya cocok), ↑ adalah `"skipx"` (nilainya diambil dari atas), ← adalah `"skipy"` (nilainya diambil dari kiri). Mengikuti panahnya mundur dari pojok kanan bawah akan merekonstruksi sebuah LCS.

	j=1 (C)	j=2 (G)	j=3 (T)	j=4 (A)	j=5 (T)
$i = 1$ **(A)**	↑	↑	↑	↖	←
$i = 2$ **(C)**	↖	←	←	↑	↑
$i = 3$ **(G)**	↑	↖	←	←	←
$i = 4$ **(G)**	↑	↖	↑	↑	↑
$i = 5$ **(T)**	↑	↑	↖	←	↖
$i = 6$ **(T)**	↑	↑	↖	↑	↖
$i = 7$ **(A)**	↑	↑	↑	↖	↑

7.9.9 Merekonstruksi LCS-nya

Mulailah di (7,5) dan ikuti panahnya sampai salah satu indeksnya menyentuh 0. Setiap ↖ yang Anda lewati menyumbang satu karakter.

1. (7,5): panah ↑. Tidak ada karakter. Pindah ke (6,5).
2. (6,5): panah ↖. **Keluarkan $x_6 = $ T.** Pindah ke (5,4).
3. (5,4): panah ←. Tidak ada karakter. Pindah ke (5,3).
4. (5,3): panah ↖. **Keluarkan $x_5 = $ T.** Pindah ke (4,2).
5. (4,2): panah ↖. **Keluarkan $x_4 = $ G.** Pindah ke (3,1).
6. (3,1): panah ↑. Tidak ada karakter. Pindah ke (2,1).
7. (2,1): panah ↖. **Keluarkan $x_2 = $ C.** Pindah ke (1,0).
8. (1,0): $j = 0$. Berhenti.

Karakternya keluar dalam urutan T, T, G, C — yaitu **terbalik**, sebab kita berjalan dari ujung untainya menuju awalnya. Membalikkannya memberi

$$\mathrm{LCS}(X, Y) = \text{"CGTT"}, \qquad \text{berpanjang } 4,$$

yang merupakan salah satu dari dua jawaban yang disebut pada **Gambar 7.18**. Periksalah langsung: "CGTT" duduk pada posisi 2, 3, 5, 6 dari "ACGGTTA" ✓ dan pada posisi 1, 2, 3, 5 dari "CGTAT" ✓.

> **MENGAPA YANG MUNCUL "CGTT" DAN BUKAN "CGTA".** Keduanya berpanjang 4 dan keduanya jawaban yang benar. Yang mana yang dihasilkan algoritmanya sepenuhnya bergantung pada aturan pemutus seri di baris 10 **Pseudokode 7.9**: dengan ≥, seri jatuh ke `skipx`. Ubahlah menjadi $>$ dan penyusurannya berbelok berbeda serta dapat menghasilkan LCS yang satunya. Algoritma untuk "sebuah *longest common subsequence*" tidak berkewajiban apa pun untuk lebih menyukai satu jawaban maksimal daripada yang lain.

Rekonstruksinya mengunjungi satu sel per langkah dan tiap langkah mengurangi $i + j$ setidaknya 1, sehingga ia memakan waktu $O(m + n)$ — tidak berarti dibandingkan $\Theta(mn)$ yang dihabiskan untuk mengisi tabelnya.

7.9.10 $\Theta(mn)$, dan varian beruang $\Theta(\min(m, n))$

Waktunya. Dua gelung bersarang, m dan n iterasi, kerja $O(1)$ per sel: $\Theta(mn)$. Tidak ada gelung minimisasi seperti pada deretan matriks, dan itulah sebabnya LCS kuadratik sedangkan deretan matriks kubik.

Ruangnya, sebagaimana ditulis. Dua tabel berisi $(m + 1)(n + 1)$ sel: $\Theta(mn)$.

Sekarang pertanyaan yang diajukan **Gambar 7.21**: *seberapa banyak kita dapat mengurangi kebutuhan ruang kita, jika kita tidak perlu merekonstruksi LCS-nya?*

Terapkan penghematan ruang §7.4.3. Lihat rekurensinya: $c[i, j]$ bersandar pada $c[i - 1, j - 1]$, $c[i - 1, j]$, dan $c[i, j - 1]$. Setiap satu di antaranya berada entah di baris $i - 1$ atau lebih awal di baris i. Tidak ada apa pun di baris $0..i - 2$ yang pernah dilihat lagi sesudah baris $i - 1$ selesai. Jadi kita hanya memerlukan **dua baris**: baris sebelumnya dan baris kini. Itu $2(n + 1)$ sel, atau $\Theta(n)$.

Satu putaran sekrup lagi. Karena $\mathrm{LCS}(X, Y) = \mathrm{LCS}(Y, X)$, kita bebas menaruh untai mana pun yang kita suka di sepanjang barisnya. Menaruh untai yang **lebih panjang** menurun pada barisnya dan yang **lebih pendek** melintang pada kolomnya membuat tiap baris selebar $\min(m, n) + 1$ sel. Karena itu:

> **HASIL SOAL RUANGNYA.** *Panjang* sebuah LCS dapat dihitung dalam waktu $\Theta(mn)$ dan ruang $\Theta(\min(m, n))$. *Subbarisannya sendiri* tidak dapat dipulihkan dengan cara ini, sebab tabel b —

> catatan pilihannya — sudah dibuang. (Ia masih dapat dipulihkan dalam ruang linear lewat penghalusan *divide and conquer* dari Hirschberg, dengan biaya waktu jalan yang berlipat dua; lihat [3, §14.4] dan Soal 7.14.)

Itulah jawaban atas pertanyaan pada **Gambar 7.21**. Untuk dua untai DNA berpanjang sejuta karakter masing-masing, tabel penuhnya akan memerlukan kira-kira 4×10^{12} bita — empat terabita — sedangkan versi dua barisnya memerlukan kira-kira delapan megabita.

7.9.11 Program lengkapnya

Kode 7.6 mengimplementasikan **LCS-Length** persis sebagaimana diberikan **Pseudokode 7.9**, mencetak kedua tabelnya, merekonstruksi subbarisannya, lalu mengadu waktu versi tabel penuh melawan versi dua baris pada untai acak.

Kode 7.6 LCS-Length, tabel panahnya, rekonstruksinya, dan varian dua barisnya, lengkap dengan perbandingan waktu dan memori. Kompilasi dengan `g++ -std=c++17 -O2 -o lcs lcs.cpp`.

```
// lcs.cpp - LCS-Length dari Subbab 7.9.6, dengan tabel c, tabel panah b,
// rekonstruksinya, dan varian hemat ruang dua baris.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <chrono>
#include <random>
using namespace std;
using namespace std::chrono;

enum Arrow { NONE, COPY, SKIPX, SKIPY };              // diagonal, atas, kiri
const char* glyph[] = {" ", "\\", "^", "<"};          // \ = copy, ^ = skip x, < = skip y

vector<vector<int>>   c;
vector<vector<Arrow>> b;

void lcsLength(const string& X, const string& Y) {
    int m = (int)X.size(), n = (int)Y.size();
    c.assign(m + 1, vector<int>(n + 1, 0));
    b.assign(m + 1, vector<Arrow>(n + 1, NONE));
    for (int i = 1; i <= m; ++i)
        for (int j = 1; j <= n; ++j) {
            if (X[i-1] == Y[j-1]) {                    // x_i = y_j : hurufnya bergabung ke
dalam LCS
                c[i][j] = c[i-1][j-1] + 1; b[i][j] = COPY;
            } else if (c[i-1][j] >= c[i][j-1]) {       // seri diputus ke arah melewatkan
x, seperti di Subbab 7.9.6
                c[i][j] = c[i-1][j];       b[i][j] = SKIPX;
            } else {
                c[i][j] = c[i][j-1];       b[i][j] = SKIPY;
            }
        }
}

void reconstruct(const string& X, int i, int j, string& out) {
    if (i == 0 || j == 0) return;
    if (b[i][j] == COPY) { reconstruct(X, i-1, j-1, out); out += X[i-1]; }
    else if (b[i][j] == SKIPX) reconstruct(X, i-1, j, out);
    else                       reconstruct(X, i, j-1, out);
}
```

Kode 7.6 (Lanjutan) *LCS-Length, tabel panahnya, rekonstruksinya, dan varian dua barisnya, lengkap dengan perbandingan waktu dan memori. Kompilasi dengan g++ -std=c++17 -O2 -o lcs lcs.cpp.*

```
// Ruang Theta(min(m,n)): dua baris saja, dan tanpa tabel b, jadi tanpa rekonstruksi.
int lcsTwoRows(const string& A, const string& B) {
    const string& X = (A.size() >= B.size()) ? A : B;  // taruh untai PENDEK melintang pada
barisnya
    const string& Y = (A.size() >= B.size()) ? B : A;
    int m = (int)X.size(), n = (int)Y.size();
    vector<int> prev(n + 1, 0), cur(n + 1, 0);
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j)
            cur[j] = (X[i-1] == Y[j-1]) ? prev[j-1] + 1 : max(prev[j], cur[j-1]);
        swap(prev, cur);
    }
    return prev[n];
}

void show(const string& X, const string& Y) {
    int m = (int)X.size(), n = (int)Y.size();
    cout << "  c table:\n        j: ";
    for (int j = 0; j <= n; ++j) cout << setw(4) << j;
    cout << "\n          y: ";
    for (int j = 0; j <= n; ++j) cout << setw(4) << (j == 0 ? '-' : Y[j-1]);
    cout << "\n";
    for (int i = 0; i <= m; ++i) {
        cout << "  i=" << i << " x=" << (i == 0 ? '-' : X[i-1]) << "  ";
        for (int j = 0; j <= n; ++j) cout << setw(4) << c[i][j];
        cout << "\n";
    }
    cout << "\n  b table  ( \\ = copy, ^ = skip x, < = skip y ):\n          j: ";
    for (int j = 1; j <= n; ++j) cout << setw(4) << Y[j-1];
    cout << "\n";
    for (int i = 1; i <= m; ++i) {
        cout << "  i=" << i << " x=" << X[i-1] << "    ";
        for (int j = 1; j <= n; ++j) cout << setw(4) << glyph[b[i][j]];
        cout << "\n";
    }
}
```

Kode 7.6 (Lanjutan) *LCS-Length, tabel panahnya, rekonstruksinya, dan varian dua barisnya, lengkap dengan perbandingan waktu dan memori. Kompilasi dengan g++ -std=c++17 -O2 -o lcs lcs.cpp.*

```
int main() {
    string X = "ACGGTTA", Y = "CGTAT";
    cout << "X = \"" << X << "\"  (m=" << X.size() << ")   Y = \""
         << Y << "\"  (n=" << Y.size() << ")\n\n";
    lcsLength(X, Y);
    show(X, Y);
    string z; reconstruct(X, (int)X.size(), (int)Y.size(), z);
    cout << "\n  LCS length          = " << c[X.size()][Y.size()] << "\n";
    cout << "  one LCS             = \"" << z << "\"\n";
    cout << "  two-row variant     = " << lcsTwoRows(X, Y) << " (same length, no
string)\n";

    // Pengukuran waktu: tabel penuh lawan dua baris, pada untai acak mirip DNA.
    cout << "\n" << left << setw(10) << "m=n" << setw(14) << "LCS len"
         << setw(14) << "full ms" << setw(16) << "full MiB"
         << setw(14) << "2-row ms" << setw(14) << "2-row MiB" << "\n";
    mt19937 rng(7);
    for (int sz : {200, 1000, 4000, 8000}) {
        string A(sz, ' '), B(sz, ' ');
        for (char& ch : A) ch = "ACGT"[rng() % 4];
        for (char& ch : B) ch = "ACGT"[rng() % 4];
        auto t0 = steady_clock::now(); lcsLength(A, B);
        double tf = duration<double,milli>(steady_clock::now()-t0).count();
        int lenF = c[sz][sz];
        auto t1 = steady_clock::now(); int lenT = lcsTwoRows(A, B);
        double tt = duration<double,milli>(steady_clock::now()-t1).count();
        if (lenF != lenT) { cout << "MISMATCH\n"; return 1; }
        double fullMiB  = (double)(sz+1)*(sz+1)*(sizeof(int)+sizeof(Arrow)) / 1048576.0;
        double tworMiB  = (double)2*(sz+1)*sizeof(int) / 1048576.0;
        cout << left << setw(10) << sz << setw(14) << lenF
             << setw(14) << fixed << setprecision(2) << tf
             << setw(16) << setprecision(3) << fullMiB
             << setw(14) << setprecision(2) << tt
             << setw(14) << setprecision(3) << tworMiB << "\n";
        c.clear(); c.shrink_to_fit(); b.clear(); b.shrink_to_fit();
    }
    return 0;
}
```

Keluarannya:

```
X = "ACGGTTA"  (m=7)   Y = "CGTAT"  (n=5)

  c table:
        j:    0   1   2   3   4   5
          y:    -   C   G   T   A   T
  i=0 x=-     0   0   0   0   0   0
  i=1 x=A     0   0   0   0   1   1
  i=2 x=C     0   1   1   1   1   1
  i=3 x=G     0   1   2   2   2   2
  i=4 x=G     0   1   2   2   2   2
  i=5 x=T     0   1   2   3   3   3
  i=6 x=T     0   1   2   3   3   4
  i=7 x=A     0   1   2   3   4   4

  b table  ( \ = copy, ^ = skip x, < = skip y ):
          j:    C   G   T   A   T
  i=1 x=A       ^   ^   ^   \   <
  i=2 x=C       \   <   <   ^   ^
  i=3 x=G       ^   \   <   <   <
  i=4 x=G       ^   \   ^   ^   ^
  i=5 x=T       ^   ^   \   <   \
  i=6 x=T       ^   ^   \   ^   \
  i=7 x=A       ^   ^   ^   \   ^

  LCS length          = 4
  one LCS             = "CGTT"
  two-row variant     = 4 (same length, no string)

m=n       LCS len       full ms       full MiB        2-row ms      2-row MiB
200       124           0.28          0.308           0.03          0.002
1000      645           6.36          7.645           0.74          0.008
4000      2598          247.23        122.131         25.07         0.031
8000      5216          1930.83       488.403         152.86        0.061
```

Tabel c milik mesin itu persis **Tabel 7.6**, sel demi sel, dan tabel b-nya persis **Tabel 7.7**, panah demi panah. Rekonstruksinya menghasilkan “CGTT”, sama seperti penyusuran tangan §7.9.9.

Blok waktunya membuat argumen ruangnya menjadi konkret. Pada $m = n = 8000$ tabel penuhnya memerlukan 488 MiB dan memakan 1 931 ms; versi dua barisnya memerlukan 0.061 MiB — kira-kira delapan ribu kali lebih sedikit — dan memakan 153 ms, dua belas kali lebih *cepat*, sebab ia muat di dalam *cache* dan tak pernah mengalokasikan setengah gigabita. Secara asimtotik keduanya $\Theta(mn)$ dalam waktu; dalam praktik yang menghormati hierarki memorilah yang menang telak.

Perhatikan pula pertumbuhan kuadratiknya, beserta batas-batasnya. Melipatduakan kedua panjang dari 4000 ke 8000 melipatkan mn sebanyak 4 kali, jadi $\Theta(mn)$ meramalkan faktor kira-kira 4; versi dua barisnya cukup dekat ($152.86/25.07 \approx 6.1$ pada jalan ini), sedangkan waktu tabel penuhnya tumbuh $1930.83/247.23 \approx 7.8$ kali, hampir dua kali ramalannya. Kelebihannya adalah tekanan memori: pada setengah gigabita, tabel penuhnya tidak lagi muat di *cache* mana pun, dan setiap baris membayar perjalanan ke memori utama.

7.10 Submasalah yang Tumpang Tindih versus Submasalah yang Saling Bebas

Kita sekarang dapat menarik garis yang sedari tadi dituju bab ini.

7.10.1 Dua ciri khasnya

> **DEFINISI.** Sebuah persoalan punya **overlapping subproblems** (submasalah yang tumpang tindih) bila algoritma rekursif untuknya menyelesaikan submasalah yang *sama* berulang kali, alih-alih selalu menghasilkan submasalah yang baru. Secara setara: banyaknya submasalah yang **berbeda** itu sedikit — biasanya polinomial dalam ukuran masukannya — sedangkan pohon rekursi yang mengunjunginya besar.

Dynamic programming berlaku ketika **kedua** hal berikut terpenuhi:

1. **Optimal substructure** — sebuah solusi optimal memuat solusi optimal bagi submasalahnya (§7.5.2). Inilah yang memungkinkan Anda menulis rekurensinya sama sekali.
2. **Overlapping subproblems** — submasalah berbedanya sedikit dan mereka berulang. Inilah yang membuat tabelnya layak dibangun.

Divide and conquer, sebaliknya, memerlukan *optimal substructure* dalam artian yang remeh saja tetapi secara khusus menginginkan **kebalikan** dari ciri kedua itu: ia ingin submasalahnya *saling bebas dan semuanya berbeda*, sebagaimana dikatakan keterangan **Gambar 7.2**. Jika submasalah Merge Sort tumpang tindih, mem-*memoization*-kannya akan menolong; karena tidak, tabel memonya akan murni biaya tambahan — setiap penelusuran meleset, setiap entri ditulis sekali dan tak pernah dibaca.

> **KEDUA CIRI ITU SAMA-SAMA DIPERLUKAN.** *Optimal substructure* tanpa tumpang tindih memberi *divide and conquer* (Merge Sort, pencarian biner): bangunlah tabel dan Anda memboroskan memori tanpa hasil. Tumpang tindih tanpa *optimal substructure* memberi persoalan yang tabelnya bisa diisi tetapi entrinya tidak tersusun menjadi keseluruhan yang optimal — lintasan sederhana terpanjang (§7.5.4) adalah contoh bakunya, dan ia NP-hard. Periksalah keduanya sebelum mengulurkan tangan ke sebuah tabel.

7.10.2 Diukur

Pembedaan itu bukan soal selera; ia dapat dihitung. **Kode 7.7** menjalankan masing-masing dari tiga rekursi dan mencatat dua bilangan: berapa banyak panggilan yang dibuat, dan berapa banyak submasalah *berbeda* yang diwakili panggilan-panggilan itu.

Kode 7.7 Mencacah panggilan melawan submasalah berbeda untuk Merge Sort, Fibonacci naif, dan rekursi deretan matriks yang polos — sebuah uji empiris untuk *overlapping subproblems*. Kompilasi dengan `g++ -std=c++17 -O2 -o overlap overlap.cpp`.

```
// overlap.cpp - garis antara divide and conquer dan dynamic programming,
// diukur: berapa banyak panggilan yang dibuat, dan ada berapa submasalah yang BERBEDA.
#include <iostream>
#include <iomanip>
#include <set>
#include <vector>
#include <utility>
using namespace std;

long long mergeCalls; set<pair<int,int>> mergeSub;      // submasalah = rentang (lo, hi)
void mergeSortShape(int lo, int hi) {
    ++mergeCalls; mergeSub.insert({lo, hi});
    if (hi - lo <= 1) return;
    int mid = lo + (hi - lo) / 2;
    mergeSortShape(lo, mid);
    mergeSortShape(mid, hi);
}

long long fibCalls; set<int> fibSub;                     // submasalah = argumen n
void fibShape(int n) {
    ++fibCalls; fibSub.insert(n);
    if (n <= 1) return;
    fibShape(n - 1);
    fibShape(n - 2);
}

long long mcmCalls; set<pair<int,int>> mcmSub;           // submasalah = deretan (i, j)
void mcmShape(int i, int j) {
    ++mcmCalls; mcmSub.insert({i, j});
    if (i == j) return;
    for (int k = i; k < j; ++k) { mcmShape(i, k); mcmShape(k + 1, j); }
}

int main() {
    cout << left << setw(22) << "algorithm" << setw(8) << "size"
         << setw(16) << "calls made" << setw(16) << "distinct subs"
         << setw(12) << "ratio" << "\n";
    for (int n : {8, 16, 24}) {
        mergeCalls = 0; mergeSub.clear(); mergeSortShape(0, n);
        cout << left << setw(22) << "Merge Sort" << setw(8) << n
             << setw(16) << mergeCalls << setw(16) << mergeSub.size()
             << setw(12) << fixed << setprecision(2)
             << (double)mergeCalls / mergeSub.size() << "\n";
    }
    for (int n : {8, 16, 24}) {
        fibCalls = 0; fibSub.clear(); fibShape(n);
```

Kode 7.7 (Lanjutan) *Mencacah panggilan melawan submasalah berbeda untuk Merge Sort, Fibonacci naif, dan rekursi deretan matriks yang polos — sebuah uji empiris untuk overlapping subproblems. Kompilasi dengan g++ -std=c++17 -O2 -o overlap overlap.cpp.*

```
        cout << left << setw(22) << "naive Fibonacci" << setw(8) << n
             << setw(16) << fibCalls << setw(16) << fibSub.size()
             << setw(12) << fixed << setprecision(2)
             << (double)fibCalls / fibSub.size() << "\n";
    }
    for (int n : {4, 8, 12}) {
        mcmCalls = 0; mcmSub.clear(); mcmShape(1, n);
        cout << left << setw(22) << "matrix chain (plain)" << setw(8) << n
             << setw(16) << mcmCalls << setw(16) << mcmSub.size()
             << setw(12) << fixed << setprecision(2)
             << (double)mcmCalls / mcmSub.size() << "\n";
    }
    cout << "\nA ratio near 1 means no subproblem is ever revisited: divide and conquer.\n"
            "A ratio that grows without bound means the recursion is redoing work: use
DP.\n";
    return 0;
}
```

Keluarannya:

```
algorithm             size    calls made      distinct subs   ratio
Merge Sort            8       15              15              1.00
Merge Sort            16      31              31              1.00
Merge Sort            24      47              47              1.00
naive Fibonacci       8       67              9               7.44
naive Fibonacci       16      3193            17              187.82
naive Fibonacci       24      150049          25              6001.96
matrix chain (plain)  4       27              10              2.70
matrix chain (plain)  8       2187            36              60.75
matrix chain (plain)  12      177147          78              2271.12

A ratio near 1 means no subproblem is ever revisited: divide and conquer.
A ratio that grows without bound means the recursion is redoing work: use DP.
```

Bacalah kolom nisbahnya. Untuk **Merge Sort** nilainya persis 1.00 pada setiap ukuran: setiap panggilan adalah submasalah baru, tidak ada yang pernah terulang. Mem-*memoization*-kan Merge Sort berarti menambahkan tabel yang ditulis sekali per entri dan tak pernah dibaca.

Untuk **Fibonacci naif** nisbahnya 7.44, lalu 187.82, lalu 6002 — ia tumbuh tanpa batas, sementara kolom submasalah berbedanya nyaris tidak bergerak (9, 17, 25 — yaitu $n + 1$).

Untuk **deretan matriks polos** polanya sama: pada $n = 12$ ada 177 147 panggilan yang hanya mencakup 78 subderetan berbeda, dan $78 = 12 \cdot 13/2$, persis $n(n + 1)/2$ dari §7.6.6.

7.10.3 Prosedur pengambilan keputusannya

> **CARA MEMUTUSKAN, DALAM PRAKTIK.** Diberikan sebuah algoritma rekursif: **1.** Sebutkan submasalahnya dengan tepat — tuple nilai apa yang menandainya? **2.** Hitunglah banyaknya nilai berbeda yang dapat diambil tuple itu. Jika cacahnya eksponensial, *dynamic programming* tidak akan menyelamatkan Anda; carilah algoritma *greedy* (Bab 9), perumusan yang lain, atau terimalah waktu eksponensialnya. **3.** Jika cacahnya polinomial tetapi rekursinya membuat panggilan jauh lebih banyak daripada itu, Anda punya *overlapping subproblems*. Lakukan *memoization* atau tabulasi. **4.** Jika cacahnya polinomial dan rekursinya membuat kira-kira sebanyak itu panggilan, submasalahnya memang sudah saling bebas. Anda punya *divide and conquer*; biarkan saja. Langkah 3 melawan langkah 4 persis adalah kolom nisbah pada **Kode 7.7**, dan Anda dapat mengukurnya dalam lima menit dengan sebuah `set` dan sebuah pencacah.

Tabel 7.8 *Divide and conquer* melawan *dynamic programming*, berdampingan. Baris pertamanyalah yang menentukan mana yang sedang Anda hadapi; segala yang lain mengikutinya.

	Divide and conquer	*Dynamic programming*
Submasalah	Saling lepas, semuanya berbeda	Tumpang tindih, sedikit yang berbeda
Arah	Hanya *top-down*	*Top-down* dengan memo, atau *bottom-up*
Memori tambahan	Tidak ada di luar tumpukan	Sebuah tabel jawaban submasalah
Bentuk khas	$T(n) = aT(n/b) + f(n)$	nilai(s) = optimum atas pilihan
Diselesaikan oleh	Teorema Master (§3.13)	Mencacah sel × biaya per sel
Contoh	Merge Sort, pencarian biner, Strassen	Fibonacci, deretan matriks, LCS
Modus kegagalan	—	Ledakan eksponensial bila tidak di-*memoization*

7.11 Profitina dalam Praktik

AI Visibility Engine di balik **app.profitina.com** memindai sebuah situs web lalu menilai seberapa siap situs itu untuk dikutip oleh mesin jawaban AI. Beberapa bagian dari pipelinenya adalah program dinamis persis dari jenis yang ada di bab ini, dan salah satunya sudah cukup menjadi alasan untuk menguasai bab ini.

Deteksi halaman yang nyaris kembar. Situs merek besar sering menerbitkan artikel yang sama di beberapa URL — jalur kategori, jalur kampanye, jalur AMP. Menilai semuanya secara terpisah memboroskan anggaran perayapan dan menyimpangkan skor situsnya. Memutuskan apakah dua halaman merupakan "artikel yang sama, disunting sedikit" adalah pertanyaan kemiripan untai, dan program dinamis *edit distance* serta *longest common subsequence* adalah perkakas klasik untuk itu. Di sini varian beruang $\Theta(\min(m, n))$ dari §7.9.10 bukan sebuah optimasi melainkan sebuah keharusan: dua artikel panjang menghasilkan tabel dengan ratusan juta sel, dan layanan yang menjawab pertanyaan itu harus berjalan di dalam anggaran memori terbatas berdampingan dengan segala hal lain di dalam mesinnya.

Saran dari asisten teknisnya. Ketika Content Engine mengusulkan penulisan ulang struktur judul sebuah halaman, antarmuka penggunanya harus memperlihatkan *apa yang berubah*, bukan menumpahkan dua blok teks berdampingan. Tampilan selisih itu adalah langkah rekonstruksi §7.9.9 — tabel panah yang disusuri mundur — dan itulah sebabnya tabel b layak memakan memorinya pada jalur kode tersebut, tetapi tidak pada pendeteksi halaman kembarnya.

Transformasi berantai di dalam pipeline penilaiannya. Menilai sebuah halaman menerapkan serangkaian transformasi pada sekumpulan sinyal terekstraksi, dan beberapa di antaranya adalah pemetaan linear yang diterapkan pada matriks pengamatan. Urutan penggabungan rantai pemetaan semacam itu mengubah ukuran hasil antaranya sehingga mengubah pula hitungan yang dikerjakan, dan itu persis persoalan deretan matriks §7.6. Rantainya pendek — $\Theta(n^3)$ milik **Matrix-Chain-Order** bukan apa-apa pada n di bawah sepuluh — dan rencananya dihitung sekali lalu disinggahkan, bukan dihitung ulang per halaman.

Penjadwalan pemindaian ulang. Tiap merek dipindai ulang pada irama yang ditetapkan tingkatannya. Memuatkan pemindaian sehari ke dalam sejumlah slot pekerja yang terbatas, sedemikian rupa sehingga tidak ada tingkatan yang meleset dari irama yang dijanjikan, adalah persoalan penjadwalan dengan *optimal substructure*. Seperti biasa: aplikasinya memberi saran, penggunanya yang memutuskan — sebuah rencana pemindaian adalah rekomendasi, tidak pernah jaminan atas sebuah skor.

Tidak ada satu pun di atas yang menyentuh data pribadi. Di mana sebuah sinyal memang menyentuhnya — halaman kontak yang memuat nama, misalnya — UU PDP 27/2022 berlaku, dan aturan yang diikuti basis kodenya sederhana saja: tidak ada medan data pribadi yang boleh masuk ke dalam perhitungan kemiripan sejak awal.

7.11.1 Studi kasus yang dikerjakan: Profitina Laundry

Profitina Laundry adalah studi kasus pengajaran dalam buku ini — sebuah laundry kecil di Surabaya dengan sebuah meja layanan, sebuah daftar harga, dan setumpuk slip tulisan tangan di penghujung tiap hari.

Persoalannya membosankan sekaligus universal. Daftar harganya punya nama-nama baku — *Cuci Setrika Reguler*, *Bed Cover Besar*, *Sepatu Kanvas*. Yang ditulis petugas meja pada slipnya adalah apa pun yang muat: `cuci kring expres`, `setrka saja`, `bedcover besar`, `cuci kering reg`. Sebelum hari itu dapat ditotal, tiap slip harus dicocokkan dengan sebuah entri katalog.

Perbandingan untai secara persis langsung gagal — `bedcover besar` tidak sama dengan `Bed Cover Besar`. Yang kita butuhkan adalah sebuah *ukuran kemiripan*, dan §7.9 sudah memberi kita satu. Definisikan Persamaan (7.15):

$$\text{kemiripan}(u, v) = \frac{|\text{LCS}(u, v)|}{\max(|u|, |v|)}, \tag{7.15}$$

dengan $|u|$ berarti panjang untai u, sesudah kedua untainya dilipat ke huruf kecil dan spasi serta tanda bacanya dibuang. Pembilangnya mencacah karakter yang muncul di keduanya, berurutan; penyebutnya menormalkan sehingga skornya terletak antara 0 (tidak ada kesamaan) dan 1 (identik). **Kode 7.8** adalah seluruh sistemnya.

Kode 7.8 Profitina Laundry: mencocokkan slip tulisan tangan dengan daftar harga memakai kemiripan LCS, dalam ruang $\Theta(\min(m, n))$. Kompilasi dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp`.

```
// laundry.cpp - Profitina Laundry: merekonsiliasi apa yang ditulis petugas meja
// dengan daftar harga bakunya, memakai LCS bab ini sebagai ukuran
// kemiripan. Tidak ada di sini yang menyentuh data pribadi: hanya nama barang dan harga.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <chrono>
using namespace std;
using namespace std::chrono;

// Waktu Theta(mn), ruang Theta(min(m,n)): di sini kita hanya perlu PANJANGnya.
int lcsLen(const string& A, const string& B) {
    const string& X = A.size() >= B.size() ? A : B;      // untai panjang menggerakkan
gelung luarnya
    const string& Y = A.size() >= B.size() ? B : A;      // untai pendek menentukan lebar
barisnya
    int m = (int)X.size(), n = (int)Y.size();
    vector<int> prev(n + 1, 0), cur(n + 1, 0);
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j)
            cur[j] = (X[i-1] == Y[j-1]) ? prev[j-1] + 1 : max(prev[j], cur[j-1]);
        swap(prev, cur);
    }
    return prev[n];
}

string normalize(const string& s) {                       // lipat huruf besar-kecil, buang
segala yang lain
    string t;
    for (char ch : s) if (isalnum((unsigned char)ch)) t += (char)tolower((unsigned
char)ch);
    return t;
}

struct Item { string name; long long pricePerKg; };

int main() {
    // Daftar harga bakunya. Harga dalam rupiah per kilogram.
    vector<Item> catalog = {
        {"Cuci Kering Reguler",  7000},
        {"Cuci Kering Express",  12000},
        {"Cuci Setrika Reguler", 10000},
        {"Cuci Setrika Express", 16000},
        {"Setrika Saja",          6000},
        {"Bed Cover Besar",      35000},
        {"Selimut Tebal",        28000},
        {"Sepatu Kanvas",        25000}
    };
```

Kode 7.8 (Lanjutan) *Profitina Laundry: mencocokkan slip tulisan tangan dengan daftar harga memakai kemiripan LCS, dalam ruang \Theta(\min(m,n)). Kompilasi dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
    // Yang sesungguhnya tertulis pada slip hari itu: disingkat, salah eja, huruf campur.
    vector<pair<string,double>> slips = {
        {"cuci setrika reguler", 3.5},
        {"cuci kring expres",    2.0},
        {"setrka saja",          1.5},
        {"bedcover besar",       1.0},
        {"cuci kering reg",      4.25},
        {"sepatu knvas",         1.0}
    };

    cout << "PROFITINA LAUNDRY - reconciling the day's slips against the price list\n\n";
    cout << left << setw(22) << "written on the slip" << setw(24) << "matched to"
         << setw(9) << "score" << setw(7) << "kg" << right << setw(12) << "rupiah" << "\n";
    cout << string(74, '-') << "\n";

    long long total = 0;
    auto t0 = steady_clock::now();
    for (auto& [written, kg] : slips) {
        string w = normalize(written);
        int bestIdx = -1; double bestScore = -1;
        for (int i = 0; i < (int)catalog.size(); ++i) {
            string cn = normalize(catalog[i].name);
            int L = lcsLen(w, cn);
            // Kemiripan di [0,1]: panjang LCS terhadap yang lebih panjang di antara kedua
namanya.
            double score = (double)L / max(w.size(), cn.size());
            if (score > bestScore) { bestScore = score; bestIdx = i; }
        }
        long long amount = (long long)(catalog[bestIdx].pricePerKg * kg + 0.5);
        total += amount;
        cout << left << setw(22) << written << setw(24) << catalog[bestIdx].name
             << setw(9) << fixed << setprecision(3) << bestScore
             << setw(7) << setprecision(2) << kg << right << setw(12) << amount << "\n";
    }
    auto t1 = steady_clock::now();
    cout << string(74, '-') << "\n";
    cout << left << setw(62) << "TOTAL" << right << setw(12) << total << "\n";
    cout << "\nAll " << slips.size() << " slips matched against " << catalog.size()
         << " catalog entries in "
         << fixed << setprecision(3) << duration<double,milli>(t1-t0).count() << " ms.\n";
```

Kode 7.8 (Lanjutan) *Profitina Laundry: mencocokkan slip tulisan tangan dengan daftar harga memakai kemiripan LCS, dalam ruang \Theta(\min(m,n)). Kompilasi dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
    // Slip yang SEHARUSNYA tidak dicocokkan: skornya tetap rendah dan manusia harus
melihatnya.
    string junk = "titip motor sebentar";
    string jn = normalize(junk);
    double worst = 0; string which;
    for (auto& it : catalog) {
        string cn = normalize(it.name);
        double s = (double)lcsLen(jn, cn) / max(jn.size(), cn.size());
        if (s > worst) { worst = s; which = it.name; }
    }
    cout << "\nGuard rail: \"" << junk << "\" scores only "
         << fixed << setprecision(3) << worst << " against its closest entry ("
         << which << ").\nBelow a threshold of 0.600 the app refuses to guess and asks a
person.\n";
    return 0;
}
```

Keluarannya:

```
PROFITINA LAUNDRY - reconciling the day's slips against the price list

written on the slip   matched to              score    kg          rupiah
-----------------------------------------------------------------------
cuci setrika reguler  Cuci Setrika Reguler    1.000    3.50         35000
cuci kring expres     Cuci Kering Express     0.882    2.00         24000
setrka saja           Setrika Saja            0.909    1.50          9000
bedcover besar        Bed Cover Besar         1.000    1.00         35000
cuci kering reg       Cuci Kering Reguler     0.765    4.25         29750
sepatu knvas          Sepatu Kanvas           0.917    1.00         25000
-----------------------------------------------------------------------
TOTAL                                                              157750

All 6 slips matched against 8 catalog entries in 0.058 ms.

Guard rail: "titip motor sebentar" scores only 0.389 against its closest entry (Cuci
Setrika Reguler).
Below a threshold of 0.600 the app refuses to guess and asks a person.
```

Setiap slip tercocokkan dengan benar. Perhatikan empat hal.

`bedcover besar` **memperoleh skor sempurna** 1.000 terhadap *Bed Cover Besar*, sebab penormalannya membuang spasi sehingga kedua untainya menjadi identik karakter demi karakter. Prapengolahan yang murah sebelum algoritma yang mahal hampir selalu berharga.

`cuci kring expres` **memperoleh** 0.882 meskipun ada tiga salah eja terpisah, sebab LCS tidak peduli di mana huruf yang hilang itu berada — ia mencacah apa yang selamat, berurutan.

`cuci kering reg` **hanya memperoleh** 0.765, terendah di antara keenamnya, sebab singkatan `reg` membuang enam dari sembilan karakter *Reguler*. Ia masih tercocokkan dengan benar, tetapi marginnya lebih tipis, dan inilah baris yang patut diperlihatkan kepada manusia bila suatu saat tokonya berselisih soal tagihan.

Pagar pengamannya yang paling penting dari semuanya. Slip bertuliskan *titip motor sebentar* — sama sekali bukan layanan laundry — memperoleh skor 0.389 terhadap entri katalog terdekatnya. Sistemnya dibangun untuk menolak di bawah 0.600 alih-alih menebak. Itulah aturan rumah Profitina, diterapkan di dalam kode: *aplikasinya memberi saran; penggunanya yang memutuskan*. Algoritma yang selalu mengembalikan tebakan terbaiknya, seburuk apa pun, akan dengan penuh percaya diri menagih pelanggan untuk layanan yang tak seorang pun pesan.

Soal biayanya: enam slip melawan delapan entri katalog berarti empat puluh delapan perhitungan LCS pada untai sepanjang kira-kira dua puluh karakter, dan seluruh rekonsiliasinya memakan 0.058 ms. Empat ratus slip sehari akan memakan kira-kira empat milidetik. $\Theta(mn)$ dari §7.9.10 masih sangat terjangkau di sini — tetapi kode yang sama dijalankan atas dokumen utuh, seperti pada pendeteksi halaman kembar di atas, persis di situlah varian dua barisnya berhenti menjadi keanggunan dan mulai menjadi beda antara bekerja dan tidak bekerja.

7.12 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. Mengira rekursinya yang menjadi persoalan. Bukan. Pengulangannyalah yang menjadi persoalan. Versi ber-*memoization* pada **Kode 7.2** tetap rekursif dan 886 000 kali lebih cepat daripada versi naifnya. Jangan menulis ulang sebuah rekursi menjadi gelung sambil berharap kecepatan; cari tahu dulu apakah ia mengulang dirinya.

2. Mengisi tabelnya dalam urutan yang salah. Pada deretan matriks, menggelung `for i` lalu `for j` akan membaca sel yang belum ditulis. Gelunglah atas **panjang** deretannya. Ujinya bersifat mekanis: untuk sel yang akan Anda hitung, daftarlah segala sesuatu yang menjadi sandarannya lalu periksa masing-masing punya nilai variabel gelung yang lebih kecil.

3. Salah menetapkan jangkauan k. Minimum deretan matriksnya berjalan atas $i \le k < j$. Menulis $k \le j$ akan menaruh seluruh deretannya di kiri dan tidak ada apa pun di kanan, dan programnya menghitung $M[i,j]$ dari $M[i,j]$ — yang entah menjadi gelung tak berhingga atau jawaban yang keliru, bergantung pada nilai awal tabel Anda.

4. Mengacaukan *subsequence* dengan *substring*. *Subsequence* boleh punya celah; *substring* tidak boleh. Jika LCS Anda untuk "ACGGTTA" dan "CGTAT" keluar sebagai 3, kemungkinan besar Anda menulis algoritma *longest common substring*, yang merupakan rekurensi yang berbeda (dan lebih sederhana).

5. Memakai *sentinel* yang mungkin disamai jawaban sungguhan. Jika "belum terhitung" dikodekan sebagai 0 dan ada submasalah yang jawaban sungguhannya memang 0, submasalah itu akan dihitung ulang setiap kali dan *memoization* Anda diam-diam tidak mengerjakan apa pun. Pakailah ∞, -1, atau bendera `computed` yang terpisah.

6. Melupakan tabel pilihannya lalu ternyata memerlukan solusinya. Tabel M memberi nilai optimalnya; hanya c yang memberi pengurungannya, dan hanya b yang memberi subbarisannya. Memutuskan belakangan bahwa Anda memerlukan solusinya berarti menjalankan semuanya dari awal lagi.

7. Salah-satu antara indeks untai dan indeks tabel. Tabelnya punya baris 0 dan kolom 0 untuk prefiks kosong, sehingga x_i adalah `X[i-1]` dalam C++. Tulislah baris itu sekali, dengan hati-hati, lalu jangan sentuh lagi.

8. Menganggap *optimal substructure* itu ada begitu saja. Buktikan dengan potong-dan-tempel, dan periksa penempelannya *sah*. Lintasan sederhana terpanjang tampak persis seperti lintasan terpendek dan sama sekali tidak punya *optimal substructure* (§7.5.4).

9. Meledakkan tumpukan dengan rekursi ber-*memoization* yang dalam. Rekursi *top-down* pada deretan berisi sejuta elemen akan meluap sebelum sempat menghitung apa pun: diukur di §7.4.5, kehancurannya terjadi antara $n = 250\,000$ dan $n = 300\,000$ pada tumpukan 8 MiB. *Bottom-up* tidak punya batas semacam itu.

10. Melaporkan jawaban yang terbalik. Rekonstruksinya berjalan dari ujung untainya secara mundur, sehingga karakternya keluar dalam urutan terbalik. Rekonstruksi rekursif pada **Kode 7.6** berekursi *sebelum* menambahkan, dan itu memperbaikinya dengan sendirinya; versi iteratif harus membalikkan untainya di akhir.

11. Membayar tabel yang tidak Anda perlukan. Jika hanya nilainya yang diperlukan, lupakan baris-baris yang tidak akan Anda baca lagi — $\Theta(n) \to \Theta(1)$ untuk Fibonacci, $\Theta(mn) \to \Theta(\min(m,n))$ untuk LCS. Pada $m = n = 8000$ itu berarti 488 MiB melawan 0.061 MiB, dan yang kecil berjalan dua belas kali lebih cepat.

12. Mengulurkan tangan ke *dynamic programming* padahal submasalahnya tidak tumpang tindih. Ukurlah nisbah panggilan terhadap submasalah berbeda, sebagaimana dilakukan **Kode 7.7**. Jika nilainya 1.00, tabel memo adalah biaya tambahan yang murni.

7.13 Ringkasan Bab

Divide and conquer membagi sebuah persoalan menjadi submasalah yang **saling lepas**, menyelesaikan masing-masing secara rekursif, lalu menggabungkan hasilnya (**Gambar 7.1**). Ia disiplin yang tepat ketika potongan-potongannya sungguh berbeda, seperti pada Merge Sort (**Gambar 7.2**).

Ketika potongan-potongannya *tidak* berbeda, rekursi yang polos menghitung ulang jawaban yang sama sebanyak eksponensial kali. Contoh terkecilnya adalah Fibonacci (**Gambar 7.3**), yang pohon rekursinya (**Gambar 7.4**) punya $2F_{n+1} - 1$ simpul yang hanya membawa $n + 1$ label berbeda, sehingga waktu jalannya $\Theta(\phi^n)$ dengan $\phi = \left(1 + \sqrt{5}\right)/2$ (**Gambar 7.5**). Diukur, itu berarti 866 juta panggilan dan dua detik untuk F_{42}.

Dua perbaikan menyelesaikannya, dan keduanya adalah *dynamic programming*: **memoization**, yang mempertahankan rekursinya tetapi memeriksa tabel jawaban yang sudah ditemukan; dan **tabulasi bottom-up**, yang meninggalkan rekursi dan membangun jawabannya mulai dari yang terkecil (**Gambar 7.6**). Keduanya membuat Fibonacci menjadi $\Theta(n)$. Sebuah jendela geser membuatnya $\Theta(1)$ dalam ruang, dan sebuah pangkat matriks membuatnya $\Theta(\lg n)$ dalam waktu (**Tabel 7.2**).

Dynamic programming berlaku pada **persoalan optimasi** (**Gambar 7.7**) yang punya **optimal substructure** — dibuktikan dengan argumen potong-dan-tempel — dan **overlapping subproblems**. Resepnya punya empat langkah (**Gambar 7.15**, **Gambar 7.16**, **Gambar 7.17**): cirikan, definisikan secara rekursif, hitung, susun.

Perkalian deretan matriks menanyakan pengurungan termurah atas $A_1 \cdots A_n$, dengan A_i berukuran $d_{i-1} \times d_i$ dan satu hasil kali berbiaya $n \cdot m \cdot k$ (**Gambar 7.8**). Pengurungannya sangat berpengaruh (**Gambar 7.9**): untuk empat matriks di §7.6.3, rentangnya 1 400 melawan 41 200. Banyaknya pengurungan sebanyak bilangan Catalan, sehingga *brute force* tidak mungkin (**Kode 7.3**). Membelah pada kurung terluar (**Gambar 7.10**) memberi rekurensinya, Persamaan (7.10) (**Gambar 7.11**)

$$M(i,j) = \min_{i \le k < j} \{M(i,k) + M(k+1,j) + d_{i-1} d_k d_j\},$$

dengan hanya $\Theta(n^2)$ submasalah berbeda (**Gambar 7.12**). **Matrix-Chain-Order** (**Pseudokode 7.5**) mengisi tabelnya menurut panjang deretan yang membesar dalam waktu $\Theta(n^3)$ — persis $(n^3 - n)/6$ iterasi terdalam — dan ruang $\Theta(n^2)$ (**Gambar 7.14**), dan **Print-Optimal-Parens** membaca pengurungannya dari tabel c (**Gambar 7.13**, **Tabel 7.3**, **Tabel 7.4**). Dikerjakan dengan tangan pada $d = \langle 10,20,3,5,30 \rangle$, jawabannya 1 950 dengan pengelompokan $((A_1 A_2)(A_3 A_4))$, dikukuhkan lewat tiga cara yang saling mandiri.

Rekurensi yang sama, ditulis sebagai rekursi ber-*memoization*, adalah **Lookup-Chain** (**Pseudokode 7.7**). Kedua bentuknya $\Theta(n^3)$; *bottom-up* menang pada tetapannya, *memoization* menang pada kemudahannya (**Tabel 7.5**).

Longest common subsequence (**Gambar 7.18**) punya *optimal substructure* dalam tiga kasus (**Gambar 7.19**), yang memberi rekurensinya, Persamaan (7.14) (**Gambar 7.20**)

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ atau } j = 0 \\ c[i-1,j-1]+1 & x_i = y_j \\ \max\{c[i,j-1], c[i-1,j]\} & x_i \neq y_j \end{cases}$$

LCS-Length (**Pseudokode 7.9**) mengisinya dalam $\Theta(mn)$, sambil mencatat panah di b. Pada untai-untai **Gambar 7.21** tabelnya adalah **Tabel 7.6** dan **Tabel 7.7**, jawabannya 4, dan mengikuti panahnya mundur memberi "CGTT". Membuang b dan menyimpan dua baris saja mengurangi ruangnya menjadi $\Theta(\min(m,n))$.

Garis antara kedua disiplin itu adalah submasalah yang **tumpang tindih versus saling bebas**, dan ia dapat diukur langsung (**Kode 7.7**, **Tabel 7.8**): nisbah panggilan terhadap submasalah berbeda milik Merge Sort persis 1.00; nisbah Fibonacci adalah 6002 pada $n = 24$.

7.13.1 Menatap ke depan

Bab 8, *Evaluasi dan Pembahasan II*, menggarap selengkapnya soal-soal yang diambil dari Bab 5 sampai 7, dan latihan pengulangannya kembali kepada dua ciri khas *dynamic programming* — submasalah yang tumpang tindih dan *optimal substructure* — yang sudah ditegakkan bab ini.

Bab 9 memperkenalkan **algoritma greedy** (rakus), teknik besar yang lain untuk persoalan optimasi. Algoritma *greedy* juga memerlukan *optimal substructure*, tetapi ia memerlukan sesuatu yang lebih kuat pula: *greedy-choice property* (sifat pilihan rakus), yang mengatakan bahwa pilihan terbaik secara lokal yang diambil sekarang merupakan bagian dari suatu solusi optimal global. Ketika sifat itu berlaku, Anda boleh mengambil pilihan yang kentara dan tak pernah menoleh ke belakang — tanpa tabel, tanpa minimisasi atas k, sering kali $\Theta(n\lg n)$ alih-alih $\Theta(n^3)$. Ketika sifat itu gagal — dan persoalan deretan matriks adalah contoh bagus persoalan yang di situ sifat itu gagal, sebab tidak ada aturan lokal sederhana yang memilih belahan yang benar — Anda kembali ke *dynamic programming*. Mengetahui mana di antara keduanya yang sedang Anda hadapi adalah keterampilan yang dibangun Bab 9, dan ia bersandar sepenuhnya pada gagasan *optimal substructure* yang didefinisikan di §7.5.2 bab ini.

7.14 Soal Latihan

7.1 Tuliskan F_0 sampai F_{12}, dengan memperlihatkan tiap penjumlahannya. Lalu hitunglah, memakai $C(n) = 2F_{n+1} - 1$, berapa banyak panggilan yang dibuat rekursi naif **Kode 7.1** untuk $n = 10$ dan untuk $n = 20$.

7.2 Buktikan bahwa pohon rekursi Fibonacci naif untuk `fib(n)` punya persis F_{n+1} daun, yang F_n di antaranya adalah `F(1)` dan F_{n-1} di antaranya adalah `F(0)` (untuk $n \geq 1$). Simpulkan bahwa nilai yang dikembalikannya, F_n, secara harfiah adalah banyaknya daun `F(1)`.

7.3 Telusuri iterasi jendela geser §7.4.3 untuk $n = 8$, satu baris satu iterasi, dalam format **Tabel 7.1**.

7.4 Pada keluaran **Kode 7.1**, nisbah cacah panggilan yang berurutan (berjarak dua langkah) adalah 2.618 pada setiap ukuran. Jelaskan persis mengapa, lalu ramalkan nisbahnya seandainya tabelnya melangkah 1 dan 3, bukan 2.

7.5 Misalkan A berukuran 5×10, B berukuran 10×2, C berukuran 2×8. Hitunglah biaya $(AB)C$ dan biaya $A(BC)$, dengan memperlihatkan tiap perkaliannya. Mana yang menang, dan dengan faktor berapa?

7.6 Memakai $P(1) = 1$ dan $P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$, hitunglah $P(6)$, $P(7)$, dan $P(8)$ dengan tangan.

7.7 Isilah tabel M dan c dengan tangan untuk deretan $d = \langle 30,35,15,5,10,20,25 \rangle$ $(n = 6)$. Berikan biaya optimalnya dan, memakai **Print-Optimal-Parens**, pengurungan optimalnya.

7.8 Memakai rumus eksak $(n^3 - n)/6$ yang diturunkan di §7.6.10, nyatakan berapa kali pernyataan terdalam **Matrix-Chain-Order** dieksekusi untuk $n = 6$ dan untuk $n = 10$. Periksalah angka $n = 6$-nya terhadap pekerjaan Anda pada Soal 7.7.

7.9 Tinjaulah heuristik *greedy* berikut: *lakukan berulang kali perkalian bersebelahan yang termurah yang tersedia.* Jalankan pada $d = \langle 10,20,3,5,30 \rangle$ lalu bandingkan dengan optimum 1 950 yang ditemukan di §7.6.8. Apa yang ditunjukkannya?

7.10 Anda hanya diberi tabel c berikut untuk deretan berisi lima matriks:

Tabel 7.9 Tabel c untuk Soal 7.10. Rekonstruksilah pengurungan yang ia kodekan.

	j=2	j=3	j=4	j=5
$i = 1$	1	1	3	3
$i = 2$	—	2	3	3
$i = 3$	—	—	3	4
$i = 4$	—	—	—	4

Jalankan **Print-Optimal-Parens** dengan tangan pada **Tabel 7.9** lalu tuliskan pengurungannya.

7.11 Hitunglah tabel c selengkapnya untuk $X = $ "ABCBDAB" dan $Y = $ "BDCABA" dengan tangan. Berikan panjang LCS-nya, dan rekonstruksilah satu LCS memakai aturan pemutus seri ≥ dari **Pseudokode 7.9**.

7.12 Untuk $X = $ "AGCAT" dan $Y = $ "GAC", bangunlah tabel c-nya lalu carilah panjang LCS-nya. Lalu carilah *semua longest common subsequence*-nya, bukan hanya satu.

7.13 Buktikan dua sifat tabel LCS mana pun: (a) $c[i,j] \geq c[i-1,j]$ dan $c[i,j] \geq c[i,j-1]$; (b) $c[i,j] \leq c[i-1,j] + 1$ dan $c[i,j] \leq c[i,j-1] + 1$. Apa yang dikatakan keduanya tentang bagaimana bilangannya boleh berubah saat Anda berjalan melintasi sebuah baris?

7.14 Varian dua baris §7.9.10 menghitung *panjang* LCS dalam ruang $\Theta(\min(m,n))$ tetapi tidak dapat merekonstruksi subbarisannya. Jelaskan dengan tepat mengapa tidak. Lalu bacalah tentang algoritma Hirschberg dan nyatakan, dalam dua kalimat, bagaimana ia memulihkan subbarisannya dalam ruang linear, dan berapa ongkosnya.

7.15 Edit distance (jarak sunting) antara dua untai adalah sesedikit mungkin penyisipan, penghapusan, dan penggantian satu karakter yang mengubah untai yang satu menjadi yang lain. Terapkan empat langkah §7.8 untuk menurunkan rekurensinya. Lalu hitunglah *edit distance* antara "kitten" dan "sitting" dengan tangan.

7.16 Tunjukkan bahwa **subbarisan menaik tegas terpanjang** (*longest strictly increasing subsequence*) dari sebuah barisan A berisi n bilangan yang berbeda-beda adalah LCS dari A dengan salinan A yang terurut. Waktu jalan berapa yang diberikannya, dan apakah itu yang terbaik yang diketahui?

7.17 **Kode 7.7** melaporkan nisbah panggilan terhadap submasalah berbeda yang persis 1.00 untuk Merge Sort pada setiap ukuran. Buktikan bahwa nisbahnya selalu persis 1, dan nyatakan berapa biaya sebuah tabel memo seandainya Anda tetap menambahkannya.

7.18 *(Tugas perancangan.)* Profitina Laundry punya kantong berkapasitas 3 kg, 5 kg, dan 8 kg. Tiap kantong yang dipakai berbiaya tarif tetap yang sama, dan sebuah kantong harus terisi *persis* sampai kapasitasnya. Terapkan empat langkah §7.8 untuk mencari sesedikit mungkin kantong yang memuat persis W kilogram. Selesaikan dengan tangan untuk setiap W dari 0 sampai 13, tulislah programnya, lalu tunjukkan bahwa aturan *greedy* "pakai kantong terbesar yang muat" gagal.

7.15 Penyelesaian Soal Latihan

7.1 $F_0 = 0$; $F_1 = 1$; $F_2 = 1 + 0 = 1$; $F_3 = 1 + 1 = 2$; $F_4 = 2 + 1 = 3$; $F_5 = 3 + 2 = 5$; $F_6 = 5 + 3 = 8$; $F_7 = 8 + 5 = 13$; $F_8 = 13 + 8 = 21$; $F_9 = 21 + 13 = 34$; $F_{10} = 34 + 21 = 55$; $F_{11} = 55 + 34 = 89$; $F_{12} = 89 + 55 = 144$.

Untuk $n = 10$: $C(10) = 2F_{11} - 1 = 2(89) - 1 = 177$ panggilan untuk menghitung bilangan 55. Untuk $n = 20$: $F_{21} = 10\,946$, sehingga $C(20) = 2(10\,946) - 1 = 21\,891$. Itu persis angka yang dicetak **Kode 7.1** untuk $n = 20$.

7.2 Misalkan $L(n)$ adalah banyaknya daun, $L_1(n)$ banyaknya daun `F(1)`, dan $L_0(n)$ banyaknya daun `F(0)`.

Daunnya. Sebuah panggilan dengan $n \leq 1$ adalah daun itu sendiri, sehingga $L(0) = L(1) = 1$. Sebuah panggilan dengan $n \geq 2$ adalah simpul dalam, dan daunnya persis adalah daun kedua anaknya, sehingga $L(n) = L(n-1) + L(n-2)$. Itu rekurensi Fibonacci dengan $L(0) = L(1) = 1$. Karena $F_1 = F_2 = 1$, barisan L adalah F yang digeser satu: $L(n) = F_{n+1}$. (Secara formal, dengan induksi kuat: ia berlaku pada $n = 0,1$, dan $L(n) = L(n-1) + L(n-2) = F_n + F_{n-1} = F_{n+1}$.)

Pembagiannya. L_1 memenuhi $L_1(0) = 0$, $L_1(1) = 1$, $L_1(n) = L_1(n-1) + L_1(n-2)$ — rekurensi Fibonacci dengan nilai awal Fibonacci sendiri — sehingga $L_1(n) = F_n$. Serupa itu $L_0(0) = 1$, $L_0(1) = 0$, memberi $L_0(n) = F_{n-1}$. Periksa: $F_n + F_{n-1} = F_{n+1} = L(n)$ ✓.

Simpulannya. Setiap simpul dalam mengembalikan jumlah anak-anaknya, dan setiap daun mengembalikan nilainya sendiri — 1 untuk daun `F(1)`, 0 untuk daun `F(0)`. Jadi nilai akarnya adalah jumlah seluruh nilai daunnya, yaitu $1 \cdot L_1(n) + 0 \cdot L_0(n) = F_n$. Algoritmanya menghitung F_n dengan secara harfiah menjumlahkan F_n buah angka satu, satu per satu. Untuk $n = 6$: delapan daun `F(1)` dan lima daun `F(0)`, dan $F_6 = 8$ — persis yang dicetak **Kode 7.1** dan yang ditunjukkan **Gambar 7.4**.

7.3 Penelusurannya adalah **Tabel 7.10**.

Tabel 7.10 Jendela geser yang menghitung $F_8 = 21$. Dua variabel, tujuh iterasi, tanpa larik.

i	prev sebelum	cur sebelum	next	prev sesudah	cur sesudah
2	0	1	1	1	1
3	1	1	2	1	2
4	1	2	3	2	3
5	2	3	5	3	5
6	3	5	8	5	8
7	5	8	13	8	13
8	8	13	21	13	21

7.4 Cacah panggilannya adalah $C(n) = 2F_{n+1} - 1$, dan $F_n \approx \phi^n/\sqrt{5}$, sehingga $C(n) \approx \left(2/\sqrt{5}\right)\phi^{n+1}$. Maka

$$\frac{C(n+2)}{C(n)} \approx \frac{\phi^{n+3}}{\phi^{n+1}} = \phi^2 = 2.6180339\ldots$$

yang cocok dengan hasil pengukuran 2.618 sampai setiap digit yang dicetak. Melangkah 1 akan memberi $\phi^1 = 1.618$; melangkah 3 akan memberi $\phi^3 = 4.236$. (Perhatikan $\phi^2 = \phi + 1$, sebab ϕ menyelesaikan $x^2 = x + 1$ — jadi nisbahnya sekadar $1.618 + 1$.)

7.5

$(AB)C$: AB berbiaya $5 \times 10 \times 2 = 100$, menghasilkan matriks 5×2; lalu $(5 \times 2)(2 \times 8)$ berbiaya $5 \times 2 \times 8 = 80$. Total $100 + 80 = \mathbf{180}$.

$A(BC)$: BC berbiaya $10 \times 2 \times 8 = 160$, menghasilkan matriks 10×8; lalu $(5 \times 10)(10 \times 8)$ berbiaya $5 \times 10 \times 8 = 400$. Total $160 + 400 = \mathbf{560}$.

$(AB)C$ menang dengan faktor $560/180 = 3.1$. Alasannya sama seperti di §7.6.3: $(AB)C$ langsung meruntuhkan ukuran dalamnya dari 10 menjadi 2 dan tak pernah membangun matriks antara 10×8.

7.6 Dengan $P(1) = P(2) = 1, P(3) = 2, P(4) = 5, P(5) = 14$ dari §7.6.4:

$P(6) = P_1P_5 + P_2P_4 + P_3P_3 + P_4P_2 + P_5P_1 = 14 + 5 + 4 + 5 + 14 = \mathbf{42}$.

$P(7) = P_1P_6 + P_2P_5 + P_3P_4 + P_4P_3 + P_5P_2 + P_6P_1 = 42 + 14 + 10 + 10 + 14 + 42 = \mathbf{132}$.

$P(8) = P_1P_7 + P_2P_6 + P_3P_5 + P_4P_4 + P_5P_3 + P_6P_2 + P_7P_1 = 132 + 42 + 28 + 25 + 28 + 42 + 132 = \mathbf{429}$.

Semuanya sesuai dengan bilangan Catalan yang dicetak **Kode 7.5**.

7.7 Dengan $d = \langle 30,35,15,5,10,20,25 \rangle$, sehingga A_1 berukuran 30×35, A_2 berukuran 35×15, A_3 berukuran 15×5, A_4 berukuran 5×10, A_5 berukuran 10×20, A_6 berukuran 20×25.

Panjang 2. $M[1,2] = 30 \cdot 35 \cdot 15 = 15\,750$ $(c = 1)$; $M[2,3] = 35 \cdot 15 \cdot 5 = 2\,625$ $(c = 2)$; $M[3,4] = 15 \cdot 5 \cdot 10 = 750$ $(c = 3)$; $M[4,5] = 5 \cdot 10 \cdot 20 = 1\,000$ $(c = 4)$; $M[5,6] = 10 \cdot 20 \cdot 25 = 5\,000$ $(c = 5)$.

Panjang 3. $M[1,3]$: $k = 1$ memberi $0 + 2625 + 30 \cdot 35 \cdot 5 = 2625 + 5250 = 7\,875$; $k = 2$ memberi $15750 + 0 + 30 \cdot 15 \cdot 5 = 15750 + 2250 = 18\,000$. Min 7 875, $c = 1$. $M[2,4]$: $k = 2$: $0 + 750 + 35 \cdot 15 \cdot 10 = 750 + 5250 = 6\,000$; $k = 3$: $2625 + 0 + 35 \cdot 5 \cdot 10 = 2625 + 1750 = 4\,375$. Min 4 375, $c = 3$. $M[3,5]$: $k = 3$: $0 + 1000 + 15 \cdot 5 \cdot 20 = 1000 + 1500 = 2\,500$; $k = 4$: $750 + 0 + 15 \cdot 10 \cdot 20 = 750 + 3000 = 3\,750$. Min 2 500, $c = 3$. $M[4,6]$: $k = 4$: $0 + 5000 + 5 \cdot 10 \cdot 25 = 5000 + 1250 = 6\,250$; $k = 5$: $1000 + 0 + 5 \cdot 20 \cdot 25 = 1000 + 2500 = 3\,500$. Min 3 500, $c = 5$.

Panjang 4. $M[1,4]$: $k = 1$: $0 + 4375 + 30 \cdot 35 \cdot 10 = 4375 + 10500 = 14\,875$; $k = 2$: $15750 + 750 + 30 \cdot 15 \cdot 10 = 16500 + 4500 = 21\,000$; $k = 3$: $7875 + 0 + 30 \cdot 5 \cdot 10 = 7875 + 1500 = 9\,375$. Min 9 375, $c = 3$. $M[2,5]$: $k = 2$: $0 + 2500 + 35 \cdot 15 \cdot 20 = 2500 + 10500 = 13\,000$; $k = 3$: $2625 + 1000 + 35 \cdot 5 \cdot 20 = 3625 + 3500 = 7\,125$; $k = 4$: $4375 + 0 + 35 \cdot 10 \cdot 20 = 4375 + 7000 = 11\,375$. Min 7 125, $c = 3$. $M[3,6]$: $k = 3$: $0 + 3500 + 15 \cdot 5 \cdot 25 = 3500 + 1875 = 5\,375$; $k = 4$: $750 + 5000 + 15 \cdot 10 \cdot 25 = 5750 + 3750 = 9\,500$; $k = 5$: $2500 + 0 + 15 \cdot 20 \cdot 25 = 2500 + 7500 = 10\,000$. Min 5 375, $c = 3$.

Panjang 5. $M[1,5]$: $k = 1$: $0 + 7125 + 30 \cdot 35 \cdot 20 = 7125 + 21000 = 28\,125$; $k = 2$: $15750 + 2500 + 30 \cdot 15 \cdot 20 = 18250 + 9000 = 27\,250$; $k = 3$: $7875 + 1000 + 30 \cdot 5 \cdot 20 = 8875 + 3000 = 11\,875$; $k = 4$: $9375 + 0 + 30 \cdot 10 \cdot 20 = 9375 + 6000 = 15\,375$. Min $11\,875$, $c = 3$. $M[2,6]$: $k = 2$: $0 + 5375 + 35 \cdot 15 \cdot 25 = 5375 + 13125 = 18\,500$; $k = 3$: $2625 + 3500 + 35 \cdot 5 \cdot 25 = 6125 + 4375 = 10\,500$; $k = 4$: $4375 + 5000 + 35 \cdot 10 \cdot 25 = 9375 + 8750 = 18\,125$; $k = 5$: $7125 + 0 + 35 \cdot 20 \cdot 25 = 7125 + 17500 = 24\,625$. Min $10\,500$, $c = 3$.

Tabel M yang sudah selesai adalah **Tabel 7.11**.

Panjang 6. $M[1,6]$: $k = 1$: $0 + 10500 + 30 \cdot 35 \cdot 25 = 10500 + 26250 = 36\,750$; $k = 2$: $15750 + 5375 + 30 \cdot 15 \cdot 25 = 21125 + 11250 = 32\,375$; $k = 3$: $7875 + 3500 + 30 \cdot 5 \cdot 25 = 11375 + 3750 = 15\,125$; $k = 4$: $9375 + 5000 + 30 \cdot 10 \cdot 25 = 14375 + 7500 = 21\,875$; $k = 5$: $11875 + 0 + 30 \cdot 20 \cdot 25 = 11875 + 15000 = 26\,875$. Min **15 125**, $c = 3$.

Tabel 7.11 Tabel M untuk Soal 7.7. Tabel c-nya adalah $c[1,2] = 1$, $c[1,3] = 1$, $c[1,4] = 3$, $c[1,5] = 3$, $c[1,6] = 3$; $c[2,3] = 2$, $c[2,4] = 3$, $c[2,5] = 3$, $c[2,6] = 3$; $c[3,4] = 3$, $c[3,5] = 3$, $c[3,6] = 3$; $c[4,5] = 4$, $c[4,6] = 5$; $c[5,6] = 5$.

	j=1	j=2	j=3	j=4	j=5	j=6
$i = 1$	0	15750	7875	9375	11875	**15125**
$i = 2$	—	0	2625	4375	7125	10500
$i = 3$	—	—	0	750	2500	5375
$i = 4$	—	—	—	0	1000	3500
$i = 5$	—	—	—	—	0	5000
$i = 6$	—	—	—	—	—	0

Rekonstruksinya: $c[1,6] = 3$ membelah menjadi (1,3) dan (4,6). $c[1,3] = 1$ membelah menjadi A_1 dan (2,3); $c[2,3] = 2$ memberi (A_2A_3). Jadi bagian kirinya adalah $(A_1(A_2A_3))$. $c[4,6] = 5$ membelah menjadi (4,5) dan A_6; $c[4,5] = 4$ memberi (A_4A_5). Jadi bagian kanannya adalah $((A_4A_5)A_6)$. Pengurungan optimalnya adalah

$$((A_1(A_2A_3))\,((A_4A_5)A_6)) \quad \text{pada 15 125 perkalian.}$$

7.8 $n = 6$: $(6^3 - 6)/6 = (216 - 6)/6 = 35$. $n = 10$: $(1000 - 10)/6 = 165$. Memeriksa angka $n = 6$-nya terhadap Soal 7.7: panjang 2, 3, 4, 5, 6 menyumbang $5 \times 1 + 4 \times 2 + 3 \times 3 + 2 \times 4 + 1 \times 5 = 5 + 8 + 9 + 8 + 5 = 35$ ✓.

7.9 Hasil kali bersebelahan yang tersedia berbiaya $A_1A_2 = 10 \cdot 20 \cdot 3 = 600$, $A_2A_3 = 20 \cdot 3 \cdot 5 = 300$, $A_3A_4 = 3 \cdot 5 \cdot 30 = 450$.

Langkah 1. Termurah adalah A_2A_3 seharga 300. Kerjakan; deretannya menjadi 10×20, 20×5, 5×30, yaitu $d = \langle 10,20,5,30 \rangle$. Total berjalan 300. *Langkah 2.* Kini pilihannya adalah $A_1(A_2A_3) = 10 \cdot 20 \cdot 5 = 1\,000$ dan $(A_2A_3)A_4 = 20 \cdot 5 \cdot 30 = 3\,000$. Termurah adalah 1 000. Total berjalan 1 300. *Langkah 3.* Tersisa satu hasil kali: $10 \cdot 5 \cdot 30 = 1\,500$. **Total** 2 800**.**

Optimumnya 1 950, jadi heuristik *greedy*-nya 43% lebih buruk. Jawabannya, 2 800, adalah pengurungan $((A_1(A_2A_3))A_4)$, yang menurut **Kode 7.3** hanya merupakan yang *ketiga* termurah di antara kelimanya, di belakang 1 950 dan 2 250. Ini menunjukkan bahwa perkalian deretan matriks tidak punya *greedy-choice property*: perkalian pertama yang termurah secara lokal bukan bagian dari solusi optimal mana pun di sini. Bandingkan dengan Bab 9, tempat persoalan yang *memang* punya sifat itu dibahas.

7.10 `parens(1,5)`: $c[1,5] = 3$, jadi `"(" + parens(1,3) + parens(4,5) + ")"`. `parens(1,3)`: $c[1,3] = 1$, jadi `"(" + parens(1,1) + parens(2,3) + ")"` = `"(A1" + parens(2,3) + ")"`. `parens(2,3)`: $c[2,3] = 2$, memberi `"(A2A3)"`. Jadi bagian kirinya adalah `"(A1(A2A3))"`. `parens(4,5)`: $c[4,5] = 4$, memberi `"(A4A5)"`. Jawabannya: $((A_1(A_2A_3))(A_4A_5))$.

7.11 $X = $ "ABCBDAB" ($m = 7$), $Y = $ "BDCABA" ($n = 6$). Tabel yang sudah lengkap adalah **Tabel 7.12**.

Tabel 7.12 Tabel c untuk Soal 7.11. Panjang LCS-nya adalah $c[7,6] = 4$.

	j=0	B	D	C	A	B	A
$i = 0$	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	**4**

Merekonstruksi dari (7,6) dengan aturan $\geq$: $x_7 =$ B, $y_6 =$ A — tidak cocok; atasnya $c[6,6] = 4$, kirinya $c[7,5] = 4$, dan $4 \geq 4$ jadi naik ke (6,6). Di sana $x_6 =$ A $= y_6 =$ A — cocok, keluarkan **A**, pindah ke (5,5). Di sana D lawan B — tidak cocok; atasnya $c[4,5] = 3$, kirinya $c[5,4] = 2$, $3 \geq 2$ jadi naik ke (4,5). Di sana B = B — cocok, keluarkan **B**, pindah ke (3,4). Di sana C lawan A — tidak cocok; atasnya $c[2,4] = 1$, kirinya $c[3,3] = 2$, $1 \geq 2$ salah, jadi ke kiri menuju (3,3). Di sana C = C — cocok, keluarkan **C**, pindah ke (2,2). Di sana B lawan D — tidak cocok; atasnya $c[1,2] = 0$, kirinya $c[2,1] = 1$, jadi ke kiri menuju (2,1). Di sana B = B — cocok, keluarkan **B**, pindah ke (1,0). Berhenti.

Yang dikeluarkan A, B, C, B; dibalik, LCS-nya adalah **"BCBA"**, berpanjang 4.

7.12 $X = $ "AGCAT", $Y = $ "GAC". Tabelnya adalah **Tabel 7.13**.

Tabel 7.13 Tabel c untuk Soal 7.12. Panjang LCS-nya adalah 2, dan tiga LCS berbeda mencapainya.

	j=0	G	A	C
$i = 0$	0	0	0	0
A	0	0	1	1
G	0	1	1	1
C	0	1	1	2
A	0	1	2	2
T	0	1	2	**2**

Panjang LCS-nya adalah 2. Ada tiga *longest common subsequence* yang berbeda:

- **"AC"** — posisi 1, 3 dari X; posisi 2, 3 dari Y.
- **"GA"** — posisi 2, 4 dari X; posisi 1, 2 dari Y.
- **"GC"** — posisi 2, 3 dari X; posisi 1, 3 dari Y.

Algoritmanya hanya mengembalikan salah satunya, ditentukan oleh aturan pemutus serinya; dengan aturan $\geq$ dari **Pseudokode 7.9** ia mengembalikan "AC". Inilah sebabnya definisi di §7.9.2 berbunyi "*sebuah* longest common subsequence".

7.13 (a) Setiap subbarisan bersama dari X_{i-1} dan Y_j juga merupakan subbarisan bersama dari X_i dan Y_j (subbarisan dari sebuah prefiks adalah subbarisan dari untai yang lebih panjang). Jadi yang terbaik untuk (i, j) setidaknya sebaik yang terbaik untuk $(i-1, j)$: $c[i, j] \geq c[i-1, j]$. Argumen yang sama dengan X dan Y dipertukarkan memberi $c[i, j] \geq c[i, j-1]$.

(b) Misalkan Z adalah LCS dari X_i dan Y_j, berpanjang $c[i, j]$. Hapus karakter terakhirnya jika karakter itu dipasangkan dengan x_i; sisanya adalah subbarisan bersama dari X_{i-1} dan Y_j, berpanjang setidaknya $c[i, j] - 1$. Jika karakter terakhirnya *tidak* dipasangkan dengan x_i, maka Z sendiri sudah merupakan subbarisan bersama dari X_{i-1} dan Y_j. Bagaimanapun juga $c[i-1, j] \geq c[i, j] - 1$, yaitu $c[i, j] \leq c[i-1, j] + 1$. Setangkup itu untuk kolomnya.

Bersama-sama keduanya mengatakan: melangkah satu ke kanan atau satu ke bawah, nilainya entah tetap sama atau naik persis 1 — ia tak pernah turun dan tak pernah melompat 2. Itulah invarian yang harus diperiksa ketika mengawasi tabel LCS dengan mata; pada **Tabel 7.6** baris untuk $i = 5$ berbunyi 0,1,2,3,3,3, dan setiap langkahnya $+1$ atau $+0$ ✓.

7.14 *Mengapa rekonstruksinya gagal.* Rekonstruksinya berjalan *mundur* menembus pilihan-pilihannya, yang berarti ia memerlukan panah (atau nilai) pada sel di baris-baris yang sudah ia lewati. Versi dua barisnya menimpa tiap baris begitu baris berikutnya terhitung, sehingga pada saat baris terakhir selesai, baris 0 sampai $m - 2$ sudah tidak ada lagi. Informasinya bukan sekadar dimampatkan; ia dimusnahkan.

Algoritma Hirschberg. Ia membelah X menjadi dua pada baris $m/2$, menghitung dalam ruang linear panjang LCS $X_{1..m/2}$ terhadap setiap prefiks Y (satu lintasan dua baris maju) dan $X_{m/2+1..m}$ terhadap setiap akhiran Y (satu lintasan dua baris mundur), lalu memilih kolom j^* yang memaksimumkan jumlahnya — yang terbukti merupakan titik yang dilewati perentetan optimalnya. Ia lalu berekursi pada kedua paruhnya, $(X_{1..m/2}, Y_{1..j^*})$ dan $(X_{m/2+1..m}, Y_{j^*+1..n})$. Ongkosnya waktu $\Theta(mn)$ — kira-kira dua kali versi polosnya, sebab kerjanya memaruh pada tiap aras dan $mn + mn/2 + mn/4 + \cdots = 2mn$ — dalam ruang $\Theta(\min(m, n))$. Ia adalah algoritma *divide and conquer* yang membungkus sebuah program dinamis: kedua teknik bab ini dalam satu. Lihat [3, §14.4] dan [6].

7.15 *Langkah 1 — cirikan.* Tinjau karakter terakhirnya x_m dan y_n. Pilihannya adalah operasi sunting terakhirnya apa. Jika $x_m = y_n$ kita boleh memasangkannya secara cuma-cuma. Selain itu ada tiga kemungkinan: **hapus** x_m (mereduksi menjadi X_{m-1} lawan Y_n), **sisipkan** y_n (mereduksi menjadi X_m lawan Y_{n-1}), atau **ganti** x_m dengan y_n (mereduksi menjadi X_{m-1} lawan Y_{n-1}). Menurut argumen potong-dan-tempel, operasi mana pun yang dipakai optimumnya, transformasi sisanya haruslah sendirinya optimal — kalau tidak, gantilah dengan yang lebih murah dan totalnya turun.

Langkah 2 — rekurensinya. Misalkan $e[i, j]$ adalah *edit distance* antara X_i dan Y_j. Maka berlaku Persamaan (7.16):

$$e[i,j] = \begin{cases} j & i = 0 \quad \text{(sisipkan} \\ i & j = 0 \quad \text{(hapus se} \\ e[i-1, j-1] & x_i = y_j \\ 1 + \min\{e[i-1, j],\ e[i, j-1],\ e[i-1, j-1]\} & x_i \neq y_j \end{cases} \tag{7.16}$$

Submasalahnya: $(m+1)(n+1) = \Theta(mn)$ — polinomial ✓.

Langkah 3 dan 4. Isi baris demi baris dalam waktu $\Theta(mn)$ dan ruang $\Theta(mn)$ (atau dua baris bila hanya jaraknya yang diinginkan); catat operasi pemenangnya untuk merekonstruksi urutan suntingannya.

"kitten" → "sitting". $m = 6, n = 7$; tabel yang sudah terisi adalah **Tabel 7.14**.

Tabel 7.14 Tabel *edit distance* untuk "kitten" dan "sitting". Jawabannya, $e[6,7] = 3$, adalah sel yang dicetak tebal.

	—	s	i	t	t	i	n	g
—	0	1	2	3	4	5	6	7
k	1	1	2	3	4	5	6	7
i	2	2	1	2	3	4	5	6
t	3	3	2	1	2	3	4	5
t	4	4	3	2	1	2	3	4
e	5	5	4	3	2	2	3	4
n	6	6	5	4	3	3	2	**3**

Jaraknya adalah **3**: ganti k→s, ganti e→i, sisipkan g.

7.16 Misalkan S adalah salinan A yang terurut. Sebuah subbarisan bersama dari A dan S adalah barisan nilai yang muncul di A dalam urutan aslinya *dan* di S dalam urutan terurut — yaitu subbarisan A yang menaik. Sebaliknya setiap subbarisan menaik dari A merupakan subbarisan keduanya. Jadi kedua himpunannya berimpit, dan anggota terpanjangnya pun berimpit.

Elemennya harus **berbeda-beda** supaya ini bekerja. Dengan pengulangan, sebuah subbarisan bersama dari A dan S bisa jadi tak-turun alih-alih menaik tegas; perbaikan bakunya adalah mengurutkan dengan duplikatnya dibuang.

Waktu jalannya: $\Theta(n\lg n)$ untuk pengurutannya ditambah $\Theta(n^2)$ untuk LCS-nya, jadi $\Theta(n^2)$. Itu **bukan** yang terbaik yang diketahui: sebuah program dinamis langsung dengan pencarian biner atas tabel nilai ekor terkecil menyelesaikan *longest increasing subsequence* dalam $\Theta(n\lg n)$; Halim *dkk.* [7, §3.5] memberikan versi itu lengkap dengan kode siap lomba. Reduksi ke LCS adalah cara yang baik untuk *melihat* persoalannya, bukan cara untuk menyelesaikannya.

7.17 Sebuah panggilan Merge Sort ditandai oleh rentang (lo, hi) yang ia urutkan. Rentang-rentang yang dihasilkannya membentuk pohon biner yang di dalamnya rentang tiap simpul dibelah menjadi dua paruh yang lalu tak pernah dihasilkan lagi oleh simpul lain mana pun: anak-anak (lo, hi) adalah (lo, mid) dan (mid, hi), dan keduanya hanyalah subrentang dari (lo, hi). Dua simpul berbeda pada kedalaman yang sama punya rentang yang saling lepas; dua simpul pada kedalaman berbeda punya rentang yang entah bersarang tegas atau saling lepas, dan pasangan yang bersarang tegas tidak mungkin sama. Karena itu tidak ada rentang yang pernah dihasilkan dua kali, sehingga submasalah berbeda = panggilan dan nisbahnya persis 1. (Ada $2n - 1$ untuk masing-masingnya bila elemennya n: cacah 15,31,47 pada **Kode 7.7** adalah $2 \cdot 8 - 1$, $2 \cdot 16 - 1$, dan $2 \cdot 24 - 1$.)

Tabel memo berkunci (lo, hi) akan memerlukan satu entri per panggilan — $\Theta(n)$ entri, masing-masing memuat sebuah larik terurut, jadi seluruhnya $\Theta(n\lg n)$ kata — dan setiap penelusurannya akan meleset. Biaya murni, manfaat nol.

7.18 *Langkah 1 — cirikan.* Pilihannya adalah *kantong mana yang diisi terakhir*. Jika kantong terakhirnya berkapasitas s, sisa muatannya seberat $W - s$ dan harus dikemas dalam sesedikit

mungkin kantong. Potong dan tempel: jika subpengemasan itu tidak minimal, gantilah dengan yang lebih kecil dan totalnya turun — kontradiksi. *Optimal substructure* ✓.

Langkah 2 — rekurensinya. Misalkan $B[w]$ adalah sesedikit mungkin kantong yang memuat persis w kilogram, atau ∞ bila w tidak dapat dibentuk persis. Maka berlaku Persamaan (7.17):

$$B[0] = 0, \qquad B[w] = 1 + \min_{s \in \{3,5,8\},\, s \le w} B[w - s]. \tag{7.17}$$

Submasalahnya: satu per w dari 0 sampai W, jadi $W + 1$ — polinomial dalam W ✓.

Langkah 3 — hitung, bottom-up dalam w yang membesar, satu baris masing-masing:

- $B[0] = 0$.
- $B[1]$: tidak ada kantong yang muat. ∞.
- $B[2]$: tidak ada kantong yang muat. ∞.
- $B[3] = 1 + B[0] = 1$. Kantong: 3.
- $B[4] = 1 + B[1] = \infty$.
- $B[5] = 1 + \min(B[2], B[0]) = 1 + 0 = 1$. Kantong: 5.
- $B[6] = 1 + \min(B[3], B[1]) = 1 + 1 = 2$. Kantong: 3+3.
- $B[7] = 1 + \min(B[4], B[2]) = \infty$.
- $B[8] = 1 + \min(B[5], B[3], B[0]) = 1 + 0 = 1$. Kantong: 8.
- $B[9] = 1 + \min(B[6], B[4], B[1]) = 1 + 2 = 3$. Kantong: 3+3+3.
- $B[10] = 1 + \min(B[7], B[5], B[2]) = 1 + 1 = 2$. Kantong: 5+5.
- $B[11] = 1 + \min(B[8], B[6], B[3]) = 1 + 1 = 2$. Kantong: 3+8.
- $B[12] = 1 + \min(B[9], B[7], B[4]) = 1 + 3 = 4$. Kantong: 3+3+3+3.
- $B[13] = 1 + \min(B[10], B[8], B[5]) = 1 + 1 = 2$. Kantong: 5+8.

Langkah 4 — susun: catat, bersama tiap $B[w]$, ukuran s yang mencapai minimumnya, lalu kupas satu kantong pada satu waktu.

Kode 7.9 adalah programnya.

Kode 7.9 Program dinamis pengemasan kantong Profitina Laundry dari Soal 7.18, lengkap dengan perbandingan *greedy*. Kompilasi dengan `g++ -std=c++17 -O2 -o bags bags.cpp`.

```
// bags.cpp - Profitina Laundry: sesedikit mungkin kantong yang memuat persis W kilogram,
// dengan kapasitas kantong yang masing-masing harus terisi persis. DP empat langkah.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <climits>
using namespace std;

int main() {
    vector<int> size = {3, 5, 8};                  // kapasitas kantong yang tersedia, dalam
kg
    int W = 13;
    const int INF = INT_MAX / 2;
    vector<int> B(W + 1, INF), choice(W + 1, -1); // B[w] = sesedikit mungkin kantong yang
memuat persis w
    B[0] = 0;                                       // nol kilogram perlu nol kantong
    for (int w = 1; w <= W; ++w)
        for (int s : size)
            if (s <= w && B[w - s] + 1 < B[w]) { B[w] = B[w - s] + 1; choice[w] = s; }

    cout << left << setw(6) << "w" << setw(10) << "bags" << "one optimal packing\n";
    for (int w = 0; w <= W; ++w) {
        cout << left << setw(6) << w;
        if (B[w] >= INF) { cout << setw(10) << "impossible" << "\n"; continue; }
        cout << setw(10) << B[w];
        for (int t = w; t > 0; t -= choice[t]) cout << choice[t] << (t - choice[t] ? " + "
: "");
        cout << "\n";
    }
    // Greedy, terbesar-dulu, sebagai pembanding.
    for (int w : {12, 13}) {
        int rem = w, used = 0; bool ok = true;
        for (int idx = (int)size.size() - 1; idx >= 0; --idx)
            while (rem >= size[idx]) { rem -= size[idx]; ++used; }
        if (rem != 0) ok = false;
        cout << "greedy largest-first on w=" << w << ": "
             << (ok ? to_string(used) + " bags" : "FAILS, " + to_string(rem) + " kg left
over")
             << "   (optimal = " << B[w] << ")\n";
    }
    return 0;
}
```

Keluarannya:

```
w     bags      one optimal packing
0     0
1     impossible
2     impossible
3     1         3
4     impossible
5     1         5
6     2         3 + 3
7     impossible
8     1         8
9     3         3 + 3 + 3
10    2         5 + 5
11    2         3 + 8
12    4         3 + 3 + 3 + 3
13    2         5 + 8
greedy largest-first on w=12: FAILS, 1 kg left over   (optimal = 4)
greedy largest-first on w=13: 2 bags   (optimal = 2)
```

Setiap barisnya cocok dengan perhitungan tangan di atas.

Mengapa greedy gagal. Pada $W = 12$ aturan *greedy*-nya mengambil kantong 8 kg, menyisakan 4 kg; ia lalu mengambil kantong 3 kg, menyisakan 1 kg, yang tidak dapat dimuat kantong mana pun — *greedy* bukan sekadar menghasilkan jawaban yang lebih buruk, ia menghasilkan jawaban *nihil*, sedangkan optimumnya adalah empat kantong 3 kg. Perhatikan pula ketidakmonotonan yang menggemaskan itu: 12 kg memerlukan empat kantong tetapi 13 kg hanya memerlukan dua. Aturan apa pun yang mengandaikan "berat lebih besar perlu kantong lebih banyak" itu keliru, dan hanya tabelnya yang tahu.

Daftar Pustaka

[1] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[2] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton Univ. Press, 1957.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[4] R. P. Stanley, *Catalan Numbers*. Cambridge, U.K.: Cambridge Univ. Press, 2015.

[5] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[6] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, vol. 18, no. 6, pp. 341–343, Jun. 1975.

[7] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, 4th ed. Raleigh, NC, USA: Lulu Press, 2020.

Bab 8
Evaluasi dan Pembahasan II: Pengurutan, Pohon Pencarian, dan Pemrograman Dinamis

Bab ini menghimpun Bab 5, 6, dan 7 menjadi satu lokakarya panjang berisi soal-soal yang diselesaikan tuntas. Di akhir bab ini Anda akan telah membangun sebuah *max-heap* dengan tangan, satu pertukaran demi satu pertukaran, lalu mengurutkan hasilnya dengan *heapsort*; melacak partisi Lomuto satu indeks demi satu indeks dan menghitung, dengan persis, banyaknya perbandingan yang dilakukan Quicksort pada kasus terbaik dan terburuknya; menulis, dari asas yang paling dasar, delapan rutin atas pohon pencarian biner dan pohon AVL — sebuah pemeriksa, dua penelusuran, sebuah pemeriksa keseimbangan, penyisipan, dua penyeimbang ulang, dan penghapusan — masing-masing disajikan sebagai program C++ utuh yang sudah dikompilasi dan dijalankan; mengerjakan keluarga soal kedua pada pohon merah-hitam; mengisi tabel rantai matriks dan tabel *longest common subsequence* sel demi sel lalu membaca jawabannya kembali dari tabel itu; dan menyelesaikan serangkaian soal lanjutan pada pohon statistik-urutan yang memaksa setiap gagasan bab ini bekerja bersamaan. Bila sebuah soal punya jawaban keliru yang menggoda, Anda akan melihatnya ditulis, dijalankan, lalu dibongkar, sebab jawaban keliru yang Anda pahami mengajarkan lebih banyak daripada jawaban benar yang Anda hafalkan. Dua puluh lima pemeriksaan cepat, empat belas soal latihan, dan penyelesaian yang dikerjakan lengkap menutup bab ini.

8.1 Di Mana Kita Berdiri

Tiga bab sudah kita lalui. Bab 5 mengajarkan dua cara mengurutkan yang mengalahkan metode kuadratik Bab 2: *heapsort*, yang menyembunyikan sebuah pohon di dalam larik, dan Quicksort, yang membagi larik menurut nilai di sekitar sebuah pivot. Bab 6 mengubah pohon dari bentuk tersembunyi menjadi struktur data yang sungguhan, yaitu pohon pencarian biner, lalu memperbaiki satu kelemahan besarnya — bahwa ia dapat tumbuh setinggi rantai — dengan pohon AVL yang menyeimbangkan dirinya sendiri. Bab 7 kembali ke rekursi yang menggerakkan semua itu dan bertanya kapan rekursi membuang tenaganya, pertanyaan yang membawa kita ke *memoization* dan pemrograman dinamis (*dynamic programming*).

Bab ini tidak memperkenalkan teori baru. Ia mengerjakan sesuatu yang sama pentingnya: ia mengambil teori ketiga bab itu dan *memakainya*, pada soal-soal yang cukup besar sehingga beberapa gagasan harus bekerja sama. Setiap soal dinyatakan, lalu dianalisis sampai isinya yang sebenarnya tampak, lalu diselesaikan langkah demi langkah dengan setiap nilai antara dituliskan, lalu diimplementasikan sebagai program utuh yang keluarannya tercantum di dalam teks persis seperti yang dihasilkan mesin. Kebanyakan soal ditutup dengan belokan keliru yang paling sering diambil pembaca, dan soal-soal baru bab ini ditutup dengan varian *periksa diri* — contoh kedua dari soal yang sama, lengkap dengan jawabannya, supaya Anda dapat menguji apakah kini Anda sanggup mengerjakannya sendiri.

Tabel 8.1 mendaftar setiap soal bernomor dalam bab ini, subbab tempat ia diselesaikan, dan bab terdahulu yang memasok teorinya.

Tabel 8.1 Soal-soal bab ini, subbab yang menyelesaikan masing-masing, dan bab yang teorinya dipakai. Nomor soal dan nomor subbab adalah dua barisan yang berbeda; tabel ini menerjemahkan yang satu ke yang lain.

Soal	Subbab	Yang ditanyakan	Teori dari
8.1	8.3	Membangun *max-heap* dengan tangan, lalu mengurutkannya dengan *heapsort*	Bab 5
8.2	8.4	Melacak partisi Quicksort; kasus terbaik dan terburuk	Bab 5
8.3	8.6	`isBST`: apakah pohon biner ini pohon pencarian?	Bab 6
8.4	8.7	`printDescending`: mencetak BST dari yang terbesar, tanpa membangun daftar	Bab 6
8.5	8.8	`max`: kunci terbesar lewat satu lintasan saja	Bab 6
8.6	8.9	`isHeightBalanced`: syarat AVL dalam satu lintasan	Bab 6, 7
8.7	8.10	`insertHB`: penyisipan yang menjaga pohon tetap seimbang	Bab 6
8.8	8.11	`rebalanceForLeft`: kasus LL dan LR	Bab 6
8.9	8.12	`rebalanceForRight`: kasus RR dan RL	Bab 6
8.10	8.13	`deleteHB`: penghapusan yang menjaga pohon tetap seimbang	Bab 6
8.11–8.15	8.14	Pohon merah-hitam: empat pemeriksa dan dua rotasi	Bab 6
8.16	8.15	Perkalian rantai matriks, tabelnya diisi dengan tangan	Bab 7
8.17	8.16	*Longest common subsequence*, tabelnya dan jalan mundurnya	Bab 7
8.18–8.25	8.18	Soal lanjutan pada pohon statistik-urutan	Bab 5, 6, 7

Soal-soal itu menimba lebih dari bab "asalnya". Soal tentang pohon tetap memerlukan invarian gelung dari Bab 2 untuk membuktikan kebenarannya dan rekurensi dari Bab 3 untuk menghitung biayanya. **Tabel 8.2** adalah petanya: apa yang disumbangkan setiap bab terdahulu, dan di bagian mana bab ini Anda melihatnya dipakai.

Tabel 8.2 Peta pengulangan. Setiap bab sebelum bab ini menyumbangkan sesuatu yang benar-benar diperlukan soal-soal bab ini.

Bab	Yang disumbangkannya	Tempat ia dipakai di sini
1	Kerangka lima fase: memahami, mengabstraksi, merancang, menganalisis, mengimplementasikan	§8.2, urutan menulis
1	Dua tujuan: benar dahulu, baru efisien	§8.6.3, pemeriksaan lokal yang cepat tetapi keliru
2	Model RAM: operasi mana yang berbiaya satu satuan	§8.8.4, mengapa "satu lintasan" adalah seluruh jawabannya
2	Invarian gelung dan ketiga kewajibannya	§8.4.3, keempat daerah partisi; §8.6.4, invarian batas
2	Pencarian biner dan penurunan $\Theta(\lg n)$	§8.8.3, lintasan paling kanan adalah sebuah penurunan
3	O, Ω, Θ, dan cara menyatakan batas dengan jujur	§8.8.5, $O(h)$ lawan $O(\lg n)$
3	Penjumlahan: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$	§8.4.5, kasus terburuk Quicksort; §8.9.3, menghitung ulang tinggi
3	Pohon rekursi dan Teorema Master	§8.4.6, kasus terbaik Quicksort; §8.9.6, kedua pemeriksaan keseimbangan
5	*Heap*, `MAX-HEAPIFY`, `BUILD-MAX-HEAP`, *heapsort*	Soal 8.1; §8.6.6, mengapa sifat heap bukan sifat BST
5	Partisi Quicksort: membagi menurut nilai, bukan menurut posisi	Soal 8.2; §8.6.4, batas-batas itu adalah partisi ruang kunci
6	Sifat BST, penelusuran, minimum dan maksimum	Soal 8.3, 8.4, 8.5
6	Faktor keseimbangan, rotasi, keempat kasus, $1.44\lg n$	Soal 8.6 sampai 8.10, dan 8.11 sampai 8.15
7	Rekursi yang mengembalikan sebuah *nilai* ke atas, bukan sekadar efek samping	§8.9.4, teknik penjaga
7	*Memoization* dan pemrograman dinamis	Soal 8.16 dan 8.17; §8.9.3, penghitungan ulang tinggi yang naif

WAWASAN KUNCI: SATU GAGASAN DI BALIK HAMPIR SETIAP SOAL DI SINI. Hampir setiap soal dalam bab ini adalah gagasan yang sama dengan pakaian berbeda: *bawa informasi turun menyusuri rekursi, atau kembalikan ke atas, sehingga tak ada sepotong kerja pun yang dikerjakan dua kali.* `MAX-HEAPIFY` membawa satu nilai yang salah tempat turun sampai ia mapan. Soal 8.3 membawa sepasang batas turun. Soal 8.6 mengembalikan tinggi ke atas. Soal 8.7 sampai 8.10 mengembalikan subpohon yang dibangun ulang ke atas. Program dinamis Soal 8.16 dan 8.17 menyimpan jawaban setiap subsoal sehingga ia dihitung sekali lalu hanya dicari. Soal 8.4 dan 8.5 tidak membawa apa pun dan tidak mengembalikan apa pun — dan keduanya adalah dua soal paling sederhana dalam bab ini. Jika Anda memahami *aliran informasi turun-dan-naik di dalam rekursi*, Anda dapat menurunkan setiap jawaban dalam bab ini dari nol.

8.2 Cara Mendekati Soal Bertingkat

Beberapa soal dalam bab ini datang berkeluarga: Soal 8.7 sampai 8.10 adalah empat rutin yang hanya bekerja bersama-sama, dan Soal 8.18 sampai 8.25 adalah delapan rutin atas satu struktur data. Soal sebesar ini menghargai sedikit metode sebelum kode apa pun ditulis. Tak satu pun dari yang berikut khusus untuk pohon; semuanya adalah kerangka lima fase Bab 1, dipadatkan agar muat untuk satu soal.

Bacalah setiap bagian sebelum menyelesaikan bagian mana pun, dan gambarlah ketergantungannya. Membaca Soal 8.7 sampai 8.10 bersama-sama menyingkapkan tiga fakta

yang tak terlihat bila soal-soal itu dibaca satu per satu: penyisipan (Soal 8.7) dan penghapusan (Soal 8.10) sama-sama *memanggil* kedua penyeimbang ulang; penyeimbang ulang yang kedua (Soal 8.9) adalah bayangan cermin yang pertama (Soal 8.8); dan penghapusan memerlukan sebuah maksimum, yang sudah ditulis oleh Soal 8.5. Jadi urutan kerja yang masuk akal adalah 8.8, lalu 8.9 (salinan cermin 8.8), lalu 8.7, lalu 8.10 — bukan urutan penomoran soalnya. Menyelesaikan soal menurut urutan ketergantungan berarti setiap rutin ditulis satu kali saja.

Sebelum menulis kode, tuliskan lima hal, dalam urutan ini.

1. **Tanda tangan fungsinya, dengan persis.** `bool isBST(MyTree t)`. Tanda tangan menetapkan apa yang diterima rutin itu dan apa yang harus dikembalikannya, dan cukup banyak jawaban keliru ternyata adalah jawaban benar untuk tanda tangan yang sedikit berbeda.
2. **Kasus dasarnya, dalam kata-kata.** "Pohon kosong adalah BST." "Pohon kosong bertinggi -1." "`max` dari pohon kosong tidak terdefinisi." Setiap rutin dalam bab ini adalah rekursi, setiap rekursi memerlukan kasus dasar, dan menamainya dengan kata-kata sebelum menuliskannya sebagai kode mencegah kegagalan yang paling sering di antara semuanya: mengakses pohon kosong.
3. **Invariannya atau kontraknya, dalam satu kalimat.** "Setiap nilai di subpohon ini terletak ketat di antara `lowerBound` dan `upperBound`." "Subpohon kiri mungkin baru bertambah tinggi satu takik; kedua subpohon sudah seimbang." Kalimat inilah rancangannya. Tuliskan ia sebagai komentar tepat di atas fungsinya.
4. **Contoh gagal yang terkecil.** Bukan masukan yang besar — melainkan masukan *terkecil* yang membuat penyelesaian naif memberi hasil keliru. Untuk Soal 8.3 itu adalah rantai tiga simpul di §8.6.3; untuk Soal 8.8 itu adalah zig-zag tiga simpul di §8.11.4. Menuliskannya memaksa Anda memeriksa penyelesaian Anda terhadapnya.
5. **Klaim kompleksitasnya, dengan pembenaran setengah kalimat.** "$\Theta(n)$: setiap simpul dikunjungi sekali." "$O(h)$: satu panggilan rekursif per aras." Jangan menulis "$O(\lg n)$" kecuali Anda dapat mengatakan mengapa pohonnya pendek; §8.8.5 menjelaskan mengapa perbedaan itu penting.

Dalam praktiknya kelima langkah ini memakan tiga menit dan menghemat jauh lebih banyak.

Ujilah kasus-kasus kecil lebih dahulu. Masukan kosong, satu elemen, dua elemen, dan kunci yang tidak ada menemukan lebih banyak kutu daripada contoh lima belas elemen, dan semuanya hanya memakan hitungan detik. Setiap program dalam bab ini mengujinya.

Katakan apa yang belum ditangani. Jika sebuah rutin belum rampung — misalnya penghapusan yang belum menangani simpul berdua anak — tuliskan hal itu di samping kodenya, bersama masukan yang membuatnya gagal. Kekurangan yang diketahui dapat ditutup besok; kekurangan yang tidak disadari ikut terkirim.

> **WAWASAN KUNCI: ASAS PENCERMINAN SEBAGAI PENGHEMAT KERJA.** Mencerminkan sebuah pohon biner dari kiri ke kanan menukar kiri dan kanan, $<$ dan $>$, minimum dan maksimum, serta kedua rotasi. Setiap rutin yang benar, bila dicerminkan simbol demi simbol, adalah rutin yang benar untuk soal cerminannya. Mengenali pasangan cermin sebelum mulai — `max` dan `min`, kedua penyeimbang ulang, kedua rotasi pohon merah-hitam — memangkas kerja menjadi separuhnya. §8.12.2 menyatakan asas ini dengan persis.

8.3 Soal 8.1 — Membangun *Max-Heap*, Langkah demi Langkah

8.3.1 Soalnya

Ambil larik sepuluh elemen

$$A = [5,\ 13,\ 2,\ 25,\ 7,\ 17,\ 20,\ 8,\ 4,\ 11].$$

1. Gambarlah A sebagai pohon biner hampir lengkap yang diwakilinya, dengan indeks setiap simpul.
2. Jalankan `BUILD-MAX-HEAP` pada A. Setelah setiap panggilan `MAX-HEAPIFY`, tuliskan seluruh lariknya, dan daftarkan setiap pertukaran yang dilakukan panggilan itu.
3. Hitunglah seluruh perbandingan kunci dan pertukaran yang dilakukan pembangunan itu.
4. Urutkan *heap* hasilnya dengan *heapsort*, dan berikan lariknya setelah setiap pengambilan maksimum.
5. Jelaskan mengapa `BUILD-MAX-HEAP` berbiaya $O(n)$ dan bukan $O(n\lg n)$, lalu periksa penjelasan itu terhadap sebuah pengukuran.

8.3.2 Yang Anda perlukan dari Bab 5

Max-heap adalah larik $A[1..n]$ yang memenuhi **sifat max-heap**: untuk setiap indeks $i > 1$, $A[\text{PARENT}(i)] \geq A[i]$. Pohonnya tidak pernah disimpan; ia dihitung dari indeksnya:

$$\text{PARENT}(i) = \lfloor i/2 \rfloor, \qquad \text{LEFT}(i) = 2i, \qquad \text{RIGHT}(i) = 2i + 1.$$

Sebagai contoh, anak-anak indeks 3 adalah indeks 6 dan 7, dan induk indeks 10 adalah indeks 5.

Mesinnya adalah `MAX-HEAPIFY`, **Pseudokode 8.1**. Prasyaratnya adalah kedua subpohon simpul i sudah berupa *max-heap*, sementara $A[i]$ sendiri mungkin terlalu kecil. Ia membiarkan satu nilai itu *tenggelam*: ia membandingkan simpulnya dengan anak-anaknya, menukarnya dengan anak yang lebih besar bila anak itu lebih besar, lalu mengulanginya satu aras di bawahnya.

Pseudokode 8.1 `MAX-HEAPIFY`: nilai di simpul i tenggelam sampai tak ada anak yang lebih besar daripadanya.

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

`BUILD-MAX-HEAP`, **Pseudokode 8.2**, memanggil `MAX-HEAPIFY` pada setiap simpul dalam, dari yang terakhir mundur sampai ke akar. Simpul $\lfloor n/2 \rfloor + 1, \ldots, n$ adalah daun, dan daun sudah merupakan *heap* satu elemen, sehingga gelungnya mulai dari $\lfloor n/2 \rfloor$. Ia menghitung *mundur* agar kedua anak sebuah simpul sudah selesai sebelum simpul itu sendiri diproses — persis prasyarat yang diperlukan `MAX-HEAPIFY`.

Pseudokode 8.2 `BUILD-MAX-HEAP`: memperbaiki simpul-simpul dalam dari bawah ke atas.

```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

8.3.3 Langkah 1: larik sebagai pohon

Di sini $n = 10$, jadi simpul dalamnya adalah $1, \ldots, \lfloor 10/2 \rfloor = 5$ dan daunnya adalah $6, \ldots, 10$. Menempatkan $A[1]$ di akar, $A[2]$ dan $A[3]$ di aras berikutnya, $A[4]$ sampai $A[7]$ di aras sesudahnya, dan $A[8]$ sampai $A[10]$ di aras terakhir, dari kiri ke kanan, menghasilkan pohon di bawah ini. Setiap simpul ditulis sebagai *nilai* dengan indeksnya di dalam kurung siku.

```
                    5 [1]
              /                 \
         13 [2]                   2 [3]
        /      \                 /      \
    25 [4]     7 [5]         17 [6]     20 [7]
    /    \       /
 8 [8]  4 [9]  11 [10]
```

Periksa aritmetika indeksnya pada dua simpul: simpul 4 beranak $2 \cdot 4 = 8$ dan $2 \cdot 4 + 1 = 9$, yang memuat 8 dan 4; simpul 5 beranak 10 dan 11, tetapi $11 > n$, jadi simpul 5 hanya punya anak kiri, yang memuat 11. Pohon ini sangat tidak teratur — akarnya memuat 5 sementara 25 duduk dua aras di bawahnya.

8.3.4 Langkah 2: pelacakannya, satu panggilan demi satu panggilan

Gelungnya berjalan untuk $i = 5,4,3,2,1$. Pada setiap panggilan kita tuliskan setiap perbandingan baris 3 dan baris 6, sebab itulah perbandingan yang diminta untuk dihitung.

Panggilan `MAX-HEAPIFY(A, 5)`. Simpul 5 memuat 7. Anak kirinya adalah indeks 10, yang memuat 11; anak kanannya semestinya indeks 11, yang tidak ada.

- Baris 3: $10 \leq 10$ dan $A[10] = 11 > A[5] = 7$? **Ya**, jadi $largest = 10$. (1 perbandingan.)
- Baris 6: $r = 11 > 10$, jadi anak kanan dilewati; tak ada perbandingan.
- Baris 8: $largest = 10 \neq 5$, jadi **tukar** $A[5] = 7$ dengan $A[10] = 11$.
- Panggilan rekursif `MAX-HEAPIFY(A, 10)` mendapati $\text{LEFT}(10) = 20 > 10$: simpul 10 adalah daun, tak ada yang dibandingkan, dan panggilan itu selesai.

$A = [5,13,2,25,\mathbf{11},17,20,8,4,\mathbf{7}]$. Satu perbandingan, satu pertukaran.

Panggilan `MAX-HEAPIFY(A, 4)`. Simpul 4 memuat 25, dengan anak $A[8] = 8$ dan $A[9] = 4$.

- Baris 3: $8 > 25$? Tidak, jadi $largest = 4$. (1 perbandingan.)
- Baris 6: $4 > A[largest] = 25$? Tidak. (1 perbandingan.)
- $largest = i$: tak ada yang perlu dikerjakan.

A tidak berubah. Dua perbandingan, tanpa pertukaran. Nilai 25 sudah lebih besar daripada semua yang ada di bawahnya.

Panggilan `MAX-HEAPIFY(A, 3)`. Simpul 3 memuat 2, dengan anak $A[6] = 17$ dan $A[7] = 20$.

- Baris 3: $17 > 2$? **Ya**, $largest = 6$. (1 perbandingan.)
- Baris 6: $20 > A[6] = 17$? **Ya**, $largest = 7$. (1 perbandingan.) Perhatikan bahwa anak kanan dibandingkan dengan *juara saat ini* 17, bukan dengan 2 yang semula.
- **Tukar** $A[3] = 2$ dengan $A[7] = 20$. Panggilan `MAX-HEAPIFY(A, 7)` mendapati simpul 7 adalah daun ($\text{LEFT}(7) = 14 > 10$) dan berhenti.

$A = [5,13,\mathbf{20},25,11,17,\mathbf{2},8,4,7]$. Dua perbandingan, satu pertukaran.

Panggilan `MAX-HEAPIFY(A, 2)`. Simpul 2 memuat 13, dengan anak $A[4] = 25$ dan $A[5] = 11$.

- Baris 3: $25 > 13$? **Ya**, $largest = 4$. Baris 6: $11 > 25$? Tidak. (2 perbandingan.)
- **Tukar** $A[2] = 13$ dengan $A[4] = 25$.
- Panggilan rekursif `MAX-HEAPIFY(A, 4)`: simpul 4 kini memuat 13, dengan anak $A[8] = 8$ dan $A[9] = 4$. $8 > 13$? Tidak. $4 > 13$? Tidak. (2 perbandingan.) Berhenti.

$A = [5,\mathbf{25},20,\mathbf{13},11,17,2,8,4,7]$. Empat perbandingan, satu pertukaran.

Panggilan `MAX-HEAPIFY(A, 1)`. Akarnya memuat 5, nilai terkecil di aras-aras atas, dan nilai itu akan tenggelam sampai ke dasar.

- Di simpul 1: anaknya $A[2] = 25$ dan $A[3] = 20$. $25 > 5$? **Ya**, $largest = 2$. $20 > 25$? Tidak. (2 perbandingan.) **Tukar** $A[1] \leftrightarrow A[2]$: kini $A[1] = 25$, $A[2] = 5$.
- Di simpul 2: anaknya $A[4] = 13$ dan $A[5] = 11$. $13 > 5$? **Ya**. $11 > 13$? Tidak. (2 perbandingan.) **Tukar** $A[2] \leftrightarrow A[4]$: kini $A[2] = 13$, $A[4] = 5$.
- Di simpul 4: anaknya $A[8] = 8$ dan $A[9] = 4$. $8 > 5$? **Ya**. $4 > 8$? Tidak. (2 perbandingan.) **Tukar** $A[4] \leftrightarrow A[8]$: kini $A[4] = 8$, $A[8] = 5$.
- Di simpul 8: $\text{LEFT}(8) = 16 > 10$, sebuah daun. Berhenti.

$A = [\mathbf{25},\mathbf{13},20,\mathbf{8},11,17,2,\mathbf{5},4,7]$. Enam perbandingan, tiga pertukaran.

Tabel 8.3 menghimpun kelima panggilan itu.

Tabel 8.3 `BUILD-MAX-HEAP` pada $A = [5,13,2,25,7,17,20,8,4,11]$, satu baris per panggilan `MAX-HEAPIFY`. Nilai di akar tenggelam tiga aras pada panggilan terakhir.

Panggilan	Nilai di simpul *i*	Perbandingan	Pertukaran	*A* Setelah panggilan
`MAX-HEAPIFY(A, 5)`	7	1	1	[5,13,2,25,11,17,20,8,4,7]
`MAX-HEAPIFY(A, 4)`	25	2	0	[5,13,2,25,11,17,20,8,4,7]
`MAX-HEAPIFY(A, 3)`	2	2	1	[5,13,20,25,11,17,2,8,4,7]
`MAX-HEAPIFY(A, 2)`	13	4	1	[5,25,20,13,11,17,2,8,4,7]
`MAX-HEAPIFY(A, 1)`	5	6	3	[25,13,20,8,11,17,2,5,4,7]
Jumlah		**15**	**6**	

Periksa hasilnya. Klaim *max-heap* diperiksa pasangan demi pasangan, induk terhadap setiap anaknya:

- simpul 1 (25): anak 13 dan 20 — keduanya lebih kecil ✓
- simpul 2 (13): anak 8 dan 11 ✓
- simpul 3 (20): anak 17 dan 2 ✓
- simpul 4 (8): anak 5 dan 4 ✓
- simpul 5 (11): anak 7 ✓

Setiap simpul dalam paling sedikit sebesar anak-anaknya, jadi [25,13,20,8,11,17,2,5,4,7] adalah *max-heap*, dan maksimumnya, 25, berada di akar, sebagaimana mestinya.

8.3.5 Langkah 3: *heapsort* pada hasilnya

Heapsort mengulang tiga gerakan: menukar $A[1]$, sang maksimum, dengan elemen terakhir *heap*; menyusutkan *heap* satu elemen, sehingga maksimum itu kini berada di luarnya dan di tempat akhirnya; lalu memanggil `MAX-HEAPIFY(A, 1)` untuk memperbaiki akarnya. Kerjakan tiga pengambilan pertama dengan tangan.

Pengambilan pertama. Tukar $A[1] = 25$ dengan $A[10] = 7$, dan susutkan *heap* menjadi 9 elemen: [7,13,20,8,11,17,2,5,4 | 25]. Kini tenggelamkan 7: anak-anaknya 13 dan 20; yang lebih besar adalah 20 di indeks 3; tukar, sehingga $A[1] = 20$, $A[3] = 7$. Di indeks 3 anak-anaknya adalah $A[6] = 17$ dan $A[7] = 2$; $17 > 7$, jadi tukar, sehingga $A[3] = 17$, $A[6] = 7$. Indeks 6 memiliki $\text{LEFT}(6) = 12 > 9$, sebuah daun. Hasil: [20,13,17,8,11,7,2,5,4 | 25].

Pengambilan kedua. Tukar $A[1] = 20$ dengan $A[9] = 4$; ukuran *heap* 8: [4,13,17,8,11,7,2,5 | 20,25]. Tenggelamkan 4: anaknya 13 dan 17, yang lebih besar 17 di indeks 3; tukar. Di indeks 3: anaknya $A[6] = 7$ dan $A[7] = 2$; $7 > 4$; tukar. Hasil: [17,13,7,8,11,4,2,5 | 20,25].

Pengambilan ketiga. Tukar $A[1] = 17$ dengan $A[8] = 5$; ukuran *heap* 7: [5,13,7,8,11,4,2 | 17,20,25]. Tenggelamkan 5: anaknya 13 dan 7, yang lebih besar 13 di indeks 2; tukar. Di indeks 2: anaknya $A[4] = 8$ dan $A[5] = 11$; yang lebih besar 11; $11 > 5$; tukar. Indeks 5 memiliki $\text{LEFT}(5) = 10 > 7$. Hasil: [13,11,7,8,5,4,2 | 17,20,25].

Enam pengambilan sisanya mengikuti pola yang sama, dan program di §8.3.7 mencetak setiap satunya. Setelah pengambilan kesembilan, *heap* hanya memuat satu elemen, yang pasti yang terkecil, dan larik itu terbaca [2,4,5,7,8,11,13,17,20,25]: terurut, di tempat, tanpa larik kedua.

8.3.6 Langkah 4: mengapa pembangunannya hanya berbiaya $O(n)$

Taksiran cepat berbunyi: $\lfloor n/2 \rfloor$ panggilan, masing-masing berbiaya $O(\lg n)$, jadi $O(n \lg n)$. Batas itu benar tetapi tidak ketat, dan alasan kelonggarannya tampak di **Tabel 8.3**: kebanyakan panggilan dilakukan pada simpul di dekat dasar, tempat hampir tak ada ruang untuk tenggelam. Panggilan di simpul 5 paling banyak dapat tenggelam satu aras; hanya panggilan di akar yang dapat tenggelam tiga.

Kita buat itu persis. Simpul dengan **tinggi** h — banyaknya sisi pada lintasan menurun terpanjang darinya ke sebuah daun — berbiaya $O(h)$ dalam `MAX-HEAPIFY`, sebab nilainya paling banyak dapat tenggelam h aras. *Heap* dengan n elemen memiliki paling banyak $\lceil n/2^{h+1} \rceil$ simpul bertinggi h (Bab 5 membuktikannya). Menjumlahkan atas semua tinggi,

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(2n) = O(n), \tag{8.1}$$

dengan memakai deret $\sum_{h \geq 0} h/2^h = 2$ dari Bab 3.

Untuk *heap* yang **sempurna** kita bahkan dapat menemukan kasus terburuknya dengan persis. Misalkan $n = 2^k - 1$, sehingga pohonnya memiliki aras $0, 1, \ldots, k-1$, dengan 2^ℓ simpul pada aras ℓ, masing-masing bertinggi $k - 1 - \ell$. Jumlah seluruh aras yang dapat ditempuh semua nilai saat tenggelam adalah jumlah tinggi-tinggi itu:

$$\sum_{\ell=0}^{k-1} 2^\ell (k - 1 - \ell) = 2^k - k - 1 = n - \lg(n+1). \tag{8.2}$$

Periksalah untuk $k = 3$, $n = 7$: akarnya bertinggi 2, kedua anaknya bertinggi 1, keempat daunnya bertinggi 0, jadi jumlahnya $2 + 1 + 1 = 4$, dan $7 - \lg 8 = 7 - 3 = 4$. ✓

Setiap aras yang ditempuh adalah satu pertukaran dan paling banyak dua perbandingan, jadi menurut Persamaan (8.2) `BUILD-MAX-HEAP` tak pernah melakukan lebih dari $n - \lg(n + 1)$ pertukaran pada *heap* sempurna — kurang dari n, apa pun masukannya. Persamaan (8.1) adalah pernyataan umum dari fakta yang sama. Program di bawah ini mengukur pertukaran pada kunci-kunci yang berurutan *naik*, yang ternyata membuat setiap nilai tenggelam sejauh mungkin; hitungan yang terukur sama persis dengan $n - \lg(n + 1)$, dari $n = 7$ sampai $n = 32\,767$.

8.3.7 Programnya

Kode 8.1 mengikuti **Pseudokode 8.1** dan **Pseudokode 8.2** baris demi baris, dengan larik berbasis 1 sehingga aritmetika indeksnya persis $2i$ dan $2i + 1$. Ia mencetak pelacakan §8.3.4, larik periksa-diri §8.3.8, *heapsort* §8.3.5, dan pengukuran §8.3.6.

Kode 8.1 Soal 8.1 selengkapnya: `MAX-HEAPIFY` dan `BUILD-MAX-HEAP` dengan pelacakan panggilan demi panggilan, *heapsort* dengan lariknya setelah setiap pengambilan, dan pengukuran batas liniernya.

```
// buildheap.cpp -- Soal 8.1: BUILD-MAX-HEAP dilacak panggilan demi panggilan, lalu
heapsort.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <utility>
using namespace std;

vector<int> A;            // A[0] tidak dipakai, sehingga heap menempati A[1..n] seperti
pada pseudokode
int heapSize = 0;
long long comparisons = 0, swaps = 0;

int LEFT(int i)  { return 2 * i; }
int RIGHT(int i) { return 2 * i + 1; }

string show() {                          // seluruh larik, bagian heap maupun ekor yang
sudah terurut
    string s = "[";
    for (size_t k = 1; k < A.size(); ++k) s += (k > 1 ? " " : "") + to_string(A[k]);
    return s + "]";
}

// MAX-HEAPIFY persis seperti pseudokodenya; setiap pertukaran dilaporkan bila trace
menyala.
void maxHeapify(int i, bool trace) {
    int l = LEFT(i), r = RIGHT(i), largest = i;
    if (l <= heapSize) { ++comparisons; if (A[l] > A[i])       largest = l; }
    if (r <= heapSize) { ++comparisons; if (A[r] > A[largest]) largest = r; }
    if (largest != i) {
        if (trace) cout << "      swap A[" << i << "]=" << A[i] << " with A[" << largest
                        << "]=" << A[largest] << "\n";
        swap(A[i], A[largest]); ++swaps;
        maxHeapify(largest, trace);          // nilainya terus tenggelam
    }
}

void buildMaxHeap(bool trace) {
    heapSize = (int)A.size() - 1;
    for (int i = heapSize / 2; i >= 1; --i) {
        if (trace) cout << "  MAX-HEAPIFY(A, " << i << ")  node value " << A[i] << "\n";
        maxHeapify(i, trace);
        if (trace) cout << "    A = " << show() << "\n";
    }
}
```

Kode 8.1 (Lanjutan) *Soal 8.1 selengkapnya: MAX-HEAPIFY dan BUILD-MAX-HEAP dengan pelacakan panggilan demi panggilan, heapsort dengan lariknya setelah setiap pengambilan, dan pengukuran batas liniernya.*

```
bool isMaxHeap() {
    for (int i = 2; i <= heapSize; ++i) if (A[i / 2] < A[i]) return false;
    return true;
}

void heapsort(bool trace) {
    buildMaxHeap(false);
    for (int i = (int)A.size() - 1; i >= 2; --i) {
        swap(A[1], A[i]); ++swaps;          // maksimum pindah ke tempat akhirnya
        --heapSize;
        maxHeapify(1, false);
        if (trace) cout << "  after extracting " << setw(2) << A[i] << ": " << show()
                        << "   heap-size = " << heapSize << "\n";
    }
}

void run(const vector<int>& input, const char* name) {
    A = input; A.insert(A.begin(), 0);
    comparisons = swaps = 0;
    cout << name << "\n  input A = " << show() << "\n";
    buildMaxHeap(true);
    cout << "  result is a max-heap: " << (isMaxHeap() ? "yes" : "NO")
         << "   comparisons = " << comparisons << ", swaps = " << swaps << "\n\n";
}

int main() {
    run({5, 13, 2, 25, 7, 17, 20, 8, 4, 11}, "BUILD-MAX-HEAP, the worked problem");
    run({3, 9, 2, 1, 4, 5}, "BUILD-MAX-HEAP, the check-yourself array");

    cout << "HEAPSORT on the worked array\n";
    A = {0, 5, 13, 2, 25, 7, 17, 20, 8, 4, 11};
    comparisons = swaps = 0;
    heapsort(true);
    cout << "  sorted: " << show() << "\n";

    cout << "\nThe linear bound: swaps made by BUILD-MAX-HEAP on n = 2^k - 1 keys in\n"
            "ascending order (every value must sink as far as it can)\n";
    cout << "      n     swaps    n - lg(n+1)\n";
    for (int k = 3; k <= 15; k += 3) {
        int n = (1 << k) - 1;
        A.assign(n + 1, 0);
        for (int v = 1; v <= n; ++v) A[v] = v;
        swaps = 0; buildMaxHeap(false);
        cout << setw(7) << n << setw(10) << swaps << setw(15) << (n - k) << "\n";
    }
    return 0;
```

Kode 8.1 (Lanjutan) *Soal 8.1 selengkapnya: MAX-HEAPIFY dan BUILD-MAX-HEAP dengan pelacakan panggilan demi panggilan, heapsort dengan lariknya setelah setiap pengambilan, dan pengukuran batas liniernya.*

```
}
```

Dikompilasi dengan `g++ -std=c++17 -O2 -Wall` lalu dijalankan, ia mencetak:

```
BUILD-MAX-HEAP, the worked problem
  input A = [5 13 2 25 7 17 20 8 4 11]
  MAX-HEAPIFY(A, 5)  node value 7
      swap A[5]=7 with A[10]=11
    A = [5 13 2 25 11 17 20 8 4 7]
  MAX-HEAPIFY(A, 4)  node value 25
    A = [5 13 2 25 11 17 20 8 4 7]
  MAX-HEAPIFY(A, 3)  node value 2
      swap A[3]=2 with A[7]=20
    A = [5 13 20 25 11 17 2 8 4 7]
  MAX-HEAPIFY(A, 2)  node value 13
      swap A[2]=13 with A[4]=25
    A = [5 25 20 13 11 17 2 8 4 7]
  MAX-HEAPIFY(A, 1)  node value 5
      swap A[1]=5 with A[2]=25
      swap A[2]=5 with A[4]=13
      swap A[4]=5 with A[8]=8
    A = [25 13 20 8 11 17 2 5 4 7]
  result is a max-heap: yes   comparisons = 15, swaps = 6

BUILD-MAX-HEAP, the check-yourself array
  input A = [3 9 2 1 4 5]
  MAX-HEAPIFY(A, 3)  node value 2
      swap A[3]=2 with A[6]=5
    A = [3 9 5 1 4 2]
  MAX-HEAPIFY(A, 2)  node value 9
    A = [3 9 5 1 4 2]
  MAX-HEAPIFY(A, 1)  node value 3
      swap A[1]=3 with A[2]=9
      swap A[2]=3 with A[5]=4
    A = [9 4 5 1 3 2]
  result is a max-heap: yes   comparisons = 7, swaps = 3

HEAPSORT on the worked array
  after extracting 25: [20 13 17 8 11 7 2 5 4 25]   heap-size = 9
  after extracting 20: [17 13 7 8 11 4 2 5 20 25]   heap-size = 8
  after extracting 17: [13 11 7 8 5 4 2 17 20 25]   heap-size = 7
  after extracting 13: [11 8 7 2 5 4 13 17 20 25]   heap-size = 6
  after extracting 11: [8 5 7 2 4 11 13 17 20 25]   heap-size = 5
  after extracting  8: [7 5 4 2 8 11 13 17 20 25]   heap-size = 4
  after extracting  7: [5 2 4 7 8 11 13 17 20 25]   heap-size = 3
  after extracting  5: [4 2 5 7 8 11 13 17 20 25]   heap-size = 2
  after extracting  4: [2 4 5 7 8 11 13 17 20 25]   heap-size = 1
  sorted: [2 4 5 7 8 11 13 17 20 25]

The linear bound: swaps made by BUILD-MAX-HEAP on n = 2^k - 1 keys in
ascending order (every value must sink as far as it can)
      n     swaps    n - lg(n+1)
      7         4              4
     63        57             57
    511       502            502
   4095      4083           4083
  32767     32752          32752
```

Bacalah keluaran itu berdampingan dengan kerja tangan. Kelima blok pelacakan pertama adalah kelima baris **Tabel 8.3**, pertukaran demi pertukaran, dan jumlahnya adalah 15 perbandingan dan 6 pertukaran. Tiga baris *heapsort* yang pertama adalah ketiga pengambilan di §8.3.5. Tabel terakhir menegaskan batas persis §8.3.6 pada lima ukuran: 32 767 kunci hanya memerlukan 32 752 pertukaran untuk menjadi *heap* — lebih sedikit pertukaran daripada kunci.

8.3.8 Periksa diri

Jalankan `BUILD-MAX-HEAP` dengan tangan pada $A = [3,9,2,1,4,5]$, sambil menghitung perbandingan dan pertukarannya, sebelum membaca jawabannya.

JAWABAN. $n = 6$, jadi gelungnya berjalan untuk $i = 3,2,1$. $i = 3$: simpul 3 memuat 2, anak satu-satunya $A[6] = 5$; $5 > 2$, tukar: [3,9,5,1,4,2] (1 perbandingan, 1 pertukaran). $i = 2$: simpul 2 memuat 9, anaknya $A[4] = 1$ dan $A[5] = 4$; tak ada yang lebih besar: tanpa perubahan (2 perbandingan). $i = 1$: akar 3 beranak 9 dan 5; 9 lebih besar; tukar: [9,3,5,1,4,2]. Di simpul 2, 3 beranak 1 dan 4; 4 lebih besar; tukar: [9,4,5,1,3,2]. Simpul 5 adalah daun (4 perbandingan, 2 pertukaran). **Hasil** [9,4,5,1,3,2], dengan **7 perbandingan dan 3 pertukaran** seluruhnya — blok kedua keluaran **Kode 8.1**.

8.3.9 Belokan keliru yang sering terjadi pada soal ini

1. **Menghitung gelungnya naik**, `for i = 1 to ⌊n/2⌋`. Akarnya lalu di-*heapify* ketika subpohon-subpohonnya masih berantakan, prasyarat `MAX-HEAPIFY` gagal, dan hasilnya bukan *heap*. Pada [5,13,2,25,7,17,20,8,4,11] gelung naik meninggalkan 25 di bawah 13.
2. **Membandingkan anak kanan dengan $A[i]$ alih-alih dengan $A[largest]$.** Di simpul 3 pada §8.3.4 ini berarti membandingkan 20 dengan 2, bukan dengan 17 — di sini kebetulan hasilnya sama, tetapi bila anak kiri yang lebih besar, perbandingan yang keliru tetap memilih anak kanan, dan sifat *heap* gagal satu aras di bawahnya.
3. **Memulai gelung di n alih-alih di $\lfloor n/2 \rfloor$.** Benar, tetapi separuh panggilannya terbuang untuk daun.
4. **Mencampur indeks berbasis 0 dan berbasis 1.** Dalam larik C++ yang dimulai di indeks 0, anak-anak i adalah $2i + 1$ dan $2i + 2$. **Kode 8.1** membiarkan $A[0]$ tidak terpakai justru agar kodenya cocok dengan pseudokodenya.
5. **Lupa menyusutkan *heap* dalam *heapsort*.** Jika `heap-size` tidak dikurangi, `MAX-HEAPIFY` berikutnya menarik kembali maksimum yang sudah diambil ke akar.

8.4 Soal 8.2 — Partisi Quicksort, Dilacak; Kasus Terbaik dan Terburuk

8.4.1 Soalnya

Ambil larik sembilan elemen

$$A = [7,\ 2,\ 9,\ 4,\ 1,\ 8,\ 3,\ 6,\ 5].$$

1. Lacaklah `PARTITION(A, 1, 9)` versi Lomuto, satu baris untuk setiap nilai indeks gelung j: berikan $A[j]$, keputusannya, nilai i yang baru, dan seluruh larik setelah langkah itu. Katakan di mana pivotnya berakhir.
2. Lacaklah `QUICKSORT(A, 1, 9)` selengkapnya: setiap panggilan, pivotnya, dan perbandingan yang dilakukannya. Berikan jumlah seluruh perbandingan.
3. Masukan berupa n kunci berbeda yang mana yang membuat Quicksort (dengan partisi ini) selambat mungkin? Hitunglah perbandingannya dengan persis.
4. Masukan yang mana yang membuatnya secepat mungkin? Hitunglah perbandingannya dengan persis untuk $n = 2^k - 1$.
5. Periksalah kedua hitungan itu terhadap sebuah pengukuran.

8.4.2 Yang Anda perlukan dari Bab 5

`PARTITION`, **Pseudokode 8.3**, mengambil elemen terakhir $x = A[r]$ sebagai **pivot** dan menata ulang $A[p..r]$ sehingga semua yang $\leq x$ datang lebih dahulu, lalu x, lalu semua yang $>x$. Ia mengembalikan indeks akhir pivot, q.

Pseudokode 8.3 `PARTITION` Lomuto: satu sapuan dari kiri ke kanan yang menumbuhkan daerah nilai $\leq x$.

```
PARTITION(A, p, r)
1  x = A[r]                        // pivotnya
2  i = p - 1                       // A[p..i] memuat nilai ≤ x yang sudah ditemukan
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]     // jatuhkan pivot ke celahnya
8  return i + 1
```

Gelungnya memelihara **invarian empat daerah** dari Bab 5: di awal setiap iterasi, $A[p..i]$ memuat nilai $\leq x$, $A[i+1..j-1]$ memuat nilai $>x$, $A[j..r-1]$ belum diperiksa, dan $A[r] = x$.

```
p              i         j                 r
+--------------+---------+-----------------+---+
|    <= x      |   > x   | belum diperiksa | x |
+--------------+---------+-----------------+---+
```

`QUICKSORT`, **Pseudokode 8.4**, mempartisi lalu mengurutkan kedua sisinya secara terpisah. Pivotnya sudah berada di tempat akhirnya dan tak pernah disentuh lagi.

Pseudokode 8.4 `QUICKSORT`: partisi, lalu rekursi pada kedua sisi pivot.

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q - 1)
4      QUICKSORT(A, q + 1, r)
```

8.4.3 Langkah 1: partisinya, satu langkah per baris

Di sini $p = 1$, $r = 9$, pivotnya $x = A[9] = 5$, dan i dimulai di $p - 1 = 0$, yang berarti "belum ada nilai ≤ 5 yang ditemukan". Gelungnya berjalan untuk $j = 1, \ldots, 8$, dan setiap iterasi melakukan tepat satu perbandingan, di baris 4.

- $j = 1$: $A[1] = 7$. Apakah $7 \leq 5$? Tidak. Tak terjadi apa-apa; 7 masuk daerah "$>x$". $A = [7,2,9,4,1,8,3,6,5]$, $i = 0$.
- $j = 2$: $A[2] = 2 \leq 5$. Ya: i menjadi 1, dan $A[1] = 7$ ditukar dengan $A[2] = 2$. $A = [\mathbf{2}, \mathbf{7}, 9,4,1,8,3,6,5]$. Nilai kecil itu melompati nilai besar.
- $j = 3$: $A[3] = 9 > 5$. Tanpa perubahan; $i = 1$.
- $j = 4$: $A[4] = 4 \leq 5$. $i = 2$; tukar $A[2] = 7$ dengan $A[4] = 4$. $A = [2, \mathbf{4}, 9, \mathbf{7}, 1,8,3,6,5]$.
- $j = 5$: $A[5] = 1 \leq 5$. $i = 3$; tukar $A[3] = 9$ dengan $A[5] = 1$. $A = [2,4, \mathbf{1}, 7, \mathbf{9}, 8,3,6,5]$.
- $j = 6$: $A[6] = 8 > 5$. Tanpa perubahan; $i = 3$.
- $j = 7$: $A[7] = 3 \leq 5$. $i = 4$; tukar $A[4] = 7$ dengan $A[7] = 3$. $A = [2,4,1, \mathbf{3}, 9,8, \mathbf{7}, 6,5]$.
- $j = 8$: $A[8] = 6 > 5$. Tanpa perubahan; $i = 4$.

Gelungnya selesai. Baris 7 menukar $A[i+1] = A[5] = 9$ dengan pivot $A[9] = 5$:

$$A = [2,\ 4,\ 1,\ 3,\ \mathbf{5},\ 8,\ 7,\ 6,\ 9], \qquad q = 5.$$

Tabel 8.4 mencatat langkah-langkah yang sama dalam satu tabel.

Tabel 8.4 `PARTITION` Lomuto pada [7,2,9,4,1,8,3,6,5] dengan pivot $x = 5$. Delapan perbandingan, empat pertukaran di dalam gelung dan satu di akhir; pivotnya berakhir di indeks $q = 5$.

j	$A[j]$	$A[j]\leq 5$?	i Sesudahnya	Pertukaran	A Setelah langkah itu
1	7	tidak	0	—	[7,2,9,4,1,8,3,6,5]
2	2	ya	1	$A[1] \leftrightarrow A[2]$	[2,7,9,4,1,8,3,6,5]
3	9	tidak	1	—	[2,7,9,4,1,8,3,6,5]
4	4	ya	2	$A[2] \leftrightarrow A[4]$	[2,4,9,7,1,8,3,6,5]
5	1	ya	3	$A[3] \leftrightarrow A[5]$	[2,4,1,7,9,8,3,6,5]
6	8	tidak	3	—	[2,4,1,7,9,8,3,6,5]
7	3	ya	4	$A[4] \leftrightarrow A[7]$	[2,4,1,3,9,8,7,6,5]
8	6	tidak	4	—	[2,4,1,3,9,8,7,6,5]
baris 7				$A[5] \leftrightarrow A[9]$	[2,4,1,3,5,8,7,6,9]

Periksa invariannya di akhir. $A[1..4] = \{2,4,1,3\}$, semuanya ≤ 5 ✓. $A[5] = 5$, sang pivot ✓. $A[6..9] = \{8,7,6,9\}$, semuanya >5 ✓. Pivotnya berada di tempat yang akan didudukinya dalam larik terurut — ada tepat empat nilai yang lebih kecil daripada 5. Partisinya melakukan tepat $r - p = 8$ perbandingan, satu per iterasi gelung; hitungan itu sama sekali tidak bergantung pada datanya.

8.4.4 Langkah 2: seluruh rekursinya

Setelah partisi pertama, `QUICKSORT` mengurutkan $A[1..4] = [2,4,1,3]$ dan $A[6..9] = [8,7,6,9]$. Setiap panggilan mempartisi rentangnya sendiri dengan cara yang sama.

- `QUICKSORT(A, 1, 4)` pada [2,4,1,3], pivot 3. $j = 1$: $2 \leq 3$, $i = 1$, $A[1]$ ditukar dengan dirinya sendiri. $j = 2$: $4 > 3$. $j = 3$: $1 \leq 3$, $i = 2$, tukar $A[2] = 4$ dengan $A[3] = 1$: [2,1,4,3]. Baris 7: tukar $A[3] = 4$ dengan $A[4] = 3$: [2,1,3,4], $q = 3$. Tiga perbandingan.
- `QUICKSORT(A, 1, 2)` pada [2,1], pivot 1. $j = 1$: $2 > 1$. Baris 7: tukar $A[1]$ dengan $A[2]$: [1,2], $q = 1$. Satu perbandingan. Kedua sisinya kini paling banyak berisi satu elemen.
- `QUICKSORT(A, 6, 9)` pada [8,7,6,9], pivot 9, yang terbesar. Setiap $A[j] \leq 9$, jadi i naik bersama setiap j dan setiap pertukaran adalah pertukaran elemen dengan dirinya sendiri; baris 7 menukar $A[9]$ dengan dirinya sendiri. $q = 9$: pembagiannya tiga elemen lawan nol. Tiga perbandingan.
- `QUICKSORT(A, 6, 8)` pada [8,7,6], pivot 6, yang terkecil. $8 > 6$, $7 > 6$; baris 7 menukar $A[6] = 8$ dengan $A[8] = 6$: [6,7,8], $q = 6$. Dua perbandingan.
- `QUICKSORT(A, 7, 8)` pada [7,8], pivot 8. $7 \leq 8$; $q = 8$. Satu perbandingan.

Setiap panggilan lainnya memiliki $p \geq r$ dan langsung kembali tanpa membandingkan apa pun. Rekursinya, digambar sebagai pohon rentang beserta pivot setiap panggilan, adalah:

```
                    [1..9] pivot 5
                  /                  \
        [1..4] pivot 3            [6..9] pivot 9
         /          \               /          \
[1..2] pivot 1   [4..4]    [6..8] pivot 6   (kosong)
                                /       \
                          (kosong)   [7..8] pivot 8
```

Jumlah perbandingan: $8 + 3 + 1 + 3 + 2 + 1 = 18$. Cabang kanan memperlihatkan apa yang salah pada kasus terburuk: pivot 9 lalu 6 adalah yang terbesar lalu yang terkecil di rentangnya, sehingga setiap partisi hanya mengupas satu elemen.

8.4.5 Langkah 3: kasus terburuk

Andaikan pivotnya selalu nilai **terbesar** di rentangnya. Maka setiap partisi atas m elemen melakukan $m - 1$ perbandingan dan meninggalkan satu sisi dengan $m - 1$ elemen dan sisi lainnya kosong. Kunci-kunci yang sudah **berurutan naik** melakukan persis hal ini dengan pivot elemen-terakhir: elemen terakhir setiap rentang adalah maksimumnya. Dengan menuliskan $C(n)$ untuk banyaknya perbandingan,

$$C(1) = 0, \qquad C(n) = C(n-1) + (n-1).$$

Uraikan: $C(n) = (n-1) + (n-2) + \cdots + 1 + 0 = \frac{n(n-1)}{2}$, jumlah segitiga dari Bab 3. Dalam bentuk asimtotik, dengan panggilan kosong berbiaya tetap yang diserap,

$$T(n) = T(n-1) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n^2). \tag{8.3}$$

Untuk $n = 1023$ ini adalah $\frac{1023 \times 1022}{2} = 522\,753$ perbandingan. Kunci-kunci yang berurutan *turun* sama buruknya, sebab pivotnya lalu menjadi yang terkecil di setiap rentang, sehingga sisi kosongnya berada di kiri. Persamaan (8.3) adalah perilaku kuadratik yang membuat Quicksort naif berbahaya pada data yang datang sudah terurut — sesuatu yang sangat sering terjadi pada data sungguhan.

8.4.6 Langkah 4: kasus terbaik

Yang terbaik yang dapat dilakukan sebuah partisi adalah menaruh pivotnya tepat di tengah, meninggalkan $\frac{m-1}{2}$ elemen di setiap sisi. Ambil $n = 2^k - 1$, sehingga setiap rentang berukuran ganjil dan terbelah tepat dua. Maka

$$C(1) = 0, \qquad C(n) = 2\,C\left(\frac{n-1}{2}\right) + (n-1).$$

Dalam bentuk asimtotik ini adalah rekurensi *merge sort*,

$$T(n) = 2T(n/2) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n\lg n), \tag{8.4}$$

menurut kasus 2 Teorema Master: $a = b = 2$, $n^{\log_b a} = n$, dan $f(n) = \Theta(n)$.

Hitungan persisnya. Hitung nilai-nilai pertamanya: $C(1) = 0$; $C(3) = 2 \cdot 0 + 2 = 2$; $C(7) = 2 \cdot 2 + 6 = 10$; $C(15) = 2 \cdot 10 + 14 = 34$. Tebak $C(2^k - 1) = (k-2)\,2^k + 2$: untuk $k = 1,2,3,4$ ini memberi 0,2,10,34 ✓. Buktikan dengan induksi pada k. Kasus dasar $k = 1$ adalah $(1-2) \cdot 2 + 2 = 0$ ✓. Untuk langkahnya, andaikan rumus itu berlaku untuk k; maka

$$\begin{aligned} C(2^{k+1} - 1) = 2[(k-2)2^k + 2] + (2^{k+1} - 2) &= (k-2)2^{k+1} + 2^{k+1} + 2 \\ &= ((k+1) - 2)2^{k+1} + 2, \end{aligned}$$

yang merupakan rumus untuk $k + 1$. ✓ Dengan menyubstitusikan $2^k = n + 1$ dan $k = \lg(n+1)$,

$$C(n) = (n+1)\lg(n+1) - 2n. \tag{8.5}$$

Untuk $n = 1023$: $1024 \times 10 - 2046 = 8\,194$ perbandingan, dibandingkan $522\,753$ pada kasus terburuk — enam puluh empat kali lebih sedikit.

Masukan mana yang mencapainya? Pivotnya harus menjadi median setiap rentang yang ditemuinya. **Kode 8.2** membangun masukan semacam itu secara rekursif: taruh mediannya di ujung; sebelumnya, susunan untuk separuh yang lebih kecil, lalu susunan untuk separuh yang lebih besar. Satu kehalusan dijelaskan dalam komentar kodenya — pertukaran terakhir Lomuto memindahkan elemen pertama daerah "$>x$" ke ujung, sehingga separuh yang lebih besar disimpan terputar satu tempat, dan partisinya memutarnya kembali.

Susunan **acak** terletak di antara keduanya, dan Bab 5 menunjukkan bahwa hitungan harapannya sekitar $2n\ln n \approx 1.39\,n\lg n$: jauh lebih dekat ke kasus terbaik daripada ke kasus terburuk. Itulah sebabnya Bab 5 menganjurkan memilih pivot secara acak — cara itu membuat biaya harapannya $\Theta(n\lg n)$ pada *setiap* masukan, termasuk yang sudah terurut.

8.4.7 Programnya

Kode 8.2 mengimplementasikan **Pseudokode 8.3** dan **Pseudokode 8.4** dengan pencacah perbandingan, mencetak pelacakan partisi §8.4.3 dan rekursi §8.4.4, mempartisi larik periksa-diri §8.4.8, dan mengukur perbandingan pada masukan terurut, masukan kasus terbaik, dan masukan acak berukuran $2^k - 1$.

Kode 8.2 Soal 8.2 selengkapnya: partisi Lomuto yang dilacak, rekursi lengkap dengan hitungan perbandingan per panggilan, pembangkit masukan kasus terbaik, dan pengukuran kasus terbaik, terburuk, dan acak.

```
// quicktrace.cpp -- Soal 8.2: PARTITION Lomuto dilacak langkah demi langkah, seluruh
// rekursi QUICKSORT, dan banyaknya perbandingan pada kasus terbaik dan terburuk.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <random>
#include <cmath>
#include <utility>
#include <algorithm>
using namespace std;

vector<int> A;                  // berbasis 1: A[0] tidak dipakai
long long comparisons = 0;
int maxDepth = 0;

string window(int p, int r) {  // A[p..r] sebagai teks
    string s = "[";
    for (int k = p; k <= r; ++k) s += (k > p ? " " : "") + to_string(A[k]);
    return s + "]";
}

// PARTITION Lomuto, persis seperti pseudokodenya. Bila trace menyala, satu baris per j.
int partition(int p, int r, bool trace) {
    int x = A[r];                                   // pivotnya adalah elemen terakhir
    int i = p - 1;                                  // ujung daerah "<= x"
    for (int j = p; j <= r - 1; ++j) {
        ++comparisons;
        bool small = A[j] <= x;
        if (small) { ++i; swap(A[i], A[j]); }
        if (trace) {
            string action = small ? "<= x: i=" + to_string(i) + ", swap A[i],A[j]"
                                  : ">  x: no change";
            cout << "    j=" << j << "  A[j]=" << setw(2) << (small ? A[i] : A[j]) << "  "
                 << left << setw(28) << action << right << window(p, r) << "\n";
        }
    }
    swap(A[i + 1], A[r]);                           // jatuhkan pivot ke celahnya
    if (trace) {
        string action = "final swap A[" + to_string(i + 1) + "] <-> A[" + to_string(r) +
"]";
        cout << "    " << left << setw(42) << action << right << window(p, r)
             << "   q = " << i + 1 << "\n";
    }
    return i + 1;
}
```

Kode 8.2 (Lanjutan) *Soal 8.2 selengkapnya: partisi Lomuto yang dilacak, rekursi lengkap dengan hitungan perbandingan per panggilan, pembangkit masukan kasus terbaik, dan pengukuran kasus terbaik, terburuk, dan acak.*

```
void quicksort(int p, int r, int depth, bool trace) {
    if (p >= r) return;                             // 0 atau 1 elemen: sudah terurut
    maxDepth = max(maxDepth, depth);
    int before = (int)comparisons;
    int q = partition(p, r, false);
    if (trace)
        cout << string(2 * depth, ' ') << "QUICKSORT(A," << p << "," << r << ")  pivot "
             << A[q] << " lands at q=" << q << "  (" << (int)comparisons - before
             << " comparison" << ((int)comparisons - before == 1 ? ")  " : "s)  ") <<
window(p, r) << "\n";
    quicksort(p, q - 1, depth + 1, trace);
    quicksort(q + 1, r, depth + 1, trace);
}

// Susun lo..hi agar pivot elemen-terakhir Lomuto selalu menjadi median rentangnya.
// Bila nilai-nilai "<= x" (L) datang lebih dulu dan nilai-nilai "> x" (R) sesudahnya,
// PARTITION membiarkan L di tempat dan pertukaran terakhirnya memindahkan nilai pertama R
// ke ujung. Maka R disimpan terputar satu tempat ke kanan, dan partisinya memutar balik.
vector<int> bestCase(int lo, int hi) {
    if (lo > hi) return {};
    int mid = lo + (hi - lo) / 2;
    vector<int> L = bestCase(lo, mid - 1), R = bestCase(mid + 1, hi), out = L;
    if (!R.empty()) {
        out.push_back(R.back());                    // kembali menjadi nilai terakhir R
        out.insert(out.end(), R.begin(), R.end() - 1);
    }
    out.push_back(mid);                             // pivotnya, di posisi terakhir
    return out;
}

long long sortCount(const vector<int>& v) {
    A = v; A.insert(A.begin(), 0);
    comparisons = 0; maxDepth = 0;
    quicksort(1, (int)v.size(), 0, false);
    return comparisons;
}

int main() {
    cout << "PARTITION(A, 1, 9) on [7 2 9 4 1 8 3 6 5], pivot x = A[9] = 5\n";
    A = {0, 7, 2, 9, 4, 1, 8, 3, 6, 5};
    comparisons = 0;
    partition(1, 9, true);
```

Kode 8.2 (Lanjutan) *Soal 8.2 selengkapnya: partisi Lomuto yang dilacak, rekursi lengkap dengan hitungan perbandingan per panggilan, pembangkit masukan kasus terbaik, dan pengukuran kasus terbaik, terburuk, dan acak.*

```
    cout << "\nThe whole QUICKSORT on the same array\n";
    A = {0, 7, 2, 9, 4, 1, 8, 3, 6, 5};
    comparisons = 0; maxDepth = 0;
    quicksort(1, 9, 0, true);
    cout << "sorted: " << window(1, 9) << "   total comparisons = " << comparisons
         << ", deepest level = " << maxDepth << "\n";

    cout << "\nCheck yourself: PARTITION on [3 8 1 6 2 7 5 4], pivot 4\n";
    A = {0, 3, 8, 1, 6, 2, 7, 5, 4};
    partition(1, 8, true);

    cout << "\n     n   sorted   n(n-1)/2   best case   (n+1)lg(n+1)-2n   random\n";
    mt19937 rng(20260923);
    for (int n : {7, 15, 31, 63, 127, 255, 511, 1023}) {
        vector<int> sorted(n);
        for (int k = 0; k < n; ++k) sorted[k] = k + 1;
        long long w = sortCount(sorted);
        long long b = sortCount(bestCase(1, n));
        vector<int> rnd = sorted; shuffle(rnd.begin(), rnd.end(), rng);
        long long a = sortCount(rnd);
        long long k = llround(log2(n + 1.0));      // n = 2^k - 1
        cout << setw(6) << n << setw(9) << w << setw(11) << (long long)n * (n - 1) / 2
             << setw(12) << b << setw(18) << (n + 1) * k - 2LL * n << setw(9) << a << "\n";
    }
    return 0;
}
```

Ia mencetak:

```
PARTITION(A, 1, 9) on [7 2 9 4 1 8 3 6 5], pivot x = A[9] = 5
    j=1  A[j]= 7  >  x: no change             [7 2 9 4 1 8 3 6 5]
    j=2  A[j]= 2  <= x: i=1, swap A[i],A[j]   [2 7 9 4 1 8 3 6 5]
    j=3  A[j]= 9  >  x: no change             [2 7 9 4 1 8 3 6 5]
    j=4  A[j]= 4  <= x: i=2, swap A[i],A[j]   [2 4 9 7 1 8 3 6 5]
    j=5  A[j]= 1  <= x: i=3, swap A[i],A[j]   [2 4 1 7 9 8 3 6 5]
    j=6  A[j]= 8  >  x: no change             [2 4 1 7 9 8 3 6 5]
    j=7  A[j]= 3  <= x: i=4, swap A[i],A[j]   [2 4 1 3 9 8 7 6 5]
    j=8  A[j]= 6  >  x: no change             [2 4 1 3 9 8 7 6 5]
    final swap A[5] <-> A[9]                  [2 4 1 3 5 8 7 6 9]   q = 5

The whole QUICKSORT on the same array
QUICKSORT(A,1,9)  pivot 5 lands at q=5  (8 comparisons)  [2 4 1 3 5 8 7 6 9]
  QUICKSORT(A,1,4)  pivot 3 lands at q=3  (3 comparisons)  [2 1 3 4]
    QUICKSORT(A,1,2)  pivot 1 lands at q=1  (1 comparison)  [1 2]
  QUICKSORT(A,6,9)  pivot 9 lands at q=9  (3 comparisons)  [8 7 6 9]
    QUICKSORT(A,6,8)  pivot 6 lands at q=6  (2 comparisons)  [6 7 8]
      QUICKSORT(A,7,8)  pivot 8 lands at q=8  (1 comparison)  [7 8]
sorted: [1 2 3 4 5 6 7 8 9]   total comparisons = 18, deepest level = 3

Check yourself: PARTITION on [3 8 1 6 2 7 5 4], pivot 4
    j=1  A[j]= 3  <= x: i=1, swap A[i],A[j]   [3 8 1 6 2 7 5 4]
    j=2  A[j]= 8  >  x: no change             [3 8 1 6 2 7 5 4]
    j=3  A[j]= 1  <= x: i=2, swap A[i],A[j]   [3 1 8 6 2 7 5 4]
    j=4  A[j]= 6  >  x: no change             [3 1 8 6 2 7 5 4]
    j=5  A[j]= 2  <= x: i=3, swap A[i],A[j]   [3 1 2 6 8 7 5 4]
    j=6  A[j]= 7  >  x: no change             [3 1 2 6 8 7 5 4]
    j=7  A[j]= 5  >  x: no change             [3 1 2 6 8 7 5 4]
    final swap A[4] <-> A[8]                  [3 1 2 4 8 7 5 6]   q = 4

     n   sorted   n(n-1)/2   best case   (n+1)lg(n+1)-2n   random
     7       21         21          10                10       10
    15      105        105          34                34       45
    31      465        465          98                98      132
    63     1953       1953         258               258      333
   127     8001       8001         642               642      839
   255    32385      32385        1538              1538     2435
   511   130305     130305        3586              3586     4892
  1023   522753     522753        8194              8194    10637
```

Blok pertama adalah **Tabel 8.4**, baris demi baris. Blok kedua adalah pohon rekursi §8.4.4, dengan 18 perbandingannya. Tabel terakhir adalah analisisnya yang dibuat kasatmata: kolom terurut sama dengan $\frac{n(n-1)}{2}$ pada setiap ukuran, sebagaimana diramalkan Persamaan (8.3), dan kolom kasus terbaik sama dengan Persamaan (8.5) pada setiap ukuran, sebagaimana diturunkan di §8.4.6. Kolom acak duduk sedikit di atas kasus terbaik, tumbuh seperti $n\lg n$, sementara kolom terurut meledak: pada $n = 1023$ masukan terurut berbiaya lima puluh kali lipat masukan yang teracak.

8.4.8 Periksa diri

Partisilah $A = [3,8,1,6,2,7,5,4]$ dengan `PARTITION(A, 1, 8)`. Berikan q, larik yang ditinggalkannya, dan banyaknya perbandingan.

> **JAWABAN.** Pivotnya $x = 4$ dan i dimulai di 0. $j = 1$: $3 \leq 4$, $i = 1$, $A[1]$ ditukar dengan dirinya sendiri. $j = 2$: $8 > 4$. $j = 3$: $1 \leq 4$, $i = 2$, tukar $A[2] = 8$ dengan $A[3] = 1$: $[3,1,8,6,2,7,5,4]$. $j = 4$: $6 > 4$. $j = 5$: $2 \leq 4$, $i = 3$, tukar $A[3] = 8$ dengan $A[5] = 2$: $[3,1,2,6,8,7,5,4]$. $j = 6,7$: $7 > 4$, $5 > 4$. Baris 7: tukar $A[4] = 6$ dengan $A[8] = 4$. **Hasil** $[3,1,2,4,8,7,5,6]$ dengan $q = 4$, setelah 7 perbandingan — blok ketiga keluaran **Kode 8.2**. Tiga nilai lebih kecil daripada 4, jadi 4 berada di indeks 4.

8.4.9 Belokan keliru yang sering terjadi pada soal ini

1. **Memulai i di p alih-alih di $p-1$.** Daerah "$\leq x$" lalu mengklaim sebuah elemen yang belum pernah diperiksanya, dan nilai kecil pertama menimpanya.
2. **Menjalankan j sampai r alih-alih sampai $r-1$.** Pivotnya dibandingkan dengan dirinya sendiri, masuk daerah "$\leq x$", dan baris 7 lalu menukarnya dengan elemen yang keliru.
3. **Berekursi pada $A[p..q]$ alih-alih pada $A[p..q-1]$.** Bila pivotnya nilai terbesar, $q = r$, dan panggilan `QUICKSORT(A, p, q)` adalah panggilan yang sama lagi: rekursi tanpa akhir.
4. **Mengatakan bahwa kasus terburuk jarang terjadi.** Untuk pivot elemen-terakhir, kasus terburuknya adalah *masukan terurut*, yang lazim. Pernyataan yang jujur adalah: Quicksort deterministik berbiaya $\Theta(n^2)$ pada kasus terburuk, dan mengacak pivotnya menjadikan $\Theta(n\lg n)$ sebagai biaya *harapan* pada setiap masukan.
5. **Mencampur nilai kembalian Lomuto dan Hoare.** Lomuto mengembalikan indeks akhir pivot, dan rekursinya mengecualikannya. Skema Hoare, yang juga ada di Bab 5, mengembalikan titik belah j dengan pivot yang belum tentu berada di j, dan rekursinya dilakukan pada $A[p..j]$ dan $A[j+1..r]$.

8.5 Tipe Data yang Dipakai Soal-Soal Pohon

Soal 8.3 sampai 8.10 semuanya menerima sebuah `MyTree`. Sebelum kita dapat menyelesaikan satu pun darinya kita harus persis tentang apa itu `MyTree`, sebab setiap satu dari kedelapan penyelesaian itu adalah rekursi atas strukturnya, dan rekursi atas struktur yang belum Anda definisikan hanyalah tebak-tebakan.

Kita nyatakan tipe itu dengan persis, lalu kita bangun. Sebuah `MyTree` adalah salah satu dari tepat dua hal:

- **pohon kosong**, yang tidak memuat apa pun; atau
- sebuah **simpul**, yang memuat satu bilangan bulat dan dua `MyTree`, yang disebut subpohon kiri dan subpohon kanan.

Itulah seluruh definisinya, dan bentuknya adalah bentuk setiap jawaban: *jika pohonnya kosong, kerjakan hal yang mudah; jika tidak, kerjakan sesuatu dengan bilangan bulatnya lalu berekursilah pada kedua upapohonnya.*

> **DEFINISI (tipe `MyTree`).** Sebuah `MyTree` adalah kosong, atau sebuah tripel (v, L, R) dengan v sebuah bilangan bulat dan L serta R adalah `MyTree`. Kita menuliskan pohon kosong sebagai $\varnothing$. **Tinggi** sebuah `MyTree` adalah

$$h(\varnothing) = -1, \qquad h((v, L, R)) = 1 + \max(h(L), h(R)). \tag{8.6}$$

> **Faktor keseimbangan** sebuah pohon tak-kosong adalah

$$\mathrm{bf}((v, L, R)) = h(R) - h(L), \tag{8.7}$$

> persis kesepakatan Bab 6: dalam Persamaan (8.7), negatif berarti berat ke kiri, positif berarti berat ke kanan, nol berarti rata.

Mengapa tinggi pohon kosong -1 dan bukan 0? Sebab dengan begitu rumus Persamaan (8.6) berlaku untuk daun tanpa kasus khusus: daun punya dua subpohon kosong, jadi $h = 1 + \max(-1, -1) = 1 + (-1) = 0$. Daun bertinggi 0; tak ada sisi di bawahnya. Sebagian penyajian — termasuk beberapa gambar yang direproduksi di Bab 6 dan di bab ini — menghitung *simpul pada lintasan terpanjang*, yang membuat daun bertinggi 1 dan pohon kosong bertinggi 0 — satu lebih

besar daripada milik kita di mana-mana. Kedua kesepakatan itu benar dan keduanya muncul dalam kepustakaan; yang penting adalah faktor keseimbangannya, yang merupakan *selisih* tinggi, keluar sama persis. Kita memakai kesepakatan penghitungan sisi di sepanjang buku ini, seperti Bab 6.

Kode 8.3 adalah tipenya, operasinya, dan sebuah pencetak, sebagai program yang dapat Anda jalankan.

Kode 8.3 Tipe `MyTree` dalam C++, dengan satu-satunya konstruktor `node()` yang menjaga tinggi tersimpannya tetap jujur, dan sebuah pencetak bentuk yang menuliskan seluruh pohon dalam satu baris.

```
// mytree.cpp -- tipe data MyTree yang menjadi dasar setiap soal pohon dalam bab ini.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

struct MyTreeNode;
using MyTree = MyTreeNode*;              // penunjuk null ITULAH pohon kosong

struct MyTreeNode {
    int    value;                        // bilangan bulat yang disimpan di simpul ini
    MyTree left, right;                  // kedua subpohon
    int    h;                            // tinggi tersimpan; pohon kosong bertinggi -1
};

MyTree emptyTree()          { return nullptr; }
bool   isEmpty(MyTree t)    { return t == nullptr; }
int    height(MyTree t)     { return isEmpty(t) ? -1 : t->h; }
int    rootValue(MyTree t)  { return t->value; }
MyTree leftOf(MyTree t)     { return t->left; }
MyTree rightOf(MyTree t)    { return t->right; }

// node(v,l,r): satu-satunya konstruktor. Ia menghitung ulang tingginya, sehingga
// pohon yang dirakit hanya lewat node() tak pernah menyimpan tinggi yang basi.
MyTree node(int v, MyTree l, MyTree r) {
    MyTree t = new MyTreeNode;
    t->value = v; t->left = l; t->right = r;
    t->h = 1 + max(height(l), height(r));
    return t;
}
MyTree leaf(int v) { return node(v, emptyTree(), emptyTree()); }

int balanceFactor(MyTree t) { return height(t->right) - height(t->left); }

// shape(t) mencetak nilai(faktorKeseimbangan){kiri,kanan}; '.' adalah pohon kosong.
string shape(MyTree t) {
    if (isEmpty(t)) return ".";
    string s = to_string(t->value) + "(" + to_string(balanceFactor(t)) + ")";
    if (isEmpty(t->left) && isEmpty(t->right)) return s;      // daun tak perlu kurung
kurawal
    return s + "{" + shape(t->left) + "," + shape(t->right) + "}";
}

int countNodes(MyTree t) {
    return isEmpty(t) ? 0 : 1 + countNodes(t->left) + countNodes(t->right);
}
```

Kode 8.3 (Lanjutan) *Tipe MyTree dalam C++, dengan satu-satunya konstruktor node() yang menjaga tinggi tersimpannya tetap jujur, dan sebuah pencetak bentuk yang menuliskan seluruh pohon dalam satu baris.*

```
void inorder(MyTree t) {
    if (isEmpty(t)) return;
    inorder(t->left);
    cout << t->value << ' ';
    inorder(t->right);
}

int main() {
    // Pohon contoh delapan simpul dari Bab 6, dirakit dengan tangan.
    MyTree t = node(13,
                    node(10, node(5, leaf(4), leaf(6)), leaf(11)),
                    node(15, emptyTree(), leaf(16)));

    cout << "shape   : " << shape(t) << "\n";
    cout << "nodes   : " << countNodes(t) << "\n";
    cout << "height  : " << height(t) << "\n";
    cout << "inorder : "; inorder(t); cout << "\n";
    cout << "empty   : height " << height(emptyTree())
         << ", isEmpty = " << (isEmpty(emptyTree()) ? "true" : "false") << "\n";
    return 0;
}
```

Dikompilasi dan dijalankan, ia mencetak:

```
shape   : 13(-1){10(-1){5(0){4(0),6(0)},11(0)},15(1){.,16(0)}}
nodes   : 8
height  : 3
inorder : 4 5 6 10 11 13 15 16
empty   : height -1, isEmpty = true
```

Belajarlah membaca notasi bentuk satu baris itu, sebab setiap hasil dalam bab ini dilaporkan dengannya. `13(-1){…,…}` berarti *simpul* 13*, yang faktor keseimbangannya* -1*, dengan subpohon kiri sebelum koma dan subpohon kanan sesudahnya*. Daun ditulis tanpa kurung kurawal: `4(0)`. Pohon kosong adalah sebuah titik. Jadi untai di atas berbunyi: akar 13, anak kiri 10 dengan anak-anak 5 dan 11, simpul 5 dengan anak-anak 4 dan 6, anak kanan 15 tanpa anak kiri dan dengan anak kanan 16. Itulah pohon delapan simpul tempat Bab 6 mengerjakan contoh-contoh penyisipannya, dan itulah pohon yang akan terus kita datangi kembali.

> **WAWASAN KUNCI: SATU KONSTRUKTOR, TAK ADA TINGGI YANG BASI.** Perhatikan bahwa `node()` adalah *satu-satunya* cara membangun pohon, dan bahwa ia menghitung ulang tingginya setiap kali. Bab 6 §6.16.1 memperingatkan bahwa kutu tunggal yang paling sering terjadi pada pohon AVL tulisan tangan adalah menyegarkan tingginya dalam urutan yang keliru setelah sebuah rotasi. Dalam gaya **Kode 8.3** kutu itu **mustahil**, sebab rotasi dinyatakan sebagai `node(...)` yang baru, dibangun dari subpohon yang sudah ada, dan `node` selalu menghitung tinggi induk setelah anak-anaknya ada. Inilah alasan yang lebih dalam mengapa `insertHB` dan `rebalanceForLeft` pada Soal 8.7 dan 8.8 *mengembalikan* sebuah `MyTree` alih-alih mengubahnya di tempat. Tanda tangan itu memang sengaja dipilih demikian.

8.6 Soal 8.3 — `isBST`: Apakah Pohon Ini Pohon Pencarian Biner?

8.6.1 Soalnya

Tulislah fungsi `bool isBST(MyTree t)` yang memutuskan apakah pohon biner `t` merupakan pohon pencarian biner (BST, *binary search tree*). Fungsinya harus rekursif, dan ia boleh

menyerahkan pekerjaannya kepada fungsi penunjang rekursif yang, selain pohonnya, menerima sebuah batas bawah dan sebuah batas atas:

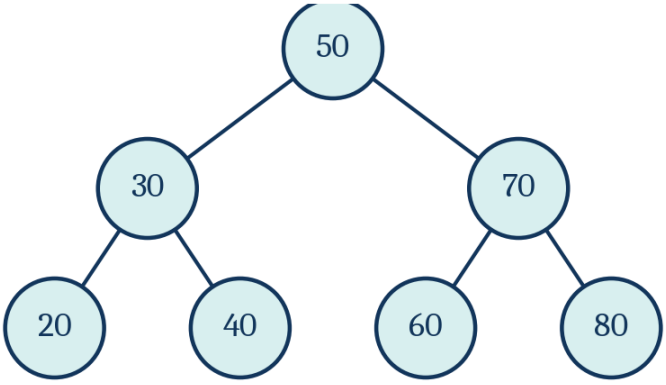
`isBST(t, lowerBound, upperBound)` memutuskan apakah `t` sebuah BST *yang semua nilainya terletak ketat di antara* `lowerBound` *dan* `upperBound`.

Kedua parameter tambahan itu adalah petunjuk yang sangat besar, dan §8.6.4 memperlihatkan mengapa.

8.6.2 Tentang apa soal ini sebenarnya

Tiga hal, menurut urutan kepentingannya.

Pertama: tahukah Anda apa yang sebenarnya dikatakan sifat BST? **Gambar 8.1** menyatakannya: cabang-cabang kiri memuat simpul-simpul yang nilainya kurang dari nilai simpul itu, dan cabang-cabang kanan memuat simpul-simpul yang nilainya lebih besar. Bentuk jamak *cabang-cabang* itulah seluruh persoalannya. Yang dimaksud bukan *anak kiri*; yang dimaksud adalah *setiap simpul di mana pun di dalam subpohon kiri*.



Gambar 8.1 Sebuah pohon pencarian biner yang menggambarkan definisinya: "Left branch(es) contains the nodes whose values $<$ the value of the node" — cabang-cabang, jamak, bukan anak.

Kedua: dapatkah Anda mengubah sifat global menjadi sifat lokal? Sifat itu sebagaimana dinyatakan mengkuantifikasi seluruh subpohon, sedangkan rekursi hanya dapat melihat satu simpul pada satu saat. Seninya adalah menemukan sepotong keterangan yang, jika dibawa turun ke tiap simpul, membuat sifat globalnya dapat diperiksa secara lokal. Potongan keterangan itu adalah sepasang batas, dan tanda tangan fungsi penunjangnya menyerahkannya kepada Anda.

Ketiga: dapatkah Anda memilih nilai penjaga untuk panggilan awalnya? Yang ini kurang penting dibandingkan kedua yang lain, tetapi di situlah penyelesaian yang tampak rampung dan paling menggoda tergelincir, dan §8.6.7 mengerjakannya.

8.6.3 Jawaban yang keliru, dan pohon terkecil yang menyingkapnya

Naluri pertama hampir semua orang adalah pemeriksaan *lokal*: di tiap simpul, bandingkan ia dengan kedua anaknya.

```
bool isBSTLocal(MyTree t) {
    if (isEmpty(t)) return true;
    if (!isEmpty(t->left)  && t->left->value  >= t->value) return false;
    if (!isEmpty(t->right) && t->right->value <= t->value) return false;
    return isBSTLocal(t->left) && isBSTLocal(t->right);
}
```

Ia pendek, ia cepat, ia berjalan dalam $\Theta(n)$, dan ia keliru. Inilah pohon terkecil yang membuktikannya. Ambillah

$$20 \text{ di akar}, \quad 10 \text{ di kiri dengan anak-anak 5 dan 25}, \quad 30 \text{ di kanan}.$$

Periksalah tiap pasangan induk–anak dengan tangan:

- Di 20: anak kirinya 10, dan $10 < 20$. ✓ Anak kanannya 30, dan $30 > 20$. ✓
- Di 10: anak kirinya 5, dan $5 < 10$. ✓ Anak kanannya 25, dan $25 > 10$. ✓
- Di 5: sebuah daun. Tak ada yang perlu diperiksa. ✓
- Di 25: sebuah daun. ✓
- Di 30: sebuah daun. ✓

Setiap uji lokal lulus. Tetapi pohon itu **bukan** BST, sebab 25 duduk di subpohon kiri milik 20 padahal $25 > 20$. Penelusuran inordernya adalah 5,10,25,20,30 — tidak menaik.

> **PERINGATAN: INILAH JEBAKANNYA.** Pemeriksaan lokal induk–anak adalah syarat *perlu* bagi sebuah BST dan bukan syarat *cukup*. Inilah penyelesaian pertama yang ditulis hampir semua orang, dan ia lulus pada setiap uji yang dibangun dari pohon kecil yang rapi. Ingatlah contoh sanggahan empat simpul di atas, bersama kalimat "pemeriksaan lokal itu perlu tetapi tidak cukup": keduanya bersama-sama adalah seluruh isi soal ini.

Empat simpul adalah minimumnya. Bisakah tiga simpul melakukannya? Dengan tiga simpul Anda punya sebuah akar dan dua simpul lain. Jika keduanya menggantung di akar sebagai anak, pemeriksaan lokal *memang* seluruh sifatnya. Jika keduanya membentuk rantai akar–anak–cucu, pemeriksaan lokal melewatkan perbandingan akar lawan cucu — misalnya akar 20, anak kiri 10, dan anak kanan 10 yaitu 25: pemeriksaan lokal lulus di 20 ($10 < 20$) dan di 10 ($25 > 10$), padahal $25 > 20$ duduk di kiri. Jadi **tiga simpul mencukupi** — akar 20, anak kiri 10, anak kanan 10 sama dengan 25 — dan contoh empat simpul kita sekadar lebih mudah diingat sebab ia juga punya daun di tiap sisi. Soal latihan 2 meminta Anda memastikan bahwa dua simpul tidak pernah dapat menyingkap kutunya.

8.6.4 Menurunkan jawaban yang benar

Kita menginginkan pernyataan tentang satu simpul yang, jika diterapkan di tiap simpul, setara dengan sifat BST global. Inilah penurunannya, satu langkah satu baris.

Langkah 1 — tuliskan sifat yang kita ingin berlaku pada seluruh pohon. Untuk pohon t berakar nilai v dengan subpohon L dan R:

$$\mathrm{BST}(t) \Leftrightarrow (\forall u \in L: u < v) \wedge (\forall u \in R: u > v) \wedge \mathrm{BST}(L) \wedge \mathrm{BST}(R). \tag{8.8}$$

Langkah 2 — sadari bahwa kedua kuantor universal pada Persamaan (8.8) **itu adalah kendala rentang.** "Setiap nilai di L kurang dari v" berkata: setiap nilai di L terletak dalam rentang $(-\infty, v)$. "Setiap nilai di R lebih besar daripada v" berkata: setiap nilai di R terletak dalam $(v, +\infty)$.

Langkah 3 — perumum klaimnya agar ia dapat dirangkai. Alih-alih bertanya "apakah t sebuah BST?", tanyakanlah pertanyaan yang lebih kuat:

$P(t, \ell, u)$**:** *t adalah BST dan setiap nilai di t terletak secara tegas di antara ℓ dan u.*

Langkah 4 — bentangkan P dengan memakai langkah 1. Misalkan $t = (v, L, R)$. Maka $P(t, \ell, u)$ berlaku tepat ketika

1. $\ell < v < u$ — akarnya sendiri berada di dalam rentangnya; dan

2. $P(L, \ell, v)$ — subpohon kirinya adalah BST yang nilai-nilainya terletak di antara ℓ dan v; dan
3. $P(R, v, u)$ — subpohon kanannya adalah BST yang nilai-nilainya terletak di antara v dan u.

Periksalah bahwa ini benar. Syarat 2 memaksa setiap nilai L berada di bawah v, yang merupakan kuantor pertama langkah 1, *sekaligus* menjaganya tetap di atas ℓ, yang memang dibutuhkan pemanggilnya. Syarat 3 melakukan cerminannya. Syarat 2 dan 3 juga secara rekursif memaksa $\mathrm{BST}(L)$ dan $\mathrm{BST}(R)$. Jadi P terbentang persis menjadi ketiga bagian definisinya, dengan rentangnya terjalin melaluinya.

Langkah 5 — kasus dasarnya. $P(\varnothing, \ell, u)$ benar untuk setiap ℓ dan u: pohon kosong adalah BST, dan klaim "setiap nilai di dalamnya terletak di antara ℓ dan u" benar secara hampa sebab tak ada nilai apa pun di dalamnya.

Langkah 6 — pulihkan pertanyaan aslinya. $\mathrm{BST}(t) \Leftrightarrow P(t, -\infty, +\infty)$, sebab dengan batas tak hingga syarat 1 terpenuhi dengan sendirinya.

Itulah algoritmanya, dan rekursi yang diuraikannya persis merupakan fungsi penunjang yang disarankan soalnya.

> **WAWASAN KUNCI: MENGAPA BATASNYA BEKERJA — SEBUAH PARTISI RUANG KUNCI.** Perhatikan apa yang sedang dikerjakan rekursinya. Di akar, selang $(-\infty, +\infty)$ dipotong di v menjadi $(-\infty, v)$ dan $(v, +\infty)$; di aras berikutnya tiap separuh dipotong lagi. Setiap simpul memiliki satu selang terbuka ruang kunci yang saling lepas, dan sifat BST persis merupakan pernyataan *setiap nilai simpul terletak di dalam selang yang dimilikinya*. Ini gerakan yang sama dengan partisi Quicksort pada Bab 5 dan Soal 8.2: membagi menurut **nilai**, bukan menurut posisi. Melihat hubungan itu berarti melihat bahwa Bab 5 dan Bab 6 berbagi satu gagasan.

8.6.5 Programnya

Kode 8.4 memuat jawaban yang benar, jawaban yang keliru sebagai pembanding, cara kedua lewat penelusuran inorder, serta versi diagnostik yang menyebut simpul pertama yang melanggar batasnya.

Kode 8.4 Soal 8.3 selengkapnya: jawaban selang-terbatas, pemeriksaan lokal yang gagal, cara kedua lewat penelusuran inorder, dan sebuah diagnostik yang menyebut simpul pertama yang bersalah.

```
// isbst.cpp -- Soal 8.3: isBST(), dan jawaban keliru yang menggoda.
#include <iostream>
#include <string>
#include <climits>
#include <algorithm>
using namespace std;

struct MyTreeNode;
using MyTree = MyTreeNode*;
struct MyTreeNode { int value; MyTree left, right; int h; };

MyTree emptyTree()       { return nullptr; }
bool   isEmpty(MyTree t) { return t == nullptr; }
int    height(MyTree t)  { return isEmpty(t) ? -1 : t->h; }
MyTree node(int v, MyTree l, MyTree r) {
    MyTree t = new MyTreeNode; t->value = v; t->left = l; t->right = r;
    t->h = 1 + max(height(l), height(r)); return t;
}
MyTree leaf(int v) { return node(v, emptyTree(), emptyTree()); }
string shape(MyTree t) {
    if (isEmpty(t)) return ".";
    string s = to_string(t->value);
    if (isEmpty(t->left) && isEmpty(t->right)) return s;
    return s + "{" + shape(t->left) + "," + shape(t->right) + "}";
}

// ---- Jawaban yang KELIRU: hanya memeriksa induk terhadap kedua anaknya. ----
bool isBSTLocal(MyTree t) {
    if (isEmpty(t)) return true;
    if (!isEmpty(t->left)  && t->left->value  >= t->value) return false;
    if (!isEmpty(t->right) && t->right->value <= t->value) return false;
    return isBSTLocal(t->left) && isBSTLocal(t->right);
}

// ---- Jawaban yang benar: bawa selang terbuka turun menyusuri rekursi. ----
// Setiap nilai di subpohon berakar t harus memenuhi lowerBound < v < upperBound.
// batas bertipe long long dipakai agar INT_MIN dan INT_MAX sah sebagai nilai pohon.
static bool isBST(MyTree t, long long lowerBound, long long upperBound) {
    if (isEmpty(t)) return true;                                // kasus dasar
    if (t->value <= lowerBound || t->value >= upperBound) return false;
    return isBST(t->left,  lowerBound,   t->value)              // kiri: (lo, v)
        && isBST(t->right, t->value,     upperBound);           // kanan: (v, hi)
}
bool isBST(MyTree t) {
    return isBST(t, (long long)INT_MIN - 1, (long long)INT_MAX + 1);
}
```

Kode 8.4 (Lanjutan) *Soal 8.3 selengkapnya: jawaban selang-terbatas, pemeriksaan lokal yang gagal, cara kedua lewat penelusuran inorder, dan sebuah diagnostik yang menyebut simpul pertama yang bersalah.*

```
// ---- Cara kedua: penelusuran inorder harus menaik secara tegas. ----
static bool inorderIncreasing(MyTree t, long long& prev) {
    if (isEmpty(t)) return true;
    if (!inorderIncreasing(t->left, prev)) return false;
    if (t->value <= prev) return false;        // satu-satunya perbandingan yang menentukan
    prev = t->value;
    return inorderIncreasing(t->right, prev);
}
bool isBSTByWalk(MyTree t) {
    long long prev = (long long)INT_MIN - 1;
    return inorderIncreasing(t, prev);
}

// ---- Versi diagnostik: sebutkan simpul dan batas yang dilanggarnya. ----
static bool audit(MyTree t, long long lo, long long hi, string& why) {
    if (isEmpty(t)) return true;
    if (t->value <= lo) {
        why = "node " + to_string(t->value) + " must be > " + to_string(lo);
        return false;
    }
    if (t->value >= hi) {
        why = "node " + to_string(t->value) + " must be < " + to_string(hi);
        return false;
    }
    return audit(t->left, lo, t->value, why) && audit(t->right, t->value, hi, why);
}

void report(const char* name, MyTree t) {
    string why = "-";
    bool ok = audit(t, (long long)INT_MIN - 1, (long long)INT_MAX + 1, why);
    cout << name << "\n";
    cout << "  shape        : " << shape(t) << "\n";
    cout << "  isBSTLocal   : " << (isBSTLocal(t)  ? "true " : "false") << "   (the wrong
method)\n";
    cout << "  isBST        : " << (isBST(t)       ? "true " : "false") << "\n";
    cout << "  isBSTByWalk  : " << (isBSTByWalk(t) ? "true " : "false") << "\n";
    cout << "  first fault  : " << (ok ? "none" : why) << "\n\n";
}

int main() {
    // BST yang sungguh-sungguh sah.
    report("good : root 20, left 10{5,15}, right 30{25,35}",
           node(20, node(10, leaf(5), leaf(15)), node(30, leaf(25), leaf(35))));
```

Kode 8.4 (Lanjutan) *Soal 8.3 selengkapnya: jawaban selang-terbatas, pemeriksaan lokal yang gagal, cara kedua lewat penelusuran inorder, dan sebuah diagnostik yang menyebut simpul pertama yang bersalah.*

```
    // Contoh sanggahan yang klasik: tiap pasangan induk-anak baik-baik saja, tetapi 25
    // berada di subpohon KIRI milik 20, padahal 25 > 20.
    report("trap : root 20, left 10{5,25}, right 30",
           node(20, node(10, leaf(5), leaf(25)), leaf(30)));

    // Nilai kembar. Definisi BST yang dipakai di sini melarang kunci yang sama.
    report("dup  : root 20, left 20, right 30",
           node(20, leaf(20), leaf(30)));

    // Satu simpul tunggal berisi INT_MIN: pengujian nilai penjaga.
    report("edge : the single node INT_MIN", leaf(INT_MIN));

    // Pohon kosong adalah BST menurut kesepakatan.
    report("empty: the empty tree", emptyTree());
    return 0;
}
```

Programnya mencetak:

```
good : root 20, left 10{5,15}, right 30{25,35}
  shape        : 20{10{5,15},30{25,35}}
  isBSTLocal   : true    (the wrong method)
  isBST        : true
  isBSTByWalk  : true
  first fault  : none

trap : root 20, left 10{5,25}, right 30
  shape        : 20{10{5,25},30}
  isBSTLocal   : true    (the wrong method)
  isBST        : false
  isBSTByWalk  : false
  first fault  : node 25 must be < 20

dup  : root 20, left 20, right 30
  shape        : 20{20,30}
  isBSTLocal   : false   (the wrong method)
  isBST        : false
  isBSTByWalk  : false
  first fault  : node 20 must be < 20

edge : the single node INT_MIN
  shape        : -2147483648
  isBSTLocal   : true    (the wrong method)
  isBST        : true
  isBSTByWalk  : true
  first fault  : none

empty: the empty tree
  shape        : .
  isBSTLocal   : true    (the wrong method)
  isBST        : true
  isBSTByWalk  : true
  first fault  : none
```

Baris ke-2 blok kedua itulah seluruh pelajarannya: `isBSTLocal` berkata **true** dan `isBST` berkata **false**, pada pohon empat simpul, dan diagnostiknya menyebut simpul yang bersalah.

Tabel 8.5 menelusuri rekursi yang benar pada pohon jebakan itu dengan tangan, sehingga Anda dapat mereproduksinya dengan pensil dan kertas. Bacalah ia sebagai tumpukan: tiap baris adalah satu panggilan, beserta batas yang diterimanya.

Tabel 8.5 Rekursi yang benar ditelusuri pada pohon jebakan $20\{10\{5,25\},30\}$. Batas $u = 20$ mencapai simpul 25 lewat *dua* aras rekursi; itulah keterangan yang dibuang oleh pemeriksaan lokal.

Panggilan	Simpul	ℓ	u	$\ell<v<u$?	Hasil
1	20	$-\infty$	$+\infty$	$-\infty < 20 < +\infty$ ✓	berekursi
2	10	$-\infty$	20	$-\infty < 10 < 20$ ✓	berekursi
3	5	$-\infty$	10	$-\infty < 5 < 10$ ✓	kedua upapohonnya kosong → **true**
4	25	10	20	$25 < 20$? **tidak**	**false**
5	—	—	—	—	panggilan 2 mengembalikan **false**
6	—	—	—	—	panggilan 1 mengembalikan **false**; simpul 30 tak pernah dikunjungi

Perhatikan baris 4: simpul 25 ditolak oleh batas atas 20, yang datang dari *kakeknya*. Itulah mekanismenya. Batas adalah cara sebuah fakta tentang leluhur mengembara turun ke keturunannya.

8.6.6 Mengapa ini bukan sifat heap

Sebuah selingan pendek tetapi berharga, sebab kedua sifat ini begitu sering tertukar. Bab 5 membangun heap, yang juga menyimpan bilangan di dalam pohon biner dan juga punya aturan pengurutan induk–anak. Aturannya tidak sama, dan bedanya persis beda antara *lokal* dan *global*.

- **Sifat heap-maks** adalah $A[\text{parent}(i)] \geq A[i]$ — aturan antara sebuah simpul dan *anak-anaknya saja*. Ia tidak mengatakan apa pun tentang kiri lawan kanan. Ia benar-benar lokal, dan itulah sebabnya pemeriksaan lokal *memang* pemeriksaan heap yang benar.
- **Sifat BST** adalah aturan antara sebuah simpul dan seluruh *subpohon*-nya. Ia global, dan itulah sebabnya pemeriksaan lokal keliru.

Jadi: heap dapat diperiksa dengan membandingkan tiap simpul dengan kedua anaknya, sedangkan BST tidak. Jika seseorang bertanya "mengapa `isBST` butuh batas padahal `isMaxHeap` tidak?", itulah jawabannya.

8.6.7 Penyelesaian yang menggoda tetapi keliru, dan mengapa ia gagal

Inilah sebuah versi yang tampak rampung. Ia persis gagasan selang-terbatas, dengan batas bertipe `int` yang diawali dengan nilai terkecil dan terbesar yang dapat ditampung sebuah `int`:

```
bool isBSTNarrow(MyTree t, int lowerBound, int upperBound) {
    if (isEmpty(t)) return true;
    if (t->value <= lowerBound || t->value >= upperBound) return false;
    return isBSTNarrow(t->left,  lowerBound, t->value)
        && isBSTNarrow(t->right, t->value,   upperBound);
}
bool isBSTNarrow(MyTree t) { return isBSTNarrow(t, INT_MIN, INT_MAX); }
```

Bandingkanlah ia dengan `isBST` pada **Kode 8.4**. Strukturnya identik, baris demi baris: kasus dasarnya, uji ganda yang tegas, dan kedua panggilan rekursif dengan nilai simpulnya sendiri disubstitusikan sebagai batas baru pada sisi yang sesuai. **Penurunan di §8.6.4 membawa kita langsung kepadanya**, dan justru itulah yang membuatnya menggoda. Perbandingan tegas `<=` dan `>=` cocok dengan definisi pada **Gambar 8.1**, yang tidak membolehkan dua simpul bernilai sama; nilai kembar ditolak, sebagaimana ditegaskan blok `dup` pada keluarannya.

Ada satu perbedaan, dan itu cacat yang sungguh-sungguh ada, bukan soal selera.

Nilai awal `INT_MIN` dan `INT_MAX` bukanlah $-\infty$ dan $+\infty$; keduanya adalah nilai terkecil dan terbesar yang benar-benar dapat ditampung sebuah `int`. Jadi tinjaulah pohon yang terdiri atas satu simpul `INT_MIN`. Fungsi penunjangnya dimasuki dengan `lowerBound = INT_MIN`, dan uji pertamanya adalah

$$\text{value} \leq \text{lowerBound} \;\Rightarrow\; \text{INT_MIN} \leq \text{INT_MIN} \;\Rightarrow\; \text{true},$$

sehingga fungsinya mengembalikan `false`. BST satu simpul yang sepenuhnya sah dinyatakan bukan BST. Hal yang sama terjadi pada pohon yang memuat `INT_MAX`. Karena kedua nilai ekstrem itu nyaris tak pernah muncul dalam uji buatan tangan, versi ini lolos dari setiap uji yang mungkin ditulis oleh pemrogram yang terburu-buru.

Kode 8.9 di §8.14 memuat versi sempit ini persis, ditulis untuk pohon merah-hitam, dan memperagakan kegagalannya; baris-baris keluarannya yang bersangkutan adalah

```
The int-bounds pitfall, on a single BLACK node holding INT_MIN
  isBST with int bounds          : false
  isBSTRepaired, wider bounds    : true
```

Ada tiga perbaikan yang bersih.

1. **Lebarkan tipe batasnya.** Lewatkan batas bertipe `long long` dan awali dengan INT_MIN − 1 dan INT_MAX + 1, nilai-nilai yang tak mungkin disamai `int` mana pun. Inilah yang dilakukan **Kode 8.4**. Ia perubahan terkecil, dan perilakunya pada setiap masukan lain tidak tersentuh.
2. **Izinkan batas tidak ada.** Lewatkan batas bertipe `std::optional<int>`, dengan optional kosong berarti "tak ada batas di sisi ini": `if (lowerBound && t->value <= *lowerBound) return false;`. Tak ada nilai yang dicadangkan sebagai penjaga, jadi tak ada yang dapat bertabrakan dengan nilai sungguhan.
3. **Beralihlah ke cara penelusuran inorder**, yang tidak memerlukan penjaganya sendiri asalkan nilai awal "belum ada yang terlihat"-nya berada di bawah setiap `int`. `isBSTByWalk` pada **Kode 8.4** adalah cara ini.

> **CONTOH: CARA MENYATAKAN PERBAIKANNYA.** *"Dengan batas bertipe `int`, penjaga `INT_MIN` dan `INT_MAX` merupakan nilai pohon yang dapat dicapai, sehingga pohon yang memuat salah satunya ditolak. Karena itu parameter batasnya dilebarkan menjadi `long long`; pada setiap masukan lain perilakunya tidak berubah."* Tiga kalimat: masukan yang gagal, penyebabnya, dan perbaikannya beserta cakupannya. Seperti itulah pemikiran kasus-tepi yang cermat bila dituliskan.

8.6.8 Cara kedua: penelusuran inorder

Bab 6 §6.6.1 membuktikan bahwa sebuah pohon biner adalah pohon pencarian biner **jika dan hanya jika** penelusuran inordernya menaik secara tegas. Teorema itu memberikan penyelesaian kedua yang sama sekali berdiri sendiri untuk Soal 8.3, dan keduanya layak ditulis: dua cara yang sepakat adalah argumen kebenaran termurah yang ada.

Caranya: telusurilah pohonnya secara berurutan, dengan mengingat hanya nilai yang tercetak sebelumnya, lalu gagalkan pada saat sebuah nilai tidak lebih besar daripada pendahulunya. `inorderIncreasing` pada **Kode 8.4** panjangnya delapan baris. Dua rincian menjadi penting.

Peubah `prev` harus dipakai bersama di sepanjang penelusuran, bukan per panggilan. Dalam C++ itu berarti `long long& prev`, sebuah rujukan. Versi yang melewatkan `prev` menurut nilai memperoleh fungsi yang membandingkan tiap simpul hanya dengan simpul di dalam upapohonnya sendiri — dan itu pemeriksaan lokal lagi, dalam penyamaran.

Perbandingannya harus `<=`, bukan `<`. Dengan `<`, nilai yang sama lolos, padahal definisinya melarangnya.

Perbandingan biayanya: kedua cara berbiaya waktu $\Theta(n)$ pada kasus terburuk dan ruang tumpukan $O(h)$. Cara batas punya satu keunggulan yang layak disebut — ia dapat berhenti lebih awal pada simpul *pertama* yang melanggar batas, tanpa lebih dahulu menelusuri seluruh sisi kiri pohonnya, dan itulah sebabnya versi diagnostiknya dapat menyebut pelakunya. Keunggulan cara penelusuran adalah ia tidak butuh penjaga sama sekali.

8.6.9 Belokan keliru yang sering terjadi pada soal ini

Diurutkan menurut seberapa sering ia muncul, dan berapa biayanya.

1. **Pemeriksaan lokal.** Rekursi dan kasus dasarnya benar, sifatnya keliru. §8.6.3.
2. **Batas yang tidak tegas.** Menulis `node.value < lowerBound` alih-alih `<=`. Ini menerima nilai kembar, yang dilarang **Gambar 8.1**, dan langsung tertemukan oleh uji dua simpul `20{20,.}`.
3. **Menukar batas di dalam panggilan rekursifnya.** Menulis `isBST(t.left, t.value, upperBound)` — nilai simpulnya sebagai batas *bawah* di *kiri*. Ini menolak setiap pohon tak-sepele, dan tertangkap dalam sepuluh detik dengan menguji pohon tiga simpul.
4. **Melupakan kasus dasarnya**, sehingga kodenya mengakses pohon kosong. Programnya gagal pada subpohon kosong pertama yang ditemuinya.
5. **Memamerkan fungsi penunjangnya.** Batas-batas itu adalah rincian implementasi; pemanggil semestinya hanya melihat `isBST(t)`. Pada **Kode 8.4** fungsi penunjangnya `static`, hanya tampak di dalam berkasnya sendiri.
6. **Memakai penjaga bertipe `int`.** §8.6.7. Kodenya benar pada setiap masukan kecuali yang memuat kedua nilai ekstrem, dan justru itulah sebabnya ia lolos dari pengujian sepintas.

8.7 Soal 8.4 — `printDescending`: Dari yang Terbesar, Tanpa Daftar

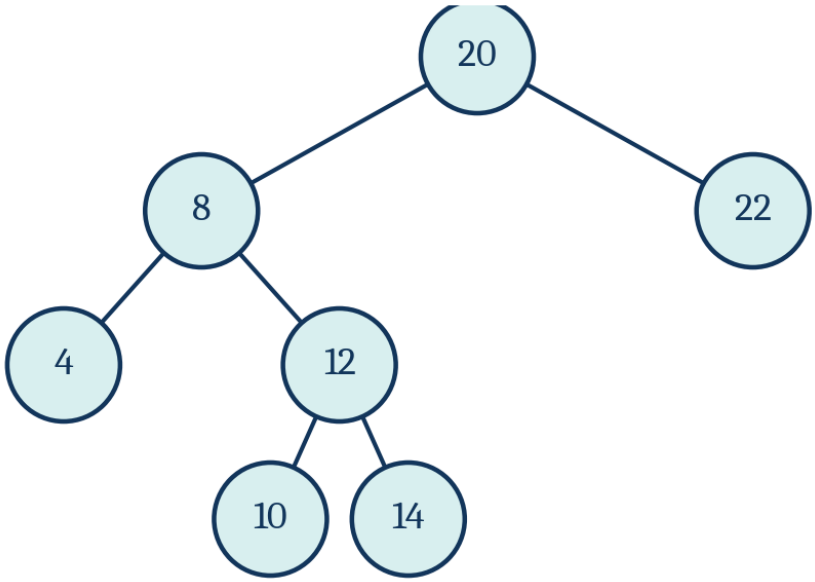
8.7.1 Soalnya

Tulislah fungsi rekursif `void printDescending(MyTree t)` yang menerima sebuah BST `t` berisi bilangan bulat dan mencetak nilai-nilainya dalam *urutan menurun*. Fungsi itu harus bekerja **tanpa membuat daftar terpisah berisi nilai-nilai `t`**.

8.7.2 Tentang apa soal ini sebenarnya

Satu gagasan, dan satu pembatasan.

Gagasannya adalah bahwa **penelusuran inorder sebuah BST itu terurut**, yang dibuktikan Bab 6 §6.6.1, dan yang diilustrasikan **Gambar 8.2**: penerus inorder 8 adalah 10, pengganti 10 adalah 12, pengganti 14 adalah 20. Bacalah hubungan-hubungan itu ujung ke ujung dan Anda memperoleh barisan terurutnya. Jika penelusuran inorder — kiri, simpul, kanan — memberikan urutan menaik, maka bayangan cerminnya — kanan, simpul, kiri — pasti memberikan urutan menurun. Itulah seluruh jawabannya, dan panjangnya empat baris.



Gambar 8.2 Penerus inorder. Merangkai para penggantinya — 8, 10, 12, 14, 20 — menjejakkan urutan terurut yang dihasilkan penelusuran inorder. Membalik rangkaiannya memberikan urutan menurun.

Pembatasannya adalah klausa *"tanpa membuat daftar terpisah berisi nilai-nilai* `t`*"*. Itu bukan hiasan. Klausa itu mengesampingkan penyelesaian yang paling sering diraih orang: kumpulkan nilai-nilainya ke dalam larik, urutkan atau balik, lalu cetak. Klausa itu ada untuk memaksa keluaran terurut **dari struktur pohonnya semata**, dan itulah sifat yang menjadi pokok soal ini.

8.7.3 Menurunkan jawabannya

Langkah 1 — tuliskan apa yang dikerjakan penelusuran inorder, dan mengapa ia terurut. Untuk pohon $t = (v, L, R)$ yang memenuhi sifat BST, setiap nilai di L kurang dari v dan setiap nilai di R lebih besar daripada v. Jadi jika kita mencetak nilai-nilai L dalam urutan menaik, lalu v, lalu nilai-nilai R dalam urutan menaik, keseluruhan keluarannya menaik. Dengan induksi atas tingginya, penelusuran inorder mencetak nilai-nilainya dalam urutan menaik.

Langkah 2 — balikkan tuntutannya lalu lihat apa yang berubah. Kita menginginkan urutan menurun. Setiap nilai di R lebih besar daripada v; setiap nilai di L kurang dari v. Jadi jika kita mencetak nilai-nilai R dalam urutan *menurun*, lalu v, lalu nilai-nilai L dalam urutan *menurun*, keseluruhan keluarannya menurun.

Langkah 3 — tuliskan itu sebagai rekursi.

$$\begin{aligned}\text{printDescending}(\varnothing) &= \text{tidak ada}, \qquad \text{printDescending}((v, L, R)) \\ &= \text{printDescending}(R);\ \text{cetak } v;\ \text{printDescending}(L).\end{aligned}$$

Langkah 4 — periksa pembatasannya. Tak ada yang dikumpulkan. Tiap nilai dicetak pada saat rekursinya mencapai simpulnya, lalu dilupakan. Satu-satunya ingatan yang terpakai adalah tumpukan rekursinya, yang memuat paling banyak $h + 1$ bingkai — dan tumpukan rekursi bukanlah "daftar nilai yang terpisah dari `t`" menurut pembacaan mana pun; ia adalah mekanisme yang membuat *setiap* penelusuran rekursif bekerja, termasuk penelusuran menaik yang tak seorang pun akan menyebutnya "daftar".

Langkah 5 — kompleksitasnya. Tiap simpul dikunjungi tepat sekali, dan di tiap simpul kerjanya adalah satu perbandingan terhadap `isEmpty`, satu pencetakan, dan dua panggilan: $\Theta(1)$. Jadi

waktunya $\Theta(n)$. Ruangnya $O(h)$ untuk tumpukannya, yang berarti $O(\lg n)$ jika pohonnya seimbang dan $O(n)$ jika ia sebuah rantai.

> **WAWASAN KUNCI: SEBUAH PENELUSURAN ADALAH TIGA PERNYATAAN DALAM SUATU URUTAN.** Setiap satu dari penelusuran klasik adalah tiga pernyataan yang sama — *berekursi ke kiri*, *kunjungi*, *berekursi ke kanan* — dalam urutan yang berbeda. Preorder adalah kunjungi, kiri, kanan. Inorder adalah kiri, kunjungi, kanan. Postorder adalah kiri, kanan, kunjungi. Inorder terbalik adalah kanan, kunjungi, kiri. Tak ada lagi yang perlu dihafalkan. Permintaan akan "urutan menurun" adalah permintaan untuk memindahkan satu baris.

8.7.4 Programnya

Kode 8.5 memberikan penyelesaian rekursifnya, penelusuran menaik sebagai pembanding, jawaban iteratif dengan tumpukan eksplisit, serta versi bertelusur kedalaman yang memperagakan klaim tumpukan $O(h)$.

Kode 8.5 Soal 8.4 selengkapnya: jawaban rekursif empat baris, penelusuran menaik di sampingnya, jawaban iteratif dengan tumpukan eksplisit, dan sebuah penelusuran yang memastikan kedalaman tumpukan $O(h)$.

```
// printdesc.cpp -- Soal 8.4: printDescending().
#include <iostream>
#include <string>
#include <stack>
#include <algorithm>
using namespace std;

struct MyTreeNode;
using MyTree = MyTreeNode*;
struct MyTreeNode { int value; MyTree left, right; int h; };

MyTree emptyTree()       { return nullptr; }
bool   isEmpty(MyTree t) { return t == nullptr; }
int    height(MyTree t)  { return isEmpty(t) ? -1 : t->h; }
MyTree node(int v, MyTree l, MyTree r) {
    MyTree t = new MyTreeNode; t->value = v; t->left = l; t->right = r;
    t->h = 1 + max(height(l), height(r)); return t;
}
MyTree leaf(int v) { return node(v, emptyTree(), emptyTree()); }

// ---- Jawabannya: bayangan cermin dari penelusuran inorder. ----
// Subpohon kanan dahulu, lalu simpulnya, lalu subpohon kiri. Tak ada daftar dibentuk:
// tiap nilai dicetak begitu rekursi sampai kepadanya.
void printDescending(MyTree t) {
    if (isEmpty(t)) return;        // kasus dasar: tak ada yang dicetak
    printDescending(t->right);     // setiap nilai di sini LEBIH BESAR daripada t->value
    cout << t->value << ' ';       // lalu t itu sendiri
    printDescending(t->left);      // setiap nilai di sini LEBIH KECIL daripada t->value
}

// Sebagai pembanding, penelusuran menaik yang biasa.
void printAscending(MyTree t) {
    if (isEmpty(t)) return;
    printAscending(t->left);
    cout << t->value << ' ';
    printAscending(t->right);
}
```

Kode 8.5 (Lanjutan) *Soal 8.4 selengkapnya: jawaban rekursif empat baris, penelusuran menaik di sampingnya, jawaban iteratif dengan tumpukan eksplisit, dan sebuah penelusuran yang memastikan kedalaman tumpukan O(h).*

```
// ---- Cara kedua: penelusuran yang sama dengan tumpukan eksplisit. ----
// Memakai ruang tambahan O(h), yang toh dipakai juga oleh rekursinya; ia tetap
// tidak membentuk daftar berisi seluruh n nilai.
void printDescendingIterative(MyTree t) {
    stack<MyTree> s;
    MyTree cur = t;
    while (cur != nullptr || !s.empty()) {
        while (cur != nullptr) { s.push(cur); cur = cur->right; }  // menyelam ke kanan
        cur = s.top(); s.pop();
        cout << cur->value << ' ';
        cur = cur->left;                                           // lalu ke kiri
    }
}

// Pencacah kedalaman, untuk menunjukkan rekursinya tak pernah lebih dalam dari h+1.
int maxDepth = 0;
void printDescendingTraced(MyTree t, int d) {
    if (isEmpty(t)) return;
    maxDepth = max(maxDepth, d);
    printDescendingTraced(t->right, d + 1);
    printDescendingTraced(t->left,  d + 1);
}

int main() {
    // Pohon delapan simpul yang sama, dipakai di sepanjang bab ini.
    MyTree t = node(13,
                    node(10, node(5, leaf(4), leaf(6)), leaf(11)),
                    node(15, emptyTree(), leaf(16)));

    cout << "ascending  : "; printAscending(t);            cout << "\n";
    cout << "descending : "; printDescending(t);           cout << "\n";
    cout << "iterative  : "; printDescendingIterative(t);  cout << "\n";

    printDescendingTraced(t, 1);
    cout << "tree height h = " << height(t)
         << ", deepest recursive call = " << maxDepth << "\n";

    cout << "single node: "; printDescending(leaf(42));    cout << "\n";
    cout << "empty tree : "; printDescending(emptyTree()); cout << "(nothing)\n";
```

Kode 8.5 (Lanjutan) *Soal 8.4 selengkapnya: jawaban rekursif empat baris, penelusuran menaik di sampingnya, jawaban iteratif dengan tumpukan eksplisit, dan sebuah penelusuran yang memastikan kedalaman tumpukan O(h).*

```
    // Rantai condong kanan 1 -> 2 -> ... -> 6: menurun harus mencetak 6..1.
    MyTree chain = emptyTree();
    for (int v = 6; v >= 1; --v) chain = node(v, emptyTree(), chain);
    cout << "chain 1..6 : "; printDescending(chain);       cout << "\n";
    return 0;
}
```

Programnya mencetak:

```
ascending  : 4 5 6 10 11 13 15 16
descending : 16 15 13 11 10 6 5 4
iterative  : 16 15 13 11 10 6 5 4
tree height h = 3, deepest recursive call = 4
single node: 42
empty tree : (nothing)
chain 1..6 : 6 5 4 3 2 1
```

Tiga hal dapat dibaca dari keluaran itu. Baris menurunnya adalah baris menaiknya dibaca terbalik, sebagaimana seharusnya. Baris iteratifnya identik dengan baris rekursifnya, dan itulah pemeriksaan termurah yang mungkin bahwa cara keduanya benar. Dan panggilan rekursif terdalamnya adalah 4 pada pohon bertinggi 3, yang memastikan batas *kedalaman* $\leq h + 1$: ada $h + 1$ aras simpul, dan panggilan pada subpohon kosong di bawah daunnya tidak mengerjakan apa-apa dan dihitung sebagai langsung kembali.

8.7.5 Cara kedua: tumpukan eksplisit

`printDescendingIterative` pada **Kode 8.5** layak ditelusuri, sebab penelusuran inorder iteratif adalah soal wawancara yang baku dan versi terbaliknya adalah cerminnya.

Invariannya adalah: *tumpukan itu memuat persis simpul-simpul yang nilainya belum dicetak tetapi yang seluruh nilai subpohon kanannya sudah dicetak.* Membaca gelungnya:

1. `while` bagian dalamnya mendorong `cur`, lalu berpindah ke `cur->right`, berulang-ulang. Jadi ia mendorong seluruh tulang punggung kanan dari `cur` ke bawah, dan berakhir dengan `cur == nullptr`.
2. `s.top()` lalu menjadi simpul terdalam pada tulang punggung itu — nilai terbesar yang belum dicetak. Keluarkan lalu cetak.
3. Tetapkan `cur` ke anak kiri simpul itu lalu ulangi. Tulang punggung kanan milik anak kiri itu didorong berikutnya, dan seterusnya.

Apakah ini melanggar klausa "tanpa daftar terpisah"? Tidak: tumpukannya memuat *simpul yang masih harus diproses*, tak pernah lebih dari $h + 1$ buah, dan ia persis data yang disimpan rekursinya secara tersirat. Namun hal itu tetap layak dinyatakan dalam sebuah komentar, sebab pembaca yang memindai mencari larik akan melihat `stack<MyTree>` lalu terhenti: *"tumpukannya memuat paling banyak $h + 1$ simpul tertunda, tak pernah berupa daftar berisi n nilai; ia adalah tumpukan rekursi yang dibuat eksplisit."*

8.7.6 Belokan keliru yang sering terjadi pada soal ini

1. **Tetap membangun daftar.** Mengumpulkan ke dalam sebuah `vector`, memanggil `sort`, lalu `reverse`. Ini cara yang dilarang, dan ia juga membuang satu-satunya sifat menarik dari masukannya — bahwa ia sudah terurut oleh strukturnya. Keluarannya benar, penyelesaiannya tidak.
2. **Mencetak menaik.** Menulis penelusuran inorder yang biasa, yang tertangkap dengan melihat baris pertama keluarannya.
3. **Urutannya benar tetapi penempatannya keliru** — menulis `printDescending(t->right); printDescending(t->left); cout << t->value;`, yang merupakan penelusuran postorder dan mencetak omong kosong. Ujilah pada pohon tiga simpul 10{5,15}: jawabannya harus `15 10 5`.
4. **Mengembalikan sebuah `string` alih-alih mencetak.** Tanda tangannya berkata `void`. Membangun lalu mengembalikan untai, pada hakikatnya, adalah membangun daftar.
5. **Melupakan kasus dasarnya** lalu mendereferensi pohon kosong.

8.8 Soal 8.5 — `max`: Kunci Terbesar Lewat Satu Lintasan

8.8.1 Soalnya

Tulislah fungsi rekursif yang **efisien** `int max(MyTree t)` yang menerima sebuah BST `t` berisi bilangan bulat dan mengembalikan nilai terbesarnya. Fungsi itu **tidak boleh mengunjungi dan membandingkan semua simpul pohonnya**: ia boleh mengikuti paling banyak satu lintasan dari akar, sehingga pada BST yang seimbang ia berjalan dalam waktu $O(\lg n)$.

Petunjuk: berdirilah di sebuah simpul `x` dari sebuah BST. Setiap nilai di subpohon kiri `x` kurang dari atau sama dengan ($\leq$) nilai `x`. Jadi maksimumnya tidak mungkin ada di subpohon kiri. Di manakah ia?

8.8.2 Tentang apa soal ini sebenarnya

Bukan "dapatkah Anda mencari maksimum" — pemrogram pekan pertama pun dapat melakukannya. Pokoknya adalah apakah Anda dapat **memakai sifat BST untuk menghindari kerja**. Kata-kata *"tidak boleh mengunjungi dan membandingkan semua simpul"* mengubah tugas sepele menjadi tugas yang sungguhan: hasil benar yang diperoleh dengan mengunjungi n simpul dikesampingkan secara tersurat, jadi inti soalnya adalah argumen pemangkasannya, bukan angkanya.

8.8.3 Menurunkan jawabannya

Langkah 1 — nyatakan fakta yang kita punya. Pada simpul x mana pun bernilai v, setiap nilai di subpohon kiri milik x kurang dari v, dan setiap nilai di subpohon kanannya lebih besar daripada v. (Petunjuknya berkata $\leq$; definisi pada **Gambar 8.1** berkata $<$ dan melarang nilai kembar. Bagaimanapun juga argumen di bawah ini tidak terpengaruh, dan §8.8.6 menjelaskan mengapa.)

Langkah 2 — tanyakan di mana maksimumnya mungkin berada. Nilai-nilai pohonnya adalah: v itu sendiri, nilai-nilai di subpohon kiri, dan nilai-nilai di subpohon kanan. Setiap nilai subpohon kiri $< v$, jadi tak ada nilai subpohon kiri yang dapat menjadi maksimum selama v ada — dan v selalu ada. **Subpohon kirinya dapat dibuang, tanpa diperiksa, di tiap simpul.**

Langkah 3 — bandingkan v dengan subpohon kanannya. Setiap nilai subpohon kanan $> v$. Jadi jika subpohon kanannya tak-kosong, maksimumnya *bukan* v; ia adalah maksimum subpohon kanan itu. Dan jika subpohon kanannya kosong, maka calon yang tersisa hanyalah v dan nilai-nilai subpohon kiri, yang semuanya lebih kecil, sehingga jawabannya adalah v.

Langkah 4 — tuliskanlah.

$$\max((v, L, \varnothing)) = v, \qquad \max((v, L, R)) = \max(R) \text{ bila } R \neq \varnothing.$$

Kasus dasarnya adalah *"subpohon kanannya kosong"*, bukan *"pohonnya kosong"*. Pohon kosong tidak punya maksimum sama sekali — tak ada bilangan bulat untuk dikembalikan — dan §8.8.7 menyebutkan apa yang harus dilakukan tentangnya.

Langkah 5 — periksa bahwa "satu lintasan" dipatuhi. Tiap panggilan membuat paling banyak satu panggilan rekursif, dan panggilan itu selalu pada subpohon kanannya. Jadi barisan simpul yang dikunjungi adalah t, anak kanan t, anak kanan simpul itu, dan seterusnya, sampai anak kanannya kosong. Itulah **lintasan terkanan** pohon itu — satu lintasan, dari akarnya, persis sebagaimana diminta.

> **DEFINISI (tulang punggung kanan). Tulang punggung kanan** sebuah pohon tak-kosong adalah barisan simpul yang diperoleh dengan mulai dari akarnya lalu berulang kali mengambil anak kanannya sampai anak kanannya kosong. Panjangnya paling banyak $h + 1$ simpul. `max` mengembalikan nilai di ujung tulang punggung kanan; `min` mengembalikan nilai di ujung tulang punggung kiri.

Langkah 6 — hitung biayanya. Tulang punggung kanannya punya paling banyak $h + 1$ simpul, satu per aras. Tiap simpul berbiaya $\Theta(1)$: satu uji kekosongan dan satu panggilan rekursif. Jadi waktu jalannya $O(h)$ dengan h tinggi pohonnya. Itulah batas yang jujur, dan §8.8.5 menjelaskan mengapa ia tidak persis seperti yang dikatakan naskahnya.

8.8.4 Programnya

Kode 8.6 mengimplementasi jawabannya secara rekursif, lalu secara iteratif, dan — sebagai pembanding — versi pemindaian penuh yang dilarang, lengkap dengan pencacah sehingga bedanya terlihat alih-alih sekadar dinyatakan.

Kode 8.6 Soal 8.5 selengkapnya, dengan pencacah kunjungan simpul yang membuat beda antara cara yang diizinkan dan cara yang dilarang menjadi terukur.

```
// bstmax.cpp -- Soal 8.5: max() pada BST dalam waktu O(h).
#include <iostream>
#include <string>
#include <stdexcept>
#include <algorithm>
using namespace std;

struct MyTreeNode;
using MyTree = MyTreeNode*;
struct MyTreeNode { int value; MyTree left, right; int h; };

MyTree emptyTree()       { return nullptr; }
bool   isEmpty(MyTree t) { return t == nullptr; }
int    height(MyTree t)  { return isEmpty(t) ? -1 : t->h; }
MyTree node(int v, MyTree l, MyTree r) {
    MyTree t = new MyTreeNode; t->value = v; t->left = l; t->right = r;
    t->h = 1 + max(height(l), height(r)); return t;
}
MyTree leaf(int v) { return node(v, emptyTree(), emptyTree()); }

long long visits = 0;      // mencacah berapa simpul yang benar-benar dilihat sebuah rutin

// ---- Jawabannya: berjalan ke kanan sampai subpohon kanan kosong. ----
int maxBST(MyTree t) {
    if (isEmpty(t)) throw invalid_argument("max() of an empty tree is undefined");
    ++visits;
    if (isEmpty(t->right)) return t->value;   // tak mungkin ada nilai yang lebih besar
    return maxBST(t->right);                  // SATU panggilan rekursif, tak pernah dua
}

// Bayangan cerminnya, gratis.
int minBST(MyTree t) {
    if (isEmpty(t)) throw invalid_argument("min() of an empty tree is undefined");
    ++visits;
    if (isEmpty(t->left)) return t->value;
    return minBST(t->left);
}

// ---- Cara kedua: penelusuran yang sama tanpa rekursi. ----
int maxBSTIterative(MyTree t) {
    if (isEmpty(t)) throw invalid_argument("max() of an empty tree is undefined");
    while (!isEmpty(t->right)) { ++visits; t = t->right; }
    ++visits;
    return t->value;
}
```

Kode 8.6 (Lanjutan) *Soal 8.5 selengkapnya, dengan pencacah kunjungan simpul yang membuat beda antara cara yang diizinkan dan cara yang dilarang menjadi terukur.*

```
// ---- Jawaban yang dilarang: melihat setiap simpul. Benar, tetapi tidak boleh. ----
int maxByFullScan(MyTree t) {
    if (isEmpty(t)) throw invalid_argument("max() of an empty tree is undefined");
    ++visits;
    int best = t->value;
    if (!isEmpty(t->left))  best = max(best, maxByFullScan(t->left));
    if (!isEmpty(t->right)) best = max(best, maxByFullScan(t->right));
    return best;
}

// Cetak lintasan terkanan, agar pembaca melihat simpul mana saja yang tersentuh.
string rightSpine(MyTree t) {
    string s;
    while (!isEmpty(t)) { s += to_string(t->value); t = t->right; if (!isEmpty(t)) s += " -
> "; }
    return s;
}

int countNodes(MyTree t) { return isEmpty(t) ? 0 : 1 + countNodes(t->left) + countNodes(t-
>right); }

// Bangun BST yang seimbang sempurna pada 1..n.
MyTree perfect(int lo, int hi) {
    if (lo > hi) return emptyTree();
    int mid = lo + (hi - lo) / 2;
    return node(mid, perfect(lo, mid - 1), perfect(mid + 1, hi));
}

int main() {
    MyTree t = perfect(1, 15);            // 15 simpul, tinggi 3
    cout << "tree        : perfectly balanced BST on 1..15\n";
    cout << "nodes n     : " << countNodes(t) << ",  height h = " << height(t) << "\n";
    cout << "right spine : " << rightSpine(t) << "\n\n";

    visits = 0; int a = maxBST(t);
    cout << "maxBST          -> " << a << "   nodes visited: " << visits << "\n";
    visits = 0; int b = maxBSTIterative(t);
    cout << "maxBSTIterative -> " << b << "   nodes visited: " << visits << "\n";
    visits = 0; int c = maxByFullScan(t);
    cout << "maxByFullScan   -> " << c << "   nodes visited: " << visits << "
(forbidden)\n";
    visits = 0; int d = minBST(t);
    cout << "minBST          -> " << d << "   nodes visited: " << visits << "\n\n";
```

Kode 8.6 (Lanjutan) *Soal 8.5 selengkapnya, dengan pencacah kunjungan simpul yang membuat beda antara cara yang diizinkan dan cara yang dilarang menjadi terukur.*

```
    // Kasus terburuk yang jujur: rantai condong kiri. h = n - 1, jadi O(h) adalah O(n).
    MyTree chain = emptyTree();
    for (int v = 1; v <= 10; ++v) chain = node(v, chain, emptyTree());
    cout << "left-leaning chain 10,9,...,1 (h = " << height(chain) << ")\n";
    visits = 0; cout << "  maxBST -> " << maxBST(chain) << "   nodes visited: " << visits
<< "\n";
    visits = 0; cout << "  minBST -> " << minBST(chain) << "   nodes visited: " << visits
<< "\n\n";

    try { maxBST(emptyTree()); }
    catch (const exception& e) { cout << "empty tree  : " << e.what() << "\n"; }
    return 0;
}
```

Keluarannya:

```
tree        : perfectly balanced BST on 1..15
nodes n     : 15,  height h = 3
right spine : 8 -> 12 -> 14 -> 15

maxBST          -> 15   nodes visited: 4
maxBSTIterative -> 15   nodes visited: 4
maxByFullScan   -> 15   nodes visited: 15  (forbidden)
minBST          -> 1   nodes visited: 4

left-leaning chain 10,9,...,1 (h = 9)
  maxBST -> 10   nodes visited: 1
  minBST -> 1   nodes visited: 10

empty tree  : max() of an empty tree is undefined
```

Empat simpul lawan lima belas, pada pohon lima belas simpul bertinggi 3: cara yang diizinkan mengunjungi $h + 1 = 4$ simpul dan yang dilarang mengunjungi seluruh $n = 15$. Rasio itulah seluruh pokok soalnya, dan cacahan terukur seperti ini jauh lebih meyakinkan daripada kalimat "versi ini lebih efisien".

Blok rantainya adalah bagian yang jujur. Pada rantai condong kiri sepuluh simpul, `maxBST` mengunjungi **satu** simpul — akarnya sudah yang terbesar — sedangkan `minBST` mengunjungi kesepuluhnya, sebab tulang punggung kiri *itulah* seluruh pohonnya. Algoritma yang sama, pohon yang sama, kerja sepuluh kali lipat. Itulah arti $O(h)$, dan itulah sebabnya subbab berikutnya penting.

8.8.5 $O(h)$, $O(\lg n)$, dan cara menyatakan batasnya agar ia benar

Soalnya meminta fungsi itu berjalan *"dalam waktu $O(\lg n)$ pada BST yang seimbang"*.

Bacalah pembatasnya. *Seimbang* — pernyataan itu diam-diam bergantung pada pohon yang pendek, sesuatu yang dijamin oleh pohon AVL pada soal-soal di sekitarnya. Untuk pohon pencarian biner sembarang klaim itu salah, dan rantai pada **Kode 8.6** adalah contoh sanggahannya: $n = 10$, $h = 9$, dan `minBST` mengunjungi 10 simpul, yang berarti $\Theta(n)$, bukan $\Theta(\lg n)$.

Rantai pernyataan yang benar, dalam bahasa Bab 3, adalah:

1. `max` berjalan dalam waktu $O(h)$, dengan h tinggi pohonnya. **Ini selalu benar.**
2. Untuk pohon pencarian biner tanpa jaminan keseimbangan, h dapat sebesar $n - 1$, sehingga batasnya merosot menjadi $O(n)$. Bab 6 §6.13 mengukur persis hal ini.
3. Untuk pohon **AVL**, Bab 6 §6.15.2 membuktikan $h \leq 1.4405\lg(n + 2) - 1.3277 = O(\lg n)$. Menyubstitusikannya ke pernyataan 1 memberikan $O(\lg n)$.
4. Karena itu: `max` berjalan dalam waktu $O(\lg n)$ **pada pohon yang seimbang-tinggi**, dan dalam waktu $O(h)$ secara umum.

> **CONTOH: CARA MENYATAKAN BATASNYA AGAR IA BENAR.** *"`max()` hanya mengunjungi tulang punggung kanan, jadi biayanya $O(h)$ dengan h tinggi pohonnya. Untuk pohon AVL pada Soal 8.6 sampai 8.10 berlaku $h \leq 1.4405\lg(n + 2) - 1.3277$, sehingga biayanya $O(\lg n)$. Untuk BST yang tak seimbang kode yang sama berbiaya $O(n)$ — uji pada rantai condong kiri sepuluh simpul mengunjungi kesepuluh simpulnya — sehingga klaim $O(\lg n)$ itu bergantung pada jaminan keseimbangan, bukan pada kodenya."*
>
> Paragraf itu sepenuhnya benar, dan ia memperlihatkan beda antara sifat sebuah algoritma dan sifat masukannya — sebuah pembedaan yang lebih berharga daripada batas tunggal mana pun.

8.8.6 Tanda ≤ pada petunjuknya

Petunjuknya berkata nilai subpohon kiri "less than or equal (≤)" terhadap nilai simpulnya, sedangkan **Gambar 8.1** berkata "<" dan bahwa tak ada dua simpul yang berbagi nilai. Keduanya adalah dua kesepakatan BST yang berbeda:

- **BST tegas**, tanpa nilai kembar: definisi yang dipakai buku ini, dan yang ditegakkan `isBST` pada **Kode 8.4**.
- **BST tak tegas**, nilai kembar diizinkan di kiri: alternatif yang lazim, dipakai sebagian buku teks.

Apakah itu mengubah penyelesaian Soal 8.5? Tidak. Di bawah kesepakatan mana pun, tak ada nilai subpohon kiri yang *melampaui* nilai simpulnya, dan hanya itulah fakta yang dipakai penurunan §8.8.3. Maksimumnya ada di ujung tulang punggung kanan bagaimanapun juga. Di bawah kesepakatan tak tegas nilai maksimumnya boleh muncul lebih dari sekali, tetapi `max` mengembalikan nilainya, bukan simpulnya, sehingga bilangan bulat yang dikembalikan tetap sama.

Dalam satu baris: *tanda ≤ pada petunjuknya dan tanda < pada definisinya memberikan jawaban yang sama di sini, sebab penurunannya hanya membutuhkan "tak ada nilai subpohon kiri yang melampaui akarnya".*

8.8.7 Pohon kosong

`int max(MyTree t)` mengembalikan sebuah `int`. Tak ada `int` yang berarti "tidak ada maksimum". Jadi apa yang seharusnya dilakukan `max(kosong)`? Ada tiga jawaban yang dapat dipertahankan, dan Anda sebaiknya menyebutkan yang mana yang Anda pilih:

1. **Lemparkan eksepsi.** `throw invalid_argument("max() of an empty tree is undefined")`. Inilah yang dilakukan **Kode 8.6**, dan inilah pilihan yang jujur: kontrak fungsinya tidak terdefinisi pada pohon kosong, jadi ia seharusnya mengatakannya dengan lantang.
2. **Kembalikan `INT_MIN`** sebagai penjaga. Murah, dan keliru karena alasan yang sama dengan §8.6.7: `INT_MIN` adalah nilai pohon yang sah, sehingga pemanggilnya tidak dapat membedakan "kosong" dari "pohonnya memuat int terkecil".
3. **Dokumentasikan sebuah prasyarat** — "pemanggil harus memastikan `t` tak-kosong" — lalu biarkan perilakunya tak terdefinisi. Dapat diterima jika dituliskan; kegagalan diam-diam tidak.

Mana pun dari ketiganya dapat dipertahankan asalkan disertai satu kalimat. Yang tidak dapat dipertahankan adalah kode yang mengakses pohon kosong tanpa komentar apa pun, sebab itu tak terbedakan dari tidak memikirkannya sama sekali.

8.8.8 Belokan keliru yang sering terjadi pada soal ini

1. **Berekursi ke kedua upapohonnya.** `return max(t->value, max(left), max(right))`. Keluarannya benar, caranya dikesampingkan secara tersurat, dan ia mengunjungi seluruh n simpul. §8.8.4 mengukurnya.
2. **Memakai kasus dasar "pohonnya kosong" alih-alih "subpohon kanannya kosong".** Ini memberikan `if (isEmpty(t)) return INT_MIN; return std::max(t->value, max(t->right));` — yang berjalan, tetapi memanggil dirinya sendiri sekali lebih banyak daripada yang perlu dan memunculkan kembali persoalan penjaganya. Pilihlah kasus dasar yang bersih.
3. **Berjalan ke kiri.** Salah ketik yang tertukar. Tertangkap dalam lima detik dengan menguji pohon tiga simpul 10{5,15}, yang maksimumnya 15.
4. **Mengklaim $O(\lg n)$ tanpa pembenaran.** §8.8.5 memperlihatkan cara menyatakan batasnya agar ia benar.

5. **Mengandaikan pohonnya adalah pohon AVL padahal soalnya hanya berkata "BST"**, lalu tidak sanggup menjawab pertanyaan lanjutan *"bagaimana jika ia sebuah rantai?"*

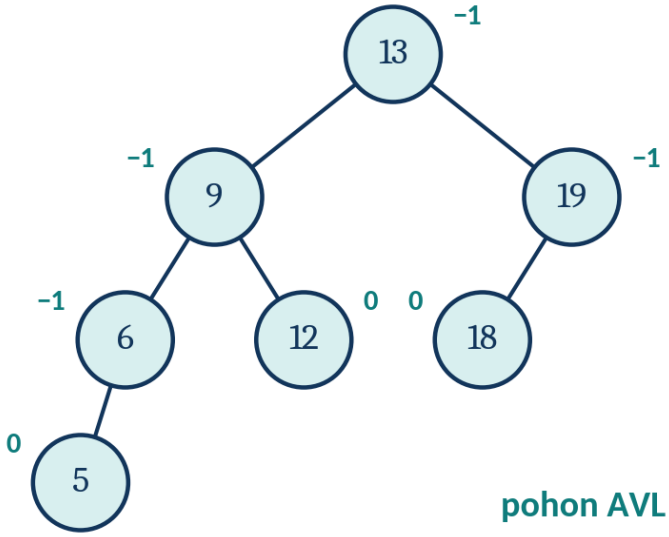
8.9 Soal 8.6 — `isHeightBalanced`: Syarat AVL dalam Satu Lintasan

8.9.1 Soalnya

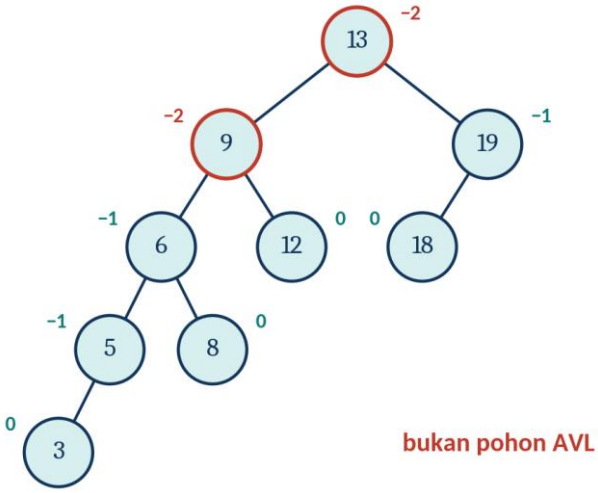
Tulislah fungsi rekursif `bool isHeightBalanced(MyTree t)` yang memeriksa apakah pohon `t` seimbang-tinggi — apakah ia memenuhi syarat AVL. Usahakan hanya satu lintasan atas pohonnya.

8.9.2 Apa itu syarat AVL

Bab 6 menyatakannya: *untuk tiap simpul, selisih tinggi antara cabang kiri dan cabang kanannya paling banyak 1*. **Gambar 8.3** memperlihatkan pohon yang memenuhinya dan **Gambar 8.4** pohon yang tidak.



Gambar 8.3 Pohon yang **merupakan** pohon AVL. Kedua subpohon tiap simpulnya berselisih tinggi paling banyak satu.



Gambar 8.4 Pohon yang **bukan** pohon AVL. Menggantungkan simpul 3 di bawah 5 mendorong dua faktor keseimbangan, milik 9 dan 13, menjadi −2, di luar jangkauan yang sah.

Dalam notasi kita, dari §8.5:

DEFINISI (seimbang-tinggi). Sebuah pohon t disebut **seimbang-tinggi** — sebuah pohon AVL — bila

$$|\mathrm{bf}(x)| \leq 1 \quad \text{untuk setiap simpul } x \text{ dari } t, \tag{8.9}$$

dengan $\mathrm{bf}(x) = h(\mathrm{right}(x)) - h(\mathrm{left}(x))$ dan pohon kosong bertinggi -1. Setara dengan itu: $\mathrm{bf}(x) \in \{-1, 0, +1\}$ di tiap simpul. Pohon kosong seimbang-tinggi secara hampa.

Kerjakanlah **Gambar 8.4** dengan tangan, memakai pohon yang diuraikan Bab 6 §6.14: akar 13; anak kiri 9 yang anak-anaknya 6 dan 12; simpul 6 dengan anak kiri 5 dan anak kanan 8; simpul baru 3 digantungkan di bawah 5; anak kanan 19 dengan anak kiri 18.

- Simpul 3: sebuah daun. $h = 0$, $\mathrm{bf} = -1 - (-1) = 0$. ✓
- Simpul 5: subpohon kirinya daun 3 ($h = 0$), subpohon kanannya kosong ($h = -1$). $\mathrm{bf} = -1 - 0 = -1$. ✓ Dan $h(5) = 1 + \max(0, -1) = 1$.
- Simpul 8: sebuah daun. $h = 0$, $\mathrm{bf} = 0$. ✓
- Simpul 6: subpohon kirinya berakar 5 ($h = 1$), subpohon kanannya daun 8 ($h = 0$). $\mathrm{bf} = 0 - 1 = -1$. ✓ Dan $h(6) = 1 + \max(1, 0) = 2$.
- Simpul 12: sebuah daun. $h = 0$, $\mathrm{bf} = 0$. ✓
- Simpul 9: subpohon kirinya berakar 6 ($h = 2$), subpohon kanannya daun 12 ($h = 0$). $\mathrm{bf} = 0 - 2 = \mathbf{-2}$. ✗ **Di luar jangkauan.**
- Simpul 13: subpohon kirinya berakar di 9 ($h = 3$), subpohon kanannya berakar di 19 ($h = 1$). $\mathrm{bf} = 1 - 3 = \mathbf{-2}$. ✗ **Juga di luar jangkauan.**

Jadi pohon itu tidak seimbang-tinggi, dan simpul pertama yang bersalah bila ditemui dari bawah adalah 9 — bukan 6, seperti yang mungkin disangka dari sekilas melihat cabang kiri yang panjang: simpul 6 masih seimbang karena anak kanannya, 8, menopangnya. Peganglah angka itu; programnya akan menemukannya.

8.9.3 Jawaban yang kentara, dan berapa biayanya

Definisinya, Persamaan (8.9), berkata "untuk tiap simpul, selisih tinggi kedua upapohonnya". Jadi kode yang kentara menuliskan persis itu:

```
int heightSlow(MyTree t) {
    if (isEmpty(t)) return -1;
    return 1 + max(heightSlow(t->left), heightSlow(t->right));
}
bool isHeightBalancedSlow(MyTree t) {
    if (isEmpty(t)) return true;
    int hl = heightSlow(t->left), hr = heightSlow(t->right);
    if (abs(hl - hr) > 1) return false;
    return isHeightBalancedSlow(t->left) && isHeightBalancedSlow(t->right);
}
```

Kode ini **benar**. Ia lulus pada setiap uji kebenaran. Yang tidak dipenuhinya adalah permintaan akan satu lintasan, dan kegagalan itulah bagian yang menarik, sebab kode ini menghitung ulang tinggi yang sama berkali-kali. Di akarnya ia menghitung tinggi kedua upapohonnya, yang bersama-sama memeriksa $n - 1$ simpul. Lalu ia berekursi ke subpohon kiri dan menghitung tinggi kedua subpohon *itu* — memeriksa simpul-simpul yang sudah diperiksanya. Dan seterusnya, sampai ke bawah.

Mari kita cacah dengan persis. Di tiap simpul x yang dicapai rekursinya, kedua panggilan `heightSlow` memeriksa setiap simpul yang benar-benar berada di bawah x — yaitu $\mathrm{size}(x) - 1$

simpul — ditambah satu pemeriksaan untuk x sendiri pada pemeriksaan keseimbangannya. Jadi jika pohonnya lulus dan setiap simpul tercapai, jumlah keseluruhan pemeriksaan simpulnya adalah

$$W(T) = \sum_{x \in T} [(\text{size}(x) - 1) + 1] = \sum_{x \in T} \text{size}\,(x).$$

Jumlah itu punya bentuk tertutup untuk pohon **sempurna**. Misalkan pohonnya sempurna bertinggi k, sehingga ada $n = 2^{k+1} - 1$ simpul, dengan 2^{ℓ} simpul di aras ℓ untuk $\ell = 0,1,\ldots,k$, dan tiap simpul di aras ℓ mengakari subpohon berukuran $2^{k-\ell+1} - 1$. Maka

$$\begin{aligned}\sum_{x \in T} \text{size}\,(x) = \sum_{\ell=0}^{k} 2^{\ell}\,(2^{k-\ell+1} - 1) &= \sum_{\ell=0}^{k} (2^{k+1} - 2^{\ell}) \\ &= (k+1)2^{k+1} - (2^{k+1} - 1).\end{aligned}$$

Kini substitusikan $2^{k+1} = n + 1$ dan $k + 1 = \lg(n + 1)$:

$$W = (n+1)\lg(n+1) - n. \tag{8.10}$$

Periksalah pada kasus terkecilnya. Untuk $k = 2, n = 7$: $W = 8 \times 3 - 7 = 17$. Untuk $k = 3, n = 15$: $W = 16 \times 4 - 15 = 49$. Untuk $n = 127$: $W = 128 \times 7 - 127 = 769$. Program pada **Kode 8.7** mencetak 17, 49, dan 769 persis untuk pohon-pohon itu, dan **Kode 8.14** di §8.23 memeriksa rumusnya terhadap programnya untuk setiap pohon sempurna sampai $n = 2047$.

Jadi cara yang kentara itu berbiaya $\Theta(n\lg n)$ pada pohon yang seimbang. Cara satu-lintasan di bawah berbiaya $\Theta(n)$. Penghematannya sebesar faktor $\lg n$ — untuk $n = 10^6$, faktor dua puluh.

> **PERINGATAN: SEBUAH KLAIM YANG AKAN ANDA TEMUI DI MANA-MANA, DAN IA TIDAK BENAR DI SINI.** Sangat lazim dinyatakan bahwa pemeriksaan keseimbangan yang naif berbiaya $\Theta(n^2)$. Ia tidak — tidak untuk predikat ini. Kasus kuadratnya membutuhkan $\Theta(n)$ simpul yang masing-masing berbiaya $\Theta(n)$, dan itu membutuhkan pohon yang tinggi; padahal pohon yang *lulus* pemeriksaan keseimbangan punya $h = O(\lg n)$, dan pohon yang *gagal* membuat pemeriksaannya kembali lebih awal. Kasus terburuk terukur untuk `isHeightBalancedSlow` adalah $\Theta(n\lg n)$, dan penurunan di atas memberikan konstanta persisnya. Tempat $\Theta(n^2)$ sungguh-sungguh muncul adalah pada rutin yang berkerabat dekat, yang melabeli *setiap* simpul dengan tingginya dengan memanggil `heightSlow` sekali per simpul, tanpa keluar lebih awal: itu berbiaya $\sum_x \text{size}\,(x)$ atas pohon yang mungkin tinggi, yang pada rantai n simpul menjadi $n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2} = \Theta(n^2)$. **Kode 8.7** mengukur itu juga, dan memperoleh 8256 untuk $n = 128$ — persis $\frac{128 \times 129}{2}$.

8.9.4 Jawabannya: satu lintasan, dengan penjaga

Pemborosannya punya sebab yang kentara: `isHeightBalancedSlow` membuang setiap tinggi yang dihitungnya. Pelajaran Bab 7 adalah *jangan pernah menghitung ulang apa yang sudah Anda ketahui*. Terapkanlah.

Langkah 1 — sadari bahwa rekursinya sudah mengunjungi tiap simpul sekali. Jika tiap kunjungan dapat *mengembalikan* tinggi upapohonnya, induknya memperoleh kedua tingginya secara gratis.

Langkah 2 — sadari bahwa induknya juga membutuhkan jawaban ya/tidak. Jadi satu panggilan rekursif harus mengantarkan dua hal: tingginya, dan apakah upapohonnya seimbang.

Langkah 3 — kemas keduanya ke dalam satu bilangan bulat. Tinggi selalu -1 atau lebih. Jadi nilai apa pun di bawah -1 bukan tinggi yang mungkin dan dapat dipakai untuk berarti "tidak seimbang". Pilihlah -2.

Langkah 4 — tuliskan fungsi penunjangnya.

$$\mathrm{ch}(\emptyset) = -1; \qquad \mathrm{ch}((v, L, R)) = \begin{cases} -2 & \text{jika } \mathrm{ch}(L) = \text{-} \\ -2 & \text{jika } \mathrm{ch}(R) = \text{-} \\ -2 & \text{jika } |\mathrm{ch}(R) - \\ 1 + \max(\mathrm{ch}(L), \mathrm{ch}(R)) & \text{jika tidak.} \end{cases} \tag{8.11}$$

Langkah 5 — jawabannya adalah isHeightBalanced$(t) \Leftrightarrow \mathrm{ch}(t) \neq -2$, dengan ch seperti pada Persamaan (8.11).

Langkah 6 — kebenarannya. Dua arah. Jika pohonnya seimbang, maka tak ada simpul yang pernah memicu -2, sehingga tiap panggilan mengembalikan $1 +$ max tinggi sejati anak-anaknya, yang dengan induksi atas tingginya adalah tinggi sejati simpul itu; jadi $\mathrm{ch}(t) = h(t) \neq -2$. Jika ada simpul yang tak seimbang, ambillah simpul semacam itu yang *terendah*, sebut x: setiap simpul di bawah x seimbang, sehingga panggilan di bawah x mengembalikan tinggi sejati, sehingga uji di x melihat tinggi yang sejati, melihat selisih yang melampaui 1, lalu mengembalikan -2; dan -2 kemudian merambat ke atas lewat dua kasus pertamanya sampai ke akarnya.

Langkah 7 — biayanya. Tiap simpul dimasuki tepat sekali dan mengerjakan kerja $\Theta(1)$, jadi waktunya $\Theta(n)$ pada kasus terburuk, dan kurang dari itu ketika keluar-awalnya menyala. Ruangnya $O(h)$ untuk tumpukannya. Ini optimal: Anda tidak dapat memutuskan apakah *setiap* simpul seimbang tanpa, pada kasus terburuk, melihat setiap simpul.

> **WAWASAN KUNCI: POLA PENJAGA, LAYAK DIINGAT SEUMUR HIDUP.** Ketika sebuah rekursi harus mengembalikan sebuah nilai *sekaligus* sebuah penanda kegagalan, dan tipe nilainya punya anggota yang tak terpakai, pakailah anggota itu sebagai penandanya alih-alih mengembalikan sepasang nilai. Tinggi selalu ≥ -1, jadi -2 gratis. Ukuran selalu ≥ 0, jadi -1 gratis. Ini peranti yang sama dengan mengembalikan -1 dari pencarian yang tidak menemukan apa-apa. Ia muncul lagi di dalam `computeBlackHeight` pada Soal 8.13 di §8.14, tempat -1 menjadi penandanya sebab tinggi-hitam selalu ≥ 0 — teknik yang sama pada invarian yang berbeda, dan itulah tanda paling pasti bahwa ia sebuah teknik, bukan sekadar akal-akalan.

8.9.5 Programnya

Kode 8.7 memuat cara yang lambat, cara satu-lintasan, rutin pelabelan tinggi yang memang sungguh kuadrat, sebuah diagnostik yang menyebut simpul yang bersalah, dan perbandingan terukur pada berbagai bentuk pohon.

Kode 8.7 Soal 8.6 selengkapnya: cara yang boros, cara satu-lintasan dengan penjaga, rutin pelabelan tinggi yang memang kuadrat, dan perbandingan terukur atas sebelas bentuk pohon.

```
// isbalanced.cpp -- Soal 8.6: isHeightBalanced().
#include <iostream>
#include <iomanip>
#include <string>
#include <algorithm>
using namespace std;

struct MyTreeNode;
using MyTree = MyTreeNode*;
struct MyTreeNode { int value; MyTree left, right; };   // TANPA tinggi tersimpan:
                                                        // pertanyaannya justru apakah
                                                        // kita bisa menghitungnya murah
MyTree emptyTree()       { return nullptr; }
bool   isEmpty(MyTree t) { return t == nullptr; }
MyTree node(int v, MyTree l, MyTree r) {
    MyTree t = new MyTreeNode; t->value = v; t->left = l; t->right = r; return t;
}
MyTree leaf(int v) { return node(v, emptyTree(), emptyTree()); }

long long work = 0;        // setiap pemeriksaan simpul, di mana pun, dicacah di sini

// ---- Jalan yang kentara tetapi boros: hitung ulang tinggi di setiap simpul. ----
int heightSlow(MyTree t) {
    if (isEmpty(t)) return -1;
    ++work;
    return 1 + max(heightSlow(t->left), heightSlow(t->right));
}
bool isHeightBalancedSlow(MyTree t) {
    if (isEmpty(t)) return true;
    ++work;
    int hl = heightSlow(t->left), hr = heightSlow(t->right);
    if (abs(hl - hr) > 1) return false;
    return isHeightBalancedSlow(t->left) && isHeightBalancedSlow(t->right);
}

// ---- Jawabannya: satu lintasan. Kembalikan tinggi, atau -2 berarti "tak seimbang". ----
// -1 sudah terpakai (tinggi pohon kosong), jadi penanda kegagalan haruslah
// nilai yang tak pernah mungkin dimiliki tinggi yang sungguhan.
const int UNBALANCED = -2;
static int checkedHeight(MyTree t) {
    if (isEmpty(t)) return -1;                  // kasus dasar
    ++work;
    int hl = checkedHeight(t->left);
    if (hl == UNBALANCED) return UNBALANCED;    // menyerah lebih awal
    int hr = checkedHeight(t->right);
    if (hr == UNBALANCED) return UNBALANCED;
```

Kode 8.7 (Lanjutan) *Soal 8.6 selengkapnya: cara yang boros, cara satu-lintasan dengan penjaga, rutin pelabelan tinggi yang memang kuadrat, dan perbandingan terukur atas sebelas bentuk pohon.*

```
    if (abs(hl - hr) > 1)  return UNBALANCED;    // simpul inilah yang melanggar
invariannya
    return 1 + max(hl, hr);                      // jika tidak, tinggi yang sebenarnya
}
bool isHeightBalanced(MyTree t) { return checkedHeight(t) != UNBALANCED; }

// ---- Bentuk diagnostik: simpul mana yang melanggarnya, dan seberapa jauh. ----
static int auditHeight(MyTree t, string& why) {
    if (isEmpty(t)) return -1;
    int hl = auditHeight(t->left,  why); if (hl == UNBALANCED) return UNBALANCED;
    int hr = auditHeight(t->right, why); if (hr == UNBALANCED) return UNBALANCED;
    if (abs(hl - hr) > 1) {
        why = "node " + to_string(t->value) + ": left height " + to_string(hl)
            + ", right height " + to_string(hr) + ", bf = " + to_string(hr - hl);
        return UNBALANCED;
    }
    return 1 + max(hl, hr);
}

// ---- Pemborosan yang sama, disendirikan: labeli tiap simpul dengan tingginya
// ---- Lewat satu panggilan heightSlow() per simpul. Biaya = jumlah size(v) atas v.
void annotateHeights(MyTree t) {
    if (isEmpty(t)) return;
    heightSlow(t);                       // menghitung ulang seluruh subpohon, setiap kali
    annotateHeights(t->left);
    annotateHeights(t->right);
}

int countNodes(MyTree t) { return isEmpty(t) ? 0 : 1 + countNodes(t->left) + countNodes(t-
>right); }

void report(const char* name, MyTree t) {
    string why = "-";
    bool ok = auditHeight(t, why) != UNBALANCED;
    work = 0; bool s = isHeightBalancedSlow(t);  long long wSlow = work;
    work = 0; bool f = isHeightBalanced(t);      long long wFast = work;
    work = 0; annotateHeights(t);                long long wAnn  = work;
    cout << left << setw(26) << name
         << " n=" << setw(4) << countNodes(t)
         << " slow:" << (s ? "yes" : "no ") << " w=" << setw(6) << wSlow
         << " fast:" << (f ? "yes" : "no ") << " w=" << setw(4) << wFast
         << " annotate w=" << setw(6) << wAnn << "\n";
    if (!ok) cout << string(26, ' ') << " first fault: " << why << "\n";
}
```

Kode 8.7 (Lanjutan) *Soal 8.6 selengkapnya: cara yang boros, cara satu-lintasan dengan penjaga, rutin pelabelan tinggi yang memang kuadrat, dan perbandingan terukur atas sebelas bentuk pohon.*

```
// BST yang seimbang sempurna pada lo..hi.
MyTree perfect(int lo, int hi) {
    if (lo > hi) return emptyTree();
    int mid = lo + (hi - lo) / 2;
    return node(mid, perfect(lo, mid - 1), perfect(mid + 1, hi));
}
// Rantai condong kanan 1 -> 2 -> ... -> n.
MyTree chain(int n) {
    MyTree t = emptyTree();
    for (int v = n; v >= 1; --v) t = node(v, emptyTree(), t);
    return t;
}

int main() {
    // Pohon AVL pada gambar: akar 13, kiri 9{6{5,.},12}, kanan 19{18,.}
    MyTree avl = node(13, node(9, node(6, leaf(5), emptyTree()), leaf(12)),
                          node(19, leaf(18), emptyTree()));
    // Simpul 8 di bawah 6, dan 3 digantung di bawah 5 -- gambar "bukan AVL".
    MyTree notAvl = node(13, node(9, node(6, node(5, leaf(3), emptyTree()), leaf(8)),
leaf(12)),
                             node(19, leaf(18), emptyTree()));

    cout << "legend: w = node inspections;  slow = recompute heights;  "
            "fast = one pass with a sentinel\n";
    report("the AVL example",          avl);
    report("the non-AVL example",      notAvl);
    report("empty tree",               emptyTree());
    report("single node",              leaf(7));
    for (int n : {7, 15, 31, 63, 127}) {
        string nm = "perfect tree, n = " + to_string(n);
        report(nm.c_str(), perfect(1, n));
    }
    for (int n : {8, 16, 32, 64, 128}) {
        string nm = "chain, n = " + to_string(n);
        report(nm.c_str(), chain(n));
    }
    return 0;
}
```

Keluarannya:

```
legend: w = node inspections;  slow = recompute heights;  fast = one pass with a sentinel
the AVL example            n=7    slow:yes w=18     fast:yes w=7    annotate w=18
the non-AVL example        n=9    slow:no  w=9      fast:no  w=7    annotate w=27
                           first fault: node 9: left height 2, right height 0, bf = -2
empty tree                 n=0    slow:yes w=0      fast:yes w=0    annotate w=0
single node                n=1    slow:yes w=1      fast:yes w=1    annotate w=1
perfect tree, n = 7        n=7    slow:yes w=17     fast:yes w=7    annotate w=17
perfect tree, n = 15       n=15   slow:yes w=49     fast:yes w=15   annotate w=49
perfect tree, n = 31       n=31   slow:yes w=129    fast:yes w=31   annotate w=129
perfect tree, n = 63       n=63   slow:yes w=321    fast:yes w=63   annotate w=321
perfect tree, n = 127      n=127  slow:yes w=769    fast:yes w=127  annotate w=769
chain, n = 8               n=8    slow:no  w=8      fast:no  w=8    annotate w=36
                           first fault: node 6: left height -1, right height 1, bf = 2
chain, n = 16              n=16   slow:no  w=16     fast:no  w=16   annotate w=136
                           first fault: node 14: left height -1, right height 1, bf = 2
chain, n = 32              n=32   slow:no  w=32     fast:no  w=32   annotate w=528
                           first fault: node 30: left height -1, right height 1, bf = 2
chain, n = 64              n=64   slow:no  w=64     fast:no  w=64   annotate w=2080
                           first fault: node 62: left height -1, right height 1, bf = 2
chain, n = 128             n=128  slow:no  w=128    fast:no  w=128  annotate w=8256
                           first fault: node 126: left height -1, right height 1, bf = 2
```

Baris ke-3 memastikan hitungan tangan §8.9.2: cacat pertama pada pohon **Gambar 8.4** ada di **simpul 9**, dengan tinggi kiri 2, tinggi kanan 0, dan faktor keseimbangan -2. Itu persis yang kita hitung dengan tangan.

Tabel 8.6 mengumpulkan angka terukurnya berhadapan dengan bentuk tertutup yang diturunkan di atas.

Tabel 8.6 Pemeriksaan simpul terukur berhadapan dengan bentuk tertutupnya. Pemeriksaan keseimbangan yang naif cocok persis dengan $(n+1)\lg(n+1)-n$ pada pohon sempurna, cara satu-lintasan cocok dengan n, dan rutin pelabelan tingginya cocok persis dengan $n(n+1)/2$ pada rantai.

Pohon	n	Lambat, terukur	$(n+1)\lg(n+1)-n$	Cepat, terukur	Pelabelan, terukur	$n(n+1)/2$
sempurna	7	17	17	7	17	—
sempurna	15	49	49	15	49	—
sempurna	31	129	129	31	129	—
sempurna	63	321	321	63	321	—
sempurna	127	769	769	127	769	—
rantai	16	16	—	16	136	136
rantai	64	64	—	64	2080	2080
rantai	128	128	—	128	8256	8256

Setiap entri pada kedua kolom "terukur" sama dengan bentuk tertutupnya. Itulah arti memverifikasi sebuah analisis alih-alih menyatakannya, dan tabel seperti ini adalah bukti sekuat-kuatnya bahwa sebuah klaim kompleksitas itu benar.

8.9.6 Rekurensinya, diselesaikan dengan Teorema Master

Demi kelengkapan, dan karena Bab 3 membangun perkakasnya persis untuk ini, berikut hasil yang sama dari rekurensinya.

Untuk cara **satu-lintasan** pada pohon yang tiap upapohonnya memuat kira-kira separuh simpulnya:

$$T(n) = 2T(n/2) + \Theta(1). \tag{8.12}$$

Cocokkan Persamaan (8.12) dengan bentuk Teorema Master $T(n) = aT(n/b) + f(n)$: $a = 2$, $b = 2$, $f(n) = \Theta(1)$. Maka $\log_b a = \log_2 2 = 1$, sehingga $n^{\log_b a} = n$. Bandingkan $f(n) = \Theta(1)$ dengan n^1: kita butuh $f(n) = O(n^{1-\varepsilon})$ untuk suatu $\varepsilon > 0$, dan $\varepsilon = 1$ berhasil, sebab $\Theta(1) = O(n^0)$. Itu **kasus 1**, jadi

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n).$$

Untuk cara yang **lambat**, kerja di sebuah simpul tidaklah konstan: ia berupa dua penghitungan tinggi, yang berbiaya $\Theta(n)$ pada simpul yang memuat n simpul. Jadi

$$T(n) = 2T(n/2) + \Theta(n), \tag{8.13}$$

dan Persamaan (8.13) adalah $a = 2$, $b = 2$, $f(n) = \Theta(n) = \Theta(n^{\log_b a})$ — **kasus 2** — yang memberikan

$$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n \lg n),$$

sesuai dengan cacahan persis Persamaan (8.10) yang diturunkan di §8.9.3 dan diukur pada **Tabel 8.6**. Dua jalan yang berdiri sendiri, satu jawaban: begitulah cara Anda tahu sebuah analisis itu benar.

8.9.7 Belokan keliru yang sering terjadi pada soal ini

1. **Memeriksa akarnya saja.** `return abs(bf(t)) <= 1;`. Definisinya berkata *setiap* simpul. Ini analog persis dengan pemeriksaan lokal Soal 8.3. Contoh sanggahannya kecil: pohon $10\{5\{3\{1,.\},.\},20\}$ seimbang di akarnya dan tidak seimbang di simpul 5.
2. **Mengembalikan tinggi padahal yang diminta jawaban ya-atau-tidak**, atau sebaliknya. Jagalah rutin penjaganya tetap `private` dan biarkan `isHeightBalanced` yang publik melakukan perbandingan dengan -2, persis seperti **Kode 8.7**.
3. **Memilih -1 sebagai penanda kegagalan.** -1 adalah tinggi pohon kosong. Memakainya membuat setiap subpohon kosong tampak tak seimbang, dan kodenya lalu melaporkan bahwa pohon satu simpul bukan pohon AVL. Ujilah simpul tunggalnya; ia harus berkata ya.
4. **Melupakan `abs`.** Menulis `hr - hl > 1` menangkap cacat berat-kanan dan melewatkan yang berat-kiri. Contoh sanggahan pada **Gambar 8.4** berat ke kiri, jadi kutu ini gagal pada pohon gambar itu sendiri.
5. **Memercayai medan tinggi tersimpan alih-alih menghitungnya.** Jika pohonnya dibangun dengan tangan atau oleh rutin yang berkutu, tinggi tersimpannya mungkin keliru, dan pemeriksa yang membacanya sama saja tidak memeriksa apa pun. Itulah sebabnya **Kode 8.7** dengan sengaja mendeklarasikan tipe simpulnya *tanpa* medan tinggi. §8.19 memperlihatkan apa yang terjadi di produksi ketika sebuah pemeriksa memercayai medan tersimpan.

8.10 Soal 8.7 — `insertHB`: Penyisipan yang Menjaga Pohon Tetap Seimbang

8.10.1 Soalnya

Pohon AVL adalah pohon *seimbang-tinggi* (*height-balanced*, HB). Tulislah fungsi rekursif `MyTree insertHB(int n, MyTree t)` yang menyisipkan `n` ke dalam `t` dan mengembalikan pohon hasilnya, dengan tetap menjaga syarat AVL.

Petunjuk: `insertHB` sebaiknya memakai dua fungsi penunjang, `rebalanceForLeft` dan `rebalanceForRight` (Soal 8.8 dan 8.9). Kedua penunjang yang sama akan melayani penghapusan, `deleteHB`, pada Soal 8.10. Kontrak keduanya adalah:

`rebalanceForLeft(t)` dipanggil ketika subpohon kiri `t` mungkin baru bertambah tinggi satu takik. Jika ia memang lebih tinggi daripada subpohon kanan sebanyak dua takik, fungsi itu mengembalikan versi `t` yang seimbang-tinggi, dengan memakai rotasi tunggal atau ganda. Kedua subpohon `t` diandaikan sudah seimbang-tinggi, dan tak ada upaya untuk menyeimbangkan ulang keduanya. `rebalanceForRight(t)` mengerjakan hal yang sama untuk subpohon kanan.

Soal 8.7, 8.8, dan 8.9 adalah satu rancangan dan sebaiknya dibaca bersama-sama. Bacalah kalimat terakhir kontraknya dua kali: *kedua subpohon `t` diandaikan sudah seimbang-tinggi, dan tak ada upaya untuk menyeimbangkan ulang keduanya.*

8.10.2 Kontraknya, dinyatakan dengan persis

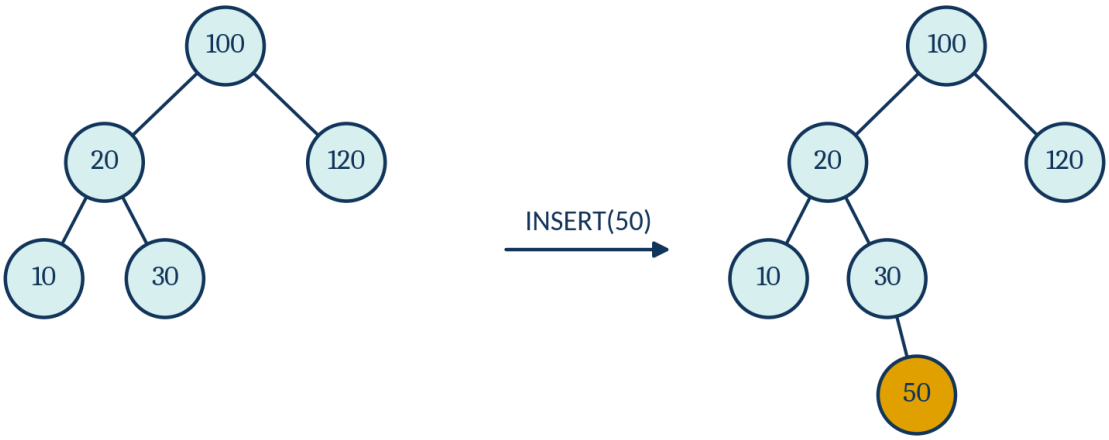
Petunjuknya memberi `rebalanceForLeft` sebuah kontrak, dan kontrak itulah isi yang dapat diujikan. Ditulis selengkapnya:

> **DEFINISI (KONTRAK `rebalanceForLeft(t)`). Prasyarat.** t tak-kosong. Kedua subpohon t sendiri sudah seimbang-tinggi. Subpohon kirinya *mungkin* baru saja bertambah tinggi satu takik, sehingga $\mathrm{bf}(t) \in \{-2, -1, 0, +1\}$ — segalanya kecuali $+2$. **Pascasyarat.** Nilai kembaliannya adalah pohon seimbang-tinggi yang memuat persis nilai-nilai yang sama dengan t, dalam urutan terurut yang sama. **Biaya.** $O(1)$: paling banyak dua rotasi, masing-masing sejumlah tetap pembangunan `node`. Tak ada subpohon yang ditelusuri. `rebalanceForRight(t)` **adalah bayangan cerminnya**, dengan $\mathrm{bf}(t) \in \{-1, 0, +1, +2\}$.

Frasa "no effort is made to rebalance them" — tidak ada upaya untuk menyeimbangkan ulang kedua upapohonnya — adalah izin yang memungkinkan biaya $O(1)$ itu. Fungsinya diperbolehkan *mengandaikan* anak-anaknya sudah baik-baik saja, sehingga ia hanya perlu membetulkan satu simpul yang ada di hadapannya.

8.10.3 Menurunkan `insertHB`

Langkah 1 — mulailah dari penyisipan BST biasa. Bab 6 §6.10 sudah memberikannya, dan **Gambar 8.5** memperlihatkannya: turunlah sambil membandingkan, sampai Anda mencapai slot kosong, lalu letakkan simpul barunya di situ sebagai daun.



Gambar 8.5 Penyisipan ke dalam BST biasa: elemen baru selalu disisipkan sebagai daun. Penurunannya adalah pencariannya, dan pencariannya adalah penempatannya.

Dalam gaya fungsional **Kode 8.3**, "letakkan di situ" menjadi "bangun ulang lintasannya":

$$\mathrm{insert}(n, \varnothing) = (n, \varnothing, \varnothing)$$

$$\mathrm{insert}(n, (v, L, R)) = \begin{cases} (v, \mathrm{insert}(n, L), R) & n < v \\ (v, L, \mathrm{insert}(n, R)) & n > v \\ (v, L, R) & n = v \quad \text{(sudah ada: tak ada }\mathrm{I} \end{cases}$$

Langkah 2 — tanyakan apa yang bisa keliru. Daun barunya bertinggi 0, dan karena itu subpohon tempat ia disisipkan mungkin satu lebih tinggi daripada sebelumnya. Setiap leluhur di lintasan dari daun baru itu ke akarnya melihat salah satu dari kedua upapohonnya mungkin satu takik lebih tinggi. Tak ada lagi yang berubah di dalam pohonnya.

Langkah 3 — betulkan persis simpul-simpul itu, dalam perjalanan naik. Setelah `insert(n, L)` kembali, simpul v sedang dibangun ulang, dan subpohon *kirinya* mungkin bertambah tinggi. Itu persis prasyarat `rebalanceForLeft`. Begitu pula di kanan.

$$\text{insertHB}(n, \varnothing) = (n, \varnothing, \varnothing)$$

$$\text{insertHB}(n, (v, L, R)) = \begin{cases} \text{rebalanceForLeft}(v, \text{insertHB}(n, L), R) & n < \\ \text{rebalanceForRight}(v, L, \text{insertHB}(n, R)) & n > \\ (v, L, R) & n = \end{cases} \tag{8.14}$$

Itulah penyelesaiannya, Persamaan (8.14), dan panjangnya sembilan baris kode. Perhatikan betapa sedikit isinya begitu kedua penyeimbang ulangnya diandaikan ada: `insertHB` adalah penyisipan BST biasa dengan satu panggilan fungsi dibungkuskan di sekeliling tiap simpul yang dibangun ulang.

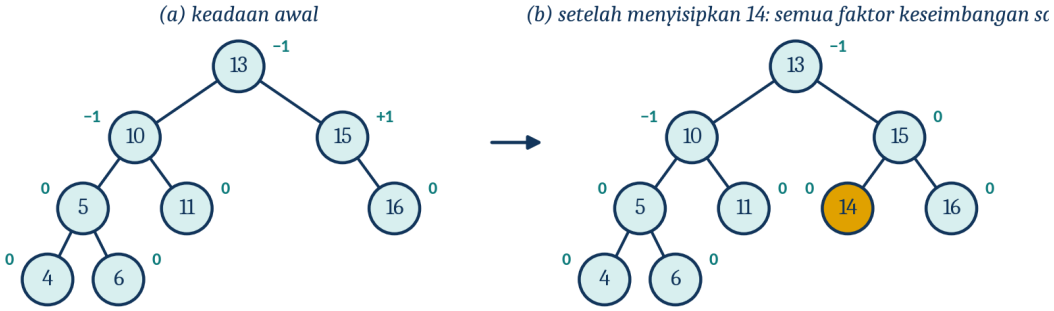
Langkah 4 — mengapa ini melestarikan invariannya? Dengan induksi atas rekursinya, dari bawah ke atas. Kasus dasarnya mengembalikan satu daun, yang seimbang-tinggi. Di simpul dalam, panggilan rekursifnya mengembalikan subpohon yang seimbang-tinggi (hipotesis induksi) yang tingginya tidak berubah atau satu lebih besar daripada sebelumnya. Subpohon yang lain tak tersentuh dan tadinya sudah seimbang. Jadi prasyarat penyeimbang ulangnya terpenuhi, dan menurut pascasyarat penyeimbang ulangnya simpul itu kembali dalam keadaan seimbang. Karena itu akarnya kembali dalam keadaan seimbang.

Langkah 5 — biayanya. Rekursinya mengikuti satu lintasan akar-ke-daun: paling banyak $h + 1$ aras. Tiap aras mengerjakan kerja $\Theta(1)$ ditambah satu penyeimbangan ulang $O(1)$. Jadi penyisipannya berbiaya $O(h) = O(\lg n)$ pada pohon AVL. Banyaknya *rotasi* paling banyak 2 untuk seluruh penyisipannya — fakta yang dibuktikan Bab 6 §6.20.2, dan yang diperagakan keluaran di bawah.

> **WAWASAN KUNCI: MENGAPA TANDA TANGANNYA MENGEMBALIKAN SEBUAH `MyTree`.** Mahasiswa yang baru pertama kali berjumpa `insertHB(int n, MyTree t)` sering bertanya mengapa ia tidak sekadar mengubah `t` lalu mengembalikan `void`. Sebab sebuah penyeimbangan ulang dapat mengubah simpul mana yang menjadi *akar* upapohonnya. Pemanggilnya harus diberi tahu akar barunya, dan mengembalikannya adalah cara yang paling bersih. Inilah alasan yang sama mengapa `rebalanceForLeft` mengembalikan sebuah `MyTree`: ia boleh menyerahkan kembali simpul yang berbeda dari yang diberikan kepadanya.

8.10.4 Sekilas pertama mesinnya bekerja

Gambar 8.6 mengerjakan satu penyisipan dengan tangan: menyisipkan 14 ke dalam pohon delapan simpul §8.5. Tak perlu penyeimbangan ulang, sebab setiap faktor keseimbangan tetap berada di dalam $\{-1, 0, +1\}$ — sebagaimana ditunjukkan hitungan Bab 6 §6.18.1 dan sebagaimana dipastikan program di bawah.



Gambar 8.6 Menyisipkan 14 ke dalam pohon delapan simpul. Penyisipannya terjadi, faktor keseimbangannya dihitung ulang menaiki lintasannya, dan tak satu pun keluar dari jangkauan yang sah — sehingga `insertHB` melakukan nol rotasi. Inilah kasus yang lazim.

8.11 Soal 8.8 — `rebalanceForLeft`: Kasus LL dan LR

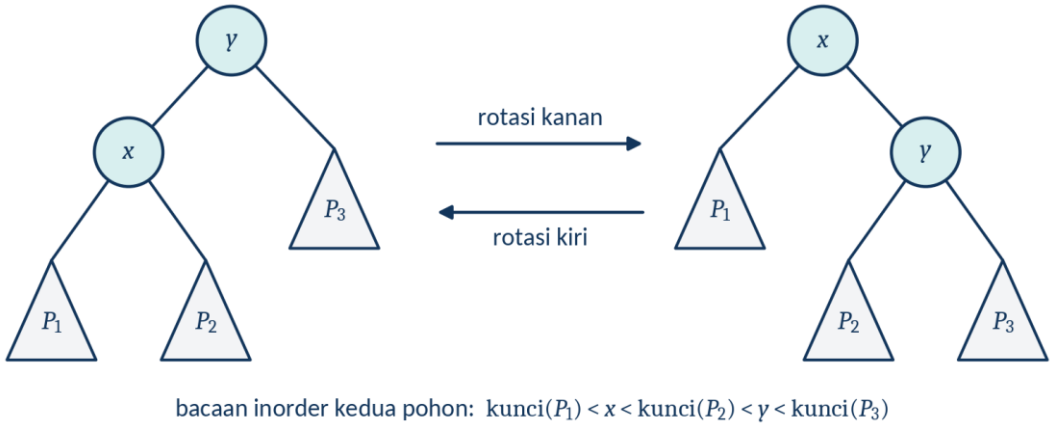
8.11.1 Soalnya

Tulislah `MyTree rebalanceForLeft(MyTree t)`, penunjang pertama Soal 8.7. Kontraknya, diulang dari §8.10.1:

`rebalanceForLeft(t)` dipanggil ketika subpohon kiri `t` mungkin baru bertambah tinggi satu takik. Jika ia memang lebih tinggi daripada subpohon kanan sebanyak dua takik, fungsi itu mengembalikan versi `t` yang seimbang-tinggi, dengan memakai rotasi tunggal atau ganda. Kedua subpohon `t` diandaikan sudah seimbang-tinggi, dan tak ada upaya untuk menyeimbangkan ulang keduanya.

8.11.2 Satu-satunya perkakas: rotasi

Rotasi adalah satu-satunya operasi yang mengubah bentuk sebuah pohon tanpa mengubah barisan inordernya. **Gambar 8.7** adalah gambarnya, dan ia membawa buktinya di bawahnya.



Gambar 8.7 Kedua rotasi. Pada kedua gambar, bacaan inordernya adalah P_1, x, P_2, y, P_3, sebagaimana dinyatakan baris di bawah gambarnya. Rotasi kanan mengubah gambar kiri menjadi gambar kanan; rotasi kiri adalah kebalikannya yang persis.

Bacalah relasi urutan yang ia nyatakan: $\text{elements}(P_1) < x < \text{elements}(P_2) < y < \text{elements}(P_3)$. Kedua pohon pada **Gambar 8.7** punya bacaan inorder yang sama itu, jadi keduanya adalah BST yang memuat nilai-nilai yang sama. Hanya bentuknya yang berbeda.

Dalam gaya fungsional, rotasi sama sekali bukan barisan penugasan penunjuk; ia satu ungkapan. **Rotasi kanan** terhadap t, yang mengangkat anak kiri t, adalah

$$\text{rotateRight}((y,(x,P_1,P_2),P_3)) = (x,P_1,(y,P_2,P_3)), \tag{8.15}$$

yang dalam kode berbunyi

```
MyTree rotateRight(MyTree t) {
    MyTree l = leftOf(t);
    return node(value(l), leftOf(l), node(value(t), rightOf(l), rightOf(t)));
}
```

Bacalah ia terhadap Persamaan (8.15). `value(l)` adalah x, simpul yang naik. `leftOf(l)` adalah P_1, yang tetap bersama x. `node(...)` bagian dalamnya adalah (y, P_2, P_3) yang baru: akar lama y, kini memegang P_2 di kiri dan subpohon kanan lamanya P_3 di kanan. Tingginya dibetulkan dengan sendirinya, sebab `node` menghitung tinggi tiap induk setelah anak-anaknya ada — panggilan bagian dalamnya berjalan sebelum panggilan bagian luarnya. Kutu urutan yang diperingatkan Bab 6 §6.16.1 tidak mungkin terjadi.

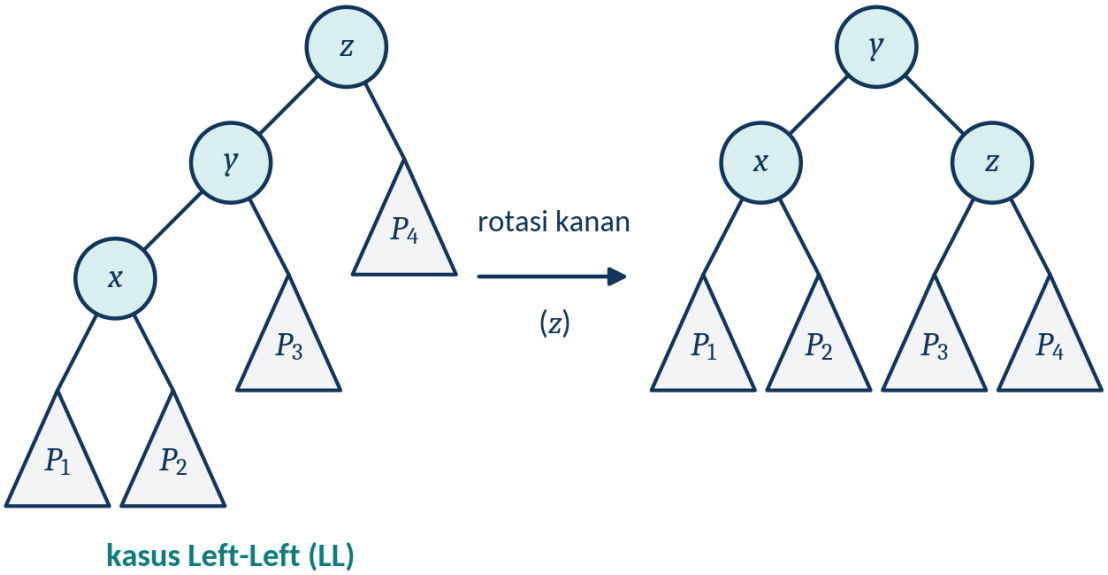
Rotasi kiri adalah cerminnya: $\text{rotateLeft}((x,P_1,(y,P_2,P_3))) = ((y,(x,P_1,P_2),P_3))$.

Tiap rotasi mengangkat satu subpohon sebesar satu aras dan menurunkan satu lainnya sebesar satu aras, dan berbiaya $\Theta(1)$: paling banyak tiga pembangunan `node`, tanpa penelusuran apa pun.

8.11.3 Kasus LL — satu rotasi kanan

Misalkan $\text{bf}(t) = -2$: subpohon kirinya dua takik lebih tinggi. Sebut L subpohon kirinya. Menurut prasyaratnya, L seimbang, jadi $\text{bf}(L) \in \{-1,0,+1\}$.

Kasus LL adalah $\text{bf}(L) \le 0$ — subpohon kirinya sendiri berat ke kiri atau rata. **Gambar 8.8** adalah gambarnya.



Gambar 8.8 Kasus Left-Left: z adalah simpul yang tak seimbang, y anak kirinya, x cucu kirinya. Satu rotasi kanan terhadap z mengangkat y ke puncak. P_1 sampai P_4 adalah keempat upapohonnya.

Kerjakanlah hitungan tingginya. Tulis k untuk tinggi subpohon kanan P_4 milik z. Maka $h(y) = k + 2$, sebab $\text{bf}(z) = -2$. Karena $\text{bf}(y) \le 0$, anak yang lebih tinggi di antara anak-anak y adalah anak kirinya x, jadi $h(x) = k + 1$ dan $h(P_3) \in \{k, k+1\}$; dengan $\text{bf}(y) \le 0$ kita peroleh $h(P_3) \le h(x)$, dan karena $h(y) = 1 + \max$ maka haruslah $h(x) = k + 1$.

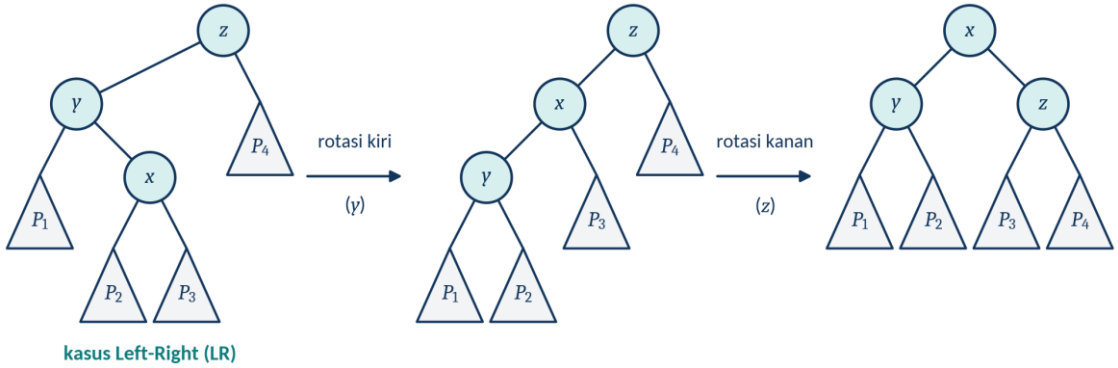
Setelah satu rotasi kanan terhadap z: y berada di puncak; anak kirinya adalah x bertinggi $k+1$; anak kanannya adalah simpul z yang baru, yang memegang P_3 (bertinggi k atau $k+1$) dan P_4 (bertinggi k).

- Jika $h(P_3) = k$: simpul z yang baru bertinggi $k+1$, dan $\text{bf}(z) = k - k = 0$. Lalu $\text{bf}(y) = (k+1) - (k+1) = 0$. Keduanya sah. ✓
- Jika $h(P_3) = k+1$ — yang hanya terjadi ketika $\text{bf}(y) = 0$: simpul z yang baru bertinggi $k+2$, dan $\text{bf}(z) = k - (k+1) = -1$. Lalu $\text{bf}(y) = (k+2) - (k+1) = +1$. Keduanya sah. ✓

Jadi satu rotasi kanan mencukupi pada kedua subkasusnya, dan itulah sebabnya ujinya $\text{bf}(L) \leq 0$ dan bukan $\text{bf}(L) < 0$. §8.11.5 kembali kepada hal ini, sebab inilah titik paling halus pada Soal 8.7 sampai 8.10.

8.11.4 Kasus LR — satu rotasi kiri, lalu satu rotasi kanan

Kasus LR adalah $\text{bf}(L) > 0$ — subpohon kirinya berat ke kanan, sehingga lintasan dari akarnya turun ke bagian yang tinggi berzig-zag: kiri, lalu kanan. **Gambar 8.9** memperlihatkan kedua langkahnya.



Gambar 8.9 Kasus Left-Right. Satu rotasi kanan tunggal terhadap z akan meninggalkan pohonnya sama tak seimbangnya ke arah sebaliknya. Pertama satu rotasi kiri terhadap y meluruskan zig-zagnya menjadi condongan kiri-kiri; lalu satu rotasi kanan terhadap z menuntaskannya.

Mengapa satu rotasi tidak berhasil di sini? Sebab rotasi kanan terhadap z mengangkat subpohon *kiri* milik y, sedangkan pada kasus LR bagian yang tinggi adalah subpohon *kanan* milik y, yang sama sekali tidak diangkat oleh rotasi itu — ia hanya dipindahkan menyeberang. Cobalah pada kasus terkecilnya: pohon tiga simpul dengan akar 10, anak kiri 5, dan anak kanan 5 yaitu 7. Di sini $\text{bf}(10) = -2$ dan $\text{bf}(5) = +1$. Rotasi kanan terhadap 10 memberikan akar 5, anak kiri kosong, anak kanan 10 yang anak kirinya 7: sekarang $\text{bf}(5) = +2$. Ketakseimbangannya berpindah sisi, bukan hilang.

Perbaikannya: rotasilah dahulu ke **kiri** terhadap L, yang mengubah zig-zagnya menjadi condongan kiri yang lurus; lalu rotasilah ke **kanan** terhadap t. Pada contoh tiga simpulnya: rotasi kiri terhadap 5 memberikan rantai $10 \to 7 \to 5$ yang condong ke kiri; lalu rotasi kanan terhadap 10 memberikan akar 7 dengan anak-anak 5 dan 10, seimbang sempurna.

Jadi:

$$\text{rebalanceForLeft}(t) = \begin{cases} t & \text{bf}(t) \geq -1 \quad \text{(tak ada yang} \\ \text{rotateRight}(t) & \text{bf}(t) = -2 \text{ dan } \text{bf}(L) \leq 0 \\ \text{rotateRight}(v, \text{rotateLeft}(L), R) & \text{bf}(t) = -2 \text{ dan } \text{bf}(L) > 0 \end{cases} \tag{8.16}$$

dan dalam kode Persamaan (8.16) itu enam baris:

```
MyTree rebalanceForLeft(MyTree t) {
    if (balanceFactor(t) >= -1) return t;              // -1, 0 atau +1: sudah sah
    MyTree l = leftOf(t);                              // bf(t) == -2, jadi l pasti ada
    if (balanceFactor(l) <= 0) return rotateRight(t);  // LL: satu rotasi
    // LR: luruskan dahulu zig-zagnya, baru rotasi kanan
    return rotateRight(node(value(t), rotateLeft(l), rightOf(t)));
}
```

8.11.5 Titik paling halus: mengapa $\leq$ dan bukan $<$

Uji yang memisahkan LL dari LR adalah `balanceFactor(l) <= 0`. Mengapa 0 disertakan di sisi LL?

Selama penyisipan, $\mathrm{bf}(L) = 0$ **tidak mungkin terjadi.** Jika subpohon kirinya bertambah satu takik lalu simpulnya menjadi tak seimbang, maka anak yang bertambah itu secara tegas lebih tinggi daripada saudaranya, sehingga $\mathrm{bf}(L) \neq 0$. Jadi untuk penyisipan saja (Soal 8.7), `<= 0` maupun `< 0` sama-sama berhasil.

Selama penghapusan, $\mathrm{bf}(L) = 0$ **sangat mungkin terjadi**, dan perbaikan yang benar saat itu adalah rotasi *tunggal*. Inilah contohnya yang terkecil. Ambillah pohon

$$z = 10, \quad L = 5\{3,7\}, \quad R = 20,$$

sehingga $h(L) = 1$, $h(R) = 0$, $\mathrm{bf}(10) = -1$: seimbang. Kini hapuslah 20. Subpohon kanannya menjadi kosong bertinggi -1, sehingga $\mathrm{bf}(10) = -1 - 1 = -2$, dan $\mathrm{bf}(L) = \mathrm{bf}(5) = 0 - 0 = 0$.

- Dengan `<= 0` kita mengambil LL lalu merotasi kanan terhadap 10: hasilnya adalah akar 5, anak kiri 3, anak kanan 10 yang memegang 7 di kiri. Tingginya: $h(3) = 0$, $h(10) = 1$, $h(5) = 2$. Faktor keseimbangannya: $\mathrm{bf}(10) = -1$, $\mathrm{bf}(5) = 1 - 0 = +1$. **Seimbang.** ✓
- Dengan `< 0` kita akan mengambil LR: rotasi kiri terhadap 5 memberikan subpohon kiri $7\{5\{3,.\},.\}$ bertinggi 2 — satu *lebih tinggi* daripada sebelumnya — lalu rotasi kanan terhadap 10 memberikan akar 7 dengan anak kiri $5\{3,.\}$ bertinggi 1 dan anak kanan 10 bertinggi 0. Kebetulan itu seimbang di sini, tetapi rotasi gandanya sudah mengerjakan kerja yang tak perlu dan, pada kasus umum ketika P_2 dan P_3 bertinggi sama dan tak nol, ia menghasilkan $\mathrm{bf} = \pm 2$ di akar barunya. **Keliru secara umum.**

> **PERINGATAN: TANDA $\leq$ ITU BUKAN SALAH KETIK DAN BUKAN HIASAN.** Petunjuk Soal 8.7 berkata kedua penunjang yang sama akan melayani penghapusan. Satu kalimat itulah yang memaksa `<= 0`. Versi yang ditulis dengan `< 0` lulus setiap uji penyisipan dan gagal pada penghapusan, dan itu persis jenis kutu yang paling sukar ditemukan larut malam. Jika Anda ditanya satu hal tentang `rebalanceForLeft`, itulah pertanyaannya.

8.12 Soal 8.9 — `rebalanceForRight`: Kasus RR dan RL

8.12.1 Soalnya

Tulislah `MyTree rebalanceForRight(MyTree t)`, penunjang kedua Soal 8.7: kontraknya sama dengan `rebalanceForLeft`, dengan peran kedua subpohon ditukar. Ia dipanggil ketika subpohon *kanan* `t` mungkin baru bertambah tinggi satu takik.

8.12.2 Asas pencerminan

Inilah soal termurah di bab ini begitu Soal 8.8 terselesaikan, dan alasannya layak dinyatakan sebagai sebuah asas.

> **WAWASAN KUNCI: ASAS PENCERMINAN.** Mencerminkan sebuah pohon pencarian biner dari kiri ke kanan memetakan: kiri ↔ kanan, $<$ ↔ $>$, $\leq$ ↔ $\geq$, bf ↔ −bf, rotateRight ↔ rotateLeft, minimum ↔ maksimum, menaik ↔ menurun. Rutin benar mana pun, dicerminkan lambang demi lambang, adalah rutin yang benar untuk soal yang dicerminkan. Jadi begitu Soal 8.8 benar, Soal 8.9 hanyalah penyalinan — dan begitu `max` benar, `min` diperoleh gratis.

Terapkanlah pencerminannya pada §8.11.4 baris demi baris.

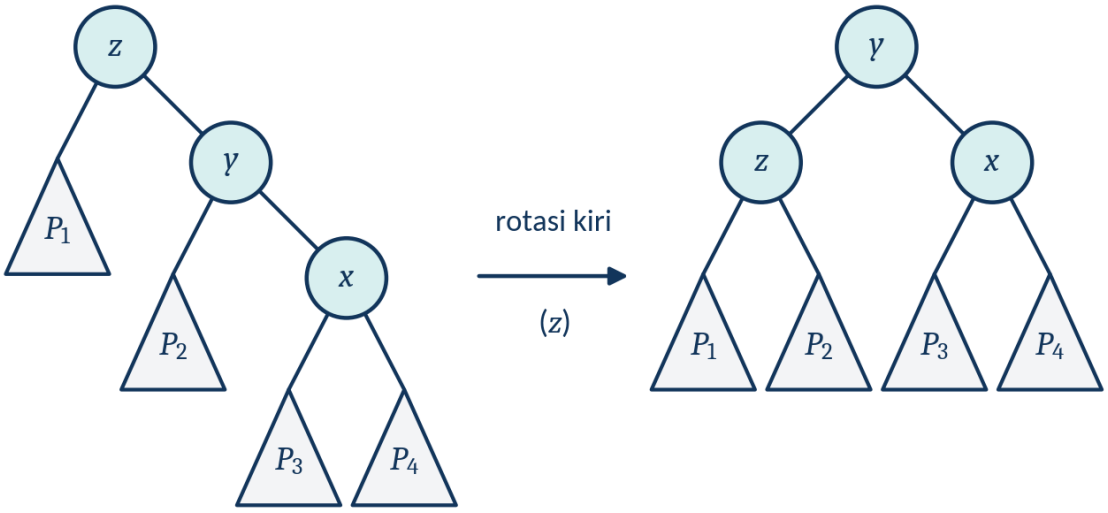
- `if (balanceFactor(t) >= -1) return t;` menjadi `if (balanceFactor(t) <= 1) return t;`
- `MyTree l = leftOf(t);` menjadi `MyTree r = rightOf(t);`
- `if (balanceFactor(l) <= 0) return rotateRight(t);` menjadi `if (balanceFactor(r) >= 0) return rotateLeft(t);`
- `return rotateRight(node(value(t), rotateLeft(l), rightOf(t)));` menjadi `return rotateLeft(node(value(t), leftOf(t), rotateRight(r)));`

Enam baris, diperoleh secara mekanis:

```
MyTree rebalanceForRight(MyTree t) {
    if (balanceFactor(t) <= 1) return t;                // sudah sah
    MyTree r = rightOf(t);                              // bf(t) == +2, jadi r pasti ada
    if (balanceFactor(r) >= 0) return rotateLeft(t);    // RR: satu rotasi
    // RL: luruskan, lalu rotasi kiri
    return rotateLeft(node(value(t), leftOf(t), rotateRight(r)));
}
```

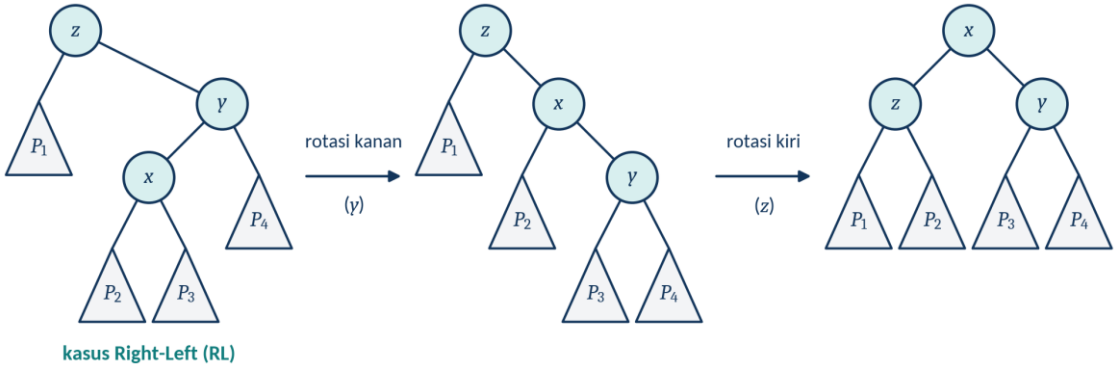
8.12.3 Kasus RR dan RL, dalam gambar

Gambar 8.10 dan **Gambar 8.11** memperlihatkan kedua kasusnya.



kasus Right-Right (RR)

Gambar 8.10 Kasus Right-Right: y adalah anak kanan z dan x adalah anak kanan y. Satu rotasi kiri terhadap z membetulkannya. Ini cermin persis dari **Gambar 8.8**.



Gambar 8.11 Kasus Right-Left: zig-zagnya kanan lalu kiri. Satu rotasi kanan terhadap y meluruskannya, lalu satu rotasi kiri terhadap z menuntaskannya. Cermin persis dari **Gambar 8.9**.

Tabel 8.7 adalah seluruh tabel keputusan untuk Soal 8.8 dan 8.9 sekaligus dalam satu halaman. Inilah yang harus dihafalkan, dan isinya empat baris.

Tabel 8.7 Keempat kasus rotasi, sebagaimana diuji di dalam kode. Perhatikan bahwa kasus nolnya tergabung dengan rotasi *tunggal* di kedua sisinya — ≤ 0 di kiri, ≥ 0 di kanan — karena alasan yang diberikan §8.11.5.

bf(t)	bf Anak yang tinggi	Kasus	Perbaikan
-2	$\text{bf}(L) \leq 0$	**LL**	satu `rotateRight(t)`
-2	$\text{bf}(L) > 0$	**LR**	`rotateLeft(L)`, lalu `rotateRight(t)`
$+2$	$\text{bf}(R) \geq 0$	**RR**	satu `rotateLeft(t)`
$+2$	$\text{bf}(R) < 0$	**RL**	`rotateRight(R)`, lalu `rotateLeft(t)`

8.12.4 Belokan keliru yang sering terjadi pada Soal 8.8 dan 8.9

1. **Membalik penjaga terluarnya.** Menulis `if (balanceFactor(t) <= -1) return t;` di dalam `rebalanceForLeft` membuatnya kembali lebih awal pada pohon dengan bf $= -2$, sehingga tak ada yang pernah dibetulkan. Ujinya: bangunlah rantai 30,20,10 lewat penyisipan lalu periksa bahwa hasilnya 20{10,30}.
2. **Memakai `<` alih-alih `<=`** pada uji kasusnya. §8.11.5. Lulus penyisipan, gagal penghapusan.
3. **Merotasi terhadap simpul yang keliru pada kasus gandanya.** Rotasi pertamanya terhadap *anaknya*, bukan terhadap simpulnya sendiri. Menulis `rotateLeft(t)` lalu `rotateRight(t)` adalah komposisi yang tidak melakukan apa-apa: keduanya saling meniadakan.
4. **Menyalin `rebalanceForLeft` ke dalam `rebalanceForRight` lalu hanya mengubah sebagian lambangnya.** Pencerminannya punya *empat* perubahan; sangat mudah hanya membuat tiga. Periksalah masing-masing terhadap daftar di §8.12.2.
5. **Melupakan bahwa penyeimbang ulangnya boleh saja secara sah tidak melakukan apa-apa.** Kedua fungsi itu dipanggil pada *setiap* simpul lintasan penyisipannya, dan sebagian besar panggilan itu mengembalikan `t` tanpa perubahan. Versi yang mengandaikan dirinya selalu dipanggil pada simpul yang tak seimbang akan mendereferensi subpohon kosong.
6. **Memamerkan kedua penunjangnya.** `rebalanceForLeft` dan `rebalanceForRight` ada untuk melayani `insertHB` dan `deleteHB`; pemanggil yang memakainya di luar kontrak itu tidak memperoleh jaminan apa pun.

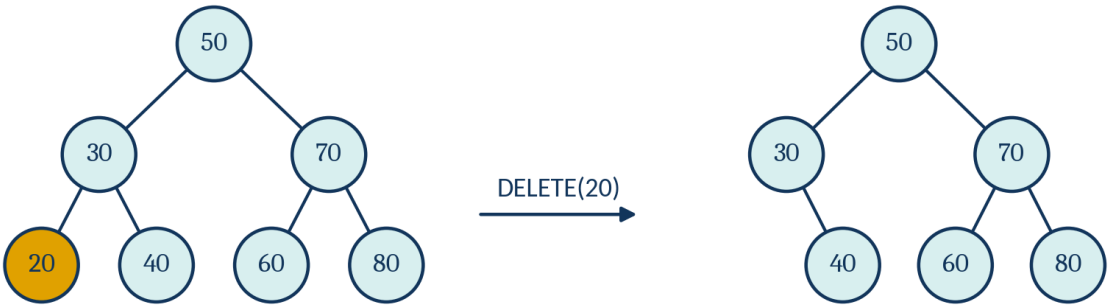
8.13 Soal 8.10 — `deleteHB`: Penghapusan yang Menjaga Pohon Tetap Seimbang

8.13.1 Soalnya

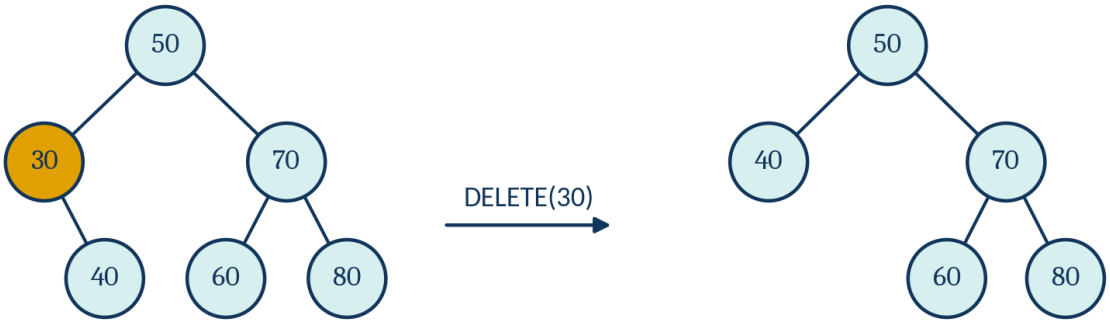
Tulislah fungsi rekursif `MyTree deleteHB(MyTree t, int x)` yang menghapus `x` dari `t` dan mengembalikan pohon hasilnya, dengan tetap menjaga syarat AVL. Fungsi itu harus memakai kedua penunjang Soal 8.8 dan 8.9.

8.13.2 Ketiga kasus penghapusan

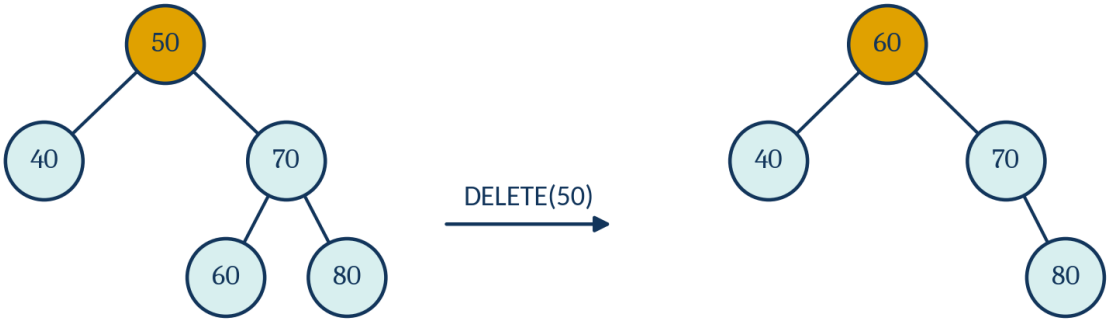
Bab 6 memberikan ketiga kasus penghapusan BST biasa; **Gambar 8.12**, **Gambar 8.13**, dan **Gambar 8.14** memperlihatkannya.



Gambar 8.12 Kasus penghapusan pertama: simpul yang akan dihapus adalah daun. Cukup singkirkan ia.



Gambar 8.13 Kasus kedua: simpulnya punya tepat satu anak. "Copy the child to the node will be deleted, then delete the child" — setara dengan itu, dalam gaya fungsional, kembalikanlah anaknya sebagai pengganti simpulnya.



Gambar 8.14 Kasus ketiga: simpulnya punya dua anak. Carilah pengganti inordernya (atau, secara simetris, pendahulunya), salinlah nilai itu ke dalam simpulnya, lalu hapuslah penggantinya dari subpohon kanannya — sebuah soal yang lebih kecil berupa kasus 1 atau kasus 2.

Dalam gaya fungsional ketiga kasusnya runtuh dengan menyenangkan, sebab "kembalikan anaknya sebagai pengganti simpulnya" menangani kasus 1 dan 2 sekaligus: subpohon kiri sebuah daun itu kosong, jadi mengembalikan subpohon kanannya berarti mengembalikan pohon kosong.

$$\mathrm{del}((v, L, R)) \text{ bila } v \text{ adalah sasarannya} = \begin{cases} R & L = \emptyset \quad \text{(kasus 1} \\ L & R = \emptyset \quad \text{(kasus 2} \\ (m, \mathrm{del}(L, m), R) & \text{jika tidak, deng} \end{cases}$$

Kita memakai **pendahulu** inorder $m = \max(L)$ alih-alih penggantinya $\min(R)$. Keduanya benar; pendahulunya adalah maksimum subpohon kiri, yang sudah diberikan Soal 8.5 kepada kita secara gratis, dan ia menjaga panggilan penyeimbangan ulangnya tetap di sisi yang sama di sepanjang prosesnya. §8.13.5 membahas pilihan itu.

8.13.3 Jebakannya: penyeimbang ulang mana yang dipanggil

Inilah kekeliruan serius yang paling sering terjadi pada penghapusan, dan ia kekeliruan satu kata.

Ketika sasarannya **lebih kecil** daripada simpul saat ini, kita berekursi ke subpohon **kiri**. Subpohon itu mungkin menjadi *lebih pendek*. Subpohon kiri yang lebih pendek berarti subpohon **kanannya** kini mungkin dua takik lebih tinggi. Jadi simpulnya harus dibetulkan dengan `rebalanceForRight`.

Bacalah sekali lagi, sebab itu kebalikan dari `insertHB`. Pada penyisipan, berekursi ke kiri mungkin membuat sisi kiri *lebih tinggi*, jadi kita memanggil `rebalanceForLeft`. Pada penghapusan, berekursi ke kiri mungkin membuat sisi kiri *lebih pendek*, jadi kita memanggil `rebalanceForRight`.

> **PERINGATAN: SISIP KIRI ⇒ `rebalanceForLeft`. HAPUS KIRI ⇒ `rebalanceForRight`.** Tuliskan kedua baris itu sebelum Anda menulis fungsinya. Penyeimbang ulangnya dinamai menurut *sisi yang mungkin bertambah tinggi*, dan menghapus dari satu sisi membuat sisi **yang lain** relatif lebih tinggi. Membaliknya menghasilkan kode yang terkompilasi, lulus uji penyisipan, dan diam-diam meninggalkan pohon yang tak seimbang — jenis kutu yang terburuk, sebab tak ada apa pun yang jatuh.

Jadi:

$$\mathrm{deleteHB}(\varnothing, x) = \varnothing$$

$$\mathrm{deleteHB}((v, L, R), x) = \begin{cases} \mathrm{rebalanceForRight}(v, \mathrm{deleteHB}(L, x), R) & x < \\ \mathrm{rebalanceForLeft}(v, L, \mathrm{deleteHB}(R, x)) & x > \\ \text{ketiga kasus pada §8.13.2} & x = \end{cases} \tag{8.17}$$

Bandingkan Persamaan (8.17) dengan Persamaan (8.14): strukturnya sama dan kedua penyeimbang ulangnya bertukar tempat. Pada kasus dua anaknya, setelah v diganti oleh $m = \max(L)$ dan m dihapus dari L, subpohon kirinya sekali lagi menjadi lebih pendek, sehingga simpul itu pun dibungkus dengan `rebalanceForRight`.

8.13.4 Mengapa penghapusan mungkin butuh banyak penyeimbangan ulang, dan penyisipan paling banyak satu

Bab 6 §6.20.2 sudah menetapkan ketaksetangkupannya; berikut alasannya dalam dua kalimat, sebab inilah salah satu fakta paling mendidik tentang pohon AVL.

Penyisipan. Penyeimbangan ulang di simpul tak seimbang yang terendah memulihkan subpohon itu ke tinggi yang dimilikinya *sebelum* penyisipannya. Jadi setiap leluhur di atasnya tidak melihat perubahan sama sekali, dan tak ada perbaikan lanjutan yang mungkin atau perlu. Paling banyak satu penyeimbangan ulang — paling banyak dua rotasi.

Penghapusan. Penyeimbangan ulang di simpul tak seimbang yang terendah mungkin meninggalkan subpohon itu *satu lebih pendek* daripada sebelumnya. Pemendekan itu sendiri merupakan perubahan, yang mungkin membuat induknya tak seimbang, yang mungkin memendek lagi, dan begitu seterusnya sampai ke akarnya. Sampai $h = \Theta(\lg n)$ penyeimbangan ulang.

Soal latihan 7 menyusun kasus terburuknya secara tersurat — pohon AVL bersimpul minimum, yaitu *pohon Fibonacci* Bab 6 §6.15.1 — dan **Kode 8.14** di §8.23 mengukur riaknya: menghapus satu kunci dari pohon AVL minimal bertinggi 12 melakukan enam rotasi, dan cacahannya tumbuh sebagai $\lfloor h/2 \rfloor$.

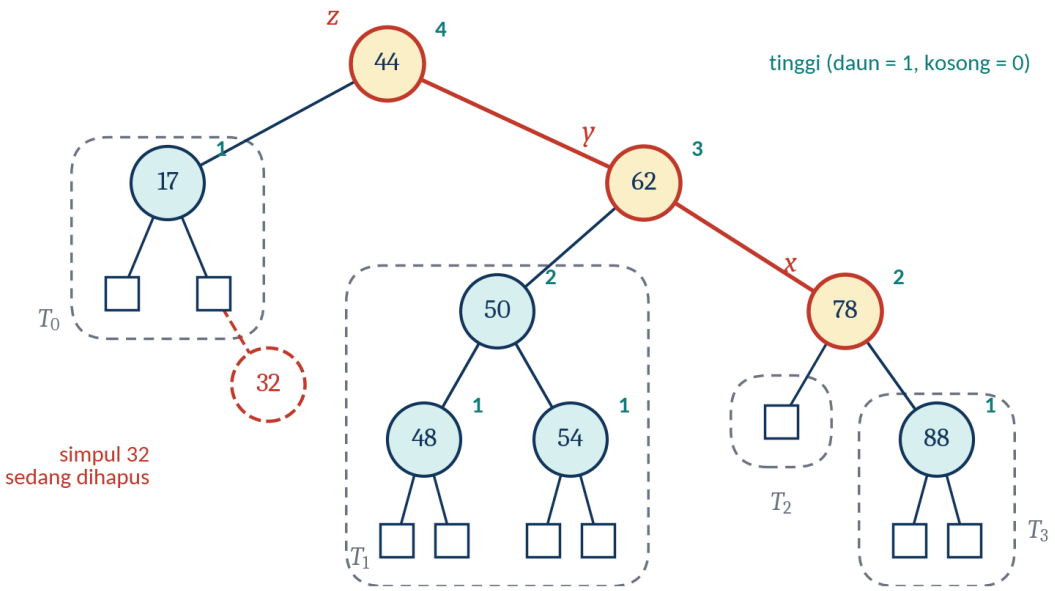
8.13.5 Pendahulu atau pengganti?

Naskahnya tidak menyebutkan yang mana yang harus dipakai, dan keduanya benar. Pertimbangannya:

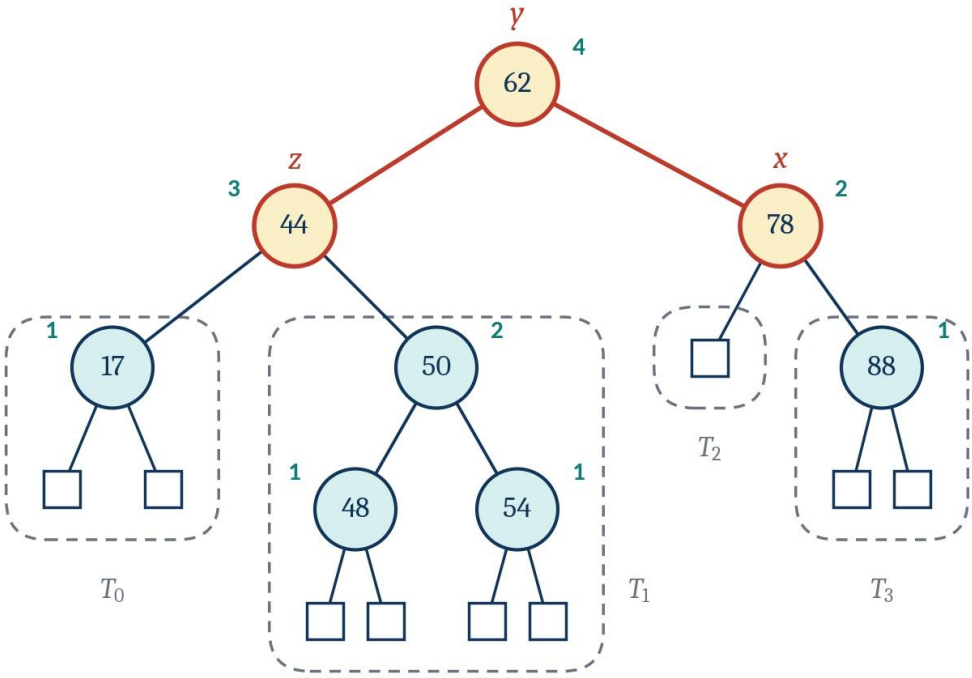
- **Pendahulu, $m = \max(L)$.** Memakai ulang `max` dari Soal 8.5 secara langsung. Penghapusan lanjutannya terjadi di subpohon kiri, sehingga simpulnya dibetulkan dengan `rebalanceForRight` — penyeimbang ulang yang *sama* dengan cabang $x < v$, yang membuat kodenya lebih seragam dan lebih mudah dibuat benar. Inilah yang dilakukan **Kode 8.8**.
- **Pengganti, $m = \min(R)$.** Pilihan buku teks klasik, sesuai dengan **Gambar 8.14** dan Bab 6 §6.11.3. Penghapusan lanjutannya terjadi di subpohon kanan, sehingga perbaikannya memakai `rebalanceForLeft`.

Memilih salah satunya lalu menyebutkan yang mana, dalam satu kalimat, sudah cukup. Memilih salah satunya lalu memanggil penyeimbang ulang yang *keliru* sesudahnya adalah jebakan §8.13.3 yang mengenakan topi berbeda.

Penghapusan terkerjakan Bab 6, yang diperlihatkan **Gambar 8.15** dan **Gambar 8.16**, adalah kasus 1: simpul 32 sebuah daun. Bagian yang menariknya adalah apa yang terjadi sesudahnya.



Gambar 8.15 Sebelum penghapusan: simpul 32 hendak dihapus, dan leluhur tak seimbang yang pertama $z = 44$, anaknya yang lebih tinggi $y = 62$, serta anak y yang lebih tinggi $x = 78$ sudah ditandai. Tinggi pada gambarnya menghitung simpul, sehingga sebuah daun bernilai 1 — satu lebih besar daripada kesepakatan kita di mana-mana.



Gambar 8.16 Setelah satu rotasi kiri terhadap 44. Simpul 62 menjadi akar yang baru, 44 turun ke kirinya sambil memegang subpohon berakar 50, dan 78 tak berubah di kanan.

Dalam kosakata Soal 8.10: menghapus 32 dari subpohon kanan milik 17 membuat sisi kanan 17 lebih pendek, sehingga 17 dibetulkan dengan `rebalanceForLeft` — yang tidak melakukan apa-apa, sebab 17 kini daun yang seimbang. Lalu sisi *kiri* milik 44 menjadi lebih pendek, sehingga 44 dibetulkan dengan `rebalanceForRight`, yang menemukan $\text{bf}(44) = +2$ dan $\text{bf}(62) = 0 \geq 0$, mengambil baris RR pada **Tabel 8.7**, lalu melakukan satu rotasi kiri. Itu persis kesimpulan yang ditarik **Gambar 8.15** — kasusnya Right-Right, jadi dilakukan rotasi kiri — yang dicapai lewat tabel keputusannya alih-alih lewat menatap gambarnya.

8.13.6 Soal 8.7 sampai 8.10 sebagai satu program

Kode 8.8 adalah keempat penyelesaian itu di dalam satu program utuh. Fungsi `main`-nya mengulangi setiap satu dari kelima contoh penyisipan Bab 6 beserta contoh penghapusannya, lalu menjalankan uji tekanan seribu elemen yang memeriksa invarian AVL-nya, keterurutannya, dan batas tinggi teoretis Bab 6.

Kode 8.8 Soal 8.7 sampai 8.10 dalam satu program: `insertHB`, `rebalanceForLeft`, `rebalanceForRight`, `deleteHB`, ditambah pemeriksaan-diri AVL, pemeriksaan keterurutan, dan uji tekanan terhadap batas tinggi teoretisnya.

```
// avlhb.cpp -- Soal 8.7 sampai 8.10: insertHB(), rebalanceForLeft(),
//              rebalanceForRight(), dan deleteHB(), dalam gaya fungsional.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <cmath>
#include <algorithm>
using namespace std;

struct MyTreeNode;
using MyTree = MyTreeNode*;
struct MyTreeNode { int value; MyTree left, right; int h; };

MyTree emptyTree()          { return nullptr; }
bool   isEmpty(MyTree t)    { return t == nullptr; }
int    height(MyTree t)     { return isEmpty(t) ? -1 : t->h; }
int    value(MyTree t)      { return t->value; }
MyTree leftOf(MyTree t)     { return t->left; }
MyTree rightOf(MyTree t)    { return t->right; }

// SATU-SATUNYA konstruktor. Ia menghitung ulang tinggi dari kedua subpohon, jadi
// pohon yang dibangun lewat node() tak pernah membawa tinggi yang basi.
MyTree node(int v, MyTree l, MyTree r) {
    MyTree t = new MyTreeNode; t->value = v; t->left = l; t->right = r;
    t->h = 1 + max(height(l), height(r)); return t;
}
int balanceFactor(MyTree t) { return height(rightOf(t)) - height(leftOf(t)); }

long long rotations = 0;    // mencacah rotasi tunggal yang dilakukan

// ---------- kedua rotasi, dalam bentuk fungsional ----------
MyTree rotateRight(MyTree t) {                  // anak kiri t naik
    ++rotations;
    MyTree l = leftOf(t);
    return node(value(l), leftOf(l), node(value(t), rightOf(l), rightOf(t)));
}
MyTree rotateLeft(MyTree t) {                   // anak kanan t naik
    ++rotations;
    MyTree r = rightOf(t);
    return node(value(r), node(value(t), leftOf(t), leftOf(r)), rightOf(r));
}
```

Kode 8.8 (Lanjutan) *Soal 8.7 sampai 8.10 dalam satu program: insertHB, rebalanceForLeft, rebalanceForRight, deleteHB, ditambah pemeriksaan-diri AVL, pemeriksaan keterurutan, dan uji tekanan terhadap batas tinggi teoretisnya.*

```
// ---------- Soal 8.8 ----------
// Dipanggil saat subpohon KIRI milik t mungkin bertambah tinggi satu takik.
MyTree rebalanceForLeft(MyTree t) {
    if (balanceFactor(t) >= -1) return t;               // -1, 0 atau +1: sudah sah
    MyTree l = leftOf(t);                               // bf(t) == -2, jadi l pasti ada
    if (balanceFactor(l) <= 0) return rotateRight(t);   // LL: satu rotasi
    // LR: luruskan dahulu zig-zagnya, baru rotasi kanan
    return rotateRight(node(value(t), rotateLeft(l), rightOf(t)));
}

// ---------- Soal 8.9 ----------
// Bayangan cermin yang persis: subpohon KANAN mungkin bertambah tinggi satu takik.
MyTree rebalanceForRight(MyTree t) {
    if (balanceFactor(t) <= 1) return t;                // sudah sah
    MyTree r = rightOf(t);                              // bf(t) == +2, jadi r pasti ada
    if (balanceFactor(r) >= 0) return rotateLeft(t);    // RR: satu rotasi
    // RL: luruskan, lalu rotasi kiri
    return rotateLeft(node(value(t), leftOf(t), rotateRight(r)));
}

// ---------- Soal 8.7 ----------
MyTree insertHB(int n, MyTree t) {
    if (isEmpty(t)) return node(n, emptyTree(), emptyTree());
    if (n < value(t))
        return rebalanceForLeft (node(value(t), insertHB(n, leftOf(t)), rightOf(t)));
    if (n > value(t))
        return rebalanceForRight(node(value(t), leftOf(t), insertHB(n, rightOf(t))));
    return t;                                           // kembar: tak ada perubahan
}

// ---------- Soal 8.10 ----------
int maxOf(MyTree t) {                       // max O(h) dari Soal 8.5
    return isEmpty(rightOf(t)) ? value(t) : maxOf(rightOf(t));
}
MyTree deleteHB(MyTree t, int x) {
    if (isEmpty(t)) return t;                           // x tidak ada di dalam pohon
    if (x < value(t))
        // sisi KIRI baru saja MEMENDEK, jadi sisi KANAN bisa dua takik lebih tinggi
        return rebalanceForRight(node(value(t), deleteHB(leftOf(t), x), rightOf(t)));
    if (x > value(t))
        return rebalanceForLeft (node(value(t), leftOf(t), deleteHB(rightOf(t), x)));
    if (isEmpty(leftOf(t)))  return rightOf(t);         // 0 atau 1 anak
    if (isEmpty(rightOf(t))) return leftOf(t);
    int m = maxOf(leftOf(t));                           // pendahulu inorder
    return rebalanceForRight(node(m, deleteHB(leftOf(t), m), rightOf(t)));
}
```

Kode 8.8 (Lanjutan) *Soal 8.7 sampai 8.10 dalam satu program: insertHB, rebalanceForLeft, rebalanceForRight, deleteHB, ditambah pemeriksaan-diri AVL, pemeriksaan keterurutan, dan uji tekanan terhadap batas tinggi teoretisnya.*

```
// ---------- pemeriksaan ----------
string shape(MyTree t) {
    if (isEmpty(t)) return ".";
    string s = to_string(value(t)) + "(" + to_string(balanceFactor(t)) + ")";
    if (isEmpty(leftOf(t)) && isEmpty(rightOf(t))) return s;
    return s + "{" + shape(leftOf(t)) + "," + shape(rightOf(t)) + "}";
}
int  countNodes(MyTree t) { return isEmpty(t) ? 0 : 1 + countNodes(leftOf(t)) +
countNodes(rightOf(t)); }
bool isAVL(MyTree t) {
    if (isEmpty(t)) return true;
    if (t->h != 1 + max(height(leftOf(t)), height(rightOf(t)))) return false;  // tinggi
yang basi
    if (abs(balanceFactor(t)) > 1) return false;
    return isAVL(leftOf(t)) && isAVL(rightOf(t));
}
void collect(MyTree t, vector<int>& v) {
    if (isEmpty(t)) return;
    collect(leftOf(t), v); v.push_back(value(t)); collect(rightOf(t), v);
}
bool isSorted(MyTree t) {
    vector<int> v; collect(t, v);
    return is_sorted(v.begin(), v.end()) && adjacent_find(v.begin(), v.end()) == v.end();
}

void step(const char* what, MyTree t) {
    cout << "  " << left << setw(12) << what << shape(t)
         << "   [h=" << height(t) << ", rot=" << rotations
         << ", AVL=" << (isAVL(t) && isSorted(t) ? "yes" : "NO") << "]\n";
}

int main() {
    cout << "--- building the eight-node tree of Chapter 6 with insertHB ---\n";
    rotations = 0;
    MyTree t = emptyTree();
    for (int v : {13, 10, 15, 5, 11, 16, 4, 6}) t = insertHB(v, t);
    step("built", t);

    cout << "--- insert 14, no rotation expected ---\n";
    rotations = 0; MyTree t14 = insertHB(14, t); step("insert 14", t14);

    cout << "--- insert 3, Left-Left expected ---\n";
    rotations = 0; MyTree t3 = insertHB(3, t); step("insert 3", t3);

    cout << "--- insert 7, Left-Right expected ---\n";
    rotations = 0; MyTree t7 = insertHB(7, t); step("insert 7", t7);
```

Kode 8.8 (Lanjutan) *Soal 8.7 sampai 8.10 dalam satu program: insertHB, rebalanceForLeft, rebalanceForRight, deleteHB, ditambah pemeriksaan-diri AVL, pemeriksaan keterurutan, dan uji tekanan terhadap batas tinggi teoretisnya.*

```
    cout << "--- root 30, left 5, right 35{32,40}; insert 45, Right-Right ---\n";
    rotations = 0;
    MyTree u = emptyTree();
    for (int v : {30, 5, 35, 32, 40}) u = insertHB(v, u);
    step("built", u);
    rotations = 0; step("insert 45", insertHB(45, u));

    cout << "--- insert 15 into 5{2{1,4{3,.}},7{6,9{.,16}}}, Right-Left ---\n";
    rotations = 0;
    MyTree w = emptyTree();
    for (int v : {5, 2, 7, 1, 4, 6, 9, 3, 16}) w = insertHB(v, w);
    step("built", w);
    rotations = 0; step("insert 15", insertHB(15, w));

    cout << "--- the deletion example of Chapter 6 ---\n";
    rotations = 0;
    MyTree d = emptyTree();
    for (int v : {44, 17, 62, 32, 50, 78, 48, 54, 88}) d = insertHB(v, d);
    step("built", d);
    rotations = 0; MyTree d2 = deleteHB(d, 32); step("delete 32", d2);
    rotations = 0; step("delete 62", deleteHB(d, 62));   // simpul beranak dua
    rotations = 0; step("delete 99", deleteHB(d, 99));   // tidak ada: tak berubah

    cout << "--- stress test: insert 1..1000 in order, then delete them all ---\n";
    rotations = 0;
    MyTree s = emptyTree();
    for (int v = 1; v <= 1000; ++v) s = insertHB(v, s);
    int n = countNodes(s);
    double bound = 1.4405 * log2((double)n + 2.0) - 1.3277;
    cout << "  n = " << n << ", height = " << height(s)
         << ", AVL = " << (isAVL(s) && isSorted(s) ? "yes" : "NO")
         << ", rotations = " << rotations << "\n";
    cout << "  theoretical bound 1.4405*lg(n+2)-1.3277 = " << fixed << setprecision(3) <<
bound
         << "  ->  height <= " << (int)floor(bound) << ": "
         << (height(s) <= bound ? "satisfied" : "VIOLATED") << "\n";
    rotations = 0;
    for (int v = 1; v <= 1000; ++v) s = deleteHB(s, v);
    cout << "  after deleting all: nodes = " << countNodes(s)
         << ", isEmpty = " << (isEmpty(s) ? "true" : "false")
         << ", rotations = " << rotations << "\n";
    return 0;
}
```

Dikompilasi dan dijalankan, ia mencetak:

```
--- building the eight-node tree of Chapter 6 with insertHB ---
  built       13(-1){10(-1){5(0){4(0),6(0)},11(0)},15(1){.,16(0)}}   [h=3, rot=0, AVL=yes]
--- insert 14, no rotation expected ---
  insert 14   13(-1){10(-1){5(0){4(0),6(0)},11(0)},15(0){14(0),16(0)}}   [h=3, rot=0,
AVL=yes]
--- insert 3, Left-Left expected ---
  insert 3    13(-1){5(0){4(-1){3(0),.},10(0){6(0),11(0)}},15(1){.,16(0)}}   [h=3, rot=1,
AVL=yes]
--- insert 7, Left-Right expected ---
  insert 7    13(-1){6(0){5(-1){4(0),.},10(0){7(0),11(0)}},15(1){.,16(0)}}   [h=3, rot=2,
AVL=yes]
--- root 30, left 5, right 35{32,40}; insert 45, Right-Right ---
  built       30(1){5(0),35(0){32(0),40(0)}}   [h=2, rot=0, AVL=yes]
  insert 45   35(0){30(0){5(0),32(0)},40(1){.,45(0)}}   [h=2, rot=1, AVL=yes]
--- insert 15 into 5{2{1,4{3,.}},7{6,9{.,16}}}, Right-Left ---
  built       5(0){2(1){1(0),4(-1){3(0),.}},7(1){6(0),9(1){.,16(0)}}}   [h=3, rot=0,
AVL=yes]
  insert 15   5(0){2(1){1(0),4(-1){3(0),.}},7(1){6(0),15(0){9(0),16(0)}}}   [h=3, rot=2,
AVL=yes]
--- the deletion example of Chapter 6 ---
  built       44(1){17(1){.,32(0)},62(0){50(0){48(0),54(0)},78(1){.,88(0)}}}   [h=3, rot=0,
AVL=yes]
  delete 32   62(-1){44(1){17(0),50(0){48(0),54(0)}},78(1){.,88(0)}}   [h=3, rot=1,
AVL=yes]
  delete 62   44(1){17(1){.,32(0)},54(0){50(-1){48(0),.},78(1){.,88(0)}}}   [h=3, rot=0,
AVL=yes]
  delete 99   44(1){17(1){.,32(0)},62(0){50(0){48(0),54(0)},78(1){.,88(0)}}}   [h=3, rot=0,
AVL=yes]
--- stress test: insert 1..1000 in order, then delete them all ---
  n = 1000, height = 9, AVL = yes, rotations = 990
  theoretical bound 1.4405*lg(n+2)-1.3277 = 13.032  ->  height <= 13: satisfied
  after deleting all: nodes = 0, isEmpty = true, rotations = 491
```

Periksalah ia terhadap hitungan tangan Bab 6, satu baris demi satu baris.

- **Sisip 14:** `rot=0`. **Gambar 8.6** tidak membutuhkan rotasi. ✓
- **Sisip 3:** `rot=1`, dan bentuk hasilnya adalah akar 13, anak kiri 5 dengan anak-anak 4 (memegang 3) dan 10 (memegang 6 dan 11). Bab 6 §6.18.2 menurunkan persis bentuk itu dengan tangan, dengan $\mathrm{bf}(4) = -1$, $\mathrm{bf}(10) = 0$, $\mathrm{bf}(5) = 0$, $\mathrm{bf}(13) = -1$ — dan bentuk tercetaknya memperlihatkan `4(-1)`, `10(0)`, `5(0)`, `13(-1)`. ✓
- **Sisip 7:** `rot=2`, rotasi ganda kasus LR, dan bentuknya adalah akar 13, anak kiri 6 dengan anak-anak 5 (memegang 4) dan 10 (memegang 7 dan 11) — persis hasil Bab 6 §6.18.4. ✓
- **Sisip 45:** `rot=1`, dan bentuknya adalah akar 35, anak kiri 30 dengan anak-anak 5 dan 32, anak kanan 40 dengan anak kanan 45 — Bab 6 §6.18.3. ✓
- **Sisip 15:** `rot=2`, RL, dan subpohon kanannya menjadi $7\{6,15\{9,16\}\}$ — Bab 6 §6.18.5. ✓
- **Hapus 32:** `rot=1`, dan bentuknya adalah `62(-1){44(1){17(0),50(0){48(0),54(0)}},78(1){.,88(0)}}` — yang merupakan pohon **Gambar 8.16**, ditulis dalam satu baris, dan karakter demi karakter merupakan keluaran yang dikutip Bab 6 §6.20.3. ✓
- **Hapus 62:** kasus dua anaknya. Pendahulu 62 di subpohon kirinya adalah 54, dan memang 54 menjadi akar baru subpohon itu; tak ada rotasi yang diperlukan. ✓
- **Hapus 99:** nilai yang tidak ada di dalam pohonnya. Bentuknya tak berubah dan tak ada rotasi yang terjadi — kasus dasar `if (isEmpty(t)) return t;` mengerjakan tugasnya. ✓
- **Uji tekanan:** seribu penyisipan menaik — urutan terburuk yang mungkin bagi BST biasa, yang akan menghasilkan rantai bertinggi 999 — menghasilkan tinggi 9. Batas teoretisnya adalah $1.4405\lg 1002 - 1.3277 = 13.032$, jadi invariannya berlaku dengan kelonggaran. Seluruh seribu

penghapusannya lalu mengosongkan pohonnya, dengan 491 rotasi, dan setiap pohon antaranya lulus baik pemeriksaan AVL maupun pemeriksaan keterurutannya.

> **CONTOH: ARGUMEN KEBENARAN DALAM SATU PARAGRAF.** *"Keempat rutin itu diperiksa terhadap keenam contoh terkerjakan Bab 6. Menyisipkan* 14 *tidak membutuhkan rotasi; menyisipkan* 3 *memerlukan satu rotasi kanan, kasus LL; menyisipkan* 45 *memerlukan satu rotasi kiri, RR; menyisipkan* 7 *memerlukan dua rotasi, LR; menyisipkan* 15 *memerlukan dua rotasi, RL; dan menghapus* 32 *memerlukan satu rotasi kiri terhadap* 44*, RR. Hitungan rotasi programnya berturut-turut* 0,1,1,2,2,1*, dan setiap bentuk yang dihasilkan cocok dengan pohon yang dikerjakan dengan tangan. Menyisipkan* 1 *sampai* 1000 *dalam urutan naik menghasilkan tinggi* 9*, di dalam batas AVL* $1.4405\lg(n+2) - 1.3277 = 14.03$*."*
>
> Paragraf itu dapat diperiksa siapa pun dalam tiga puluh detik, dan ia lebih berharga daripada prosa sepanjang apa pun tentang cara kerja algoritmanya.

8.13.7 Belokan keliru yang sering terjadi pada soal ini

1. **Memanggil penyeimbang ulang yang keliru setelah penghapusan rekursif.** §8.13.3. Inilah kekeliruan yang menghasilkan pohon tak seimbang secara diam-diam.
2. **Lupa menyeimbangkan ulang setelah penggantian pada kasus dua anak.** Panggilan rekursif `deleteHB(L, m)` memendekkan subpohon kirinya, jadi simpul itu juga butuh `rebalanceForRight`. Sangat mudah menangani kedua cabang yang mudah lalu melupakan yang ketiga.
3. **Menghapus *simpul* penggantinya alih-alih berekursi.** Kode yang melepaskan penggantinya lewat bedah penunjuk kehilangan penyeimbangan ulang di sepanjang lintasan turun ke simpul itu. Berekursilah; jangan menyambung-nyambung penunjuk.
4. **Tidak menangani "x tidak ada di dalam pohonnya".** Kasus dasarnya harus mengembalikan pohonnya tanpa perubahan, bukan jatuh dan bukan menghapus sesuatu yang lain. Ujilah `deleteHB(t, 99)`.
5. **Hanya menangani kasus daunnya** lalu mengembalikan `t` tanpa perubahan untuk simpul yang beranak, sehingga nilainya tak pernah benar-benar tersingkir. Ujilah selalu dengan menghapus *akarnya*.

8.14 Soal 8.11 sampai 8.15 — Pohon Merah-Hitam

Pohon AVL adalah salah satu cara menjaga pohon pencarian tetap pendek. Bab 6 juga menjumpai cara termasyhur yang lain, yaitu **pohon merah-hitam**, yang menyeimbangkan dengan *warna* alih-alih dengan selisih tinggi. Subbab ini mengerjakan lima soal pada pohon merah-hitam. Bersama `isBST` — yang terbawa dari Soal 8.3 tanpa perubahan, sebab pohon merah-hitam pertama-tama adalah pohon pencarian biner — kelimanya membentuk pemeriksa yang lengkap bagi struktur itu, disusul kedua rotasi yang menjadi bahan setiap penyisipan dan penghapusan merah-hitam.

8.14.1 Soal-soalnya, dan hubungannya dengan soal-soal AVL

Setiap soal dalam subbab ini bekerja pada tipe simpul yang sama: sebuah kunci, sebuah bendera `red`, penunjuk `left`, `right`, dan `parent`, serta satu penjaga bersama `NIL` yang berwarna hitam dan mewakili setiap subpohon kosong. **Tabel 8.8** mendaftar kelima soal itu.

Tabel 8.8 Kelima soal merah-hitam dalam subbab ini. Kolom terakhir mengatakan apakah sifatnya dapat diperiksa hanya dengan melihat sebuah simpul dan anak-anaknya — pertanyaan yang menentukan Soal 8.3.

Soal	Rutin	Yang harus dikerjakannya	Lokal atau global?
8.11	`countBlackNodes(t)`	Menghitung simpul hitam, tanpa menghitung `NIL`	cacahan global, tak mungkin dipangkas
8.12	`validateProperty4(t)`	Memeriksa bahwa tak ada simpul merah yang beranak merah	lokal: induk dan anak-anaknya
8.13	`validateBlackHeight(t)`	Memeriksa bahwa setiap lintasan menurun membawa simpul hitam yang sama banyak	global, diselesaikan dalam satu lintasan dengan penjaga
8.14	`isValidRBTree(t)`	Memeriksa kelima sifat merah-hitam dan sifat pohon pencarian	perakitan dari yang lain
8.15	`leftRotate(t, x)`, `rightRotate(t, y)`	Merotasi di tempat, sambil membetulkan semua penunjuk induk	bedah lokal atas enam penunjuk

Tabel 8.9 menjajarkan soal-soal merah-hitam dengan soal-soal AVL. Keduanya jelas satu keluarga: sebuah soal pemeriksaan, dua soal rotasi, dan pemeriksaan-pemeriksaan struktural di antaranya.

Tabel 8.9 Soal-soal AVL dan soal-soal merah-hitam berdampingan. Pemeriksaan pohon pencarian dan rotasinya pada hakikatnya sama; soal-soal di tengahnya berbeda sebab invarian keseimbangannya berbeda.

	Soal-soal AVL	Soal-soal merah-hitam
Pemeriksaan pohon pencarian	`isBST` (Soal 8.3)	`isBST` yang sama, logika identik
Pencacahan	`printDescending` (Soal 8.4)	`countBlackNodes` (Soal 8.11)
Sebuah pemeriksaan lokal	`max` (Soal 8.5)	`validateProperty4` (Soal 8.12)
Pemeriksaan satu lintasan dengan penjaga	`isHeightBalanced` (Soal 8.6)	`validateBlackHeight` (Soal 8.13)
Perakitan	`insertHB` (Soal 8.7)	`isValidRBTree` (Soal 8.14)
Rotasi	`rebalanceForLeft`, `rebalanceForRight` (8.8, 8.9)	`leftRotate`, `rightRotate` (8.15)
Penghapusan	`deleteHB` (Soal 8.10)	—
Invarian keseimbangan	$\lvert \mathrm{bf}(x) \rvert \leq 1$	warna, ditambah tinggi-hitam yang seragam
Gaya	fungsional: bangun ulang lalu kembalikan	imperatif: ubah di tempat, dengan penunjuk `parent`

8.14.2 Kelima sifat merah-hitam

Pohon merah-hitam adalah pohon pencarian biner yang setiap simpulnya membawa satu bit warna tambahan, merah atau hitam, dan yang di dalamnya:

1. Setiap simpul berwarna merah atau hitam.
2. Akarnya hitam.
3. Setiap daun — di sini, setiap penjaga `NIL` — hitam.
4. Jika sebuah simpul merah, kedua anaknya hitam. (Setara dengan: tak ada simpul merah yang beranak merah.)

5. Untuk tiap simpul, setiap lintasan sederhana dari simpul itu turun ke daun keturunannya memuat simpul hitam yang sama banyak. Bilangan itu, tanpa menghitung simpul itu sendiri, adalah **tinggi-hitam** (*black-height*) simpul itu.

Sifat 4 dan 5 bersama-sama memaksa pohonnya pendek. Argumennya, dalam satu baris: lintasan akar-ke-daun yang terpanjang dapat berselang-seling merah dan hitam, dan yang terpendek dapat seluruhnya hitam, jadi yang terpanjang paling banyak dua kali yang terpendek, sehingga $h \leq 2\lg(n+1)$. Bab 6 §6.21.1 membandingkannya dengan batas AVL $1.4405\lg(n+2) - 1.3277$: pohon merah-hitam boleh lebih tinggi, dan sebagai gantinya ia lebih jarang menyeimbangkan ulang [1].

8.14.3 Soal 8.11 — mencacah simpul hitam

Soalnya. Tulislah `int countBlackNodes(const RBTree& t)`, yang mengembalikan banyaknya simpul hitam `t`, tanpa menghitung penjaga `NIL`.

Penyelesaiannya. Pencacahan postorder biasa: 0 di `NIL`, jika tidak maka 1 bila simpul ini hitam, ditambah cacahan kedua subpohonnya. $\Theta(n)$, dan tidak ada cara yang lebih cerdik — setiap simpul harus diperiksa, sebab warna tidak dapat disimpulkan dari posisi. Satu-satunya keputusan yang perlu dinyatakan adalah bahwa penjaga `NIL` *tidak* dihitung, dan itu pilihan yang tepat: penjaga itu bersifat struktural, bukan data.

8.14.4 Soal 8.12 — tak ada simpul merah yang beranak merah

Soalnya. Tulislah `bool validateProperty4(const RBTree& t)`, yang mengembalikan `true` tepat ketika sifat 4 berlaku.

Penyelesaiannya. Di tiap simpul: jika simpul ini merah, tak satu pun anaknya boleh merah. Ujinya melihat sebuah simpul dan *anak-anaknya* — sebuah uji lokal — dan kali ini uji lokalnya benar, sebab sifat 4 memang benar-benar pernyataan tentang pasangan induk–anak. Bandingkan dengan §8.6.3, tempat uji lokalnya keliru karena sifat BST bukanlah pernyataan semacam itu. Mengetahui *kapan* uji lokal mencukupi adalah seluruh keterampilannya, dan Soal 8.3 serta 8.12, bila dijajarkan, menegaskan hal itu dengan tepat.

8.14.5 Soal 8.13 — tinggi-hitam yang seragam

Soalnya. Tulislah `bool validateBlackHeight(const RBTree& t)`, yang mengembalikan `true` tepat ketika sifat 5 berlaku, dalam satu lintasan.

Penyelesaiannya. Teknik penjaga §8.9.4, persis. Fungsi penunjang `computeBlackHeight` mengembalikan tinggi-hitam subpohonnya, atau -1 yang berarti "sifatnya sudah gagal di suatu tempat di bawah". Itu sah sebab tinggi-hitam yang sebenarnya tak pernah negatif. Penunjang itu mengembalikan -1 jika salah satu anaknya mengembalikan -1, atau jika tinggi-hitam kedua anaknya berbeda; jika tidak, ia mengembalikan tinggi-hitam anaknya, ditambah satu bila simpul ini hitam. Satu lintasan, $\Theta(n)$.

8.14.6 Soal 8.14 — pemeriksa yang lengkap

Soalnya. Tulislah `bool isValidRBTree(const RBTree& t)`, yang mengembalikan `true` tepat ketika `t` adalah pohon pencarian biner yang memenuhi kelima sifat merah-hitam.

Penyelesaiannya. Perakitan. Sifat 1 gratis — warnanya sebuah `bool`, jadi ia tak mungkin apa pun selain itu. Sifat 3 gratis — `NIL` dibangun hitam dan dipakai bersama. Sifat 2 adalah satu uji pada akarnya. Sifat pohon pencarian, sifat 4, dan sifat 5 adalah Soal 8.3, 8.12, dan 8.13. Rutinnya hanya

empat baris; isi sebenarnya adalah kesanggupan mengatakan *sifat mana yang tidak perlu diperiksa kodenya, dan mengapa*.

8.14.7 Soal 8.15 — rotasi dengan penunjuk induk

Soalnya. Tulislah `void leftRotate(RBTree& t, RBNode* x)`, yang mengangkat anak kanan x, yaitu y, ke tempat x dan menjadikan x anak kiri y, sambil memperbarui setiap penunjuk `parent` dan, bila x adalah akar, akar `t`. Lalu tulislah cerminannya, `rightRotate`.

Penyelesaiannya. Rotasi imperatif klasik dari Cormen dkk. [2, bab 13], dalam empat langkah bernomor: tetapkan y; pindahkan subpohon dalam milik y ke x; tautkan induk x ke y; taruh x di bawah y. Penjaga `NIL` membuktikan gunanya di sini: karena `NIL` adalah simpul sungguhan, uji `if (x->parent == NIL) t.root = y;` tidak memerlukan pemeriksaan null. Bandingkan dengan rotasi fungsional §8.11.2, yang sama sekali tidak memerlukan penunjuk induk sebab akar subpohon yang baru *dikembalikan* kepada pemanggilnya — sembilan baris lawan enam belas, dan tak ada penunjuk yang dapat terlupa.

> **PERINGATAN: PEMBARUAN PENUNJUK YANG DILUPAKAN SEMUA ORANG.** Di dalam rotasi imperatif ada *enam* penunjuk yang harus dibetulkan, dalam tiga pasang: $x.$right dan $y.$left.parent; $y.$parent dan tautan anak milik induknya; $y.$left dan $x.$parent. Sangat mudah membetulkan penunjuk yang ke bawah lalu melupakan yang ke atas, sehingga lahir pohon yang tercetak dengan benar tetapi berjalan ke atas menuju tempat yang keliru. Uji-uji pada **Kode 8.9** memeriksa `10's parent pointer was fixed` justru karena alasan ini.

8.14.8 Kelima soal sebagai satu program yang berjalan

Kode 8.9 memuat kelima penyelesaian, versi `isBST` berbatas `int` yang sempit dari §8.6.7 beserta perbaikannya, dan serangkaian pemeriksaan yang mencetak `PASS` atau `FAIL`.

Kode 8.9 Soal 8.11 sampai 8.15 dalam satu program: pemeriksa-pemeriksa merah-hitam, rotasi imperatif dengan penunjuk induk, serangkaian pemeriksaan, dan peragaan jebakan batas `int` pada §8.6.7.

```
// rbtree.cpp -- Soal 8.11 sampai 8.15: pemeriksa merah-hitam dan rotasi dengan
// penunjuk induk, bersama isBST dari Soal 8.3.
#include <iostream>
#include <string>
#include <vector>
#include <climits>
using namespace std;

// ---------------- RBNode ----------------
struct RBNode {
    int     key   = 0;
    bool    red   = false;                // true = MERAH, false = HITAM
    RBNode *left = nullptr, *right = nullptr, *parent = nullptr;
    bool isRed()   const { return red; }
    bool isBlack() const { return !red; }
};
RBNode* makeNil() {                       // penjaga bersama: HITAM, menunjuk dirinya
sendiri
    RBNode* n = new RBNode;
    n->left = n->right = n->parent = n;
    return n;
}
RBNode* const NIL = makeNil();
RBNode* newNode(int key) {                // simpul baru selalu MERAH
    RBNode* n = new RBNode;
    n->key = key; n->red = true;
    n->left = n->right = n->parent = NIL;
    return n;
}
string label(RBNode* n) {
    return n == NIL ? "NIL:B" : to_string(n->key) + ":" + (n->red ? "R" : "B");
}

// ---------------- RBTree ----------------
struct RBTree {
    RBNode* root = NIL;
    void bstInsert(int key) {             // penyisipan BST biasa, tanpa perbaikan warna
        RBNode *z = newNode(key), *y = NIL, *x = root;
        while (x != NIL) {
            y = x;
            if      (z->key < x->key) x = x->left;
            else if (z->key > x->key) x = x->right;
            else return;                  // kunci kembar: tidak melakukan apa-apa
        }
        z->parent = y;
        if      (y == NIL)        root = z;
        else if (z->key < y->key) y->left  = z;
```

Kode 8.9 (Lanjutan) *Soal 8.11 sampai 8.15 dalam satu program: pemeriksa-pemeriksa merah-hitam, rotasi imperatif dengan penunjuk induk, serangkaian pemeriksaan, dan peragaan jebakan batas int pada §8.6.7.*

```
        else                        y->right = z;
    }
};
RBNode* findNode(RBNode* n, int key) {
    if (n == NIL || key == n->key) return n;
    return key < n->key ? findNode(n->left, key) : findNode(n->right, key);
}
RBTree buildForTest(const vector<int>& keys, const vector<bool>& reds) {
    RBTree t;
    for (int k : keys) t.bstInsert(k);
    for (size_t i = 0; i < keys.size(); ++i) findNode(t.root, keys[i])->red = reds[i];
    return t;
}
void printTree(RBNode* n, const string& pre, bool isLeft) {
    if (n == NIL) return;
    cout << pre << (isLeft ? "|-- " : "`-- ") << label(n) << "\n";
    printTree(n->left,  pre + (isLeft ? "|   " : "    "), true);
    printTree(n->right, pre + (isLeft ? "|   " : "    "), false);
}

// ---------------- pemeriksa ----------------
// isBST() dari Soal 8.3, dalam bentuk sempit yang menggoda di subbab 8.6.7: batas int
// diawali INT_MIN dan INT_MAX. Sengaja dipertahankan untuk memperlihatkan jebakannya.
static bool isBST(RBNode* n, int lowerBound, int upperBound) {
    if (n == NIL) return true;
    if (n->key <= lowerBound || n->key >= upperBound) return false;
    return isBST(n->left, lowerBound, n->key) && isBST(n->right, n->key, upperBound);
}
bool isBST(const RBTree& t) { return t.root == NIL || isBST(t.root, INT_MIN, INT_MAX); }

// Perbaikannya: batas yang selangkah lebih lebar daripada int mana pun.
static bool isBSTFixed(RBNode* n, long long lo, long long hi) {
    if (n == NIL) return true;
    if (n->key <= lo || n->key >= hi) return false;
    return isBSTFixed(n->left, lo, n->key) && isBSTFixed(n->right, n->key, hi);
}
bool isBSTRepaired(const RBTree& t) {
    return isBSTFixed(t.root, (long long)INT_MIN - 1, (long long)INT_MAX + 1);
}
```

Kode 8.9 (Lanjutan) *Soal 8.11 sampai 8.15 dalam satu program: pemeriksa-pemeriksa merah-hitam, rotasi imperatif dengan penunjuk induk, serangkaian pemeriksaan, dan peragaan jebakan batas int pada §8.6.7.*

```
// Soal 8.11: countBlackNodes(). Penjaga NIL tidak dihitung.
static int countBlack(RBNode* n) {
    if (n == NIL) return 0;
    return (n->isBlack() ? 1 : 0) + countBlack(n->left) + countBlack(n->right);
}
int countBlackNodes(const RBTree& t) { return countBlack(t.root); }

// Soal 8.12: validateProperty4(): tak ada simpul MERAH yang beranak MERAH.
static bool prop4(RBNode* n) {
    if (n == NIL) return true;
    if (n->isRed() && (n->left->isRed() || n->right->isRed())) return false;
    return prop4(n->left) && prop4(n->right);
}
bool validateProperty4(const RBTree& t) { return prop4(t.root); }

// Soal 8.13: validateBlackHeight(). -1 adalah bendera kegagalan, sebab tinggi-hitam
// yang sebenarnya tak pernah negatif -- teknik penjaga Soal 8.6 sekali lagi.
int computeBlackHeight(RBNode* n) {
    if (n == NIL) return 0;
    int l = computeBlackHeight(n->left), r = computeBlackHeight(n->right);
    if (l == -1 || r == -1 || l != r) return -1;
    return l + (n->isBlack() ? 1 : 0);
}
bool validateBlackHeight(const RBTree& t) {
    return t.root == NIL || computeBlackHeight(t.root) != -1;
}

// Soal 8.14: isValidRBTree(): kelima sifat sekaligus.
bool isValidRBTree(const RBTree& t) {
    if (t.root == NIL)   return true;   // sifat 1 otomatis: penandanya bertipe bool
    if (t.root->isRed()) return false;  // sifat 2: akarnya hitam
    return isBST(t) && validateProperty4(t) && validateBlackHeight(t);
}                                       // sifat 3 berlaku karena cara NIL dibangun

// ---------------- Soal 8.15: rotasi ----------------
// leftRotate(t, x): anak kanan x, yaitu y, naik; x menjadi anak kiri y.
void leftRotate(RBTree& t, RBNode* x) {
    RBNode* y = x->right;                   // 1. tetapkan y
    x->right = y->left;                     // 2. subpohon kiri y menjadi kanan x
    if (y->left != NIL) y->left->parent = x;
    y->parent = x->parent;                  // 3. sambungkan induk x ke y
    if      (x->parent == NIL)     t.root = y;
    else if (x == x->parent->left) x->parent->left  = y;
    else                           x->parent->right = y;
    y->left = x;                            // 4. letakkan x di kiri y
    x->parent = y;
```

Kode 8.9 (Lanjutan) *Soal 8.11 sampai 8.15 dalam satu program: pemeriksa-pemeriksa merah-hitam, rotasi imperatif dengan penunjuk induk, serangkaian pemeriksaan, dan peragaan jebakan batas int pada §8.6.7.*

```
}
// rightRotate(t, y): bayangan cermin yang persis.
void rightRotate(RBTree& t, RBNode* y) {
    RBNode* x = y->left;
    y->left = x->right;
    if (x->right != NIL) x->right->parent = y;
    x->parent = y->parent;
    if      (y->parent == NIL)     t.root = x;
    else if (y == y->parent->left) y->parent->left  = x;
    else                           y->parent->right = x;
    x->right = y;
    y->parent = x;
}

// ---------------- pemeriksaan ----------------
int passed = 0, failed = 0;
void check(const string& name, bool cond) {
    (cond ? passed : failed)++;
    cout << (cond ? "  PASS: " : "  FAIL: ") << name << "\n";
}

int main() {
    RBTree valid = buildForTest({26, 17, 41, 14, 21, 47},
                                {false, true, false, false, false, true});
    cout << "The valid test tree:\n";
    printTree(valid.root, "", true);

    cout << "\nValidators (Problems 8.3, 8.11 to 8.14)\n";
    check("8.3  valid RB tree is a BST", isBST(valid));
    check("8.3  empty tree is a BST", isBST(RBTree()));
    RBTree bad = buildForTest({10, 5, 15}, {false, true, true});
    bad.root->left->key = 20;                       // anak kiri kini melampaui akarnya
    check("8.3  broken BST detected", !isBST(bad));
    check("8.11 four black nodes", countBlackNodes(valid) == 4);
    check("8.11 empty tree has none", countBlackNodes(RBTree()) == 0);
    check("8.12 no red-red in the valid tree", validateProperty4(valid));
    RBTree redRed = buildForTest({10, 5, 15}, {false, true, true});
    redRed.root->left->left = newNode(3);           // 3 MERAH di bawah 5 MERAH
    redRed.root->left->left->parent = redRed.root->left;
    check("8.12 red-red violation detected", !validateProperty4(redRed));
    check("8.13 uniform black-height", validateBlackHeight(valid));
    check("8.14 valid tree is a valid RB tree", isValidRBTree(valid));
    check("8.14 a red root is rejected", !isValidRBTree(buildForTest({10}, {true})));
    cout << "  black-height of the root = " << computeBlackHeight(valid.root) << "\n";
```

Kode 8.9 (Lanjutan) *Soal 8.11 sampai 8.15 dalam satu program: pemeriksa-pemeriksa merah-hitam, rotasi imperatif dengan penunjuk induk, serangkaian pemeriksaan, dan peragaan jebakan batas int pada §8.6.7.*

```
    cout << "\nThe int-bounds pitfall, on a single BLACK node holding INT_MIN\n";
    RBTree edge = buildForTest({INT_MIN}, {false});
    cout << "  isBST with int bounds          : " << (isBST(edge) ? "true" : "false") <<
"\n";
    cout << "  isBSTRepaired, wider bounds    : " << (isBSTRepaired(edge) ? "true" :
"false") << "\n";

    cout << "\nProblem 8.15: leftRotate on the chain 10 -> 20 -> 30\n";
    RBTree t1; for (int k : {10, 20, 30}) t1.bstInsert(k);
    t1.root->red = false;
    printTree(t1.root, "", true);
    leftRotate(t1, t1.root);
    cout << "  after leftRotate(10):\n";
    printTree(t1.root, "", true);
    check("root is now 20", t1.root->key == 20);
    check("20's children are 10 and 30",
          t1.root->left->key == 10 && t1.root->right->key == 30);
    check("10's parent pointer was fixed", t1.root->left->parent == t1.root);

    cout << "\nProblem 8.15: rightRotate on the chain 30 <- 20 <- 10\n";
    RBTree t2; for (int k : {30, 20, 10}) t2.bstInsert(k);
    t2.root->red = false;
    rightRotate(t2, t2.root);
    printTree(t2.root, "", true);
    check("root is now 20", t2.root->key == 20);
    check("20's children are 10 and 30",
          t2.root->left->key == 10 && t2.root->right->key == 30);
    check("30's parent pointer was fixed", t2.root->right->parent == t2.root);

    cout << "\nProblem 8.15: a non-root rotation, leftRotate(30)\n";
    RBTree t3 = buildForTest({20, 10, 30, 25, 35, 33, 40},
                             {false, false, true, false, false, true, true});
    leftRotate(t3, findNode(t3.root, 30));
    printTree(t3.root, "", true);
    check("root is still 20", t3.root->key == 20);
    check("20's right is now 35", t3.root->right->key == 35);
    check("35's left is 30", t3.root->right->left->key == 30);
    check("30's right is 33", t3.root->right->left->right->key == 33);

    cout << "\n=== " << passed << " passed, " << failed << " failed ===\n";
    return 0;
}
```

Ia mencetak:

```
The valid test tree:
|-- 26:B
|   |-- 17:R
|   |   |-- 14:B
|   |   `-- 21:B
|   `-- 41:B
|       `-- 47:R

Validators (Problems 8.3, 8.11 to 8.14)
  PASS: 8.3  valid RB tree is a BST
  PASS: 8.3  empty tree is a BST
  PASS: 8.3  broken BST detected
  PASS: 8.11 four black nodes
  PASS: 8.11 empty tree has none
  PASS: 8.12 no red-red in the valid tree
  PASS: 8.12 red-red violation detected
  PASS: 8.13 uniform black-height
  PASS: 8.14 valid tree is a valid RB tree
  PASS: 8.14 a red root is rejected
  black-height of the root = 2

The int-bounds pitfall, on a single BLACK node holding INT_MIN
  isBST with int bounds          : false
  isBSTRepaired, wider bounds    : true

Problem 8.15: leftRotate on the chain 10 -> 20 -> 30
|-- 10:B
|   `-- 20:R
|       `-- 30:R
  after leftRotate(10):
|-- 20:R
|   |-- 10:B
|   `-- 30:R
  PASS: root is now 20
  PASS: 20's children are 10 and 30
  PASS: 10's parent pointer was fixed

Problem 8.15: rightRotate on the chain 30 <- 20 <- 10
|-- 20:R
|   |-- 10:R
|   `-- 30:B
  PASS: root is now 20
  PASS: 20's children are 10 and 30
  PASS: 30's parent pointer was fixed

Problem 8.15: a non-root rotation, leftRotate(30)
|-- 20:B
|   |-- 10:B
|   `-- 35:B
|       |-- 30:R
|       |   |-- 25:B
|       |   `-- 33:R
|       `-- 40:R
  PASS: root is still 20
  PASS: 20's right is now 35
  PASS: 35's left is 30
  PASS: 30's right is 33

=== 20 passed, 0 failed ===
```

Periksalah tinggi-hitamnya dengan tangan terhadap pohon yang tercetak. Menyusuri kirinya: `26:B` hitam, `17:R` merah, `14:B` hitam, lalu `NIL` — dua simpul hitam, menghitung 26 dan 14 tetapi tidak 17 yang merah dan tidak `NIL` itu sendiri, jadi tinggi-hitamnya 2. Menyusuri kanannya: `26:B`, `41:B`, `47:R`, `NIL` — lagi-lagi dua simpul hitam, 26 dan 41. Setiap lintasan memberikan 2, dan itulah

yang dicetak programnya. Ada empat simpul hitam di dalam pohonnya seluruhnya — 26, 41, 14, 21 — dan itulah yang dilaporkan Soal 8.11.

Lalu perhatikan rotasi bukan-akarnya: sebelum `leftRotate(30)` subpohon kanannya adalah $30\{25,35\{33,40\}\}$; sesudahnya ia menjadi $35\{30\{25,33\},40\}$. Akar 20 tak tersentuh, 33 berpindah dari menjadi anak kiri 35 menjadi anak kanan 30, dan bacaan inordernya 10,20,25,30,33,35,40 tidak berubah — dan itulah teorema **Gambar 8.7**, diverifikasi pada pohon yang konkret.

8.15 Soal 8.16 — Perkalian Rantai Matriks, Tabelnya Diisi dengan Tangan

8.15.1 Soalnya

Lima matriks hendak dikalikan:

$$A_1 : 4 \times 10, \quad A_2 : 10 \times 3, \quad A_3 : 3 \times 12, \quad A_4 : 12 \times 20, \quad A_5 : 20 \times 7.$$

1. Tuliskan barisan dimensi d, lalu isilah tabel biaya M dan tabel pemisah c dari `MATRIX-CHAIN-ORDER` dengan tangan, sel demi sel, dalam urutan pengisian algoritmanya.
2. Bacalah pengurungan optimalnya dari c.
3. Pastikan biaya optimalnya dengan menjabarkan pengurungan itu, lalu bandingkan dengan urutan naif dari kiri ke kanan.
4. Hitunglah panggilan yang dilakukan penyelesaian rekursif polos dan penyelesaian bermemo, lalu turunkan kedua hitungan itu dalam bentuk tertutup.

8.15.2 Yang Anda perlukan dari Bab 7

Mengalikan matriks $p \times q$ dengan matriks $q \times r$ memerlukan $p\,q\,r$ perkalian skalar dan menghasilkan matriks $p \times r$. Untuk sebuah rantai, dimensinya didaftar sekali saja sebagai $d = \langle d_0, d_1, \ldots, d_n \rangle$, dengan A_i berukuran $d_{i-1} \times d_i$. Misalkan $M[i,j]$ adalah biaya terkecil untuk menghitung $A_i A_{i+1} \cdots A_j$. Perkalian terakhirnya membelah rantai itu di suatu k dengan $i \le k < j$ menjadi $A_i \cdots A_k$ (matriks $d_{i-1} \times d_k$) dan $A_{k+1} \cdots A_j$ (matriks $d_k \times d_j$), dan substruktur optimal memberikan

$$M[i,j] = \begin{cases} 0 & \text{jika } i = j, \\ \min\limits_{i \le k < j} \{M[i,k] + M[k+1,j] + d_{i-1}\, d_k\, d_j\} & \text{jika } i < j. \end{cases} \tag{8.18}$$

Tabel pemisah $c[i,j]$ mencatat k yang mencapai minimum itu. **Pseudokode 8.5** mengisi kedua tabel menurut panjang rantai l yang naik, sehingga setiap entri yang diperlukan sebuah sel sudah dihitung lebih dahulu.

Pseudokode 8.5 `MATRIX-CHAIN-ORDER`: setiap subrantai diselesaikan sekali, yang terpendek lebih dahulu, dan pemisah pemenangnya dicatat.

```
MATRIX-CHAIN-ORDER(d, n)
1  for i = 1 to n
2      M[i, i] = 0
3  for l = 2 to n                        // l adalah panjang rantai
4      for i = 1 to n - l + 1
5          j = i + l - 1
6          M[i, j] = ∞
7          for k = i to j - 1
8              q = M[i, k] + M[k + 1, j] + d[i - 1] · d[k] · d[j]
9              if q < M[i, j]
10                 M[i, j] = q
11                 c[i, j] = k
12 return M and c
```

8.15.3 Langkah 1: dimensi dan diagonalnya

Membaca ukuran-ukurannya berurutan: A_1 berukuran 4×10, jadi $d_0 = 4$ dan $d_1 = 10$; A_2 berukuran 10×3, jadi $d_2 = 3$; lalu $d_3 = 12$, $d_4 = 20$, $d_5 = 7$. Karena itu

$$d = \langle 4,10,3,12,20,7 \rangle, \qquad n = 5.$$

Baris 1–2 menetapkan $M[1,1] = M[2,2] = \cdots = M[5,5] = 0$: "rantai" satu matriks tidak memerlukan perkalian.

8.15.4 Langkah 2: rantai sepanjang 2

Rantai dua matriks hanya punya satu pemisah, jadi setiap sel adalah satu hasil kali $d_{i-1}d_id_{i+1}$.

- $M[1,2]$, $k = 1$: $0 + 0 + d_0d_1d_2 = 4 \cdot 10 \cdot 3 = 120$. Jadi $M[1,2] = 120$, $c[1,2] = 1$.
- $M[2,3]$, $k = 2$: $d_1d_2d_3 = 10 \cdot 3 \cdot 12 = 360$. $M[2,3] = 360$, $c[2,3] = 2$.
- $M[3,4]$, $k = 3$: $d_2d_3d_4 = 3 \cdot 12 \cdot 20 = 720$. $M[3,4] = 720$, $c[3,4] = 3$.
- $M[4,5]$, $k = 4$: $d_3d_4d_5 = 12 \cdot 20 \cdot 7 = 1\,680$. $M[4,5] = 1\,680$, $c[4,5] = 4$.

8.15.5 Langkah 3: rantai sepanjang 3

Setiap sel kini punya dua calon pemisah. Setiap suku di ruas kanan adalah sel sepanjang 1 atau 2, yang sudah terisi.

$M[1,3]$, rantai $A_1A_2A_3$:

- $k = 1$, yaitu $A_1(A_2A_3)$: $M[1,1] + M[2,3] + d_0d_1d_3 = 0 + 360 + 4 \cdot 10 \cdot 12 = 360 + 480 = 840$.
- $k = 2$, yaitu $(A_1A_2)A_3$: $M[1,2] + M[3,3] + d_0d_2d_3 = 120 + 0 + 4 \cdot 3 \cdot 12 = 120 + 144 = 264$.

Yang lebih kecil adalah 264: $M[1,3] = 264$, $c[1,3] = 2$. Mengalikan A_1A_2 lebih dahulu mengecilkan 10 menjadi 3, dan setiap perkalian sesudahnya menjadi lebih murah karenanya.

$M[2,4]$, rantai $A_2A_3A_4$:

- $k = 2$: $0 + M[3,4] + d_1d_2d_4 = 720 + 10 \cdot 3 \cdot 20 = 720 + 600 = 1\,320$.
- $k = 3$: $M[2,3] + 0 + d_1d_3d_4 = 360 + 10 \cdot 12 \cdot 20 = 360 + 2\,400 = 2\,760$.

$M[2,4] = 1\,320$, $c[2,4] = 2$.

$M[3,5]$, rantai $A_3A_4A_5$:

- $k = 3$: $0 + M[4,5] + d_2d_3d_5 = 1\,680 + 3 \cdot 12 \cdot 7 = 1\,680 + 252 = 1\,932$.
- $k = 4$: $M[3,4] + 0 + d_2d_4d_5 = 720 + 3 \cdot 20 \cdot 7 = 720 + 420 = 1\,140$.

$M[3,5] = 1\,140$, $c[3,5] = 4$.

8.15.6 Langkah 4: rantai sepanjang 4

Masing-masing tiga calon.

$M[1,4]$, rantai $A_1A_2A_3A_4$:

- $k = 1$: $0 + M[2,4] + d_0d_1d_4 = 1\,320 + 4 \cdot 10 \cdot 20 = 1\,320 + 800 = 2\,120$.
- $k = 2$: $M[1,2] + M[3,4] + d_0d_2d_4 = 120 + 720 + 4 \cdot 3 \cdot 20 = 840 + 240 = 1\,080$.
- $k = 3$: $M[1,3] + 0 + d_0d_3d_4 = 264 + 4 \cdot 12 \cdot 20 = 264 + 960 = 1\,224$.

$M[1,4] = 1\,080$, $c[1,4] = 2$.

$M[2,5]$, rantai $A_2A_3A_4A_5$:

- $k = 2$: $0 + M[3,5] + d_1d_2d_5 = 1\,140 + 10 \cdot 3 \cdot 7 = 1\,140 + 210 = 1\,350$.
- $k = 3$: $M[2,3] + M[4,5] + d_1d_3d_5 = 360 + 1\,680 + 10 \cdot 12 \cdot 7 = 2\,040 + 840 = 2\,880$.
- $k = 4$: $M[2,4] + 0 + d_1d_4d_5 = 1\,320 + 10 \cdot 20 \cdot 7 = 1\,320 + 1\,400 = 2\,720$.

$M[2,5] = 1\,350$, $c[2,5] = 2$.

8.15.7 Langkah 5: seluruh rantai

$M[1,5]$, empat calon:

- $k = 1$: $0 + M[2,5] + d_0d_1d_5 = 1\,350 + 4 \cdot 10 \cdot 7 = 1\,350 + 280 = 1\,630$.
- $k = 2$: $M[1,2] + M[3,5] + d_0d_2d_5 = 120 + 1\,140 + 4 \cdot 3 \cdot 7 = 1\,260 + 84 = \mathbf{1\,344}$.
- $k = 3$: $M[1,3] + M[4,5] + d_0d_3d_5 = 264 + 1\,680 + 4 \cdot 12 \cdot 7 = 1\,944 + 336 = 2\,280$.
- $k = 4$: $M[1,4] + 0 + d_0d_4d_5 = 1\,080 + 4 \cdot 20 \cdot 7 = 1\,080 + 560 = 1\,640$.

$M[1,5] = 1\,344$, $c[1,5] = 2$. **Tabel 8.10** dan **Tabel 8.11** memperlihatkan tabel-tabel yang sudah jadi.

Tabel 8.10 Tabel biaya M untuk $d = \langle 4,10,3,12,20,7 \rangle$. Setiap diagonal diisi sebelum diagonal berikutnya; jawabannya ada di pojok kanan atas, $M[1,5] = 1\,344$.

$M[i,j]$	j=1	j=2	j=3	j=4	j=5
$i = 1$	0	120	264	1 080	**1 344**
$i = 2$		0	360	1 320	1 350
$i = 3$			0	720	1 140
$i = 4$				0	1 680
$i = 5$					0

Tabel 8.11 Tabel pemisah c untuk rantai yang sama. Kedua entri tebal adalah entri yang dibaca oleh rekonstruksi di §8.15.8.

$c[i,j]$	j=2	j=3	j=4	j=5
$i = 1$	1	2	2	**2**
$i = 2$		2	2	2
$i = 3$			3	**4**
$i = 4$				4

8.15.8 Langkah 6: membaca pengurungannya, dan memeriksanya

`PRINT-OPTIMAL-PARENS` dari Bab 7 membaca c dari atas:

- $c[1,5] = 2$: perkalian terakhirnya adalah $(A_1A_2) \cdot (A_3A_4A_5)$.
- Bagian kiri A_1A_2 hanya punya satu pengurungan, (A_1A_2).
- Bagian kanan: $c[3,5] = 4$, jadi ia adalah $(A_3A_4) \cdot A_5$.

Karena itu urutan optimalnya adalah $((A_1A_2)((A_3A_4)A_5))$. Jabarkanlah untuk memastikan biayanya, sambil melacak ukuran setiap hasil antara:

1. A_1A_2: $(4 \times 10)(10 \times 3)$ berbiaya $4 \cdot 10 \cdot 3 = 120$ dan menghasilkan matriks 4×3.
2. A_3A_4: $(3 \times 12)(12 \times 20)$ berbiaya $3 \cdot 12 \cdot 20 = 720$ dan menghasilkan 3×20.
3. $(A_3A_4)A_5$: $(3 \times 20)(20 \times 7)$ berbiaya $3 \cdot 20 \cdot 7 = 420$ dan menghasilkan 3×7.
4. Perkalian terakhir: $(4 \times 3)(3 \times 7)$ berbiaya $4 \cdot 3 \cdot 7 = 84$ dan menghasilkan 4×7.

Jumlah: $120 + 720 + 420 + 84 = 1\,344$ ✓, persis $M[1,5]$.

Bandingkan dengan urutan yang akan dipakai program tanpa berpikir, kiri ke kanan, $(((A_1A_2)A_3)A_4)A_5$: 120 untuk A_1A_2, lalu $4 \cdot 3 \cdot 12 = 144$, lalu $4 \cdot 12 \cdot 20 = 960$, lalu $4 \cdot 20 \cdot 7 = 560$, seluruhnya 1 784 — sepertiga lebih banyak kerja untuk hasil kali yang persis sama. Urutan optimal menang dengan menjaga dimensi kecil 3 tetap di bagian dalam selama mungkin.

8.15.9 Langkah 7: rekursi polos lawan *memoization*

Terjemahkan Persamaan (8.18) langsung menjadi fungsi rekursif, dan fungsi itu menghitung ulang subrantai yang sama berkali-kali. Misalkan $R(n)$ adalah banyaknya panggilan yang dilakukannya pada rantai n matriks. Rantai satu matriks melakukan satu panggilan; rantai yang lebih panjang melakukan panggilannya sendiri ditambah, untuk setiap $n - 1$ pemisah, satu panggilan pada bagian kiri sepanjang k dan satu pada bagian kanan sepanjang $n - k$:

$$R(1) = 1, \qquad R(n) = 1 + \sum_{k=1}^{n-1} (R(k) + R(n-k)) = 1 + 2\sum_{k=1}^{n-1} R\,(k).$$

Kurangkan persamaan yang sama untuk $n - 1$: $R(n) - R(n-1) = 2R(n-1)$, jadi $R(n) = 3R(n-1)$ untuk $n \geq 3$; dan secara langsung $R(2) = 1 + 2 \cdot 1 = 3$. Karena itu $R(n) = 3^{n-1}$: untuk $n = 5$ kita, $3^4 = 81$ panggilan, dan untuk $n = 17$, lebih dari empat puluh tiga juta.

Versi bermemo dari Bab 7 menyimpan setiap $M[i,j]$ pada kali pertama ia dihitung. Setiap pasangan $i < j$ lalu *dihitung* sekali, dan perhitungannya melakukan $2(j - i)$ panggilan, dua per pemisah; tambahkan 1 untuk panggilan teratas. Dengan mengelompokkan pasangan-pasangan itu menurut selisihnya $\delta = j - i$, ada $n - \delta$ pasangan dengan selisih itu, sehingga jumlahnya adalah

$$1 + \sum_{\delta=1}^{n-1} 2\,\delta(n - \delta) = 1 + \frac{n^3 - n}{3}. \tag{8.19}$$

Untuk $n = 5$: $2(1 \cdot 4 + 2 \cdot 3 + 3 \cdot 2 + 4 \cdot 1) = 2 \cdot 20 = 40$, ditambah 1, adalah 41 panggilan. Eksponensial lawan kubik — seluruh Bab 7 dalam dua bilangan.

8.15.10 Programnya

Kode 8.10 mengimplementasikan **Pseudokode 8.5** dengan pelacakan setiap calon, mencetak kedua tabel dan pengurungannya, menghitung panggilan kedua penyelesaian rekursif, dan menyelesaikan rantai periksa-diri di §8.15.11.

Kode 8.10 Soal 8.16 selengkapnya: `MATRIX-CHAIN-ORDER` dengan pelacakan setiap calon pemisah, `PRINT-OPTIMAL-PARENS`, dan hitungan panggilan rekursi polos dan rekursi bermemo.

```
// mcmtable.cpp -- Soal 8.16: MATRIX-CHAIN-ORDER diisi sel demi sel, pengurungan optimal
// dibaca dari tabel pemisah, dan memoisasi dibandingkan dengan rekursi polos.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <climits>
using namespace std;

vector<long long> d;                    // dimensi: A_i berukuran d[i-1] x d[i]
vector<vector<long long>> M;            // M[i][j] = biaya terkecil untuk A_i..A_j
vector<vector<int>> c;                  // c[i][j] = pemisah k yang mencapainya
long long calls = 0;                    // menghitung setiap panggilan pemecah rekursif

void matrixChainOrder(int n, bool trace) {
    M.assign(n + 1, vector<long long>(n + 1, 0));
    c.assign(n + 1, vector<int>(n + 1, 0));
    for (int l = 2; l <= n; ++l) {                  // panjang rantai, yang terpendek dulu
        if (trace) cout << "length " << l << "\n";
        for (int i = 1; i <= n - l + 1; ++i) {
            int j = i + l - 1;
            M[i][j] = LLONG_MAX;
            for (int k = i; k <= j - 1; ++k) {
                long long q = M[i][k] + M[k + 1][j] + d[i - 1] * d[k] * d[j];
                if (trace)
                    cout << "  M[" << i << "," << j << "] k=" << k << ": " << M[i][k] << "
+ "
                         << M[k + 1][j] << " + " << d[i - 1] << "*" << d[k] << "*" << d[j]
                         << " = " << q << (q < M[i][j] ? "  <- best so far" : "") << "\n";
                if (q < M[i][j]) { M[i][j] = q; c[i][j] = k; }
            }
        }
    }
}

string parens(int i, int j) {                       // PRINT-OPTIMAL-PARENS
    if (i == j) return "A" + to_string(i);
    return "(" + parens(i, c[i][j]) + parens(c[i][j] + 1, j) + ")";
}
```

Kode 8.10 (Lanjutan) *Soal 8.16 selengkapnya: MATRIX-CHAIN-ORDER dengan pelacakan setiap calon pemisah, PRINT-OPTIMAL-PARENS, dan hitungan panggilan rekursi polos dan rekursi bermemo.*

```
// Rekursi polos: mencoba setiap pemisah dan melupakan setiap jawaban.
long long naive(int i, int j) {
    ++calls;
    if (i == j) return 0;
    long long best = LLONG_MAX;
    for (int k = i; k < j; ++k)
        best = min(best, naive(i, k) + naive(k + 1, j) + d[i - 1] * d[k] * d[j]);
    return best;
}

// Rekursi bermemo: kode yang sama, tetapi tiap jawaban disimpan pada kali pertama.
vector<vector<long long>> memo;
long long lookup(int i, int j) {
    ++calls;
    if (memo[i][j] >= 0) return memo[i][j];        // sudah diketahui: tanpa kerja
    long long best = 0;
    if (i < j) {
        best = LLONG_MAX;
        for (int k = i; k < j; ++k)
            best = min(best, lookup(i, k) + lookup(k + 1, j) + d[i - 1] * d[k] * d[j]);
    }
    return memo[i][j] = best;
}

void printTable(int n, bool costs) {
    cout << (costs ? "  M   " : "  c   ");
    for (int j = 1; j <= n; ++j) cout << setw(7) << j;
    cout << "\n";
    for (int i = 1; i <= n; ++i) {
        cout << "  " << setw(2) << i << "  ";
        for (int j = 1; j <= n; ++j) {
            if (j < i || (!costs && j == i)) cout << setw(7) << "-";
            else cout << setw(7) << (costs ? M[i][j] : (long long)c[i][j]);
        }
        cout << "\n";
    }
}
```

Kode 8.10 (Lanjutan) *Soal 8.16 selengkapnya: MATRIX-CHAIN-ORDER dengan pelacakan setiap calon pemisah, PRINT-OPTIMAL-PARENS, dan hitungan panggilan rekursi polos dan rekursi bermemo.*

```
void solve(const vector<long long>& dims, bool trace) {
    d = dims;
    int n = (int)d.size() - 1;
    cout << "d = <";
    for (size_t k = 0; k < d.size(); ++k) cout << (k ? ", " : "") << d[k];
    cout << ">, n = " << n << "\n";
    matrixChainOrder(n, trace);
    printTable(n, true);
    printTable(n, false);
    cout << "  least cost M[1," << n << "] = " << M[1][n]
         << ", bracketing " << parens(1, n) << "\n";
}

int main() {
    solve({4, 10, 3, 12, 20, 7}, true);

    cout << "\nMemoization against plain recursion on the same chain\n";
    int n = 5;
    calls = 0; long long a = naive(1, n);
    cout << "  plain recursion : cost " << a << " after " << calls << " calls\n";
    memo.assign(n + 1, vector<long long>(n + 1, -1));
    calls = 0; long long b = lookup(1, n);
    cout << "  memoized        : cost " << b << " after " << calls << " calls\n";

    cout << "\nGrowth of the call counts (dimensions do not matter for the count)\n";
    cout << "   n   plain recursion   memoized\n";
    for (int m : {5, 8, 11, 14, 17}) {
        d.assign(m + 1, 2);
        calls = 0; naive(1, m); long long p = calls;
        memo.assign(m + 1, vector<long long>(m + 1, -1));
        calls = 0; lookup(1, m); long long q = calls;
        cout << setw(4) << m << setw(18) << p << setw(11) << q << "\n";
    }

    cout << "\nCheck yourself\n";
    solve({2, 3, 6, 4, 5}, false);
    return 0;
}
```

Ia mencetak:

```
d = <4, 10, 3, 12, 20, 7>, n = 5
length 2
  M[1,2] k=1: 0 + 0 + 4*10*3 = 120  <- best so far
  M[2,3] k=2: 0 + 0 + 10*3*12 = 360  <- best so far
  M[3,4] k=3: 0 + 0 + 3*12*20 = 720  <- best so far
  M[4,5] k=4: 0 + 0 + 12*20*7 = 1680  <- best so far
length 3
  M[1,3] k=1: 0 + 360 + 4*10*12 = 840  <- best so far
  M[1,3] k=2: 120 + 0 + 4*3*12 = 264  <- best so far
  M[2,4] k=2: 0 + 720 + 10*3*20 = 1320  <- best so far
  M[2,4] k=3: 360 + 0 + 10*12*20 = 2760
  M[3,5] k=3: 0 + 1680 + 3*12*7 = 1932  <- best so far
  M[3,5] k=4: 720 + 0 + 3*20*7 = 1140  <- best so far
length 4
  M[1,4] k=1: 0 + 1320 + 4*10*20 = 2120  <- best so far
  M[1,4] k=2: 120 + 720 + 4*3*20 = 1080  <- best so far
  M[1,4] k=3: 264 + 0 + 4*12*20 = 1224
  M[2,5] k=2: 0 + 1140 + 10*3*7 = 1350  <- best so far
  M[2,5] k=3: 360 + 1680 + 10*12*7 = 2880
  M[2,5] k=4: 1320 + 0 + 10*20*7 = 2720
length 5
  M[1,5] k=1: 0 + 1350 + 4*10*7 = 1630  <- best so far
  M[1,5] k=2: 120 + 1140 + 4*3*7 = 1344  <- best so far
  M[1,5] k=3: 264 + 1680 + 4*12*7 = 2280
  M[1,5] k=4: 1080 + 0 + 4*20*7 = 1640
  M         1      2      3      4      5
   1        0    120    264   1080   1344
   2        -      0    360   1320   1350
   3        -      -      0    720   1140
   4        -      -      -      0   1680
   5        -      -      -      -      0
  c         1      2      3      4      5
   1        -      1      2      2      2
   2        -      -      2      2      2
   3        -      -      -      3      4
   4        -      -      -      -      4
   5        -      -      -      -      -
  least cost M[1,5] = 1344, bracketing ((A1A2)((A3A4)A5))

Memoization against plain recursion on the same chain
  plain recursion : cost 1344 after 81 calls
  memoized        : cost 1344 after 41 calls

Growth of the call counts (dimensions do not matter for the count)
   n   plain recursion   memoized
   5                81         41
   8              2187        169
  11             59049        441
  14           1594323        911
  17          43046721       1633

Check yourself
d = <2, 3, 6, 4, 5>, n = 4
  M         1      2      3      4
   1        0     36     84    124
   2        -      0     72    132
   3        -      -      0    120
   4        -      -      -      0
  c         1      2      3      4
   1        -      1      2      3
   2        -      -      2      3
   3        -      -      -      3
   4        -      -      -      -
  least cost M[1,4] = 124, bracketing (((A1A2)A3)A4)
```

Setiap baris "best so far" adalah salah satu calon di §8.15.4 sampai §8.15.7, dalam urutan yang sama, dan kedua tabel yang tercetak adalah **Tabel 8.10** dan **Tabel 8.11**. Hitungan panggilannya adalah 3^{n-1} dan Persamaan (8.19) pada setiap ukuran: 81 dan 41 untuk $n = 5$, serta 43 046 721 lawan 1 633 untuk $n = 17$.

8.15.11 Periksa diri

Selesaikan rantai $d = \langle 2,3,6,4,5 \rangle$: empat matriks, berukuran 2×3, 3×6, 6×4, dan 4×5.

> **JAWABAN.** Panjang 2: $M[1,2] = 2 \cdot 3 \cdot 6 = 36$, $M[2,3] = 3 \cdot 6 \cdot 4 = 72$, $M[3,4] = 6 \cdot 4 \cdot 5 = 120$. Panjang 3: $M[1,3] = \min(0 + 72 + 2 \cdot 3 \cdot 4,\ 36 + 0 + 2 \cdot 6 \cdot 4) = \min(96,84) = 84$ dengan $c[1,3] = 2$; $M[2,4] = \min(0 + 120 + 3 \cdot 6 \cdot 5,\ 72 + 0 + 3 \cdot 4 \cdot 5) = \min(210,132) = 132$ dengan $c[2,4] = 3$. Panjang 4: $M[1,4] = \min(0 + 132 + 2 \cdot 3 \cdot 5,\ 36 + 120 + 2 \cdot 6 \cdot 5,\ 84 + 0 + 2 \cdot 4 \cdot 5) = \min(162,216,124) = 124$ dengan $c[1,4] = 3$. **Biaya terkecil** 124**, pengurungan** $(((A_1A_2)A_3)A_4)$ — di sini urutan kiri ke kanan kebetulan optimal, sebab dimensi pertama 2 adalah yang terkecil dan setiap perkalian menjaganya tetap di sisi luar. Blok terakhir keluaran **Kode 8.10** menegaskan kedua tabelnya.

8.15.12 Belokan keliru yang sering terjadi pada soal ini

1. **Mengisi tabel baris demi baris.** $M[1,4]$ memerlukan $M[2,4]$, yang berada di baris sesudahnya. Gelung terluarnya harus berjalan atas panjang rantai l.
2. **Membiarkan k mencapai j.** Rentangnya $i \le k < j$; dengan $k = j$ bagian kanannya kosong.
3. **Memakai d_i alih-alih d_{i-1}.** Faktor kiri $A_i \cdots A_k$ memiliki d_{i-1} baris. Itulah sebabnya d berisi $n + 1$ entri dan dimulai di indeks 0.
4. **Hanya menyimpan M.** Biayanya saja tidak mengatakan cara mengurungnya. Simpanlah c, atau Anda harus menelusuri tabelnya lagi untuk membangun ulang jawabannya.
5. **Mengira *memoization* mengubah jawabannya.** Ia hanya mengubah banyaknya kerja. Kedua versi rekursif di **Kode 8.10** mengembalikan 1 344.

8.16 Soal 8.17 — *Longest Common Subsequence*, Tabelnya dan Jalan Mundurnya

8.16.1 Soalnya

Misalkan $X = $ SORTED dan $Y = $ STORE.

1. Isilah tabel panjang c dan tabel panah b dari `LCS-LENGTH` dengan tangan, baris demi baris, satu sel per baris. Putuskan seri dengan bergerak ke atas, seperti Bab 7.
2. Ikutilah panah-panahnya mundur dari pojok kanan bawah dan tuliskan subbarisan bersama terpanjang yang dihasilkannya.
3. Apakah itu satu-satunya subbarisan bersama terpanjang?
4. Nyatakan biaya setiap fase dalam $m = |X|$ dan $n = |Y|$.

8.16.2 Yang Anda perlukan dari Bab 7

Sebuah **subbarisan** (*subsequence*) dari sebuah untai adalah apa yang tersisa setelah mencoret sebagian karakternya dan mempertahankan sisanya dalam urutan semula; sebuah **subbarisan bersama** dari X dan Y adalah subbarisan dari keduanya. Misalkan X_i adalah awalan $x_1 \cdots x_i$ dan

$c[i, j]$ panjang *longest common subsequence* (subbarisan bersama terpanjang, LCS) dari X_i dan Y_j. Substruktur optimal memberikan

$$c[i,j] = \begin{cases} 0 & \text{jika } i = 0 \text{ atau } j = 0, \\ c[i-1, j-1] + 1 & \text{jika } i, j > 0 \text{ dan } x_i = y_j, \\ \max\{c[i-1, j],\ c[i, j-1]\} & \text{jika } i, j > 0 \text{ dan } x_i \neq y_j. \end{cases} \tag{8.20}$$

Dengan kata-kata: jika karakter terakhirnya cocok, karakter itu mengakhiri sebuah LCS dari kedua awalan, dan kita menambahkan satu pada jawaban untuk awalan-awalan tanpa karakter itu; jika tidak cocok, paling sedikit satu dari kedua karakter terakhir itu tak berguna, dan kita mengambil yang lebih baik antara membuang x_i atau membuang y_j. **Pseudokode 8.6** mengisi c baris demi baris dan mencatat di b kasus mana yang berlaku: ↖ untuk kecocokan (nilai dari diagonal), ↑ untuk "buang x_i" (nilai dari atas), ← untuk "buang y_j" (nilai dari kiri). Seri diputuskan ke atas.

Pseudokode 8.6 `LCS-LENGTH`: tabel panjang dan tabel panah, diisi baris demi baris, dengan seri diputuskan ke atas.

```
LCS-LENGTH(X, Y, m, n)
for i = 0 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i - 1, j - 1] + 1;  b[i, j] = "↖"
        elseif c[i - 1, j] ≥ c[i, j - 1]
            c[i, j] = c[i - 1, j];          b[i, j] = "↑"
        else
            c[i, j] = c[i, j - 1];          b[i, j] = "←"
return c and b
```

`PRINT-LCS`, **Pseudokode 8.7**, dimulai di pojok kanan bawah dan mengikuti panah-panahnya mundur. Setiap ↖ menyumbangkan karakternya; karena jalannya mundur, karakter-karakter itu ditemukan dari yang terakhir dan harus dicetak terbalik.

Pseudokode 8.7 `PRINT-LCS`: ikuti panah-panahnya mundur; setiap panah diagonal adalah satu karakter jawaban.

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i - 1, j - 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i - 1, j)
else PRINT-LCS(b, X, i, j - 1)
```

8.16.3 Langkah 1: tabelnya, satu sel per baris

Beri nomor karakter-karakternya: $X = x_1 x_6 = $ S, O, R, T, E, D dan $Y = y_1 y_5 = $ S, T, O, R, E. Baris 0 dan kolom 0 seluruhnya 0. Untuk setiap sel kita bandingkan x_i dengan y_j; bila cocok kita ambil diagonal ditambah satu, bila tidak kita ambil yang lebih besar antara "atas" $c[i-1, j]$ dan "kiri" $c[i, j-1]$, dengan bergerak ke atas bila seri.

Baris 1, $x_1 = $ S.

- $c[1,1]$: S = S, cocok. Diagonal $c[0,0] = 0$, jadi $c[1,1] = 1$, ↖.
- $c[1,2]$: S ≠ T. Atas $c[0,2] = 0$, kiri $c[1,1] = 1$. Kiri lebih besar: 1, ←.
- $c[1,3]$: S ≠ O. Atas 0, kiri 1: 1, ←.
- $c[1,4]$: S ≠ R. Atas 0, kiri 1: 1, ←.
- $c[1,5]$: S ≠ E. Atas 0, kiri 1: 1, ←.

Baris 2, $x_2 = $ O.

- $c[2,1]$: O ≠ S. Atas $c[1,1] = 1$, kiri $c[2,0] = 0$: 1, ↑.
- $c[2,2]$: O ≠ T. Atas 1, kiri 1 — seri, jadi ke atas: 1, ↑.
- $c[2,3]$: O = O, cocok. Diagonal $c[1,2] = 1$: $c[2,3] = 2$, ↖.
- $c[2,4]$: O ≠ R. Atas $c[1,4] = 1$, kiri $c[2,3] = 2$: 2, ←.
- $c[2,5]$: O ≠ E. Atas 1, kiri 2: 2, ←.

Baris 3, $x_3 = $ R.

- $c[3,1]$: R ≠ S. Atas 1, kiri 0: 1, ↑.
- $c[3,2]$: R ≠ T. Atas 1, kiri 1, seri: 1, ↑.
- $c[3,3]$: R ≠ O. Atas $c[2,3] = 2$, kiri $c[3,2] = 1$: 2, ↑.
- $c[3,4]$: R = R, cocok. Diagonal $c[2,3] = 2$: $c[3,4] = 3$, ↖.
- $c[3,5]$: R ≠ E. Atas $c[2,5] = 2$, kiri $c[3,4] = 3$: 3, ←.

Baris 4, $x_4 = $ T.

- $c[4,1]$: T ≠ S. Atas 1, kiri 0: 1, ↑.
- $c[4,2]$: T = T, cocok. Diagonal $c[3,1] = 1$: $c[4,2] = 2$, ↖.
- $c[4,3]$: T ≠ O. Atas $c[3,3] = 2$, kiri $c[4,2] = 2$, seri: 2, ↑.
- $c[4,4]$: T ≠ R. Atas $c[3,4] = 3$, kiri 2: 3, ↑.
- $c[4,5]$: T ≠ E. Atas $c[3,5] = 3$, kiri 3, seri: 3, ↑.

Baris 5, $x_5 = $ E.

- $c[5,1]$: E ≠ S. Atas 1, kiri 0: 1, ↑.
- $c[5,2]$: E ≠ T. Atas $c[4,2] = 2$, kiri 1: 2, ↑.
- $c[5,3]$: E ≠ O. Atas 2, kiri 2, seri: 2, ↑.
- $c[5,4]$: E ≠ R. Atas 3, kiri 2: 3, ↑.
- $c[5,5]$: E = E, cocok. Diagonal $c[4,4] = 3$: $c[5,5] = 4$, ↖.

Baris 6, $x_6 = $ D. D tidak muncul di mana pun dalam Y, jadi tak satu sel pun di baris ini yang cocok.

- $c[6,1]$: atas 1, kiri 0: 1, ↑.
- $c[6,2]$: atas 2, kiri 1: 2, ↑.
- $c[6,3]$: atas 2, kiri 2, seri: 2, ↑.
- $c[6,4]$: atas 3, kiri 2: 3, ↑.
- $c[6,5]$: atas $c[5,5] = 4$, kiri 3: 4, ↑.

Panjang sebuah subbarisan bersama terpanjang adalah $c[6,5] = 4$. **Tabel 8.12** memperlihatkan tabel-tabel yang sudah jadi, setiap sel membawa nilai dan panahnya.

Tabel 8.12 Tabel c dan b untuk $X = $ SORTED dan $Y = $ STORE. Sel-sel tebal adalah keempat panah diagonal yang dilewati jalan mundur di §8.16.4.

	j=0	j=1 (S)	j=2 (T)	j=3 (O)	j=4 (R)	j=5 (E)
$i = 0$	0	0	0	0	0	0
$i = 1$ (S)	0	**1** ↖	1 ←	1 ←	1 ←	1 ←
$i = 2$ (O)	0	1 ↑	1 ↑	**2** ↖	2 ←	2 ←
$i = 3$ (R)	0	1 ↑	1 ↑	2 ↑	**3** ↖	3 ←
$i = 4$ (T)	0	1 ↑	2 ↖	2 ↑	3 ↑	3 ↑
$i = 5$ (E)	0	1 ↑	2 ↑	2 ↑	3 ↑	**4** ↖
$i = 6$ (D)	0	1 ↑	2 ↑	2 ↑	3 ↑	4 ↑

8.16.4 Langkah 2: jalan mundurnya

Mulailah di $(i, j) = (6,5)$ dan ikuti panah-panahnya sampai salah satu indeks mencapai 0:

1. (6,5): ↑. Pindah ke (5,5).
2. (5,5): ↖. **Ambil $x_5 = $ E.** Pindah ke (4,4).
3. (4,4): ↑. Pindah ke (3,4).
4. (3,4): ↖. **Ambil $x_3 = $ R.** Pindah ke (2,3).
5. (2,3): ↖. **Ambil $x_2 = $ O.** Pindah ke (1,2).
6. (1,2): ←. Pindah ke (1,1).
7. (1,1): ↖. **Ambil $x_1 = $ S.** Pindah ke (0,0) dan berhenti.

Karakter-karakter itu diambil dalam urutan E, R, O, S; dibalik, jawabannya adalah **SORE**. Periksalah dengan tangan: dalam SORTED, S, O, R, E muncul dalam urutan itu (dengan mencoret T dan D); dalam STORE, S, O, R, E muncul dalam urutan itu (dengan mencoret T). Panjangnya, 4, cocok dengan $c[6,5]$.

8.16.5 Langkah 3: apakah SORE satu-satunya jawaban?

Tabelnya hanya mengatakan bahwa panjang terbesarnya 4. Untuk melihat bahwa SORE adalah satu-satunya subbarisan bersama sepanjang itu, perhatikan di mana huruf T dapat duduk. Dalam $Y = $ STORE huruf T datang *sebelum* O dan R, sedangkan dalam $X = $ SORTED ia datang *sesudah* keduanya. Jadi subbarisan bersama yang memuat T tidak dapat memuat O maupun R; yang terpanjang di antaranya adalah STE, sepanjang 3. Karena itu setiap subbarisan bersama sepanjang 4 menghindari T, dan juga menghindari D, yang tidak ada dalam Y — yang tersisa hanyalah huruf-huruf S, O, R, E dari X dalam satu-satunya urutan mereka. Jadi SORE adalah satu-satunya subbarisan bersama terpanjang. Menjabarkan setiap subbarisan kedua untai dengan mesin menegaskan hal ini, dan juga bahwa subbarisan bersama sepanjang 3 tepat ORE, SOE, SOR, SRE, dan STE.

Bila memang *ada* beberapa subbarisan bersama terpanjang, yang dihasilkan jalan mundur bergantung pada aturan serinya: ke atas bila seri, seperti di sini, atau ke kiri, dapat membawa ke jawaban yang berbeda — dan sama panjang. Algoritmanya menjanjikan *sebuah* subbarisan bersama terpanjang, bukan subbarisan tertentu.

8.16.6 Langkah 4: biayanya

Mengisi tabelnya mengunjungi setiap sel dari $m \times n$ sel sekali dan mengerjakan sejumlah tetap kerja di sana, jadi `LCS-LENGTH` memakan waktu $\Theta(mn)$ dan ruang $\Theta(mn)$: di sini $6 \times 5 = 30$ sel. Jalan mundur mengurangi i, j, atau keduanya pada setiap langkah, jadi ia memakan paling banyak $m + n$ langkah — di sini 7 langkah, dengan batas 11. Bab 7 menunjukkan cara menyimpan hanya dua baris bila yang diinginkan hanya panjangnya, sehingga ruangnya turun menjadi $\Theta(\min(m, n))$; jalan mundur memerlukan seluruh tabel b, dan itulah sebabnya programnya menyimpannya.

8.16.7 Programnya

Kode 8.11 mengimplementasikan **Pseudokode 8.6** dengan satu baris pelacakan per sel, mencetak tabel gabungan dengan `D`, `U`, dan `L` mewakili ↖, ↑, dan ←, serta mengimplementasikan **Pseudokode 8.7** sebagai sebuah gelung.

Kode 8.11 Soal 8.17 selengkapnya: `LCS-LENGTH` dengan pelacakan setiap sel, tabel gabungan nilai dan panah, dan jalan mundur yang menghasilkan satu subbarisan bersama terpanjang.

```
// lcstable.cpp -- Soal 8.17: LCS-LENGTH diisi sel demi sel, tabel panah,
// dan rekonstruksi satu subbarisan bersama terpanjang dengan berjalan mundur.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

// b[i][j]: 'D' = diagonal (salin), 'U' = atas (lewati x_i), 'L' = kiri (lewati y_j)
vector<vector<int>>  C;
vector<vector<char>> B;

void lcsLength(const string& X, const string& Y, bool trace) {
    int m = (int)X.size(), n = (int)Y.size();
    C.assign(m + 1, vector<int>(n + 1, 0));        // baris 0 dan kolom 0 tetap 0
    B.assign(m + 1, vector<char>(n + 1, ' '));
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (X[i - 1] == Y[j - 1]) {                 // string di C++ berbasis 0
                C[i][j] = C[i - 1][j - 1] + 1; B[i][j] = 'D';
            } else if (C[i - 1][j] >= C[i][j - 1]) {    // seri ke atas, seperti di Bab 7
                C[i][j] = C[i - 1][j];         B[i][j] = 'U';
            } else {
                C[i][j] = C[i][j - 1];         B[i][j] = 'L';
            }
            if (trace) {
                cout << "  c[" << i << "," << j << "]  x" << i << "=" << X[i - 1]
                     << " y" << j << "=" << Y[j - 1] << "  ";
                if (B[i][j] == 'D')
                    cout << "match:    c[" << i - 1 << "," << j - 1 << "]+1 = " << C[i][j]
<< "  D\n";
                else
                    cout << "no match: max(up " << C[i - 1][j] << ", left " << C[i][j - 1]
                         << ") = " << C[i][j] << "  " << B[i][j] << "\n";
            }
        }
    }
}
```

Kode 8.11 (Lanjutan) *Soal 8.17 selengkapnya: LCS-LENGTH dengan pelacakan setiap sel, tabel gabungan nilai dan panah, dan jalan mundur yang menghasilkan satu subbarisan bersama terpanjang.*

```
// PRINT-LCS: ikuti panah mundur dari (i,j); setiap D menyumbang satu karakter.
string walkBack(const string& X, int i, int j, bool trace) {
    string out;
    while (i > 0 && j > 0) {
        char a = B[i][j];
        if (trace) cout << "  at (" << i << "," << j << ") arrow " << a;
        if (a == 'D') {
            out = X[i - 1] + out;
            if (trace) cout << "  take " << X[i - 1] << "\n";
            --i; --j;
        } else {
            if (trace) cout << "\n";
            if (a == 'U') --i; else --j;
        }
    }
    return out;
}

void printTable(const string& X, const string& Y) {
    int m = (int)X.size(), n = (int)Y.size();
    cout << "      j:" << setw(4) << 0 << "  ";                        // nomor kolom, lalu
huruf-huruf Y
    for (int j = 1; j <= n; ++j) cout << setw(3) << j << "  ";
    cout << "\n              ";
    for (int j = 1; j <= n; ++j) cout << "  " << Y[j - 1] << "  ";
    cout << "\n  i=0    ";
    for (int j = 0; j <= n; ++j) cout << setw(3) << 0 << "  ";
    cout << "\n";
    for (int i = 1; i <= m; ++i) {
        cout << "  i=" << i << " " << X[i - 1] << "  " << setw(3) << 0 << "  ";
        for (int j = 1; j <= n; ++j) cout << setw(3) << C[i][j] << B[i][j] << " ";
        cout << "\n";
    }
}

void solve(const string& X, const string& Y, bool trace) {
    cout << "X = " << X << " (m = " << X.size() << "), Y = " << Y
         << " (n = " << Y.size() << ")\n";
    lcsLength(X, Y, trace);
    printTable(X, Y);
    string z = walkBack(X, (int)X.size(), (int)Y.size(), trace);
    cout << "  LCS length = " << C[X.size()][Y.size()] << ", one LCS = " << z << "\n";
}
```

Kode 8.11 (Lanjutan) *Soal 8.17 selengkapnya: LCS-LENGTH dengan pelacakan setiap sel, tabel gabungan nilai dan panah, dan jalan mundur yang menghasilkan satu subbarisan bersama terpanjang.*

```
int main() {
    solve("SORTED", "STORE", true);
    cout << "\nCheck yourself\n";
    solve("HEAPS", "SHAPE", false);
    return 0;
}
```

Ia mencetak:

```
X = SORTED (m = 6), Y = STORE (n = 5)
  c[1,1]  x1=S y1=S  match:    c[0,0]+1 = 1  D
  c[1,2]  x1=S y2=T  no match: max(up 0, left 1) = 1  L
  c[1,3]  x1=S y3=O  no match: max(up 0, left 1) = 1  L
  c[1,4]  x1=S y4=R  no match: max(up 0, left 1) = 1  L
  c[1,5]  x1=S y5=E  no match: max(up 0, left 1) = 1  L
  c[2,1]  x2=O y1=S  no match: max(up 1, left 0) = 1  U
  c[2,2]  x2=O y2=T  no match: max(up 1, left 1) = 1  U
  c[2,3]  x2=O y3=O  match:    c[1,2]+1 = 2  D
  c[2,4]  x2=O y4=R  no match: max(up 1, left 2) = 2  L
  c[2,5]  x2=O y5=E  no match: max(up 1, left 2) = 2  L
  c[3,1]  x3=R y1=S  no match: max(up 1, left 0) = 1  U
  c[3,2]  x3=R y2=T  no match: max(up 1, left 1) = 1  U
  c[3,3]  x3=R y3=O  no match: max(up 2, left 1) = 2  U
  c[3,4]  x3=R y4=R  match:    c[2,3]+1 = 3  D
  c[3,5]  x3=R y5=E  no match: max(up 2, left 3) = 3  L
  c[4,1]  x4=T y1=S  no match: max(up 1, left 0) = 1  U
  c[4,2]  x4=T y2=T  match:    c[3,1]+1 = 2  D
  c[4,3]  x4=T y3=O  no match: max(up 2, left 2) = 2  U
  c[4,4]  x4=T y4=R  no match: max(up 3, left 2) = 3  U
  c[4,5]  x4=T y5=E  no match: max(up 3, left 3) = 3  U
  c[5,1]  x5=E y1=S  no match: max(up 1, left 0) = 1  U
  c[5,2]  x5=E y2=T  no match: max(up 2, left 1) = 2  U
  c[5,3]  x5=E y3=O  no match: max(up 2, left 2) = 2  U
  c[5,4]  x5=E y4=R  no match: max(up 3, left 2) = 3  U
  c[5,5]  x5=E y5=E  match:    c[4,4]+1 = 4  D
  c[6,1]  x6=D y1=S  no match: max(up 1, left 0) = 1  U
  c[6,2]  x6=D y2=T  no match: max(up 2, left 1) = 2  U
  c[6,3]  x6=D y3=O  no match: max(up 2, left 2) = 2  U
  c[6,4]  x6=D y4=R  no match: max(up 3, left 2) = 3  U
  c[6,5]  x6=D y5=E  no match: max(up 4, left 3) = 4  U
      j:   0    1    2    3    4    5
                S    T    O    R    E
  i=0      0    0    0    0    0    0
  i=1 S    0    1D   1L   1L   1L   1L
  i=2 O    0    1U   1U   2D   2L   2L
  i=3 R    0    1U   1U   2U   3D   3L
  i=4 T    0    1U   2D   2U   3U   3U
  i=5 E    0    1U   2U   2U   3U   4D
  i=6 D    0    1U   2U   2U   3U   4U
  at (6,5) arrow U
  at (5,5) arrow D  take E
  at (4,4) arrow U
  at (3,4) arrow D  take R
  at (2,3) arrow D  take O
  at (1,2) arrow L
  at (1,1) arrow D  take S
  LCS length = 4, one LCS = SORE

Check yourself
X = HEAPS (m = 5), Y = SHAPE (n = 5)
      j:   0    1    2    3    4    5
                S    H    A    P    E
  i=0      0    0    0    0    0    0
  i=1 H    0    0U   1D   1L   1L   1L
  i=2 E    0    0U   1U   1U   1U   2D
  i=3 A    0    0U   1U   2D   2L   2U
  i=4 P    0    0U   1U   2U   3D   3L
  i=5 S    0    1D   1U   2U   3U   3U
  LCS length = 3, one LCS = HAP
```

Ketiga puluh baris pelacakan adalah ketiga puluh sel di §8.16.3, dalam urutan yang sama dan dengan panah yang sama; tabelnya adalah **Tabel 8.12**; dan ketujuh baris "at" adalah ketujuh langkah di §8.16.4.

8.16.8 Periksa diri

Temukan sebuah subbarisan bersama terpanjang dari $X = $ HEAPS dan $Y = $ SHAPE dengan mengisi tabelnya dengan tangan.

> **JAWABAN.** Baris-barisnya, sebagai nilai dan panah untuk $j = 1, \ldots, 5$ (S, H, A, P, E): H: 0 ↑, 1 ↖, 1 ←, 1 ←, 1 ←. E: 0 ↑, 1 ↑, 1 ↑, 1 ↑, 2 ↖. A: 0 ↑, 1 ↑, 2 ↖, 2 ←, 2 ↑. P: 0 ↑, 1 ↑, 2 ↑, 3 ↖, 3 ←. S: 1 ↖, 1 ↑, 2 ↑, 3 ↑, 3 ↑. Panjangnya $c[5,5] = 3$, dan jalan mundur dari (5,5) naik ke (4,5), ke kiri ke (4,4), mengambil P di sana, mengambil A di (3,3), bergerak diagonal ke (2,2), naik ke (1,2), dan mengambil H: **HAP**. Ia satu-satunya subbarisan bersama sepanjang 3: S berada paling depan dalam Y tetapi paling belakang dalam X, dan E paling belakang dalam Y tetapi kedua dalam X, sehingga keduanya tidak dapat bergabung dengan H, A, dan P. Blok kedua keluaran **Kode 8.11** menegaskan tabelnya.

8.16.9 Belokan keliru yang sering terjadi pada soal ini

1. **Mengambil diagonal saat tidak cocok.** Diagonal hanya dipakai bila $x_i = y_j$; saat tidak cocok, nilainya datang dari atas atau dari kiri, tidak pernah dari diagonal ditambah apa pun.
2. **Mencetak karakter jalan mundur dalam urutan ditemukannya.** Karakter-karakter itu keluar dari yang terakhir: EROS alih-alih SORE. Baliklah, atau cetak sesudah panggilan rekursifnya seperti yang dilakukan **Pseudokode 8.7**.
3. **Selisih satu antara tabel dan untainya.** Tabelnya diberi indeks mulai 1 untuk karakter, tetapi untai C++ dimulai di 0, jadi x_i adalah `X[i-1]`. **Kode 8.11** menandai baris itu.
4. **Mengira aturan seri mengubah panjangnya.** Aturan itu mengubah subbarisan terpanjang mana yang ditemukan, tidak pernah panjangnya.
5. **Mengacaukan subbarisan dengan subuntai.** Subuntai (*substring*) harus bersambung; SORE bukan subuntai dari SORTED, tetapi ia subbarisannya.

8.17 Dua Puluh Lima Pemeriksaan Cepat

Dua puluh lima pertanyaan dengan jawaban satu baris, meliputi gagasan-gagasan pokok Bab 1 sampai 7. Tutuplah jawabannya, kerjakan daftarnya, lalu periksalah diri Anda sendiri. Apa pun yang meleset menyebutkan subbab yang sebaiknya Anda baca ulang.

1. **Apa kelima fase kerangka pemecahan masalah buku ini?** *Understand* (memahami), *abstract* (mengabstraksi), *design* (merancang), *analyze* (menganalisis), *implement* (mengimplementasi). (Bab 1)
2. **Berapa biaya subskrip larik menurut model RAM?** Satu satuan — waktu konstan, seperti aritmetika atau perbandingan mana pun. (Bab 2)
3. **Nyatakan invarian gelung pengurutan sisip.** Pada awal tiap iterasi gelung luarnya, upalarik $A[1..j-1]$ memuat unsur-unsur asli $A[1..j-1]$ dalam keadaan terurut. (Bab 2)
4. **Tiga kewajiban sebuah invarian gelung?** Inisialisasi, pemeliharaan, terminasi. (Bab 2)
5. **Kasus terbaik dan terburuk pengurutan sisip, beserta masukan yang menyebabkan masing-masing?** $\Theta(n)$ pada larik yang sudah terurut; $\Theta(n^2)$ pada larik yang terurut terbalik. (Bab 2)
6. **Berapa banyak perbandingan yang dilakukan pencarian biner pada n unsur?** $\Theta(\lg n)$ — persisnya $\lfloor \lg n \rfloor + 1$ pada kasus terburuk. (Bab 2)
7. **Definisikan $f(n) = O(g(n))$.** Terdapat konstanta $c > 0$ dan n_0 sedemikian sehingga $0 \leq f(n) \leq c\, g(n)$ untuk semua $n \geq n_0$. (Bab 3)

8. **Beda antara O dan o?** O memperbolehkan batasnya ketat; o menuntut $f(n)/g(n) \to 0$, sehingga batasnya longgar secara tegas. (Bab 3)
9. **Apakah $2^{n+1} = O(2^n)$? Apakah $2^{2n} = O(2^n)$?** Ya dan tidak: $2^{n+1} = 2 \cdot 2^n$, tetapi $2^{2n} = (2^n)^2$ tumbuh lebih cepat daripada kelipatan konstanta mana pun. (Bab 3)
10. **Hitunglah $\sum_{i=1}^{n} i$ dan $\sum_{i=0}^{n} 2^i$.** $\frac{n(n+1)}{2}$ dan $2^{n+1} - 1$. (Bab 3)
11. **Ketiga kasus Teorema Master, masing-masing satu baris.** $f(n) = O(n^{\log_b a - \varepsilon}) \Rightarrow \Theta(n^{\log_b a})$; $f(n) = \Theta(n^{\log_b a}) \Rightarrow \Theta(n^{\log_b a} \lg n)$; $f(n) = \Omega(n^{\log_b a + \varepsilon})$ dengan syarat keteraturan $\Rightarrow \Theta(f(n))$. (Bab 3)
12. **Selesaikan $T(n) = 2T(n/2) + n$.** $a = b = 2$, $\log_b a = 1$, $f(n) = \Theta(n^1)$: kasus 2, jadi $T(n) = \Theta(n \lg n)$. (Bab 3)
13. **Selesaikan $T(n) = T(n/2) + \Theta(1)$.** $\log_b a = 0$, $f(n) = \Theta(n^0)$: kasus 2, jadi $T(n) = \Theta(\lg n)$ — pencarian biner. (Bab 3)
14. **Mengapa pengurutan gabung $\Theta(n \lg n)$ pada *setiap* kasus?** Pembelahannya menurut posisi, jadi kedua separuhnya selalu punya $n/2$ unsur tanpa memandang datanya; rekurensinya tidak punya varian terbaik atau terburuk. (Bab 3)
15. **Nyatakan sifat heap-maks, dan mengapa ia bukan sifat BST.** $A[\text{parent}(i)] \geq A[i]$ — aturan induk-dan-anak tanpa pembedaan kiri/kanan, sehingga pemeriksaan yang murni lokal benar untuk heap dan keliru untuk BST. (Bab 5, §8.6.6)
16. **Mengapa `BUILD-MAX-HEAP` $\Theta(n)$ dan bukan $\Theta(n \lg n)$?** Sebagian besar simpulnya dekat ke dasar, tempat `MAX-HEAPIFY` hampir tak berbiaya apa pun; menjumlahkan $\sum_h \lceil n/2^{h+1} \rceil O(h)$ memberikan $O(n)$. (Bab 5, §8.3.6)
17. **Waktu jalan dan ruang heapsort?** $\Theta(n \lg n)$ selalu, dan ruang tambahan $O(1)$ — ia mengurutkan di tempat. (Bab 5)
18. **Kasus terburuk quicksort, dan masukan yang menyebabkannya dengan pivot elemen-terakhir?** $\Theta(n^2)$, pada larik yang sudah terurut: tepat $\frac{n(n-1)}{2}$ perbandingan. (Bab 5, §8.4.5)
19. **Apa yang dibeli oleh pengacakan pivotnya, dan apa yang tidak?** $\Theta(n \lg n)$ harapan pada *setiap* masukan; ia tidak mengubah kasus terburuk $\Theta(n^2)$, hanya peluangnya. (Bab 5)
20. **Batas bawah pengurutan berbasis perbandingan, dan mengapa?** $\Omega(n \lg n)$: pohon keputusan dengan $n!$ daun bertinggi paling sedikit $\lg(n!) = \Theta(n \lg n)$. (Bab 5)
21. **Mengapa penelusuran inorder sebuah BST itu terurut?** Setiap nilai subpohon kiri berada di bawah akarnya dan setiap nilai subpohon kanan di atasnya, jadi kiri–akar–kanan memancarkannya dalam urutan menaik; dengan induksi, seluruh penelusurannya terurut. (Bab 6, §8.7.3)
22. **Batas tinggi AVL?** $h \leq 1.4405 \lg(n+2) - 1.3277$, dari rekurensi simpul-minimum $N(h) = 1 + N(h-1) + N(h-2)$, yang penyelesaiannya $N(h) = F(h+3) - 1$. (Bab 6)
23. **Berapa banyak rotasi yang dapat disebabkan satu penyisipan AVL? Satu penghapusan?** Paling banyak dua untuk penyisipan; sampai $\Theta(\lg n)$ rotasi tunggal untuk penghapusan. (Bab 6, §8.13.4)
24. **Apa yang membuat sebuah persoalan cocok untuk pemrograman dinamis alih-alih bagi-dan-kuasai biasa?** Adanya submasalah yang tumpang tindih bersama substruktur optimal; bagi dan kuasai cocok untuk submasalah yang *saling bebas*. (Bab 7, Soal 8.16 dan 8.17)
25. **Memoisasi lawan tabulasi bawah-atas — bedanya dalam satu baris?** Keduanya menyimpan jawaban tiap submasalah sekali; memoisasi mempertahankan rekursinya dan mengisi sesuai permintaan, tabulasi menyingkirkan rekursinya dan mengisi menurut urutan kebergantungan. (Bab 7, §8.15.9)

8.18 Soal-Soal Lanjutan: Pohon Statistik-Urutan

Kedelapan soal subbab ini, Soal 8.18 sampai 8.25, berbentuk sama dengan Soal 8.3 sampai 8.10 tetapi dengan invarian yang berbeda, sehingga mengingat penyelesaian terdahulu tidak akan cukup, sedangkan memahaminya akan cukup. Cobalah setiap soal sebelum membaca penyelesaiannya.

8.18.1 Soal-soalnya

Tipe kerjanya adalah **pohon statistik-urutan** (*order-statistic tree*): pohon pencarian biner yang setiap simpulnya menyimpan, selain nilai bulatnya, dua medan turunan — `h`, tinggi subpohonnya sendiri, dan `size`, banyaknya simpul di subpohonnya sendiri. Kedua medan itu dipelihara oleh satu-satunya konstruktor `node(v, l, r)`.

Soal 8.18. Tulislah `isOrderStatisticTree(OSTree t)`, dengan sebuah fungsi penunjang rekursif, yang mengembalikan `true` tepat ketika *ketiga* hal berikut berlaku di setiap simpul: sifat pohon pencarian biner; `size` yang tersimpan sama dengan $1 + \text{size(kiri)} + \text{size(kanan)}$; dan `h` yang tersimpan sama dengan $1 + \max(h(\text{kiri}), h(\text{kanan}))$. Fungsi penunjangnya harus menempuh **satu lintasan**. Nyatakan kompleksitasnya.

Soal 8.19. Tulislah `printBetween(OSTree t, int lo, int hi)`, yang mencetak, dalam urutan **menaik**, setiap nilai v dari t dengan $lo \leq v \leq hi$. Fungsi itu **tidak** boleh membangun daftar nilai, dan **tidak** boleh mengunjungi subpohon yang tak mungkin memuat nilai di dalam rentangnya.

Soal 8.20. Tulislah `select(OSTree t, int k)`, yang mengembalikan nilai terkecil ke-k dari t, dengan k dihitung mulai 1. Fungsi itu harus mengikuti **paling banyak satu lintasan** dari akar. Nyatakan mengapa caranya $O(h)$.

Soal 8.21. Tulislah `isWeightBalanced(OSTree t)`, yang mengembalikan `true` tepat ketika, di setiap simpul, $\text{size(kiri)} \leq 2\,\text{size(kanan)} + 1$ dan $\text{size(kanan)} \leq 2\,\text{size(kiri)} + 1$. Satu lintasan. Lalu jawablah dalam satu kalimat: *apakah setiap pohon AVL seimbang-bobot dalam pengertian ini?* Benarkan dengan contoh atau bukti.

Soal 8.22. Tulislah `insertOS(int n, OSTree t)`, yang menyisipkan n sambil menjaga invarian keseimbangan-**tinggi** (AVL) *dan* kebenaran kedua medan tersimpannya. Fungsi itu harus memakai kedua fungsi Soal 8.23 dan 8.24.

Soal 8.23. Tulislah `rebalanceForLeft(OSTree t)`, yang dipanggil ketika subpohon kiri t mungkin baru bertambah tinggi satu takik.

Soal 8.24. Tulislah `rebalanceForRight(OSTree t)`, bayangan cerminnya.

Soal 8.25. Tulislah `deleteOS(OSTree t, int x)`, yang menghapus x sambil menjaga invarian AVL dan kedua medan tersimpannya.

Tabel 8.13 memasangkan setiap soal dengan soal terdahulu yang diperluasnya, dan menyebutkan apa yang baru.

Tabel 8.13 Soal 8.18 sampai 8.25 dibandingkan dengan soal-soal yang diperluasnya. Bentuknya sama; hanya invariannya yang berubah.

Soal	Memperluas	Yang baru
8.18	8.3 `isBST`	dua medan turunan juga harus diperiksa
8.19	8.4 `printDescending`	pemangkasan, bukan sekadar urutan penelusuran
8.20	8.5 `max`	ukuran subpohon mengemudikan penurunannya
8.21	8.6 `isHeightBalanced`	gagasan keseimbangan yang *berbeda*
8.22	8.7 `insertHB`	medan `size` harus bertahan melewati rotasi
8.23	8.8 `rebalanceForLeft`	strukturnya tak berubah
8.24	8.9 `rebalanceForRight`	strukturnya tak berubah
8.25	8.10 `deleteHB`	strukturnya tak berubah

8.18.2 Bagaimana tiap penyelesaian diturunkan

Soal 8.18. Perluaslah rekursi selang-terbatas §8.6.4 sehingga ia *mengembalikan* tinggi dan ukuran yang sejati alih-alih sekadar sebuah boolean — gagasan penjaga §8.9.4, diperumum menjadi sebuah rekaman kecil. Di tiap simpul: periksa batasnya, berekursi ke kiri dengan batas atas diganti nilai simpulnya, berekursi ke kanan dengan batas bawahnya diganti, lalu bandingkan kedua medan terhitungnya terhadap kedua medan tersimpannya, lalu periksa $|h_R - h_L| \leq 1$. Satu lintasan, waktu $\Theta(n)$, tumpukan $O(h)$.

Soal 8.19. Penelusuran inorder biasa, dengan dua uji pemangkasan. Turunlah ke kiri hanya bila nilai simpulnya melampaui lo — jika simpulnya sudah $\leq lo$, setiap nilai di kirinya lebih kecil lagi dan tak satu pun dapat berada dalam rentang. Turunlah ke kanan hanya bila nilai simpulnya di bawah hi, menurut argumen cerminnya. Biayanya: $O(h + k)$ dengan k banyaknya nilai yang dilaporkan, dan itu optimal.

Soal 8.20. Medan `size` mengubah pemilihan menjadi sebuah penurunan. Di sebuah simpul, sebut $\ell = \text{size(kiri)}$. Jika $k \leq \ell$, jawabannya adalah nilai terkecil ke-k dari subpohon kirinya. Jika $k = \ell + 1$, jawabannya adalah nilai simpul ini — ada tepat ℓ nilai di bawahnya. Jika tidak, jawabannya adalah nilai terkecil ke-$(k - \ell - 1)$ dari subpohon kanannya, sebab kita sudah membuang ℓ nilai kirinya *dan* simpulnya sendiri. Satu panggilan rekursif per aras, jadi $O(h)$.

Soal 8.21. Teknik penjaga dengan ukuran: kembalikan ukuran upapohonnya, atau -1 untuk berarti "sudah gagal". -1 gratis, sebab ukuran tak pernah negatif. $\Theta(n)$.

Dan pertanyaan satu kalimatnya: **tidak**, pohon AVL tidak harus seimbang-bobot dalam pengertian ini. Contoh sanggahannya disusun, bukan ditemukan. Ambillah sebuah akar yang subpohon kirinya *sempurna* bertinggi 2 — tujuh simpul — dan subpohon kanannya pohon AVL *minimal* bertinggi 1 — dua simpul. Tingginya berselisih tepat 1, jadi akarnya seimbang-tinggi dan begitu pula segala yang di bawahnya; tetapi $\text{size(kiri)} = 7$ dan $2\,\text{size(kanan)} + 1 = 2 \cdot 2 + 1 = 5$, sedangkan $7 > 5$. Keseimbangan tinggi membatasi *kedalaman* kedua sisinya; ia tidak membatasi *populasinya*, sebab subpohon dengan tinggi tertentu boleh memuat sesedikit $N(h)$ simpul atau sebanyak $2^{h+1} - 1$.

Soal 8.22 sampai 8.25. Secara struktural identik dengan §§8.10–8.13. Satu-satunya kewajiban barunya adalah bahwa `node(v,l,r)` harus membetulkan `size` sekaligus `h`, dan ia memang melakukannya — dan itulah sebabnya rotasinya, kedua penyeimbang ulangnya, `insertOS`, dan

`deleteOS` sama sekali tidak perlu menyebut `size`. Itulah buah dari disiplin satu-konstruktor §8.5: menambahkan medan turunan kedua berbiaya satu baris.

8.18.3 Penyelesaian lengkap

Kode 8.12 menyelesaikan kedelapan soal dalam satu program. Fungsi `main`-nya membangun pohon sebelas simpul, dengan sengaja merusak satu medan tersimpan untuk memperlihatkan pemeriksa Soal 8.18 bekerja, lalu diakhiri uji tekanan berupa dua ribu penyisipan dan penghapusan.

Kode 8.12 Penyelesaian lengkap Soal 8.18 sampai 8.25, dengan perusakan sengaja pada satu medan tersimpan untuk memperlihatkan pemeriksa Soal 8.18 menangkapnya, serta uji tekanan dua ribu elemen.

```
// ostree.cpp -- penyelesaian lengkap Soal 8.18 sampai 8.25 di subbab 8.18.
// Pohon statistik terurut: pohon AVL yang setiap simpulnya juga menyimpan
// banyaknya simpul di dalam upapohonnya sendiri.
#include <iostream>
#include <iomanip>
#include <string>
#include <stdexcept>
#include <vector>
#include <algorithm>
using namespace std;

struct OSNode;
using OSTree = OSNode*;
struct OSNode { int value; OSTree left, right; int h, size; };

OSTree emptyTree()       { return nullptr; }
bool   isEmpty(OSTree t) { return t == nullptr; }
int    height(OSTree t)  { return isEmpty(t) ? -1 : t->h; }
int    sizeOf(OSTree t)  { return isEmpty(t) ?  0 : t->size; }
int    value(OSTree t)   { return t->value; }
OSTree leftOf(OSTree t)  { return t->left; }
OSTree rightOf(OSTree t) { return t->right; }

// Satu-satunya konstruktor. Ia membetulkan KEDUA medan turunan, jadi tak ada yang basi.
OSTree node(int v, OSTree l, OSTree r) {
    OSTree t = new OSNode;
    t->value = v; t->left = l; t->right = r;
    t->h    = 1 + max(height(l), height(r));
    t->size = 1 + sizeOf(l) + sizeOf(r);
    return t;
}
OSTree leaf(int v) { return node(v, emptyTree(), emptyTree()); }
int balanceFactor(OSTree t) { return height(rightOf(t)) - height(leftOf(t)); }

string shape(OSTree t) {
    if (isEmpty(t)) return ".";
    string s = to_string(value(t)) + "[" + to_string(sizeOf(t)) + "]";
    if (isEmpty(leftOf(t)) && isEmpty(rightOf(t))) return s;
    return s + "{" + shape(leftOf(t)) + "," + shape(rightOf(t)) + "}";
}
```

Kode 8.12 (Lanjutan) *Penyelesaian lengkap Soal 8.18 sampai 8.25, dengan perusakan sengaja pada satu medan tersimpan untuk memperlihatkan pemeriksa Soal 8.18 menangkapnya, serta uji tekanan dua ribu elemen.*

```
// ===== Soal 8.18: periksa ketiga invarian dalam satu lintasan =====
// Mengembalikan (tinggi, ukuran) sejati subpohon, atau melaporkan cacat pertama.
struct Fields { int h, size; };
static bool validate(OSTree t, long long lo, long long hi, Fields& out, string& why) {
    if (isEmpty(t)) { out = {-1, 0}; return true; }
    if (value(t) <= lo) { why = "node " + to_string(value(t)) + " must exceed " +
to_string(lo); return false; }
    if (value(t) >= hi) { why = "node " + to_string(value(t)) + " must be below " +
to_string(hi); return false; }
    Fields L, R;
    if (!validate(leftOf(t),  lo, value(t), L, why)) return false;
    if (!validate(rightOf(t), value(t), hi, R, why)) return false;
    int trueH = 1 + max(L.h, R.h), trueSize = 1 + L.size + R.size;
    if (t->h != trueH) {
        why = "node " + to_string(value(t)) + " stores height " + to_string(t->h)
            + " but the truth is " + to_string(trueH); return false;
    }
    if (t->size != trueSize) {
        why = "node " + to_string(value(t)) + " stores size " + to_string(t->size)
            + " but the truth is " + to_string(trueSize); return false;
    }
    if (abs(R.h - L.h) > 1) {
        why = "node " + to_string(value(t)) + " has balance factor " + to_string(R.h -
L.h);
        return false;
    }
    out = {trueH, trueSize};
    return true;
}
bool isOrderStatisticTree(OSTree t, string& why) {
    Fields f; why = "none";
    return validate(t, -2147483649LL, 2147483648LL, f, why);
}

// ===== Soal 8.19: cetak setiap nilai di [lo,hi], naik, tanpa membangun daftar =====
void printBetween(OSTree t, int lo, int hi) {
    if (isEmpty(t)) return;
    if (value(t) > lo) printBetween(leftOf(t), lo, hi);   // pangkas: tak ada yang < lo di
kiri
    if (lo <= value(t) && value(t) <= hi) cout << value(t) << ' ';
    if (value(t) < hi) printBetween(rightOf(t), lo, hi);  // pangkas: tak ada yang > hi di
kanan
}
```

Kode 8.12 (Lanjutan) *Penyelesaian lengkap Soal 8.18 sampai 8.25, dengan perusakan sengaja pada satu medan tersimpan untuk memperlihatkan pemeriksa Soal 8.18 menangkapnya, serta uji tekanan dua ribu elemen.*

```
// ===== Soal 8.20: nilai terkecil ke-k, satu lintasan menurun, O(h) =====
long long probes = 0;
int select_(OSTree t, int k) {                  // k dihitung mulai dari 1
    if (isEmpty(t)) throw invalid_argument("k is out of range");
    ++probes;
    int leftCount = sizeOf(leftOf(t));
    if (k <= leftCount)      return select_(leftOf(t), k);
    if (k == leftCount + 1)  return value(t);
    return select_(rightOf(t), k - leftCount - 1);   // buang sisi kiri DAN t
}

// ===== Soal 8.21: predikat seimbang-bobot, satu lintasan =====
// Invarian: di tiap simpul, size(kiri) <= 2*size(kanan)+1 dan sebaliknya.
const int WB_FAIL = -1;
static int checkedSize(OSTree t, string& why) {
    if (isEmpty(t)) return 0;
    int sl = checkedSize(leftOf(t),  why); if (sl == WB_FAIL) return WB_FAIL;
    int sr = checkedSize(rightOf(t), why); if (sr == WB_FAIL) return WB_FAIL;
    if (sl > 2 * sr + 1 || sr > 2 * sl + 1) {
        why = "node " + to_string(value(t)) + ": left size " + to_string(sl)
            + ", right size " + to_string(sr);
        return WB_FAIL;
    }
    return 1 + sl + sr;
}
bool isWeightBalanced(OSTree t, string& why) { why = "none"; return checkedSize(t, why) !=
WB_FAIL; }

// ===== Soal 8.23 dan 8.24: kedua penyeimbang ulang =====
long long rotations = 0;
OSTree rotateRight(OSTree t) {
    ++rotations; OSTree l = leftOf(t);
    return node(value(l), leftOf(l), node(value(t), rightOf(l), rightOf(t)));
}
OSTree rotateLeft(OSTree t) {
    ++rotations; OSTree r = rightOf(t);
    return node(value(r), node(value(t), leftOf(t), leftOf(r)), rightOf(r));
}
OSTree rebalanceForLeft(OSTree t) {
    if (balanceFactor(t) >= -1) return t;
    OSTree l = leftOf(t);
    if (balanceFactor(l) <= 0) return rotateRight(t);
    return rotateRight(node(value(t), rotateLeft(l), rightOf(t)));
}
OSTree rebalanceForRight(OSTree t) {
    if (balanceFactor(t) <= 1) return t;
    OSTree r = rightOf(t);
```

Kode 8.12 (Lanjutan) *Penyelesaian lengkap Soal 8.18 sampai 8.25, dengan perusakan sengaja pada satu medan tersimpan untuk memperlihatkan pemeriksa Soal 8.18 menangkapnya, serta uji tekanan dua ribu elemen.*

```
    if (balanceFactor(r) >= 0) return rotateLeft(t);
    return rotateLeft(node(value(t), leftOf(t), rotateRight(r)));
}

// ===== Soal 8.22: sisip, sambil menjaga keseimbangan tinggi dan medan size =====
OSTree insertOS(int n, OSTree t) {
    if (isEmpty(t)) return leaf(n);
    if (n < value(t)) return rebalanceForLeft (node(value(t), insertOS(n, leftOf(t)),
rightOf(t)));
    if (n > value(t)) return rebalanceForRight(node(value(t), leftOf(t), insertOS(n,
rightOf(t))));
    return t;
}

// ===== Soal 8.25: hapus, sambil menjaga keseimbangan tinggi dan medan size =====
int maxOf(OSTree t) { return isEmpty(rightOf(t)) ? value(t) : maxOf(rightOf(t)); }
OSTree deleteOS(OSTree t, int x) {
    if (isEmpty(t)) return t;
    if (x < value(t)) return rebalanceForRight(node(value(t), deleteOS(leftOf(t), x),
rightOf(t)));
    if (x > value(t)) return rebalanceForLeft (node(value(t), leftOf(t),
deleteOS(rightOf(t), x)));
    if (isEmpty(leftOf(t)))  return rightOf(t);
    if (isEmpty(rightOf(t))) return leftOf(t);
    int m = maxOf(leftOf(t));
    return rebalanceForRight(node(m, deleteOS(leftOf(t), m), rightOf(t)));
}

int main() {
    OSTree t = emptyTree();
    for (int v : {50, 30, 70, 20, 40, 60, 80, 10, 35, 45, 65}) t = insertOS(v, t);
    string why;
    cout << "8.18 shape            : " << shape(t) << "\n";
    cout << "8.18 valid OS tree    : " << (isOrderStatisticTree(t, why) ? "yes" : "no")
         << "   (fault: " << why << ")\n";
    t->size = 99;                                   // rusakkan satu medan dengan sengaja
    cout << "8.18 after corrupting : " << (isOrderStatisticTree(t, why) ? "yes" : "no")
         << "   (fault: " << why << ")\n";
    t->size = 11;                                   // kembalikan seperti semula
    cout << "8.18 repaired         : " << (isOrderStatisticTree(t, why) ? "yes" : "no") <<
"\n\n";

    cout << "8.19 all values       : "; printBetween(t, -1000, 1000); cout << "\n";
    cout << "8.19 in [30,60]       : "; printBetween(t,    30,   60); cout << "\n";
    cout << "8.19 in [41,44]       : "; printBetween(t,    41,   44); cout <<
"(nothing)\n\n";
```

Kode 8.12 (Lanjutan) *Penyelesaian lengkap Soal 8.18 sampai 8.25, dengan perusakan sengaja pada satu medan tersimpan untuk memperlihatkan pemeriksa Soal 8.18 menangkapnya, serta uji tekanan dua ribu elemen.*

```
    cout << "8.20 k-th smallest    :";
    for (int k = 1; k <= sizeOf(t); ++k) cout << ' ' << select_(t, k);
    cout << "\n";
    probes = 0; int v6 = select_(t, 6);
    cout << "8.20 select(6) = " << v6 << " after " << probes
         << " node probes (n = " << sizeOf(t) << ", h = " << height(t) << ")\n\n";

    cout << "8.21 weight-balanced  : " << (isWeightBalanced(t, why) ? "yes" : "no")
         << "   (fault: " << why << ")\n";
    // Pohon AVL yang TIDAK seimbang-bobot: kiri sempurna bertinggi 2 (7 simpul),
    // kanan pohon AVL minimal bertinggi 1 (2 simpul).
    OSTree skew = node(100,
                       node(40, node(20, leaf(10), leaf(30)), node(60, leaf(50),
leaf(70))),
                       node(200, emptyTree(), leaf(300)));
    cout << "8.21 handcrafted tree : " << shape(skew) << "\n";
    cout << "8.21 is it AVL?       : " << (isOrderStatisticTree(skew, why) ? "yes" : "no")
<< "\n";
    cout << "8.21 weight-balanced? : " << (isWeightBalanced(skew, why) ? "yes" : "no")
         << "   (fault: " << why << ")\n\n";

    cout << "8.22 insert 5, 15, 25 one at a time\n";
    OSTree u = t;
    for (int v : {5, 15, 25}) {
        rotations = 0; u = insertOS(v, u);
        cout << "      after " << setw(2) << v << ": rot=" << rotations
             << ", h=" << height(u) << ", n=" << sizeOf(u)
             << ", valid=" << (isOrderStatisticTree(u, why) ? "yes" : "NO") << "\n";
    }
    cout << "      shape: " << shape(u) << "\n\n";

    cout << "8.25 delete 50 (two children), then 10, then 999 (absent)\n";
    for (int v : {50, 10, 999}) {
        rotations = 0; u = deleteOS(u, v);
        cout << "      after " << setw(3) << v << ": rot=" << rotations
             << ", h=" << height(u) << ", n=" << sizeOf(u)
             << ", valid=" << (isOrderStatisticTree(u, why) ? "yes" : "NO") << "\n";
    }
    cout << "      shape: " << shape(u) << "\n\n";
```

Kode 8.12 (Lanjutan) *Penyelesaian lengkap Soal 8.18 sampai 8.25, dengan perusakan sengaja pada satu medan tersimpan untuk memperlihatkan pemeriksa Soal 8.18 menangkapnya, serta uji tekanan dua ribu elemen.*

```
    cout << "stress: insert 1..2000 ascending, select every 200th, delete all\n";
    rotations = 0;
    OSTree s = emptyTree();
    for (int v = 1; v <= 2000; ++v) s = insertOS(v, s);
    cout << "      n=" << sizeOf(s) << ", h=" << height(s)
         << ", rot=" << rotations
         << ", valid=" << (isOrderStatisticTree(s, why) ? "yes" : "NO") << "\n";
    cout << "      select(k) for k = 200,400,...,2000 :";
    for (int k = 200; k <= 2000; k += 200) cout << ' ' << select_(s, k);
    cout << "\n";
    for (int v = 1; v <= 2000; ++v) s = deleteOS(s, v);
    cout << "      after deleting all: n=" << sizeOf(s)
         << ", isEmpty=" << (isEmpty(s) ? "true" : "false") << "\n";
    return 0;
}
```

Programnya mencetak:

```
8.18 shape            :
50[11]{30[6]{20[2]{10[1],.},40[3]{35[1],45[1]}},70[4]{60[2]{.,65[1]},80[1]}}
8.18 valid OS tree    : yes   (fault: none)
8.18 after corrupting : no   (fault: node 50 stores size 99 but the truth is 11)
8.18 repaired         : yes

8.19 all values       : 10 20 30 35 40 45 50 60 65 70 80
8.19 in [30,60]       : 30 35 40 45 50 60
8.19 in [41,44]       : (nothing)

8.20 k-th smallest    : 10 20 30 35 40 45 50 60 65 70 80
8.20 select(6) = 45 after 4 node probes (n = 11, h = 3)

8.21 weight-balanced  : yes   (fault: none)
8.21 handcrafted tree :
100[10]{40[7]{20[3]{10[1],30[1]},60[3]{50[1],70[1]}},200[2]{.,300[1]}}
8.21 is it AVL?       : yes
8.21 weight-balanced? : no   (fault: node 100: left size 7, right size 2)

8.22 insert 5, 15, 25 one at a time
      after  5: rot=1, h=3, n=12, valid=yes
      after 15: rot=0, h=4, n=13, valid=yes
      after 25: rot=0, h=4, n=14, valid=yes
      shape:
50[14]{30[9]{10[5]{5[1],20[3]{15[1],25[1]}},40[3]{35[1],45[1]}},70[4]{60[2]{.,65[1]},80[1]}
}

8.25 delete 50 (two children), then 10, then 999 (absent)
      after  50: rot=0, h=4, n=13, valid=yes
      after  10: rot=1, h=4, n=12, valid=yes
      after 999: rot=0, h=4, n=12, valid=yes
      shape:
45[12]{30[7]{20[4]{5[2]{.,15[1]},25[1]},40[2]{35[1],.}},70[4]{60[2]{.,65[1]},80[1]}}

stress: insert 1..2000 ascending, select every 200th, delete all
      n=2000, h=10, rot=1989, valid=yes
      select(k) for k = 200,400,...,2000 : 200 400 600 800 1000 1200 1400 1600 1800 2000
      after deleting all: n=0, isEmpty=true
```

8.18.4 Membaca keluarannya sebagai pemeriksaan

Setiap blok keluaran menjawab satu pertanyaan tentang penyelesaian Anda sendiri.

Soal 8.18. Pemeriksa yang benar berkata *tidak* terhadap pohon yang dirusakkan dan menyebut simpul 50. Pemeriksa yang menghitung ulang ukurannya alih-alih membandingkannya terhadap medan tersimpannya akan melaporkan "yes" — ia tidak memeriksa apa pun.

Soal 8.19. Ketiga barisnya harus berbunyi `10 20 30 35 40 45 50 60 65 70 80`, `30 35 40 45 50 60`, dan tidak ada apa-apa sama sekali. Tuntutan pemangkasannya hanya terpenuhi bila kodenya punya kedua penjaganya; versi tanpa keduanya mencetak baris yang sama dan mengunjungi setiap simpul.

Soal 8.20. `select(6) = 45` setelah **4** penyelidikan pada pohon bertinggi 3 — itu $h + 1$, satu simpul per aras. Cacahan penyelidikan 11 berarti seluruh pohon ditelusuri dan tuntutan "satu lintasan" tidak terpenuhi.

Soal 8.21. Pohon yang disusun tangan itu harus kembali dengan *AVL ya, seimbang-bobot tidak*, dengan cacatnya dilaporkan di simpul 100 sebagai "left size 7, right size 2". Jika jawaban Anda atas pertanyaan satu kalimatnya adalah "ya, setiap pohon AVL seimbang-bobot", keluaran inilah contoh sanggahannya.

Soal 8.22 sampai 8.25. Setiap `valid=` harus berbunyi `yes`, termasuk sesudah menghapus akarnya dan sesudah menghapus nilai yang tidak ada. Pada uji tekanannya, dua ribu penyisipan menaik harus memberikan tinggi 10, bukan 1999: batasnya adalah $1.4405\lg 2002 - 1.3277 \approx 14.5$, dan $10 \leq 15$. Menghapus kedua ribunya harus meninggalkan `n=0`.

> **WAWASAN KUNCI: MENGAPA SOAL-SOAL INI LEBIH SUKAR DARIPADA KELIHATANNYA.** Rotasi pada Soal 8.23 dan 8.24 karakter demi karakter sama dengan yang ada pada **Kode 8.8** — dan toh keduanya kini juga memelihara medan `size` yang tak pernah disebut Soal 8.8 dan 8.9. Itu bukan kebetulan. Itu akibat dari menyalurkan setiap pembangunan lewat satu fungsi yang membetulkan setiap medan turunan. Pelajarannya layak dinyatakan sebagai aturan: *taruhlah perbaikan invariannya di dalam konstruktor, maka algoritma-algoritmanya berhenti peduli berapa banyak invarian yang ada.*

8.19 Profitina dalam Praktik

Profitina adalah platform visibilitas AI — GEO, *generative engine optimization* — yang berjalan di produksi pada `app.profitina.com`. Ia memindai situs web sebuah merek, menilai seberapa siap situs itu untuk **disitir oleh mesin jawaban AI** dari sinyal yang nyata seperti keterekstrakan konten, data terstruktur, dan otoritas teknis, lalu membantu membetulkan apa yang ditemukannya. Ia aplikasi Python dan FastAPI di atas PostgreSQL; seluruh kerja AI-nya berjalan pada model lokal lewat Ollama dan Qwen2.5, sehingga data pengguna tak pernah meninggalkan servernya untuk pemrosesan AI. Ia menyimpan riwayat skor mingguan per merek, dan sebuah penjadwal memindai ulang tiap situs pada irama sesuai tingkatannya.

Ketiga bab yang dihimpun bab ini masing-masing punya tempat di dalam produk itu. Uraian berikut menggambarkan *bentuk* setiap persoalan dan *golongan* penyelesaiannya.

Antrean pemindaian adalah antrean prioritas. Pada setiap saat banyak situs menunggu pemindaian berikutnya, masing-masing dengan waktu jatuh temponya. Pertanyaan penjadwalnya selalu sama — *pemindaian mana yang jatuh tempo lebih dahulu?* — dan sebuah *max-heap* yang dikunci menurut urgensi (atau, sama saja, sebuah *min-heap* yang dikunci menurut waktu jatuh tempo) menjawabnya dalam $O(1)$ dan menyerap setiap pemindaian baru atau yang dijadwal ulang dalam $O(\lg n)$. Ketika antrean itu harus dibangun ulang dari nol, misalnya setelah dimulai ulang, `BUILD-MAX-HEAP` mengerjakannya dalam waktu linear — Soal 8.1, dalam skala besar.

Membandingkan dua versi sebuah halaman adalah persoalan penjajaran barisan. Tren skor mingguan menjadi berguna ketika ia dapat mengatakan *apa yang berubah* pada halaman itu di antara dua pemindaian. Menjajarkan dua versi sebuah teks sehingga bagian yang tak berubah saling cocok dan sisanya tampil sebagai penyisipan dan penghapusan adalah keluarga persoalan Soal 8.17: *longest common subsequence*-nya persis bagian yang tidak berubah. Tabel berukuran $m \times n$ sel terjangkau untuk judul atau paragraf sebuah halaman, tidak untuk karakter-karakternya satu per satu, sehingga perbandingan yang praktis bekerja pada satuan yang lebih kasar — pertukaran yang sama antara ukuran tabel dan kerincian yang dibahas Bab 7.

Soal-soal pohon bab ini pun punya tempat, dan tempat itu lebih sempit dan, di produksi, lebih berharga: **`isBST` dan `isHeightBalanced` bukan latihan kelas; keduanya adalah bentuk sebuah fitur yang dikapalkan.**

Setiap indeks di dalam layanan yang berjalan lama harus sanggup mengaudit dirinya sendiri. Penjadwal pemindaian Profitina menyimpan indeks waktu-jatuh-tempo — pemindaian mana yang berikutnya, mana yang jatuh tempo antara pukul 02:00 dan 04:00 malam ini — dan indeks itu bertahan melewati proses yang dimulai ulang, rilis bertahap, migrasi skema, dan

sesekali koreksi manual oleh operatornya. Setiap satu di antaranya adalah kesempatan bagi indeks itu untuk berhenti menjadi indeks. Pohon yang urutannya diam-diam rusak tidak jatuh; ia mengembalikan *jawaban yang keliru dengan cepat*, dan itu jauh lebih buruk, sebab tak ada yang memberi peringatan. `isBST` dan `isHeightBalanced` adalah dua fungsi yang mengubah "strukturnya mungkin baik-baik saja" menjadi "strukturnya diperiksa pukul 03:00 dan inilah simpul yang gagal". Keduanya berbiaya $\Theta(n)$ dan keduanya berjalan saat sistemnya sedang senggang.

Jangan percaya medan tersimpan; hitunglah ulang. Peringatan §8.9.7 — bahwa pemeriksa yang membaca medan tinggi tersimpan sama saja tidak memeriksa apa pun — punya padanan di produksi. Rilis bertahap Profitina baru melaporkan keberhasilan setelah versi yang terpasang menjawab `/api/health`; pemeriksaan kesehatan yang sekadar melaporkan nilai sebuah peubah yang ditetapkan saat mula-jalan tidak memastikan apa pun sama sekali. Pemeriksaan yang sungguhan mengerjakan kerjanya lagi lalu membandingkan. Itulah persis yang dituntut Soal 8.18, dan persis pula sebabnya **Kode 8.7** mendeklarasikan tipe simpulnya tanpa medan tinggi.

Sebuah pemeriksa harus menyebut cacatnya, bukan sekadar melaporkan adanya cacat. "Indeksnya rusak" mengirim seorang insinyur berburu. "Pesanan PL-0431 pada menit 500 terarsip di tempat yang hanya untuk menit di bawah 20" mengirimnya ke satu baris. Varian diagnostik pada **Kode 8.4** dan **Kode 8.7** berbiaya empat baris tambahan dan mengubah sebuah boolean menjadi laporan insiden.

Dan pemulihannya tidak boleh bergantung pada invarian yang rusak. Ketika sebuah urutan sudah gagal, *pencarian* menyusuri pohonnya tidak dapat diandalkan — tetapi penelusuran inorder tetap mengunjungi setiap simpul, sebab penelusuran mengikuti penunjuk, bukan perbandingan. Jadi perbaikannya selalu sama: telusuri seluruh strukturnya, kumpulkan rekamannya, urutkan menurut kunci yang sebenarnya, bangun ulang. **Kode 8.13** mengerjakan persis itu.

> **PERINGATAN: DUA ATURAN YANG TIDAK KAMI TAWAR.** Pertama, platformnya *menasihati*; pelanggannya yang memutuskan. Skor visibilitas adalah pengukuran yang disertai penjelasan, tak pernah janji tentang bagaimana sebuah mesin jawaban akan berperilaku. Kedua, apa pun yang menyentuh data pribadi mengikuti UU PDP 27/2022 — persetujuan tersurat, ekspor data, penghapusan akun secara menyeluruh. Penghapusan menyeluruh berarti setiap indeks yang memegang rujukan ke akun itu harus mendukung penghapusan, dan itu satu lagi alasan praktis mengapa Soal 8.10 penting: struktur yang hanya dapat ditambahi tidak dapat dibersihkan.

8.19.1 Studi kasus terkerjakan: Profitina Laundry

Profitina Laundry adalah studi kasus pengajaran untuk buku ini — sebuah laundry kecil di Keputih, Surabaya, bukan produk yang sesungguhnya.

Pemiliknya menerima pesanan sepanjang hari dan menjanjikan waktu pengambilan untuk masing-masingnya. Ia membutuhkan empat hal dari bukunya: pengambilan paling awal yang masih tertunggak, yang paling akhir, antrean menurut urutan tenggat, dan kemampuan mencoret sebuah pesanan ketika sudah diambil atau menambah satu ketika pelanggan datang. Itu persis himpunan operasi yang disediakan Soal 8.5, 8.4, 8.7, dan 8.10, atas sebuah pohon berkunci tenggat pengambilan dalam menit setelah pukul 07:00.

Kode 8.13 membangun indeksnya dengan `insertHB`, menjawab keempat pertanyaan pemiliknya, lalu mengerjakan hal yang penting: seseorang mengetik ulang sebuah tenggat *di tempat*, tanpa

mengarsipkan ulang pesanannya, dan auditnya menangkap itu, menyebutnya, lalu indeksnya dibangun ulang dari sebuah penelusuran inorder. Harganya Rp 7.000 per kilogram, ditagih per kilogram bulat, dengan minimum dua kilogram.

Kode 8.13 Indeks pengambilan Profitina Laundry, dibangun seluruhnya dari kedelapan penyelesaian soal pohon bab ini, dengan audit-diri yang menyebut rekaman yang bercacat dan pembangunan ulang yang selamat dari urutan yang rusak.

```
// laundry.cpp -- Profitina Laundry: indeks pengambilan yang dapat diaudit.
// Setiap rutin yang dipakai di sini adalah salah satu penyelesaian soal pohon bab ini.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

struct Order { int due; string id; int grams; };     // due = menit setelah pukul 07:00

struct LNode;
using Index = LNode*;
struct LNode { Order v; Index left, right; int h; };

Index  emptyIndex()      { return nullptr; }
bool   isEmpty(Index t)  { return t == nullptr; }
int    height(Index t)   { return isEmpty(t) ? -1 : t->h; }
int    key(Index t)      { return t->v.due; }
Index  leftOf(Index t)   { return t->left; }
Index  rightOf(Index t)  { return t->right; }
Index  node(const Order& v, Index l, Index r) {
    Index t = new LNode; t->v = v; t->left = l; t->right = r;
    t->h = 1 + max(height(l), height(r)); return t;
}
int balanceFactor(Index t) { return height(rightOf(t)) - height(leftOf(t)); }

// ---- Soal 8.8 dan 8.9 ----
Index rotateRight(Index t) {
    Index l = leftOf(t);
    return node(l->v, leftOf(l), node(t->v, rightOf(l), rightOf(t)));
}
Index rotateLeft(Index t) {
    Index r = rightOf(t);
    return node(r->v, node(t->v, leftOf(t), leftOf(r)), rightOf(r));
}
Index rebalanceForLeft(Index t) {
    if (balanceFactor(t) >= -1) return t;
    Index l = leftOf(t);
    if (balanceFactor(l) <= 0) return rotateRight(t);
    return rotateRight(node(t->v, rotateLeft(l), rightOf(t)));
}
Index rebalanceForRight(Index t) {
    if (balanceFactor(t) <= 1) return t;
    Index r = rightOf(t);
    if (balanceFactor(r) >= 0) return rotateLeft(t);
```

Kode 8.13 (Lanjutan) *Indeks pengambilan Profitina Laundry, dibangun seluruhnya dari kedelapan penyelesaian soal pohon bab ini, dengan audit-diri yang menyebut rekaman yang bercacat dan pembangunan ulang yang selamat dari urutan yang rusak.*

```
    return rotateLeft(node(t->v, leftOf(t), rotateRight(r)));
}
// ---- Soal 8.7 ----
Index insertHB(const Order& o, Index t) {
    if (isEmpty(t)) return node(o, emptyIndex(), emptyIndex());
    if (o.due < key(t)) return rebalanceForLeft (node(t->v, insertHB(o, leftOf(t)),
rightOf(t)));
    if (o.due > key(t)) return rebalanceForRight(node(t->v, leftOf(t), insertHB(o,
rightOf(t))));
    return t;                                        // menit yang sama: pertahankan yang
pertama
}
// ---- Soal 8.5: pengambilan terakhir hari ini, satu lintasan saja ----
const Order& latest(Index t) { return isEmpty(rightOf(t)) ? t->v : latest(rightOf(t)); }
const Order& earliest(Index t){ return isEmpty(leftOf(t)) ? t->v : earliest(leftOf(t)); }
// ---- Soal 8.10 ----
Index deleteHB(Index t, int due) {
    if (isEmpty(t)) return t;
    if (due < key(t)) return rebalanceForRight(node(t->v, deleteHB(leftOf(t), due),
rightOf(t)));
    if (due > key(t)) return rebalanceForLeft (node(t->v, leftOf(t), deleteHB(rightOf(t),
due)));
    if (isEmpty(leftOf(t)))  return rightOf(t);
    if (isEmpty(rightOf(t))) return leftOf(t);
    Order m = latest(leftOf(t));
    return rebalanceForRight(node(m, deleteHB(leftOf(t), m.due), rightOf(t)));
}

// ---- Soal 8.3: apakah indeksnya masih pohon pencarian? ----
static bool auditBST(Index t, long long lo, long long hi, string& why) {
    if (isEmpty(t)) return true;
    if (key(t) <= lo || key(t) >= hi) {
        why = "order " + t->v.id + " (minute " + to_string(key(t))
            + ") is filed where only minutes in (" + to_string(lo) + ","
            + to_string(hi) + ") belong";
        return false;
    }
    return auditBST(leftOf(t), lo, key(t), why) && auditBST(rightOf(t), key(t), hi, why);
}
// ---- Soal 8.6: apakah ia masih seimbang-tinggi? ----
static int auditHeight(Index t, string& why) {
    if (isEmpty(t)) return -1;
    int hl = auditHeight(leftOf(t),  why); if (hl == -2) return -2;
    int hr = auditHeight(rightOf(t), why); if (hr == -2) return -2;
    if (t->h != 1 + max(hl, hr)) {
        why = "order " + t->v.id + " stores height " + to_string(t->h)
            + " but the truth is " + to_string(1 + max(hl, hr));
        return -2;
    }
    if (abs(hr - hl) > 1) {
        why = "order " + t->v.id + " has balance factor " + to_string(hr - hl);
```

Kode 8.13 (Lanjutan) *Indeks pengambilan Profitina Laundry, dibangun seluruhnya dari kedelapan penyelesaian soal pohon bab ini, dengan audit-diri yang menyebut rekaman yang bercacat dan pembangunan ulang yang selamat dari urutan yang rusak.*

```
        return -2;
    }
    return 1 + max(hl, hr);
}
int countNodes(Index t) { return isEmpty(t) ? 0 : 1 + countNodes(leftOf(t)) +
countNodes(rightOf(t)); }

// Penelusuran inorder menjangkau tiap simpul apa pun kata kuncinya, jadi itulah cara
// yang aman untuk menyelamatkan isi indeks yang urutannya sudah rusak.
void collectAll(Index t, vector<Order>& out) {
    if (isEmpty(t)) return;
    collectAll(leftOf(t), out);
    out.push_back(t->v);
    collectAll(rightOf(t), out);
}

string clockOf(int due) {                 // menit setelah pukul 07:00 -> HH:MM
    int m = 7 * 60 + due;
    char buf[16]; snprintf(buf, sizeof buf, "%02d:%02d", m / 60, m % 60);
    return string(buf);
}
string rupiah(long long v) {
    string s = to_string(v), out; int c = 0;
    for (int i = (int)s.size() - 1; i >= 0; --i) { out += s[i]; if (++c % 3 == 0 && i > 0)
out += '.'; }
    return "Rp " + string(out.rbegin(), out.rend());
}
// Rp 7.000 per kilogram, ditagih per kilogram bulat, minimum dua kilogram.
long long price(int grams) {
    int kg = (grams + 999) / 1000;
    if (kg < 2) kg = 2;
    return 7000LL * kg;
}
// ---- Soal 8.4: antreannya, tenggat terakhir lebih dahulu ----
void printLatestFirst(Index t) {
    if (isEmpty(t)) return;
    printLatestFirst(rightOf(t));
    cout << "    " << clockOf(key(t)) << "  " << setw(7) << left << t->v.id
         << right << setw(6) << t->v.grams << " g   " << rupiah(price(t->v.grams)) << "\n";
    printLatestFirst(leftOf(t));
}
void audit(const char* when, Index t) {
    string w1 = "-", w2 = "-";
    bool bst = auditBST(t, -1, 100000, w1);
    bool bal = auditHeight(t, w2) != -2;
    cout << "  audit " << when << ": orders=" << countNodes(t)
         << ", height=" << height(t)
         << ", search-tree=" << (bst ? "OK" : "BROKEN")
```

Kode 8.13 (Lanjutan) *Indeks pengambilan Profitina Laundry, dibangun seluruhnya dari kedelapan penyelesaian soal pohon bab ini, dengan audit-diri yang menyebut rekaman yang bercacat dan pembangunan ulang yang selamat dari urutan yang rusak.*

```
         << ", height-balanced=" << (bal ? "OK" : "BROKEN") << "\n";
    if (!bst) cout << "      " << w1 << "\n";
    if (!bal) cout << "      " << w2 << "\n";
}

int main() {
    vector<Order> book = {
        {  0, "PL-0431", 3200}, { 95, "PL-0432", 7400}, { 40, "PL-0433", 1500},
        {185, "PL-0434", 5000}, { 20, "PL-0435", 2600}, {260, "PL-0436", 12000},
        {130, "PL-0437",  900}, {305, "PL-0438", 4100}, { 75, "PL-0439", 6300},
        {215, "PL-0440", 2000}
    };
    Index idx = emptyIndex();
    for (const Order& o : book) idx = insertHB(o, idx);

    cout << "Profitina Laundry, Keputih -- pickup index for today\n";
    audit("after load", idx);
    cout << "  earliest pickup : " << clockOf(earliest(idx).due) << "  " <<
earliest(idx).id << "\n";
    cout << "  latest pickup   : " << clockOf(latest(idx).due)   << "  " << latest(idx).id
<< "\n";
    cout << "  queue, latest deadline first:\n";
    printLatestFirst(idx);

    long long total = 0;
    for (const Order& o : book) total += price(o.grams);
    cout << "  billed today    : " << rupiah(total) << "\n\n";

    cout << "PL-0437 collected at 09:10; PL-0433 collected at 07:40\n";
    idx = deleteHB(idx, 130);
    idx = deleteHB(idx, 40);
    audit("after two collections", idx);

    cout << "\nA walk-in at 11:00 with 8.5 kg, promised for 13:20 (minute 380)\n";
    idx = insertHB({380, "PL-0441", 8500}, idx);
    audit("after the walk-in", idx);
    cout << "  new latest pickup: " << clockOf(latest(idx).due) << "  " << latest(idx).id
<< "\n";

    cout << "\nNow a bad hand-edit: somebody retypes one deadline in place.\n";
    Index victim = idx;                     // cari simpul terkiri lalu geser tenggatnya
jauh ke belakang
    while (!isEmpty(leftOf(victim))) victim = leftOf(victim);
    cout << "  editing " << victim->v.id << " from " << clockOf(victim->v.due)
         << " to " << clockOf(500) << " without re-filing it\n";
    victim->v.due = 500;
    audit("after the bad edit", idx);
```

Kode 8.13 (Lanjutan) *Indeks pengambilan Profitina Laundry, dibangun seluruhnya dari kedelapan penyelesaian soal pohon bab ini, dengan audit-diri yang menyebut rekaman yang bercacat dan pembangunan ulang yang selamat dari urutan yang rusak.*

```
    cout << "\nRepair: read the index out in order and rebuild it with insertHB.\n";
    vector<Order> salvage;
    collectAll(idx, salvage);      // penelusuran inorder tetap menjangkau tiap pesanan
    sort(salvage.begin(), salvage.end(), [](const Order& a, const Order& b){ return a.due <
b.due; });
    Index rebuilt = emptyIndex();
    for (const Order& o : salvage) rebuilt = insertHB(o, rebuilt);
    audit("after rebuild", rebuilt);
    cout << "  queue, latest deadline first:\n";
    printLatestFirst(rebuilt);
    return 0;
}
```

Keluarannya:

```
Profitina Laundry, Keputih -- pickup index for today
  audit after load: orders=10, height=3, search-tree=OK, height-balanced=OK
  earliest pickup : 07:00  PL-0431
  latest pickup   : 12:05  PL-0438
  queue, latest deadline first:
    12:05  PL-0438  4100 g   Rp 35.000
    11:20  PL-0436 12000 g   Rp 84.000
    10:35  PL-0440  2000 g   Rp 14.000
    10:05  PL-0434  5000 g   Rp 35.000
    09:10  PL-0437   900 g   Rp 14.000
    08:35  PL-0432  7400 g   Rp 56.000
    08:15  PL-0439  6300 g   Rp 49.000
    07:40  PL-0433  1500 g   Rp 14.000
    07:20  PL-0435  2600 g   Rp 21.000
    07:00  PL-0431  3200 g   Rp 28.000
  billed today    : Rp 350.000

PL-0437 collected at 09:10; PL-0433 collected at 07:40
  audit after two collections: orders=8, height=3, search-tree=OK, height-balanced=OK

A walk-in at 11:00 with 8.5 kg, promised for 13:20 (minute 380)
  audit after the walk-in: orders=9, height=3, search-tree=OK, height-balanced=OK
  new latest pickup: 13:20  PL-0441

Now a bad hand-edit: somebody retypes one deadline in place.
  editing PL-0431 from 07:00 to 15:20 without re-filing it
  audit after the bad edit: orders=9, height=3, search-tree=BROKEN, height-balanced=OK
      order PL-0431 (minute 500) is filed where only minutes in (-1,20) belong

Repair: read the index out in order and rebuild it with insertHB.
  audit after rebuild: orders=9, height=3, search-tree=OK, height-balanced=OK
  queue, latest deadline first:
    15:20  PL-0431  3200 g   Rp 28.000
    13:20  PL-0441  8500 g   Rp 63.000
    12:05  PL-0438  4100 g   Rp 35.000
    11:20  PL-0436 12000 g   Rp 84.000
    10:35  PL-0440  2000 g   Rp 14.000
    10:05  PL-0434  5000 g   Rp 35.000
    08:35  PL-0432  7400 g   Rp 56.000
    08:15  PL-0439  6300 g   Rp 49.000
    07:20  PL-0435  2600 g   Rp 21.000
```

Empat hal yang perlu diperhatikan, masing-masing adalah salah satu soal bab ini yang sedang mengenakan celemek.

Hitungan penagihannya layak diperiksa dengan tangan, sebab ia jenis aturan yang memang dimiliki toko yang sesungguhnya. PL-0437 seberat 900 g — di bawah satu kilogram — tetapi

tagihan minimumnya dua kilogram, jadi ia ditagih $2 \times$ Rp 7.000 = Rp 14.000, sama dengan PL-0440 yang tepat 2.000 g dan PL-0433 yang 1.500 g dibulatkan ke atas menjadi 2 kg. PL-0436 seberat 12.000 g, tepat 12 kg, jadi $12 \times$ Rp 7.000 = Rp 84.000. Menjumlahkan kesepuluhnya: $28 + 56 + 14 + 35 + 21 + 84 + 14 + 35 + 49 + 14$ = Rp 350.000 dalam ribuan, dan itulah yang dicetak programnya.

Auditnya berbunyi `OK` tiga kali lalu `BROKEN` sekali, dan baris yang rusak itu menyebut pesanannya dan selang yang dilanggarnya — "order PL-0431 (minute 500) is filed where only minutes in (-1,20) belong". Batas atas 20 datang dari PL-0435, dua aras di atasnya. Itulah mekanisme **Tabel 8.5**, di dalam sebuah toko.

`height-balanced` tetap `OK` di sepanjang kerusakannya. Tentu saja demikian: memindahkan sebuah nilai tidak mengubah bentuknya. Inilah gunanya menjalankan *kedua* auditnya — masing-masing buta terhadap ragam kegagalan yang lain, dan sistem yang hanya memeriksa satu invarian adalah sistem yang memercayai separuh dari yang seharusnya.

Pembangunan ulangnya memulihkan kesembilan pesanannya. Penelusuran inorder menjangkau setiap simpul apa pun kata kuncinya, dan setelah diurutkan menurut tenggat yang sebenarnya lalu disisipkan ulang dengan `insertHB`, indeksnya benar kembali — dan PL-0431, yang tenggatnya memang sungguh berpindah ke pukul 15:20, kini sudah sepatutnya menjadi pengambilan terakhir hari itu. Pelanggannya terlambat; bukunya kini mengatakan begitu.

8.20 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Daftar di akhir tiap soal menyebutkan kekeliruan yang khas bagi soal itu. Yang berikut ini adalah kekeliruan yang memotong seluruh bab.

Membaca soalnya sekali saja. Kebanyakan soal di sini membawa kendala yang tidak ada di dalam nama fungsinya: *rekursif*, *tanpa membuat daftar terpisah*, *paling banyak satu lintasan*, *harus memakai kedua penunjang*, *satu lintasan*. Setiap kendala adalah isi sebenarnya dari soalnya, dan setiap kendala mudah terlewat pada pembacaan yang sambil lalu. Garis bawahi setiap kendala sebelum Anda mulai.

Menguji hanya jalur yang mulus. Pohon kosong, simpul tunggal, pohon dua simpul, dan kunci yang tidak ada menemukan lebih banyak kutu daripada contoh lima belas simpul mana pun. Program-program pohon bab ini mengujinya semua; tak satu pun kutu yang ditemukannya memakan lebih dari sepuluh detik untuk dibetulkan begitu ia ditemukan.

Memercayai medan tersimpan. Tinggi yang tersimpan di dalam sebuah simpul adalah *klaim*, bukan fakta. Pemeriksa harus menghitung ulang; itulah seluruh maksud **Kode 8.7** mendeklarasikan tipe simpulnya tanpa medan tinggi, dan maksud Soal 8.18 pada **Kode 8.12** membandingkan ukuran tersimpannya terhadap kebenarannya.

Memeriksa secara lokal apa yang benar secara global, atau secara global apa yang benar secara lokal. Sifat BST butuh batas; sifat heap tidak; sifat 4 merah-hitam tidak; sifat 5 merah-hitam butuh nilai yang dikembalikan dari bawah. Sebelum menulis pemeriksa apa pun, tanyakan: *apakah ini pernyataan tentang sebuah simpul dan anak-anaknya, atau tentang sebuah simpul dan seluruh upapohonnya?*

Memanggil penyeimbang ulang yang dinamai menurut sisi tempat Anda berekursi. Pada penyisipan, ya. Pada penghapusan, **tidak** — justru kebalikannya. §8.13.3.

Menyatakan sebuah kompleksitas alih-alih mengukurnya. Setiap klaim dalam bab ini diperiksa oleh pencacah di dalam sebuah program: 4 083 pertukaran untuk membangun *heap* dari 4 095 kunci pada Soal 8.1, 522 753 perbandingan lawan 8 194 untuk kasus terburuk dan terbaik Quicksort pada Soal 8.2, 4 simpul lawan 15 pada Soal 8.5, 769 lawan 127 pada Soal 8.6, 3^{n-1} panggilan lawan $1 + (n^3 - n)/3$ pada Soal 8.16, dan 6 rotasi pada pohon Fibonacci bertinggi 12 pada Soal Latihan 7. Pencacah berbiaya tiga baris dan mengubah pendapat menjadi bukti.

Mengatakan $O(\lg n)$ padahal yang Anda maksud $O(h)$. Benar hanya dengan jaminan keseimbangan. §8.8.5.

Melakukan heapify atau partisi dengan batas gelung yang keliru. `BUILD-MAX-HEAP` harus menghitung *mundur* dari $\lfloor n/2 \rfloor$; gelung Lomuto menjalankan j dari p sampai $r - 1$ dengan i dimulai di $p - 1$; Quicksort berekursi pada $A[p..q-1]$ dan $A[q+1..r]$. Setiap kekeliruan selisih-satu ini menghasilkan program yang berjalan dan keliru. §8.3.9 dan §8.4.9.

Mengisi tabel pemrograman dinamis dalam urutan yang keliru. Sebuah sel hanya boleh dihitung setelah setiap sel yang dibacanya. Untuk rantai matriks itu berarti panjang rantai yang naik, bukan baris demi baris; untuk *longest common subsequence*, baris demi baris berhasil sebab setiap sel hanya membaca baris di atasnya dan sel di kirinya. §8.15.12 dan §8.16.9.

Menyimpan nilainya dan membuang pilihannya. $M[1,5] = 1\,344$ mengatakan berapa biaya pengurungan terbaik, bukan pengurungan itu sendiri; $c[6,5] = 4$ mengatakan berapa panjang LCS-nya, bukan LCS itu sendiri. Simpanlah tabel pemisah dan tabel panahnya, atau jawabannya tak dapat dibangun ulang.

8.21 Ringkasan Bab

- Bab ini menghimpun Bab 5 sampai 7 lewat **dua puluh lima soal yang diselesaikan tuntas**, yang didaftar dalam **Tabel 8.1**, dan menimba dari setiap bab terdahulu, sebagaimana diperlihatkan **Tabel 8.2**. Hampir setiap penyelesaian adalah satu gagasan: bawa informasi turun menyusuri rekursi atau kembalikan ke atas, sehingga tak ada kerja yang dikerjakan dua kali.
- **Soal bertingkat diselesaikan menurut urutan ketergantungannya**, bukan urutan nomornya, dan setiap bagiannya dimulai dengan lima hal tertulis: tanda tangan fungsi, kasus dasar, invarian, contoh gagal terkecil, dan klaim kompleksitas (§8.2).
- **Soal 8.1, `BUILD-MAX-HEAP`.** Lakukan *heapify* pada simpul-simpul dalam dari $\lfloor n/2 \rfloor$ turun sampai 1; pada [5,13,2,25,7,17,20,8,4,11] itu berarti 15 perbandingan dan 6 pertukaran (**Tabel 8.3**). Biayanya $O(n)$ karena kebanyakan simpul berada di bawah (Persamaan (8.1)), dan pada *heap* sempurna pertukarannya tak pernah melebihi $n - \lg(n+1)$. **Kode 8.1**.
- **Soal 8.2, Quicksort.** Partisi Lomuto melakukan tepat $r - p$ perbandingan dan meninggalkan pivotnya di tempat akhirnya (**Tabel 8.4**). Masukan terurut berbiaya $\frac{n(n-1)}{2}$ perbandingan (Persamaan (8.3)); pembelahan yang seimbang sempurna berbiaya $(n+1)\lg(n+1) - 2n$ (Persamaan (8.4)); urutan acak mendarat dekat kasus terbaik. **Kode 8.2**.
- **Soal 8.3, `isBST`.** Sifat BST bersifat global, sehingga pemeriksaan induk–anak itu perlu tetapi tidak cukup; perbaikannya adalah membawa selang terbuka (ℓ, u) turun menyusuri rekursinya, dengan memotongnya di nilai setiap simpul. Mengawali batas bertipe `int` dengan `INT_MIN` dan `INT_MAX` adalah penyelesaian keliru yang menggoda — ia menolak pohon sah yang memuat salah satunya — dan melebarkan tipe batasnya memperbaikinya. Penelusuran inorder adalah cara kedua yang berdiri sendiri. **Kode 8.4**.
- **Soal 8.4, `printDescending`.** Inorder terbalik: kanan, simpul, kiri. Empat baris, tanpa daftar, waktu $\Theta(n)$ dan tumpukan $O(h)$. **Kode 8.5**.

- **Soal 8.5, `max`.** Buanglah setiap subpohon kiri tanpa memeriksanya lalu berjalanlah menyusuri tulang punggung kanannya. $O(h)$, yang menjadi $O(\lg n)$ hanya karena pohonnya seimbang — katakanlah begitu. **Kode 8.6**.
- **Soal 8.6, `isHeightBalanced`.** Kembalikan tingginya dan gagalkan dengan penjaga -2, yang gratis sebab tak ada tinggi sungguhan di bawah -1. Satu lintasan, $\Theta(n)$, berhadapan dengan $(n+1)\lg(n+1) - n$ untuk versi hitung-ulang-semuanya (Persamaan (8.10)) — sebuah fakta yang diturunkan, lalu diukur, lalu ditegaskan Teorema Master. **Kode 8.7**, **Tabel 8.6**.
- **Soal 8.7 sampai 8.10** adalah satu rancangan. `node()` adalah satu-satunya konstruktor dan ia membetulkan setiap medan turunan, sehingga rotasinya berupa ungkapan tunggal dan tinggi yang basi menjadi mustahil. `insertHB` adalah penyisipan BST biasa dengan sebuah penyeimbang ulang membungkus setiap simpul yang dibangun ulang; `rebalanceForLeft` adalah empat kasus yang diringkas menjadi dua; `rebalanceForRight` cerminnya; `deleteHB` memanggil penyeimbang ulang **yang berlawanan** dengan sisi tempat ia berekursi. Uji kasusnya harus ≤ 0, bukan < 0, sebab penghapusan dapat menyodorkan anak yang rata. **Kode 8.8**, **Tabel 8.7**.
- **Soal 8.11 sampai 8.15, pohon merah-hitam**, berkerangka sama dengan soal-soal AVL: `isBST` yang sama, pemeriksaan lokal yang *memang* benar (sifat 4), pemeriksaan satu lintasan dengan penjaga (tinggi-hitam), sebuah perakitan, dan dua rotasi — kali ini imperatif, dengan enam penunjuk induk yang harus dibetulkan. **Kode 8.9**, **Tabel 8.8**, **Tabel 8.9**.
- **Soal 8.16, rantai matriks.** Mengisi M menurut panjang rantai yang naik, Persamaan (8.18) memberikan $M[1,5] = 1\,344$ untuk $d = \langle 4,10,3,12,20,7 \rangle$ dengan pengurungan $((A_1A_2)((A_3A_4)A_5))$, dibandingkan $1\,784$ dari kiri ke kanan. Rekursi polos melakukan 3^{n-1} panggilan; *memoization* melakukan $1 + (n^3 - n)/3$. **Kode 8.10**, **Tabel 8.10**, **Tabel 8.11**.
- **Soal 8.17, *longest common subsequence*.** Mengisi c baris demi baris dengan Persamaan (8.20) lalu mengikuti panah-panahnya mundur memberikan SORE untuk SORTED dan STORE, dalam waktu $\Theta(mn)$ untuk tabelnya dan paling banyak $m + n$ langkah untuk jalan mundurnya. **Kode 8.11**, **Tabel 8.12**.
- **Soal 8.18 sampai 8.25** mengulang soal-soal pohon pada pohon statistik-urutan dan memperlihatkan bahwa medan turunan kedua hanya berbiaya satu baris bila setiap pembangunan melewati satu konstruktor. **Kode 8.12**, **Tabel 8.13**.
- Segala yang ada di sini diverifikasi alih-alih dinyatakan: setiap pelacakan tangan direproduksi sebuah program, setiap cacahan bentuk tertutup dibandingkan dengan cacahan terukur, dan rutin-rutin AVL diperiksa terhadap keenam contoh terkerjakan Bab 6 dan uji tekanan seribu elemen terhadap batas tinggi $1.4405\lg(n+2) - 1.3277$.

8.21.1 Menatap ke depan

Bab 9 meninggalkan pohon menuju **algoritma greedy** (rakus) — pemilihan aktivitas, pengodean Huffman, dan teori matroid yang mengatakan kapan kerakusan itu aman. Kebiasaan yang dibangun bab ini, yaitu bertanya *invarian apa yang dipelihara kode saya, dan bagaimana saya akan memeriksanya?*, adalah kebiasaan yang paling dibutuhkan Bab 9, sebab sebuah algoritma greedy benar tepat ketika sebuah invarian pertukaran tertentu selamat melewati setiap langkahnya.

8.22 Soal Latihan

1. Telusurilah `isBST` pada pohon $40\{20\{10,45\}, 60\{50,70\}\}$. Berikan pasangan (ℓ, u) yang diterima tiap panggilan, dalam urutan panggilannya dibuat, lalu sebutkan simpul mana yang pertama melanggar batasnya.

2. Berapa jumlah simpul paling sedikit di dalam sebuah pohon yang membuat pemeriksaan lokal induk–anak lulus padahal pohonnya bukan BST? Buktikan bahwa dua simpul tidak pernah dapat melakukannya.

3. Ubahlah `printDescending` sehingga ia hanya mencetak k nilai terbesar lalu berhenti, tanpa mengunjungi simpul lain mana pun. Nyatakan waktu jalannya dalam k dan h.

4. Tulislah `secondLargest(MyTree t)`, yang mengembalikan nilai terbesar kedua sebuah BST dalam waktu $O(h)$. Tanganilah: maksimumnya punya subpohon kiri; maksimumnya sebuah daun; akarnya sendiri adalah maksimumnya; pohonnya punya kurang dari dua nilai.

5. Hitunglah, dengan bentuk tertutup §8.9.3, banyaknya pemeriksaan simpul yang dilakukan `isHeightBalancedSlow` pada pohon sempurna bertinggi 4, lalu periksalah terhadap **Kode 8.14**.

6. Sisipkanlah 1,2,3,4,5,6,7 dalam urutan itu ke dalam pohon AVL kosong dengan memakai `insertHB`. Berikan bentuk dan cacahan rotasinya setelah tiap penyisipan, lalu sebutkan nama kasus untuk tiap rotasinya.

7. Bangunlah pohon AVL bersimpul minimum bertinggi 6 — pohon Fibonacci $N(6)$. Berapa simpul yang dimilikinya? Hapuslah kunci terbesarnya lalu cacahlah penyeimbangan ulangnya. Lalu katakan bagaimana cacahannya tumbuh terhadap h.

8. Susunlah sebuah pohon AVL sembilan simpul beserta sebuah penghapusan darinya sedemikian sehingga `rebalanceForLeft` yang ditulis dengan `< 0` alih-alih `<= 0` menghasilkan pohon yang bukan pohon AVL. Berikan kedua hasilnya.

9. Untuk pohon merah-hitam pada **Kode 8.9** — akar 26:B, dengan 17:R memegang 14:B dan 21:B, serta 41:B memegang 47:R — hitunglah `countBlackNodes`, tinggi-hitam akarnya, dan tinggi-hitam simpul 41. Lalu warnailah ulang 41 menjadi merah dan sebutkan sifat mana yang rusak.

10. Selesaikan $T(n) = 2T(n/2) + \Theta(n)$ dan $T(n) = 2T(n/2) + \Theta(1)$ dengan Teorema Master, sebutkan kasusnya masing-masing, lalu katakan rutin Soal 8.6 yang mana yang diuraikan oleh masing-masingnya.

11. Jalankan `BUILD-MAX-HEAP` dengan tangan pada $A = [1,2,3,4,5,6,7]$. Berikan lariknya setelah setiap panggilan `MAX-HEAPIFY`, hitung perbandingan dan pertukarannya, lalu bandingkan cacahan pertukarannya dengan batas $n - \lg(n + 1)$ di §8.3.6.

12. Terapkan `PARTITION(A, 1, 5)` pada $A = [5,5,5,5,5]$. Di mana pivotnya berakhir? Berapa banyak perbandingan yang dilakukan `QUICKSORT` selengkapnya pada n kunci yang sama, dan apa artinya bagi masukan yang banyak memuat nilai berulang?

13. Selesaikan rantai matriks $d = \langle 5,4,6,2,7 \rangle$ dengan tangan: isilah M dan c, berikan biaya terkecil dan pengurungan optimalnya, lalu bandingkan dengan urutan dari kiri ke kanan.

14. Temukan sebuah *longest common subsequence* dari $X = $ LAUNDRY dan $Y = $ SUNDAY dengan mengisi tabel **Pseudokode 8.6** lalu mengikuti panah-panahnya mundur. Apakah ia satu-satunya?

8.23 Penyelesaian Soal Latihan

1. **Tabel 8.14** mendaftar panggilannya, berurutan, beserta batas yang diterima masing-masing.

Tabel 8.14 Soal latihan 1: rekursinya berhenti di simpul 45, yang batas atasnya 40 datang dari akarnya dua aras di atas. Simpul 60, 50, dan 70 tak pernah dikunjungi.

Panggilan	Simpul	ℓ	u	Putusan
1	40	$-\infty$	$+\infty$	di dalam; berekursi
2	20	$-\infty$	40	di dalam; berekursi
3	10	$-\infty$	20	di dalam; daun; **true**
4	45	20	40	$45 \geq 40$ — **melanggar batas atasnya**

Simpul 45 adalah pelanggaran yang pertama. Pemeriksaan lokal akan lulus di mana-mana: $10 < 20 < 45$ di simpul 20, dan $20 < 40 < 60$ di akarnya.

2. Tiga simpul mencukupi, sebagaimana ditunjukkan §8.6.3: akar 20, anak kiri 10, dan anak kanan 10 yaitu 25. Pemeriksaan lokalnya lulus ($10 < 20$ dan $25 > 10$) tetapi $25 > 20$ terletak di subpohon kirinya.

Dua simpul tidak bisa. Dengan dua simpul ada sebuah akar dan satu simpul lain, dan simpul itu adalah anak kiri atau anak kanan akarnya. Sifat BST untuk pohon semacam itu mengatakan tepat satu hal — *anak kirinya lebih kecil* atau *anak kanannya lebih besar* — dan itu persis pemeriksaan lokalnya. Tak ada relasi leluhur lawan cucu yang dapat terlewat, sebab tak ada cucu. Karena itu pemeriksaan lokalnya setara dengan sifat BST pada setiap pohon berukuran paling banyak dua simpul, dan tiga adalah minimumnya. (Satu simpul dan nol simpul sepele-sepele saja sama halnya.)

3.

```
void printKLargest(MyTree t, int& k) {
    if (isEmpty(t) || k == 0) return;      // berhenti begitu kuotanya terpenuhi
    printKLargest(t->right, k);
    if (k == 0) return;
    cout << t->value << ' '; --k;
    printKLargest(t->left, k);
}
```

Pencacah `k` harus dipakai bersama di sepanjang rekursinya — sebuah rujukan di C++ — karena alasan yang diberikan §8.6.8. Waktu jalannya: penelusurannya menuruni tulang punggung kanan sekali, berbiaya $O(h)$, dan sesudah itu masing-masing dari k nilai yang tercetak berbiaya $O(1)$ teramortisasi ditambah paling banyak satu penurunan tambahan, sehingga menjadi $O(h + k)$. Ruangnya $O(h)$.

4. Pengamatan kuncinya: maksimumnya ada di ujung tulang punggung kanan. Pendahulunya — yang terbesar kedua — adalah maksimum subpohon kiri *milik maksimum itu*, jika ia punya, atau kalau tidak maka induknya. Tiga kasus, satu lintasan, $O(h)$. `secondLargest` pada **Kode 8.14** adalah kodenya. Keluarannya meliputi ketiga bentuknya:

```
=== Exercise 4: second largest, in O(h) ===
  50(0){30(0){20(0),40(0)},70(0){60(0),80(0)}}
  largest = 80, second largest = 70
  50(1){30(0),70(-1){65(0),.}}
  largest = 70, second largest = 65
  50(-1){30(0),.}
  largest = 50, second largest = 30
```

Baris 2: maksimumnya 80 adalah daun, jadi jawabannya induknya, 70. Baris 4: maksimumnya 70 punya anak kiri, jadi jawabannya adalah maksimum subpohon itu, 65. Baris 6: akar 50 adalah maksimumnya, jadi jawabannya adalah maksimum subpohon kirinya, 30.

5. Pohon sempurna bertinggi $k = 4$ punya $n = 2^5 - 1 = 31$ simpul. Bentuk tertutupnya memberikan

$$W = (n+1)\lg(n+1) - n = 32 \times 5 - 31 = 160 - 31 = 129.$$

Kode 8.14 mencetak `31 129 129 31 match`. Cara satu-lintasannya memeriksa 31 simpul, satu per simpul, sebagaimana seharusnya.

6. Tabel 8.15 memberikan jawabannya, dihasilkan oleh `insertHB` yang sama dengan **Kode 8.8**.

Tabel 8.15 Soal latihan 6: menyisipkan 1 sampai 7 dalam urutan menaik. Setiap rotasinya adalah kasus Right-Right, sebab masukan menaik selalu condong ke kanan, dan empat rotasi tunggal sudah cukup untuk menjaga tingginya tetap 2 alih-alih 6.

Sisip	Rotasi	Kasus	Bentuk, beserta faktor keseimbangannya
1	0	—	`1(0)`
2	0	—	`1(1){.,2(0)}`
3	1	RR	`2(0){1(0),3(0)}`
4	0	—	`2(1){1(0),3(1){.,4(0)}}`
5	1	RR	`2(1){1(0),4(0){3(0),5(0)}}`
6	1	RR	`4(0){2(0){1(0),3(0)},5(1){.,6(0)}}`
7	1	RR	`4(0){2(0){1(0),3(0)},6(0){5(0),7(0)}}`

7. $N(h) = 1 + N(h-1) + N(h-2)$ dengan $N(0) = 1$ dan $N(1) = 2$ memberikan $N(2) = 4$, $N(3) = 7, N(4) = 12, N(5) = 20, N(6) = \mathbf{33}$ — cocok dengan $N(h) = F(h+3) - 1$, sebab $F(9) = 34$. **Kode 8.14** mencetak untuk $h = 6$: 33 simpul, **3 rotasi**, tingginya turun dari 6 ke 5, pohonnya masih AVL. Tabel yang dicetaknya untuk $h = 2$ sampai 12 memberikan cacahan rotasi 1,1,2,2,3,3,4,4,5,5,6 — itu $\lfloor h/2 \rfloor$, dan karena $h = \Theta(\lg n)$ maka banyaknya penyeimbangan ulang yang disebabkan satu penghapusan adalah $\Theta(\lg n)$. Bandingkan dengan paling-banyak-satu penyeimbangan ulang pada penyisipan (§8.13.4).

8. Pohon sembilan simpul

$$50\{\,20\{10\{5,15\},\ 30\{.,40\}\},\ 70\{60,.\}\}$$

adalah pohon AVL: $\mathrm{bf}(10) = 0$, $\mathrm{bf}(30) = +1$, $\mathrm{bf}(20) = 0$, $\mathrm{bf}(70) = -1$, $\mathrm{bf}(50) = 1 - 2 = -1$. Hapuslah 60. Simpul 70 menjadi daun, sehingga di simpul 50 kita peroleh $\mathrm{bf}(50) = 0 - 2 = -2$ dengan $\mathrm{bf}(20) = 0$ — anak yang rata, yang hanya dapat dihasilkan oleh penghapusan. **Kode 8.14** menjalankan kedua versinya:

```
=== Exercise 8: why the case test needs <= and not < ===
  start          : 50(-1){20(0){10(0){5(0),15(0)},30(1){.,40(0)}},70(-1){60(0),.}}
AVL=yes
  delete 60, <=0 : 20(1){10(0){5(0),15(0)},50(-1){30(1){.,40(0)},70(0)}}   AVL=yes
  delete 60, <0  : 30(-1){20(-2){10(0){5(0),15(0)},.},50(0){40(0),70(0)}}   AVL=NO
```

Dengan `<= 0` cabang LL melakukan satu rotasi kanan dan hasilnya adalah pohon AVL yang sah. Dengan `< 0` cabang LR melakukan rotasi ganda, dan simpul 20 berakhir dengan faktor keseimbangan -2: hasilnya adalah pohon pencarian biner dengan nilai yang benar dalam urutan yang benar, dan ia tidak seimbang-tinggi. Inilah kutu yang diramalkan §8.11.5, diperagakan pada pohon terkecil yang memperlihatkannya.

9. Simpul hitamnya: 26, 14, 21, 41 — **empat**. Tinggi-hitam akarnya: setiap lintasan ke sebuah `NIL` melewati dua simpul hitam tanpa menghitung `NIL`-nya; misalnya 26 → 17 → 14 → NIL mencacah 26 dan 14, dan 26 → 41 → 47 → NIL mencacah 26 dan 41. Jadi **2**. Tinggi-hitam simpul 41: lintasan 41 → NIL dan 41 → 47 → NIL masing-masing mencacah 41 saja, jadi **1**. **Kode 8.9** mencetak `black-height of the root = 2`.

Warnailah ulang 41 menjadi merah. Sifat 4 masih terpenuhi — anak-anak 41 adalah `NIL` yang hitam dan 47 yang merah — bukan: 47 *memang* merah, jadi 41 yang merah dengan anak merah 47 seketika merusak **sifat 4**. Sifat 5 juga rusak: lintasan 26 → 41 → NIL kini hanya mencacah 26, sehingga tinggi-hitamnya 1 di sisi itu berhadapan dengan 2 di sisi kiri. Baik `validateProperty4` maupun `validateBlackHeight` menolaknya.

10. $T(n) = 2T(n/2) + \Theta(n)$: $a = 2$, $b = 2$, $\log_b a = 1$, $n^{\log_b a} = n$, dan $f(n) = \Theta(n) = \Theta(n^{\log_b a})$ — **kasus 2** — jadi $T(n) = \Theta(n\lg n)$. Ini `isHeightBalancedSlow`, yang membayar $\Theta(n)$ di tiap simpul untuk menghitung ulang tingginya.

$T(n) = 2T(n/2) + \Theta(1)$: a, b, dan $\log_b a$ yang sama, tetapi $f(n) = \Theta(1) = O(n^{1-\varepsilon})$ dengan $\varepsilon = 1$ — **kasus 1** — jadi $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$. Ini `checkedHeight`, cara satu-lintasan dengan penjaga.

11. Di sini $n = 7$, jadi gelungnya berjalan untuk $i = 3,2,1$.

- $i = 3$: simpul 3 memuat 3, dengan anak $A[6] = 6$ dan $A[7] = 7$; yang lebih besar, 7, melampaui 3, jadi tukar: [1,2,7,4,5,6,3]. Simpul 7 adalah daun. (2 perbandingan, 1 pertukaran.)
- $i = 2$: simpul 2 memuat 2, dengan anak 4 dan 5; tukar dengan 5: [1,5,7,4,2,6,3]. Simpul 5 adalah daun. (2 perbandingan, 1 pertukaran.)
- $i = 1$: akar 1 beranak 5 dan 7; tukar dengan 7: [7,5,1,4,2,6,3]. Di simpul 3 nilai 1 beranak 6 dan 3; tukar dengan 6: [7,5,6,4,2,1,3]. Simpul 6 adalah daun. (4 perbandingan, 2 pertukaran.)

Heap-nya adalah [7,5,6,4,2,1,3], setelah **8 perbandingan dan 4 pertukaran**. Batasnya adalah $n - \lg(n+1) = 7 - 3 = 4$, jadi masukan menaik mencapainya dengan persis: setiap nilai tenggelam sejauh mungkin. Inilah baris pertama tabel pengukuran yang dicetak **Kode 8.1**.

12. Setiap $A[j] = 5 \le 5$, jadi i maju di setiap langkah dan setiap pertukaran adalah pertukaran elemen dengan dirinya sendiri; baris 7 menukar $A[5]$ dengan dirinya sendiri dan mengembalikan $q = 5$. Pivotnya mendarat di ujung kanan, dan pembelahannya $n - 1$ lawan 0 — kasus terburuk §8.4.5. Rekursinya mengulang hal ini pada $n - 1$ kunci yang sama, lalu $n - 2$, dan seterusnya, sehingga pengurutan selengkapnya melakukan

$$C(n) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$$

perbandingan: 10 untuk $n = 5$, dan $\Theta(n^2)$ pada umumnya. Karena itu partisi Lomuto merosot pada masukan yang banyak memuat nilai berulang — persoalan yang nyata, sebab sebuah kolom harga atau tanggal sering memuat deretan panjang entri yang sama. Partisi Hoare dari Bab 5, yang berhenti pada kunci yang sama dari kedua sisi sehingga membelah deretan kunci yang sama di dekat tengahnya, atau partisi tiga-arah yang menyisihkan sekaligus semua kunci yang sama dengan pivot, menghindari persoalan itu.

13. $d_0, \ldots, d_4 = 5,4,6,2,7$.

- Panjang 2: $M[1,2] = 5 \cdot 4 \cdot 6 = 120$; $M[2,3] = 4 \cdot 6 \cdot 2 = 48$; $M[3,4] = 6 \cdot 2 \cdot 7 = 84$.
- $M[1,3] = \min(0 + 48 + 5 \cdot 4 \cdot 2,\ 120 + 0 + 5 \cdot 6 \cdot 2) = \min(88,180) = 88$, dengan $c[1,3] = 1$.

- $M[2,4] = \min(0 + 84 + 4 \cdot 6 \cdot 7,\ 48 + 0 + 4 \cdot 2 \cdot 7) = \min(252,104) = 104$, dengan $c[2,4] = 3$.
- $M[1,4] = \min(0 + 104 + 5 \cdot 4 \cdot 7,\ 120 + 84 + 5 \cdot 6 \cdot 7,\ 88 + 0 + 5 \cdot 2 \cdot 7) = \min(244,414,158) = 158$, dengan $c[1,4] = 3$.

Biaya terkecilnya 158. Membaca c: $c[1,4] = 3$ memisahkan A_4; $c[1,3] = 1$ memisahkan A_1 dari A_2A_3. Pengurungannya $((A_1(A_2A_3))A_4)$: mula-mula A_2A_3 seharga 48, menghasilkan 4×2; lalu A_1 dikalikan hasil itu seharga $5 \cdot 4 \cdot 2 = 40$, menghasilkan 5×2; lalu dikalikan A_4 seharga $5 \cdot 2 \cdot 7 = 70$. Jumlahnya $48 + 40 + 70 = 158$ ✓. Dari kiri ke kanan berbiaya $120 + 60 + 70 = 250$. Urutan pemenangnya mencapai dimensi kecil 2 lebih dahulu lalu mempertahankannya.

14. Dengan $X = $ L, A, U, N, D, R, Y (baris) dan $Y = $ S, U, N, D, A, Y (kolom), baris-baris tabelnya adalah:

- L: 0 ↑, 0 ↑, 0 ↑, 0 ↑, 0 ↑, 0 ↑ (L tidak ada dalam SUNDAY; setiap seri ke atas)
- A: 0 ↑, 0 ↑, 0 ↑, 0 ↑, 1 ↖, 1 ←
- U: 0 ↑, 1 ↖, 1 ←, 1 ←, 1 ↑, 1 ↑
- N: 0 ↑, 1 ↑, 2 ↖, 2 ←, 2 ←, 2 ←
- D: 0 ↑, 1 ↑, 2 ↑, 3 ↖, 3 ←, 3 ←
- R: 0 ↑, 1 ↑, 2 ↑, 3 ↑, 3 ↑, 3 ↑
- Y: 0 ↑, 1 ↑, 2 ↑, 3 ↑, 3 ↑, 4 ↖

Panjangnya $c[7,6] = 4$. Jalan mundurnya: (7,6) ↖, ambil Y; (6,5) ↑; (5,5) ←; (5,4) ↖, ambil D; (4,3) ↖, ambil N; (3,2) ↖, ambil U; (2,1) ↑; (1,1) ↑; berhenti. Dibalik: **UNDY**. Ia satu-satunya yang sepanjang 4: satu-satunya huruf bersama yang lain, A, datang *sebelum* U, N, dan D dalam LAUNDRY tetapi *sesudah* ketiganya dalam SUNDAY, sehingga ia tak dapat bergabung dengan satu pun di antaranya, dan subbarisan bersama yang memuat A paling panjang 2 (AY).

Kode 8.14 Penyelesaian yang dapat dijalankan untuk soal latihan 4, 5, 7, dan 8, dengan sakelar `STRICT` yang mereproduksi kutu `< 0` soal latihan 8 sesuai permintaan.

```
// exercises.cpp -- penyelesaian yang dapat dijalankan untuk soal latihan 4, 5 dan 7 bab 8.
#include <iostream>
#include <iomanip>
#include <string>
#include <stdexcept>
#include <vector>
#include <cmath>
#include <algorithm>
using namespace std;

struct MyTreeNode;
using MyTree = MyTreeNode*;
struct MyTreeNode { int value; MyTree left, right; int h; };

MyTree emptyTree()       { return nullptr; }
bool   isEmpty(MyTree t) { return t == nullptr; }
int    height(MyTree t)  { return isEmpty(t) ? -1 : t->h; }
int    value(MyTree t)   { return t->value; }
MyTree leftOf(MyTree t)  { return t->left; }
MyTree rightOf(MyTree t) { return t->right; }
MyTree node(int v, MyTree l, MyTree r) {
    MyTree t = new MyTreeNode; t->value = v; t->left = l; t->right = r;
    t->h = 1 + max(height(l), height(r)); return t;
}
MyTree leaf(int v) { return node(v, emptyTree(), emptyTree()); }
int balanceFactor(MyTree t) { return height(rightOf(t)) - height(leftOf(t)); }

long long rotations = 0;
bool STRICT = false;        // Soal latihan 8: true membuat uji kasusnya memakai < alih-
alih <=
MyTree rotateRight(MyTree t) { ++rotations; MyTree l = leftOf(t);
    return node(value(l), leftOf(l), node(value(t), rightOf(l), rightOf(t))); }
MyTree rotateLeft(MyTree t)  { ++rotations; MyTree r = rightOf(t);
    return node(value(r), node(value(t), leftOf(t), leftOf(r)), rightOf(r)); }
MyTree rebalanceForLeft(MyTree t) {
    if (balanceFactor(t) >= -1) return t;
    MyTree l = leftOf(t);
    bool single = STRICT ? (balanceFactor(l) < 0) : (balanceFactor(l) <= 0);
    if (single) return rotateRight(t);
    return rotateRight(node(value(t), rotateLeft(l), rightOf(t)));
}
MyTree rebalanceForRight(MyTree t) {
    if (balanceFactor(t) <= 1) return t;
    MyTree r = rightOf(t);
    bool single = STRICT ? (balanceFactor(r) > 0) : (balanceFactor(r) >= 0);
    if (single) return rotateLeft(t);
    return rotateLeft(node(value(t), leftOf(t), rotateRight(r)));
```

Kode 8.14 (Lanjutan) *Penyelesaian yang dapat dijalankan untuk soal latihan 4, 5, 7, dan 8, dengan sakelar STRICT yang mereproduksi kutu < 0 soal latihan 8 sesuai permintaan.*

```
}
MyTree insertHB(int n, MyTree t) {
    if (isEmpty(t)) return node(n, emptyTree(), emptyTree());
    if (n < value(t)) return rebalanceForLeft (node(value(t), insertHB(n, leftOf(t)),
rightOf(t)));
    if (n > value(t)) return rebalanceForRight(node(value(t), leftOf(t), insertHB(n,
rightOf(t))));
    return t;
}
int maxOf(MyTree t) { return isEmpty(rightOf(t)) ? value(t) : maxOf(rightOf(t)); }
MyTree deleteHB(MyTree t, int x) {
    if (isEmpty(t)) return t;
    if (x < value(t)) return rebalanceForRight(node(value(t), deleteHB(leftOf(t), x),
rightOf(t)));
    if (x > value(t)) return rebalanceForLeft (node(value(t), leftOf(t),
deleteHB(rightOf(t), x)));
    if (isEmpty(leftOf(t)))  return rightOf(t);
    if (isEmpty(rightOf(t))) return leftOf(t);
    int m = maxOf(leftOf(t));
    return rebalanceForRight(node(m, deleteHB(leftOf(t), m), rightOf(t)));
}
string shape(MyTree t) {
    if (isEmpty(t)) return ".";
    string s = to_string(value(t)) + "(" + to_string(balanceFactor(t)) + ")";
    if (isEmpty(leftOf(t)) && isEmpty(rightOf(t))) return s;
    return s + "{" + shape(leftOf(t)) + "," + shape(rightOf(t)) + "}";
}
int countNodes(MyTree t) { return isEmpty(t) ? 0 : 1 + countNodes(leftOf(t)) +
countNodes(rightOf(t)); }
bool isAVL(MyTree t) {
    if (isEmpty(t)) return true;
    return abs(balanceFactor(t)) <= 1 && isAVL(leftOf(t)) && isAVL(rightOf(t));
}

// ===== Soal latihan 4: nilai terbesar kedua sebuah BST, dalam O(h) =====
// Nilai terbesar ada di ujung tulang punggung kanan. Terbesar kedua adalah
// induk simpul itu, atau nilai terbesar di subpohon KIRI simpul itu.
int secondLargest(MyTree t) {
    if (isEmpty(t)) throw invalid_argument("the tree is empty");
    if (isEmpty(rightOf(t))) {                        // akarnya ITULAH yang terbesar
        if (isEmpty(leftOf(t))) throw invalid_argument("fewer than two values");
        return maxOf(leftOf(t));                      // terbesar di subpohon kiri
    }
    MyTree parent = t, cur = rightOf(t);
    while (!isEmpty(rightOf(cur))) { parent = cur; cur = rightOf(cur); }
    // cur memuat maksimumnya; parent adalah induknya
    if (!isEmpty(leftOf(cur))) return maxOf(leftOf(cur));
    return value(parent);
}
```

Kode 8.14 (Lanjutan) *Penyelesaian yang dapat dijalankan untuk soal latihan 4, 5, 7, dan 8, dengan sakelar STRICT yang mereproduksi kutu < 0 soal latihan 8 sesuai permintaan.*

```
// ===== Soal latihan 7: pohon AVL bersimpul minimum bertinggi h (pohon Fibonacci) =====
// N(-1)=0, N(0)=1, N(h)=1+N(h-1)+N(h-2). Kunci dibagikan secara inorder, jadi
// hasilnya benar-benar sebuah BST.
int nextKey;
MyTree fibTree(int h) {
    if (h < 0) return emptyTree();
    if (h == 0) return node(nextKey++, emptyTree(), emptyTree());
    MyTree l = fibTree(h - 1);
    int v = nextKey++;
    MyTree r = fibTree(h - 2);
    return node(v, l, r);
}

// ===== Soal latihan 5: biaya persis kedua pemeriksaan keseimbangan =====
long long work = 0;
int heightSlow(MyTree t) {
    if (isEmpty(t)) return -1;
    ++work;
    return 1 + max(heightSlow(leftOf(t)), heightSlow(rightOf(t)));
}
bool isHBSlow(MyTree t) {
    if (isEmpty(t)) return true;
    ++work;
    if (abs(heightSlow(leftOf(t)) - heightSlow(rightOf(t))) > 1) return false;
    return isHBSlow(leftOf(t)) && isHBSlow(rightOf(t));
}
int checkedHeight(MyTree t) {
    if (isEmpty(t)) return -1;
    ++work;
    int hl = checkedHeight(leftOf(t));  if (hl == -2) return -2;
    int hr = checkedHeight(rightOf(t)); if (hr == -2) return -2;
    if (abs(hl - hr) > 1) return -2;
    return 1 + max(hl, hr);
}
MyTree perfect(int lo, int hi) {
    if (lo > hi) return emptyTree();
    int mid = lo + (hi - lo) / 2;
    return node(mid, perfect(lo, mid - 1), perfect(mid + 1, hi));
}
```

Kode 8.14 (Lanjutan) *Penyelesaian yang dapat dijalankan untuk soal latihan 4, 5, 7, dan 8, dengan sakelar STRICT yang mereproduksi kutu < 0 soal latihan 8 sesuai permintaan.*

```
int main() {
    cout << "=== Exercise 4: second largest, in O(h) ===\n";
    {
        MyTree t = emptyTree();
        for (int v : {50, 30, 70, 20, 40, 60, 80}) t = insertHB(v, t);
        cout << "  " << shape(t) << "\n";
        cout << "  largest = " << maxOf(t) << ", second largest = " << secondLargest(t) <<
"\n";
        MyTree u = emptyTree();
        for (int v : {50, 30, 70, 65}) u = insertHB(v, u);      // maksimumnya punya anak
kiri
        cout << "  " << shape(u) << "\n";
        cout << "  largest = " << maxOf(u) << ", second largest = " << secondLargest(u) <<
"\n";
        MyTree w = emptyTree();
        for (int v : {50, 30}) w = insertHB(v, w);              // akarnya adalah
maksimumnya
        cout << "  " << shape(w) << "\n";
        cout << "  largest = " << maxOf(w) << ", second largest = " << secondLargest(w) <<
"\n";
    }

    cout << "\n=== Exercise 8: why the case test needs <= and not < ===\n";
    {
        MyTree base = node(50,
                           node(20, node(10, leaf(5), leaf(15)), node(30, emptyTree(),
leaf(40))),
                           node(70, leaf(60), emptyTree()));
        cout << "  start          : " << shape(base)
             << "   AVL=" << (isAVL(base) ? "yes" : "NO") << "\n";
        STRICT = false; MyTree a = deleteHB(base, 60);
        cout << "  delete 60, <=0 : " << shape(a) << "   AVL=" << (isAVL(a) ? "yes" : "NO")
<< "\n";
        STRICT = true;  MyTree b = deleteHB(base, 60);
        cout << "  delete 60, <0  : " << shape(b) << "   AVL=" << (isAVL(b) ? "yes" : "NO")
<< "\n";
        STRICT = false;
    }

    cout << "\n=== Exercise 7: one deletion, many rebalancings ===\n";
    cout << "  h   N(h)   deleting the largest key: rotations\n";
    for (int h = 2; h <= 12; ++h) {
        nextKey = 1;
        MyTree f = fibTree(h);
        int n = countNodes(f);
        rotations = 0;
        MyTree g = deleteHB(f, n);              // n adalah kunci terbesar menurut cara
membangunnya
        cout << "  " << setw(2) << h << setw(7) << n << setw(28) << rotations
             << "   (still AVL: " << (isAVL(g) ? "yes" : "no")
             << ", height " << height(f) << " -> " << height(g) << ")\n";
    }
```

Kode 8.14 (Lanjutan) *Penyelesaian yang dapat dijalankan untuk soal latihan 4, 5, 7, dan 8, dengan sakelar STRICT yang mereproduksi kutu < 0 soal latihan 8 sesuai permintaan.*

```
    cout << "\n=== Exercise 5: node inspections on a perfect tree ===\n";
    cout << "     n      slow    (n+1)lg(n+1)-n     fast\n";
    for (int k = 2; k <= 10; ++k) {
        int n = (1 << (k + 1)) - 1;
        MyTree p = perfect(1, n);
        work = 0; isHBSlow(p);        long long ws = work;
        work = 0; checkedHeight(p);   long long wf = work;
        long long closed = (long long)(n + 1) * (k + 1) - n;
        cout << setw(6) << n << setw(10) << ws << setw(18) << closed << setw(9) << wf
             << (ws == closed ? "   match" : "   MISMATCH") << "\n";
    }
    return 0;
}
```

Keluarannya selengkapnya:

```
=== Exercise 4: second largest, in O(h) ===
  50(0){30(0){20(0),40(0)},70(0){60(0),80(0)}}
  largest = 80, second largest = 70
  50(1){30(0),70(-1){65(0),.}}
  largest = 70, second largest = 65
  50(-1){30(0),.}
  largest = 50, second largest = 30

=== Exercise 8: why the case test needs <= and not < ===
  start          : 50(-1){20(0){10(0){5(0),15(0)},30(1){.,40(0)}},70(-1){60(0),.}}
AVL=yes
  delete 60, <=0 : 20(1){10(0){5(0),15(0)},50(-1){30(1){.,40(0)},70(0)}}   AVL=yes
  delete 60, <0  : 30(-1){20(-2){10(0){5(0),15(0)},.},50(0){40(0),70(0)}}   AVL=NO

=== Exercise 7: one deletion, many rebalancings ===
  h   N(h)   deleting the largest key: rotations
   2      4                           1   (still AVL: yes, height 2 -> 1)
   3      7                           1   (still AVL: yes, height 3 -> 2)
   4     12                           2   (still AVL: yes, height 4 -> 3)
   5     20                           2   (still AVL: yes, height 5 -> 4)
   6     33                           3   (still AVL: yes, height 6 -> 5)
   7     54                           3   (still AVL: yes, height 7 -> 6)
   8     88                           4   (still AVL: yes, height 8 -> 7)
   9    143                           4   (still AVL: yes, height 9 -> 8)
  10    232                           5   (still AVL: yes, height 10 -> 9)
  11    376                           5   (still AVL: yes, height 11 -> 10)
  12    609                           6   (still AVL: yes, height 12 -> 11)

=== Exercise 5: node inspections on a perfect tree ===
     n      slow    (n+1)lg(n+1)-n     fast
     7        17                17        7   match
    15        49                49       15   match
    31       129               129       31   match
    63       321               321       63   match
   127       769               769      127   match
   255      1793              1793      255   match
   511      4097              4097      511   match
  1023      9217              9217     1023   match
  2047     20481             20481     2047   match
```

Blok terakhirnya adalah bentuk tertutup §8.9.3 yang diperiksa terhadap mesinnya pada sembilan ukuran yang berbeda, dan ia cocok pada setiap satunya.

Daftar Pustaka

[1] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. 19th Annual Symposium on Foundations of Computer Science (FOCS)*, Ann Arbor, MI, USA, 1978, pp. 8–21.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[3] J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964.

[4] R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, vol. 7, no. 12, p. 701, 1964.

[5] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[6] G. M. Adelson-Velsky and E. M. Landis, "An algorithm for the organization of information," *Doklady Akademii Nauk SSSR*, vol. 146, no. 2, pp. 263–266, 1962. (English translation: *Soviet Mathematics Doklady*, vol. 3, pp. 1259–1263, 1962.)

[7] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.

[8] N. Wirth, *Algorithms + Data Structures = Programs*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1976.

[9] C. Okasaki, *Purely Functional Data Structures*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[10] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton Univ. Press, 1957.

[11] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974.

[12] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, 4th ed. Boston, MA, USA: Pearson, 2014.

[13] ISO/IEC 14882:2024, *Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2024.

Prosedur *heap* pada Soal 8.1 serta partisi dan analisis pada Soal 8.2 mengikuti Cormen dkk. [2, bab 6–7]; *heapsort* berasal dari Williams [3], pembangunan bawah-atas berwaktu linear dari Floyd [4], dan Quicksort dari Hoare [5], yang makalahnya juga memperkenalkan partisi dua-penunjuk di Bab 5. Invarian AVL dan batas tinggi $1.4405\lg(n+2) - 1.3277$ berasal dari [6] dan dianalisis dalam [7, §6.2.3]; *heapsort* dianalisis dalam [7, §5.2.3]. Rumusan *fungsional* yang dipakai untuk soal-soal pohon — satu konstruktor tunggal, rotasi sebagai ungkapan, `insertHB` dan `deleteHB` yang mengembalikan pohon yang dibangun ulang — adalah gaya [8] dan, dalam bentuk modernnya, [9]; itulah sebabnya kutu penyegaran tinggi di Bab 6 tak dapat terjadi di sini. Pohon merah-hitam dalam bentuk berwarnanya berasal dari Guibas dan Sedgewick [1], dan rotasi imperatif dengan penunjuk induk pada Soal 8.15 mengikuti [2, bab 13]. Pemrograman dinamis sebagai metode berasal dari Bellman [10]; algoritma rantai matriks dan *longest common subsequence* pada Soal 8.16 dan 8.17 mengikuti [2, bab 14], dan pendekatan pengisian tabel untuk membandingkan dua untai berakar pada Wagner dan Fischer [11]. Panduan praktis implementasi pohon seimbang ada di [12, bab 4], dan aturan bahasa yang dipatuhi setiap kode adalah [13].

Bab 9
Algoritma Rakus

Di akhir bab ini Anda akan sanggup memandang sebuah persoalan optimasi lalu mengatakan apakah mengambil pilihan yang jelas-jelas menarik, sekarang juga, tanpa memikirkan masa depan, kebetulan aman atau tidak. Anda akan sanggup menuliskan sebuah algoritma rakus dalam tiga baris dan — yang jauh lebih penting — *membuktikan* bahwa ia benar, dengan argumen penukaran (*exchange argument*), teknik pembuktian paling berharga dalam buku ini. Anda akan sanggup mengenali dua bahan yang harus dimiliki sebuah persoalan sebelum kerakusan boleh dipakai: sifat pilihan rakus dan substruktur optimal. Anda akan sanggup menyelesaikan persoalan pemilihan aktivitas secara rekursif maupun iteratif dalam $\Theta(n\lg n)$ dan membuktikan kedua bentuknya optimal. Anda akan sanggup menyebutkan dengan tepat mengapa kerakusan menyelesaikan masalah ransel pecahan dan justru merusak ransel 0-1, dengan angka-angkanya terbentang di depan mata. Anda akan sanggup membangun kode Huffman dengan tangan dan dengan C++, menyandikan sebuah berkas, menguraikan sandinya kembali, serta membuktikan kodenya optimal. Anda akan tahu apa itu matroid, mengapa matroid adalah jawaban terdalam yang kita punyai atas pertanyaan "kapan kerakusan berhasil?", dan bagaimana menjadwalkan tugas bertenggat dan berdenda dengan memakai satu matroid. Dan Anda akan tahu, ketika kelak berjumpa dengan Kruskal, Prim, dan Dijkstra di bab-bab berikutnya, bahwa Anda sudah pernah melihat bentuk bukti kebenaran mereka.

9.1 Dari Mana Kita Melanjutkan

Bab 7 mengajari kita *dynamic programming* (pemrograman dinamis). Bab itu memberi kita resep berlangkah empat (§7.8.4): telaah struktur sebuah solusi optimal, definisikan nilainya secara rekursif, hitung nilai itu dalam urutan yang menghormati kebergantungannya, lalu susunlah solusinya dari tabel berisi pilihan-pilihan yang tercatat. Bab itu memberi kita sifat yang mengizinkan seluruh usaha itu, yaitu *optimal substructure* (substruktur optimal) (§7.5.2): *sebuah solusi optimal bagi suatu persoalan memuat di dalam dirinya solusi-solusi optimal bagi submasalah penyusunnya.* Dan bab itu memberi kita cara baku untuk membuktikan sifat tersebut, yaitu **argumen potong-dan-tempel** (§7.5.3): andaikan sepotong bagian dari sebuah solusi optimal ternyata tidak optimal bagi submasalahnya sendiri, potonglah bagian itu, tempelkan potongan yang lebih baik, periksalah bahwa hasilnya masih sah, lalu perhatikanlah bahwa Anda baru saja mengalahkan sesuatu yang tadi Anda anggap tak terkalahkan.

Bab 7 juga mengajari kita berapa harga yang dituntut pemrograman dinamis. Algoritma perkalian rantai matriks di §7.6.7 mengisi tabel berisi $\Theta(n^2)$ sel, dan menghitung tiap sel berarti mengambil minimum atas sebanyak-banyaknya n calon titik belah, sehingga keseluruhannya $\Theta(n^3)$ — tepatnya $(n^3 - n)/6$ iterasi terdalam, sebagaimana dihitung §7.6.10. Algoritma *longest common subsequence* di §7.9.6 memakan waktu $\Theta(mn)$ dan, tanpa siasat penghematan ruang, memakan ingatan $\Theta(mn)$ pula; pada $m = n = 8000$ tabel itu berukuran 488 MiB.

Mengapa semahal itu? Karena pada setiap sel program dinamis itu *tidak tahu pilihan mana yang benar*, jadi ia mencoba semuanya. Pada perkalian rantai matriks ia mencoba setiap titik belah k antara i dan j. Pada *longest common subsequence* ia mencoba membuang karakter terakhir X dan mencoba membuang karakter terakhir Y, lalu mengambil mana yang ternyata lebih baik.

Tabelnya ada justru karena pilihannya tak dapat dijatuhkan tanpa lebih dulu menyelesaikan submasalahnya.

9.1.1 Bagaimana jika pilihannya sudah gamblang?

Sekarang andaikan pada tiap langkah tak ada pilihan yang perlu dipergumulkan. Andaikan salah satu calonnya *kasatmata* memang yang benar — benar tanpa perlu menyelesaikan satu submasalah pun, benar sekadar dengan memandangnya, benar karena alasan yang dapat Anda nyatakan dalam satu kalimat. Maka Anda tak memerlukan tabelnya. Anda akan mengambil calon itu, menyusutkan persoalannya menjadi satu persoalan yang lebih kecil, mengambil calon yang kasatmata benar di sana, dan terus begitu sampai tak ada yang tersisa. Tak ada peminimuman atas k. Tak ada $\Theta(n^2)$ sel. Sering kali tak ada ingatan tambahan sama sekali di luar satu salinan masukan yang sudah terurut, dan waktu jalan $\Theta(n\lg n)$ yang seluruhnya didominasi oleh pengurutan itu.

Itulah **algoritma rakus** (*greedy algorithm*), dan bab ini membicarakannya. Paragraf penutup Bab 7 sudah menjanjikannya, dan sudah menyebut bahan tambahan yang membuatnya sah: *sifat pilihan rakus*. Kita sekarang akan mendefinisikan sifat itu, membuktikannya bagi tiga persoalan yang berlainan, dan — ini yang genting — melihatnya gagal, supaya Anda tak pernah lagi keliru menyangka aturan yang menarik sebagai aturan yang benar.

9.2 Apa Itu Algoritma Rakus

9.2.1 Definisinya

> **DEFINISI.** Sebuah **algoritma rakus** menyelesaikan persoalan optimasi dengan menjatuhkan serangkaian pilihan. Pada tiap titik keputusan ia menjatuhkan pilihan **yang tampak terbaik pada saat itu**, dinilai dengan aturan tetap yang hanya memeriksa keadaan persoalan saat ini — tak pernah memeriksa solusi submasalah yang akan dilahirkan pilihan itu, dan tak pernah memeriksa pilihan mana pun di masa depan. Setelah menjatuhkan sebuah pilihan, ia tak pernah meninjaunya ulang.

Tiga ungkapan di dalam definisi itu memikul seluruh bebannya, jadi mari kita ambil satu per satu.

"Yang tampak terbaik pada saat itu." Aturannya bersifat *lokal*. Ia memandang apa yang ada di depannya lalu memeringkat pilihan yang tersedia. Pada pemberian uang kembalian ia berkata: ambil koin terbesar yang muat. Pada persoalan pemilihan aktivitas ia akan berkata: ambil aktivitas yang paling awal selesai. Pada penyandian Huffman ia akan berkata: gabungkan dua lambang yang paling jarang muncul. Masing-masingnya adalah aturan satu baris yang dapat Anda titipkan kepada seorang penjaga warung.

"Tak pernah memeriksa solusi submasalah." Inilah titik persis tempat kerakusan berpisah jalan dengan pemrograman dinamis. Pertentangan itu dapat dinyatakan tanpa berbelit: pemrograman dinamis *menyelesaikan submasalahnya sebelum menjatuhkan pilihan pertama*; algoritma rakus *menjatuhkan pilihan pertamanya sebelum menyelesaikan satu submasalah pun*. Sebuah pilihan rakus boleh bergantung pada pilihan yang sudah terjatuh — lagi pula pilihan itu sudah mempersempit persoalannya — tetapi ia tak boleh bergantung pada pilihan mana pun di masa depan atau pada nilai submasalah mana pun.

"Tak pernah meninjau ulang." Tak ada penelusuran mundur. Algoritma rakus adalah jalan satu arah. Di situlah kecepatannya berasal, dan di situ pula sebabnya ia bisa keliru begitu parah: kesalahan yang dibuat pada langkah ke-3 tak akan pernah dapat dibatalkan pada langkah ke-40.

DENGAN BAHASA SEDERHANA. Anda berada di pasar dengan anggaran tertentu dan ingin mengisi keranjang sebaik mungkin. Seorang pemrogram dinamis akan menghitung lebih dulu, untuk setiap kemungkinan anggaran dan setiap awalan deretan lapak, keranjang terbaiknya seperti apa, dan baru sesudah itu mulai berbelanja. Seorang pembelanja rakus menghampiri lapak pertama, menyambar apa pun yang tampak paling berharga di meja, membayar, lalu berjalan terus. Si pembelanja rakus sudah sampai di rumah sebelum jam makan siang. Apakah si pembelanja rakus benar, itu sepenuhnya bergantung pada apa arti "paling berharga" dan pada bagaimana lapak-lapaknya tersusun — dan itulah seluruh pokok bahasan bab ini.

9.2.2 Kenyataan yang tidak nyaman

Inilah kalimat yang dikubur kebanyakan buku ajar dan yang diletakkan bab ini paling depan.

ALGORITMA RAKUS BIASANYA KELIRU. Ambillah sembarang persoalan optimasi dan sebuah aturan lokal yang masuk akal, maka algoritma yang Anda peroleh, secara umum, tidak akan menghasilkan solusi optimal. Hal ini layak dikatakan dua kali, dalam dua rumusan: algoritma rakus *tidak selalu menghasilkan solusi optimal*, dan strategi rakus, bila diterapkan pada sembarang persoalan, *tidak selalu menghasilkan solusi optimal*. Tak ada apa pun dalam *bentuk* sebuah algoritma rakus yang membuatnya benar. Kerakusan adalah harapan, bukan metode.

Hal ini berakibat pada cara Anda harus bekerja, dan inilah pokok kaidah kerja yang paling penting dalam bab ini.

BUKTINYA ADALAH SELURUH PEKERJAANNYA. Menulis algoritma rakus itu sepele — Anda mengurutkan masukannya lalu mengambil barangnya berurutan. Memutuskan urutan *yang mana*, dan *membuktikan* bahwa mengambil barang dalam urutan itu aman, itulah seluruh pekerjaannya. Algoritma rakus tanpa bukti kebenaran bukanlah algoritma; ia tebakan yang berlagak sopan. Sebaliknya, algoritma rakus *dengan* buktinya biasanya merupakan algoritma benar yang tercepat dan tersederhana yang dipunyai persoalan itu.

Bandingkan keadaan ini dengan keadaan di Bab 7. Sebuah program dinamis yang mempertimbangkan setiap titik belah k sudah benar begitu Anda membuktikan substruktur optimalnya dan menyusun rekurensinya dengan betul, karena ia tak pernah berjudi: ia mencoba semuanya. Algoritma rakus berjudi pada setiap langkah. Buktinyalah yang mengubah judi itu menjadi kepastian.

9.2.3 Apa yang dikerjakan sisa bab ini

Rencananya karena itu begini:

1. **§9.3** memperlihatkan kerakusan gagal, pada persoalan paling sehari-hari yang ada — memberi uang kembalian — lalu memperlihatkan *aturan yang sama* berhasil pada sekumpulan koin yang lain. Inilah pelajaran itu dalam bentuk mungilnya: aturannya bukan intinya, buktinyalah intinya.
2. **§9.4** menggarap persoalan pemilihan aktivitas secara tuntas: persoalannya, contoh berjalannya, empat calon aturan rakus yang tiga di antaranya keliru, bukti substruktur optimalnya, lalu **argumen penukaran** yang membuktikan aturan yang selamat itu benar. Kuasailah argumen tersebut. Ia muncul lagi di §9.6, di §9.7, di §9.8, dan di Bab 11 serta Bab 13.
3. **§9.5** menyarikan resepnya: dua bahan yang diperlukan sebuah persoalan, dan cara mengujinya.
4. **§9.6** menjajarkan kerakusan dan pemrograman dinamis pada dua masalah ransel, lengkap dengan hitungannya.
5. **§9.7** membangun kode Huffman dan membuktikannya optimal.

6. **§9.8** mengajukan pertanyaan yang paling dalam — *adakah sebuah struktur yang mencirikan kapan kerakusan berhasil?* — dan menjawabnya dengan matroid.
7. **§9.9** mendaftar algoritma rakus yang akan Anda jumpai di Bab 11 dan Bab 13, supaya Anda mengenalinya saat mereka tiba.

9.3 Kegagalan yang Memantik: Memberi Uang Kembalian

9.3.1 Persoalannya

Anda kasir di sebuah warung. Seorang pembeli harus diberi uang kembalian sejumlah tertentu, dan Anda ingin menyerahkan **sesedikit mungkin keping uang** — laci Anda kecil dan Anda lebih suka tidak berpisah dengan lima puluh keping seribuan.

> **DEFINISI (persoalan uang kembalian).** Anda diberi sehimpunan pecahan koin $d_1, d_2, \ldots, d_k$, semuanya bilangan bulat positif, dan sebuah nilai A, juga bilangan bulat positif. Anda punya persediaan tak terbatas untuk tiap pecahan. Carilah *multihimpunan* koin yang jumlah nilainya tepat A dan yang **cacah koinnya** sekecil mungkin.

Ada dua lambang di sana, keduanya baru, jadi mari kita nyatakan dengan jelas. Sebuah *pecahan* adalah nilai nominal satu jenis koin — di Indonesia, per September 2026, Bank Indonesia menerbitkan uang kertas pecahan 1000, 2000, 5000, 10 000, 20 000, 50 000, dan 100 000 rupiah (emisi 2022) serta uang logam pecahan 100, 200, 500, dan 1000 rupiah [1]. Jika ketiga koin kecil, yang jarang berpindah tangan, kita kesampingkan, pecahan yang kita pakai membayar adalah 1000, 2000, 5000, 10 000, 20 000, 50 000, dan 100 000 rupiah, sehingga $k = 7$ dan $d_1 = 1000$, $d_2 = 2000$, dan seterusnya. Sebuah *multihimpunan* mirip himpunan, hanya saja pengulangan diperbolehkan dan dihitung: $\{5000,5000,1000\}$ adalah jawaban yang sah dengan tiga keping. Nilai A adalah yang kita utang.

9.3.2 Aturan rakusnya

Semua orang menemukan aturan yang sama, dan nalurinya memang tepat:

> **ATURAN RAKUS UNTUK UANG KEMBALIAN.** Selama masih ada utang, serahkan **koin terbesar yang tidak melebihi sisa yang terutang**. Ulangi.

Ia lokal: ia hanya memandang sisa yang terutang. Ia tak pernah meninjau ulang: begitu sekeping koin keluar dari laci, ia tetap di luar. Ia jelas cepat. Dan ia terasa benar, karena koin besar bekerja lebih banyak daripada koin kecil.

Kerjakan dengan tangan untuk $A = 78\,000$ rupiah dengan pecahan Indonesia.

- Terutang 78 000. Koin terbesar yang muat: 50 000. Serahkan. Terutang $78\,000 - 50\,000 = 28\,000$.
- Terutang 28 000. Koin terbesar yang muat: 20 000. Serahkan. Terutang $28\,000 - 20\,000 = 8\,000$.
- Terutang 8 000. Koin terbesar yang muat: 5 000. Serahkan. Terutang $8\,000 - 5\,000 = 3\,000$.
- Terutang 3 000. Koin terbesar yang muat: 2 000. Serahkan. Terutang $3\,000 - 2\,000 = 1\,000$.
- Terutang 1 000. Koin terbesar yang muat: 1 000. Serahkan. Terutang 0. Berhenti.

Lima keping: $50\,000 + 20\,000 + 5\,000 + 2\,000 + 1\,000 = 78\,000$. Dan memang tak ada empat keping yang sanggup, sebagaimana akan kita pastikan dengan mesin di §9.3.5.

9.3.3 Contoh sanggahannya

Sekarang ganti koinnya. Andaikan — dan ini bukan khayalan, ini kira-kira keadaan Inggris sebelum desimalisasi dan persis keadaan pada beberapa permainan papan — pecahannya adalah

$$d_1 = 1, \qquad d_2 = 3, \qquad d_3 = 4,$$

dan yang terutang adalah $A = 6$.

Jalankan aturan rakusnya, satu langkah satu baris, dengan hitungannya ditampakkan.

- Terutang 6. Koin terbesar yang tidak melebihi 6: itu 4, sebab $4 \leq 6$ sedangkan pecahan berikutnya yang lebih besar tidak ada. Serahkan 4. Terutang $6 - 4 = 2$.
- Terutang 2. Koin terbesar yang tidak melebihi 2: bukan 4 (terlalu besar), bukan 3 (terlalu besar), jadi 1. Serahkan 1. Terutang $2 - 1 = 1$.
- Terutang 1. Koin terbesar yang muat: 1. Serahkan 1. Terutang $1 - 1 = 0$. Berhenti.

Kerakusan telah menyerahkan $4 + 1 + 1$, yaitu **tiga keping**.

Sekarang lihatlah apa yang dilakukan kasir yang berpikir:

$$3 + 3 = 6.$$

Dua keping. Kerakusan memakai tiga keping padahal dua sudah cukup — lima puluh persen lebih banyak daripada yang perlu, pada nilai enam, dengan tiga pecahan. Ini bukan konstruksi ganjil berisi seratus pecahan; ini contoh kasus menarik yang terkecil yang ada.

> **MENGAPA KERAKUSAN GAGAL DI SINI.** Aturan rakus mengambil 4 karena $4 > 3$, dan koin yang lebih besar *seolah-olah* membuat kemajuan lebih banyak. Tetapi mengambil 4 menyisakan utang 2, dan 2 adalah jumlah utang yang menyedihkan pada sistem ini: satu-satunya cara membayarnya adalah dua keping 1. Mengambil koin yang lebih kecil, yaitu 3, menyisakan utang 3, yang terbayar dengan sekeping koin saja. Aturan rakus itu mengoptimalkan hal yang keliru. Ia memaksimumkan kemajuan *langkah ini* lalu karenanya merusak pilihan-pilihan *langkah berikutnya* — dan, karena tak punya penelusuran mundur, ia tak dapat pulih.

Itulah anatomi umum kegagalan rakus, dan ia layak diberi nama.

> **BENTUK SETIAP KEGAGALAN RAKUS.** Sebuah aturan rakus gagal ketika pilihan yang menarik secara lokal merusak submasalah yang ia tinggalkan, dengan kerusakan yang lebih besar daripada keuntungan lokalnya. Aturan itu tak dapat melihat kerusakan tersebut, sebab menurut definisinya ia tidak memandang submasalahnya.

Satu contoh sanggahan kedua yang bahkan lebih masyhur: pecahan $\{1,10,25\}$ — koin Amerika dengan keping lima sennya dicabut — dan $A = 30$. Kerakusan mengambil 25, menyisakan 5, yang harus dibayar sebagai $1 + 1 + 1 + 1 + 1$: enam keping seluruhnya. Optimumnya $10 + 10 + 10$: tiga keping. Kerakusan memakai **dua kali lipat** dari yang perlu.

9.3.4 Sistem koin tempat kerakusan berhasil

Sekarang separuh pelajaran yang lain. Kembalikan keping lima sennya: pecahan $\{1,5,10,25\}$, yaitu himpunan koin Amerika yang sesungguhnya. Cobalah mematahkan kerakusan dan Anda tak akan berhasil — untuk *setiap* nilai, aturan rakus menghasilkan multihimpunan berukuran minimum. Hal yang sama berlaku bagi himpunan Indonesia $\{1,2,5,10,20,50,100\}$ dalam ribuan rupiah, dan bagi pangkat-pangkat dua $\{1,2,4,8,16,32,64\}$.

> **DEFINISI.** Sehimpunan pecahan disebut **kanonis** jika aturan rakus untuk uang kembalian mengembalikan multihimpunan berukuran minimum bagi setiap nilai. {1,5,10,25} kanonis. {1,3,4} tidak.

Jadi aturan yang sama persis — "koin terbesar yang muat" — *terbukti optimal* pada satu himpunan koin dan *terbukti keliru* pada himpunan koin yang lain. Tak ada apa pun pada aturannya yang berubah. Yang berubah adalah struktur masukannya, dan karena itu ada atau tidaknya sebuah bukti.

> **PELAJARAN BAGIAN INI, DALAM SATU KALIMAT.** Strateginya bukan intinya; buktinyalah intinya. "Jadilah rakus" bukanlah teknik perancangan algoritma yang berdiri sendiri — ia baru menjadi teknik ketika dipasangkan dengan sebuah argumen bahwa kerakusan yang ini, pada persoalan yang ini, tak mungkin meleset.

Perlu ditambahkan bahwa memutuskan apakah sebuah sistem koin sembarang bersifat kanonis itu sendiri merupakan persoalan sungguhan, bukan hal sepele. Tak ada pola sederhana yang dapat dibaca langsung dari pecahannya. (Algoritma Pearson memutuskannya dalam waktu $O(k^3)$ untuk k pecahan, dengan memeriksa sehimpunan terbatas calon contoh sanggahan [2]; dan perhatikanlah bahwa persoalan uang kembalian yang umum, bagi sistem yang tidak kanonis, justru persis jenis persoalan yang menjadi alasan dibangunnya pemrograman dinamis di Bab 7.)

9.3.5 Diukur

Kode 9.1 mengaudit enam sistem koin. Untuk tiap sistem dan untuk setiap nilai dari 1 sampai 200 ia menghitung jawaban rakus dan optimum sejatinya — yang terakhir dengan program dinamis Bab 7, sebuah tabel berdimensi satu dengan `best[a] = 1 + min atas koin c dari best[a-c]` — lalu melaporkan nilai terkecil, jika ada, tempat keduanya berselisih.

Kode 9.1 Mengaudit aturan rakus uang kembalian terhadap optimum sejatinya, untuk enam sistem koin dan setiap nilai sampai 200. Kompilasi dengan `g++ -std=c++17 -O2 -o coins coins.cpp` lalu jalankan `./coins`.

```cpp
// coins.cpp — Benarkah aturan rakus "koin terbesar yang muat" memberi koin paling sedikit?
// Untuk tiap sistem koin kita hitung, bagi setiap nilai uang, cacah koin versi
// rakus dan optimum sejatinya (dengan pemrograman dinamis), lalu melaporkan nilai
// terkecil tempat keduanya berselisih.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <climits>
using namespace std;

// Rakus: berulang kali ambil koin terbesar yang tak melebihi sisa yang terutang.
// Mengembalikan koin yang dipakai, atau vektor kosong bila rakus macet.
vector<int> greedyChange(vector<int> coin, int amount) {
    sort(coin.rbegin(), coin.rend());            // terbesar dahulu
    vector<int> used;
    for (int c : coin)
        while (amount >= c) { amount -= c; used.push_back(c); }
    if (amount != 0) return {};                  // rakus macet: tak ada kembalian pas
    return used;
}

// Optimum dengan pemrograman dinamis: best[a] = koin tersedikit yang jumlahnya tepat a.
// choice[a] mengingat satu koin pada penyelesaian optimal agar dapat kita cetak.
pair<int, vector<int>> optimalChange(const vector<int>& coin, int amount) {
    const int INF = INT_MAX / 2;
    vector<int> best(amount + 1, INF), choice(amount + 1, -1);
    best[0] = 0;
    for (int a = 1; a <= amount; ++a)
        for (int c : coin)
            if (c <= a && best[a - c] + 1 < best[a]) {
                best[a] = best[a - c] + 1;
                choice[a] = c;
            }
    if (best[amount] >= INF) return {-1, {}};
    vector<int> used;
    for (int a = amount; a > 0; a -= choice[a]) used.push_back(choice[a]);
    return {best[amount], used};
}
```

Kode 9.1 (Lanjutan) *Mengaudit aturan rakus uang kembalian terhadap optimum sejatinya, untuk enam sistem koin dan setiap nilai sampai 200. Kompilasi dengan g++ -std=c++17 -O2 -o coins coins.cpp lalu jalankan ./coins.*

```
string show(const vector<int>& v) {
    string s;
    for (size_t i = 0; i < v.size(); ++i) { if (i) s += " + "; s += to_string(v[i]); }
    return s.empty() ? "(none)" : s;
}

void audit(const string& name, vector<int> coin, int limit) {
    cout << "System " << name << " = {";
    for (size_t i = 0; i < coin.size(); ++i) { if (i) cout << ", "; cout << coin[i]; }
    cout << "}\n";
    int firstBad = -1;
    int badCount = 0;
    for (int a = 1; a <= limit; ++a) {
        auto g = greedyChange(coin, a);
        auto o = optimalChange(coin, a);
        if (o.first < 0) continue;               // nilai ini sama sekali tak terbayarkan
        int gc = g.empty() && a > 0 ? -1 : (int)g.size();
        if (gc != o.first) { ++badCount; if (firstBad < 0) firstBad = a; }
    }
    if (firstBad < 0) {
        cout << "  greedy is optimal for every amount from 1 to " << limit << "\n\n";
        return;
    }
    auto g = greedyChange(coin, firstBad);
    auto o = optimalChange(coin, firstBad);
    cout << "  greedy FAILS.  smallest counterexample: amount = " << firstBad << "\n";
    cout << "    greedy : " << show(g) << "   = " << g.size() << " coins\n";
    cout << "    optimal: " << show(o.second) << "   = " << o.first << " coins\n";
    cout << "  greedy is wrong for " << badCount << " of the " << limit
         << " amounts tested\n\n";
}

int main() {
    audit("A (textbook trap)",  {1, 3, 4},                 200);
    audit("B (no nickel)",      {1, 10, 25},               200);
    audit("C (US coins)",       {1, 5, 10, 25},            200);
    audit("D (rupiah, x1000)",  {1, 2, 5, 10, 20, 50, 100}, 200);
    audit("E (powers of 2)",    {1, 2, 4, 8, 16, 32, 64},  200);
    audit("F (rupiah minus 2)", {1, 5, 10, 20, 50, 100},   200);
    return 0;
}
```

Programnya mencetak:

```
System A (textbook trap) = {1, 3, 4}
  greedy FAILS.  smallest counterexample: amount = 6
    greedy : 4 + 1 + 1   = 3 coins
    optimal: 3 + 3   = 2 coins
  greedy is wrong for 49 of the 200 amounts tested

System B (no nickel) = {1, 10, 25}
  greedy FAILS.  smallest counterexample: amount = 30
    greedy : 25 + 1 + 1 + 1 + 1 + 1   = 6 coins
    optimal: 10 + 10 + 10   = 3 coins
  greedy is wrong for 70 of the 200 amounts tested

System C (US coins) = {1, 5, 10, 25}
  greedy is optimal for every amount from 1 to 200

System D (rupiah, x1000) = {1, 2, 5, 10, 20, 50, 100}
  greedy is optimal for every amount from 1 to 200

System E (powers of 2) = {1, 2, 4, 8, 16, 32, 64}
  greedy is optimal for every amount from 1 to 200

System F (rupiah minus 2) = {1, 5, 10, 20, 50, 100}
  greedy is optimal for every amount from 1 to 200
```

Bacalah angka-angka itu baik-baik, sebab semuanya lebih mengkhawatirkan daripada yang disiratkan contoh sanggahan yang kita kerjakan dengan tangan tadi. Pada sistem {1,3,4}, kerakusan bukan keliru pada satu nilai yang apes — ia keliru pada **49 dari 200 nilai pertama**, kira-kira satu dari empat. Pada {1,10,25} ia keliru pada **70 dari 200**, lebih dari satu dari tiga. Aturan rakus yang gagal biasanya tidak gagal sesekali; ia gagal secara sistematis, dan kegagalannya tak kelihatan kecuali Anda mengujinya terhadap cara yang sudah diketahui benar. **Tabel 9.1** mengumpulkan hasilnya.

Tabel 9.1 Aturan rakus yang sama pada enam sistem koin, diaudit oleh **Kode 9.1** terhadap optimum pemrograman dinamis untuk setiap nilai dari 1 sampai 200. Aturannya tak berubah; hanya struktur masukannya yang berubah.

Sistem koin	Kanonis?	Contoh sanggahan terkecil	Rakus / optimal di sana	Keliru pada
{1,3,4}	tidak	$A = 6$	3 keping lawan 2	49 dari 200
{1,10,25}	tidak	$A = 30$	6 keping lawan 3	70 dari 200
{1,5,10,25}	ya	—	—	0 dari 200
{1,2,5,10,20,50,100}	ya	—	—	0 dari 200
{1,2,4,8,16,32,64}	ya	—	—	0 dari 200
{1,5,10,20,50,100}	ya	—	—	0 dari 200

Perhatikan baris terakhirnya. Membuang lembar 2000 rupiah dari himpunan Indonesia menyisakan {1,5,10,20,50,100}, yang *tetap* kanonis — jadi mencabut sebuah pecahan tak harus mematahkan kerakusan. Dan perhatikan pula, sebaliknya, bahwa {1,10,25} berbeda dari {1,5,10,25} yang kanonis itu hanya pada satu pecahan, dan satu pencabutan itu mematahkannya pada 70 nilai dari 200. Tak ada jalan pintas. Anda harus membuktikannya.

> **GEMA YANG SUDAH AKRAB.** Soal Latihan 7.18 meminta Anda memaketkan pesanan Profitina Laundry ke dalam kantong 3 kg, 5 kg, dan 8 kg, terisi tepat, dengan kantong sesedikit mungkin, dan menunjukkan bahwa "pakai kantong terbesar yang muat" gagal. Ia memang gagal, dan sekarang Anda tahu nama gejalanya: {3,5,8} tidak kanonis. Untuk pesanan 9 kg, kerakusan mengambil kantong 8 kg lalu tertinggal dengan 1 kg yang sama sekali tak dapat dikantongi; tiga kantong 3 kg menyelesaikan pekerjaannya.

9.4 Persoalan Pemilihan Aktivitas

Inilah persoalan yang di atasnya seluruh metodenya dibangun, dan kita akan mengerjakannya secara tuntas: pernyataannya, datanya, empat calon aturan yang tiga di antaranya keliru, substruktur optimal yang dibuktikan, sifat pilihan rakus yang dibuktikan dengan argumen penukaran, algoritma rekursifnya, algoritma iteratifnya, penelusurannya, dan waktu jalannya.

9.4.1 Persoalannya

DEFINISI (persoalan pemilihan aktivitas). Kita diberi sehimpunan $S = \{a_1, a_2, \dots, a_n\}$ berisi n **aktivitas** yang diusulkan, yang semuanya menghendaki penggunaan satu sumber daya bersama secara eksklusif — satu ruang rapat, satu mesin cuci, satu teleskop. Aktivitas a_i punya **waktu mulai** s_i dan **waktu selesai** f_i dengan $0 \le s_i < f_i < \infty$. Jika a_i terpilih, ia menempati sumber dayanya selama **selang setengah terbuka** $[s_i, f_i)$. Dua aktivitas a_i dan a_j disebut **selaras** jika selangnya tidak bertindih, yang terjadi tepat ketika

$$s_i \ge f_j \qquad \text{atau} \qquad s_j \ge f_i.$$

Tujuannya adalah memilih himpunan bagian **berukuran maksimum** berisi aktivitas yang saling selaras.

Setiap lambang dalam definisi itu perlu sepatah penjelasan, sebab rinciannya penting.

S hanyalah nama bagi seluruh kumpulan permintaan itu. n adalah berapa banyaknya. s_i dan f_i adalah dua bilangan yang melekat pada permintaan bernomor i — waktu jam saat ia hendak mulai dan waktu jam saat ia akan selesai. Keduanya boleh dalam menit setelah jam buka, atau dalam jam, atau satuan apa pun; yang penting hanyalah kita dapat membandingkannya. Syarat $s_i < f_i$ mengatakan sebuah aktivitas memakan waktu yang positif, dan $f_i < \infty$ mengatakan ia akhirnya memang berakhir.

Mengapa setengah terbuka? Notasi $[s_i, f_i)$ berarti "dari s_i termasuk sampai f_i tidak termasuk". Kurung siku menyertakan ujungnya; kurung bundar mengecualikannya. Inilah kesepakatan yang persis tepat untuk penjadwalan, karena ia membuat aktivitas yang selesai pukul 10:00 dan aktivitas lain yang mulai pukul 10:00 menjadi selaras — sumber dayanya diserahterimakan pada detik pukul 10:00 dan tak ada waktu yang terpesan dua kali. Seandainya kita memakai selang tertutup $[s_i, f_i]$, kedua aktivitas itu secara teknis bertindih pada satu titik waktu pukul 10:00, dan bukan begitu cara sebuah *laundry* atau sebuah ruang rapat bekerja.

Mengapa uji keselarasan yang itu? Kedua klausanya meliputi satu-satunya dua cara dua selang dapat saling luput: entah a_i mulai pada atau sesudah a_j selesai ($s_i \ge f_j$, jadi a_j lebih dulu), entah a_j mulai pada atau sesudah a_i selesai ($s_j \ge f_i$, jadi a_i lebih dulu). Jika tak satu pun berlaku, masing-masing mulai sebelum yang lain selesai, dan keduanya bertindih.

Perhatikan apa yang *tidak* dimaksimumkan. Kita memaksimumkan **cacah** aktivitas, bukan total waktu sumber dayanya terpakai, dan bukan pula gagasan nilai atau pendapatan apa pun. Ini teramat penting: §9.14 (Soal Latihan 9.15) memperlihatkan bahwa begitu aktivitas membawa nilai yang berbeda-beda, aturan rakus yang sebentar lagi kita buktikan benar itu menjadi keliru, dan persoalannya kembali menuntut pemrograman dinamis.

Akhirnya, kita memakai satu anggapan tetap yang akan membayar dirinya berkali-kali:

$$f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n.$$

(9.1)

Dengan kata lain, Persamaan (9.1) menyatakan bahwa aktivitasnya diberikan dalam keadaan **terurut menaik monoton menurut waktu selesai**. Jika belum, urutkanlah; biayanya $O(n\lg n)$ dan, sebagaimana akan kita lihat di §9.4.11, ternyata itulah biaya yang mendominasi seluruh algoritmanya.

9.4.2 Contoh berjalan

Gambar 9.1 adalah contoh kasus yang akan kita pakai dari awal sampai akhir bagian ini.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Gambar 9.1 Sebelas aktivitas contoh berjalan, dengan waktu mulai s_i dan waktu selesai f_i, sudah terurut sehingga $f_1 \leq f_2 \leq \cdots \leq f_{11}$. Tabel yang satu ini menjadi contoh berjalan bagi seluruh §9.4.

Bacalah tabelnya sebagai tiga baris: nomor aktivitas i, waktu mulainya s_i, dan waktu selesainya f_i. Jadi a_1 berjalan selama [1,4), a_2 selama [3,5), a_3 selama [0,6), dan seterusnya sampai a_{11} selama [12,16). Periksa urutan yang dituntut Persamaan (9.1): waktu selesainya terbaca 4,5,6,7,9,9,10,11,12,14,16 — menaik monoton (dengan satu seri pada 9, yang diperbolehkan, sebab urutannya *tak menurun*).

Tabel 9.2 menyatakan ulang data yang sama dengan durasinya dihitung, sebab kita akan memerlukannya di §9.4.4.

Tabel 9.2 Contoh kasus **Gambar 9.1** dengan durasi tiap aktivitas dihitung. Aktivitas a_{10} adalah raksasanya: dua belas satuan waktu, hampir sepanjang hari.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16
durasi $f_i - s_i$	3	2	6	2	6	4	4	3	4	12	4

Ada dua fakta tentang contoh kasus ini yang patut diperhatikan, dan keduanya mendidik.

Pertama: himpunan bagian $\{a_3, a_9, a_{11}\}$ terdiri atas aktivitas yang saling selaras. Periksalah. $a_3 =$ [0,6) dan $a_9 =$ [8,12): apakah $s_9 \geq f_3$? Ya, $8 \geq 6$. Selaras. $a_9 =$ [8,12) dan $a_{11} =$ [12,16): apakah $s_{11} \geq f_9$? Ya, $12 \geq 12$ — dan di sinilah kesepakatan setengah terbuka itu memperlihatkan gunanya. Selaras. a_3 dan a_{11}: $12 \geq 6$. Selaras. Jadi ketiga pasangannya selaras dan himpunannya sah. Tetapi ia **bukan** himpunan maksimum: isinya hanya tiga aktivitas.

Kedua: himpunan bagian $\{a_1, a_4, a_8, a_{11}\}$ dan $\{a_2, a_4, a_9, a_{11}\}$ masing-masing berisi empat aktivitas, dan empat adalah yang terbanyak yang mungkin. Periksa yang pertama: $a_1 =$ [1,4), $a_4 =$ [5,7) — $5 \geq 4$ ✓; a_4, $a_8 =$ [8,11) — $8 \geq 7$ ✓; a_8, $a_{11} =$ [12,16) — $12 \geq 11$ ✓. Dan karena keselarasan di sini hanya berarti "masing-masing mulai setelah yang sebelumnya selesai", pasangan yang tidak bersebelahan otomatis selaras.

> **MAKSIMAL BUKAN MAKSIMUM.** Sebuah himpunan disebut **maksimal** jika Anda tak dapat menambahkan aktivitas apa pun lagi ke dalamnya. Sebuah himpunan disebut **maksimum** jika tak ada himpunan sah lain yang lebih besar. $\{a_3, a_9, a_{11}\}$ maksimal — cobalah menambahkan apa saja dan ia akan bentrok — tetapi ia bukan maksimum, sebab $\{a_1, a_4, a_8, a_{11}\}$ lebih besar. Maksimal adalah sifat lokal yang dapat Anda periksa sekadar dengan memandang; maksimum adalah sifat global yang menuntut sebuah argumen. Setiap algoritma rakus menghasilkan

> solusi maksimal tanpa susah payah; seluruh kesulitannya terletak pada memperoleh yang maksimum.

Perhatikan pula bahwa himpunan maksimumnya **tidak tunggal**: $\{a_1, a_4, a_8, a_{11}\}$ dan $\{a_2, a_4, a_9, a_{11}\}$ dua-duanya optimal. Ini lumrah, dan ini penting bagi bukti-bukti yang akan datang: sifat pilihan rakus akan mengatakan bahwa *ada sebuah* solusi optimal yang memuat pilihan rakusnya, bukan bahwa *setiap* solusi optimal memuatnya.

9.4.3 Empat calon aturan rakus

Sebelum membuktikan apa pun, mari kita jujur tentang betapa banyaknya aturan yang masuk akal. Masing-masing di bawah ini adalah aturan lokal satu baris; masing-masing pernah diusulkan mahasiswa; tiga di antaranya keliru.

1. **Waktu mulai terawal.** Ambil aktivitas yang paling cepat mulai, buang semua yang bentrok dengannya, ulangi. *Alasannya:* bergeraklah sedini mungkin.
2. **Durasi terpendek.** Ambil aktivitas yang menempati sumber daya paling sebentar, buang yang bentrok, ulangi. *Alasannya:* aktivitas yang pendek menyisakan ruang lebih banyak untuk yang lain.
3. **Bentrokan tersedikit.** Hitunglah, untuk tiap aktivitas, berapa banyak aktivitas tersisa lainnya yang bentrok dengannya; ambil yang bentrokannya paling sedikit, buang yang bentrok, ulangi. *Alasannya:* aktivitas yang paling sedikit mengganggu semestinya paling aman.
4. **Waktu selesai terawal.** Ambil aktivitas yang paling cepat selesai, buang yang bentrok, ulangi. *Alasannya:* bebaskan sumber dayanya kembali sedini mungkin.

Jalankan keempatnya pada contoh berjalan itu dan terjadilah sesuatu yang canggung: dua dari yang keliru justru memperoleh jawaban yang benar. Dari keluaran terverifikasi **Kode 9.2**:

```
=== Four greedy rules on the running example ===
  earliest finish time  -> {a1, a4, a8, a11}, size 4
  earliest start time   -> {a3, a7, a11}, size 3
  shortest duration     -> {a2, a4, a8, a11}, size 4
  fewest conflicts      -> {a1, a4, a8, a11}, size 4
```

Mulai-terawal sudah gagal pada contoh kasus ini juga: ia menyambar $a_3 = [0,6)$, yang memblokir a_1, a_2, a_4, a_5, a_6, dan a_{10}, lalu $a_7 = [6,10)$, lalu a_{11}, sehingga seluruhnya tiga, bukan empat. Itu persis bentuk kegagalan yang dinamai di §9.3.3: a_3 mulai paling awal tetapi memborong enam satuan waktu.

Durasi-terpendek dan bentrokan-tersedikit, sebaliknya, memperoleh empat pada contoh kasus ini. **Inilah jebakannya.** Satu contoh kasus tidak membuktikan apa pun. Untuk menunjukkan sebuah aturan keliru Anda perlu contoh sanggahan; untuk menunjukkan ia benar Anda perlu bukti. **Kode 9.2** mencari secara tuntas contoh sanggahan terkecil bagi masing-masing, dan menemukan yang berikut ini:

```
=== Smallest counterexample for each incorrect rule ===
  earliest start time on an instance with n = 3:
      activity :       a1       a2       a3
      [s, f)   :    [0,8)    [1,4)    [0,1)
      the rule picks 1 activities {a1}
      brute force says the optimum is 2  ->  the rule is WRONG

  shortest duration  on an instance with n = 3:
      activity :       a1       a2       a3
      [s, f)   :    [5,8)    [1,5)    [4,6)
      the rule picks 1 activities {a3}
      brute force says the optimum is 2  ->  the rule is WRONG

  fewest conflicts   on an instance with n = 9:
      activity :       a1       a2       a3       a4       a5       a6       a7       a8
a9
      [s, f)   :    [4,7)   [9,13)    [2,5)    [1,4)    [1,2)   [6,10)    [0,2)   [7,15)
[10,11)
      the rule picks 3 activities {a1, a2, a4}
      brute force says the optimum is 4  ->  the rule is WRONG
```

Telusurilah contoh sanggahan durasi-terpendek dengan tangan, sebab isinya hanya tiga aktivitas dan ia benar-benar meyakinkan. Aktivitasnya adalah $a_1 = [5,8)$ berdurasi 3, $a_2 = [1,5)$ berdurasi 4, dan $a_3 = [4,6)$ berdurasi 2. Yang terpendek adalah a_3, durasi 2. Ambil ia. Sekarang $a_1 = [5,8)$: apakah $5 \geq 6$? Tidak. Apakah $8 \leq 4$? Tidak. Keduanya bertindih pada $[5,6)$, jadi a_1 dibuang. Dan $a_2 = [1,5)$: apakah $4 \geq 5$? Tidak. Apakah $5 \leq 1$? Tidak. Keduanya bertindih pada $[4,5)$, jadi a_2 pun dibuang. Aturan itu berakhir dengan **satu** aktivitas. Padahal $a_2 = [1,5)$ dan $a_1 = [5,8)$ selaras ($5 \geq 5$ ✓), jadi **dua** sebenarnya tersedia. Aktivitas yang pendek itu duduk di tengah dan meracuni kedua tetangganya.

Contoh sanggahan bentrokan-tersedikit memerlukan sembilan aktivitas — tak ada contoh kasus yang lebih kecil yang mematahkannya, dan justru itulah sebabnya aturan ini begitu memikat — dan penelusurannya dikerjakan lengkap di §9.14, Soal Latihan 9.4.

> **MENGAPA SELESAI-TERAWAL BERBEDA.** Inilah gerak hatinya, dan inilah benih buktinya. *Kita sebaiknya memilih aktivitas yang menyisakan sumber dayanya tersedia bagi sebanyak mungkin aktivitas lain.* Dari semua aktivitas yang akhirnya kita pilih, salah satunya pastilah yang pertama selesai. Aktivitas di S dengan **waktu selesai terawal** membebaskan sumber dayanya sejak saat yang paling dini yang mungkin — tak ada pilihan lain yang dapat membebaskannya lebih cepat. Karena aktivitasnya terurut menurut waktu selesai, pilihan rakusnya sekadar a_1.

Itu argumen tentang *gerak hati*, bukan bukti. Sekarang kita menjadikannya bukti. Ada dua hal yang harus ditegakkan, dan keduanya sepadan persis dengan dua bahan yang dibutuhkan setiap bukti rakus: **substruktur optimal**, dan **sifat pilihan rakus**.

9.4.4 Substruktur optimal, dibuktikan

Mula-mula kita harus memandang persoalannya sebagai sekumpulan submasalah. Definisikan, untuk dua indeks i dan j,

$$S_{ij} = \{ a_k \in S : f_i \leq s_k \text{ dan } f_k \leq s_j \}, \tag{9.2}$$

dengan kata lain: S_{ij} **adalah himpunan aktivitas yang mulai setelah aktivitas** a_i **selesai dan yang selesai sebelum aktivitas** a_j **mulai.** Ia himpunan aktivitas yang dapat disisipkan ke dalam sela antara a_i dan a_j.

Agar definisi Persamaan (9.2) bermakna bagi seluruh persoalannya, kita tambahkan dua aktivitas fiktif: a_0, dengan $f_0 = 0$, dan a_{n+1}, dengan $s_{n+1} = \infty$. Maka $S_{0,n+1}$ adalah seluruh himpunan S, sebab setiap aktivitas yang sungguhan mulai setelah waktu 0 dan selesai sebelum waktu ∞. Dengan akal-akalan ini, "selesaikan persoalan aslinya" dan "selesaikan submasalah S_{ij}" menjadi kalimat yang sejenis, dan justru itulah yang kita perlukan untuk sebuah rekursi.

Sekarang teoremanya.

TEOREMA 9.1 (substruktur optimal pemilihan aktivitas). Misalkan A_{ij} sebuah himpunan berukuran maksimum berisi aktivitas yang saling selaras di dalam S_{ij}, dan andaikan $a_k \in A_{ij}$ untuk suatu aktivitas a_k. Definisikan

$$A_{ik} = A_{ij} \cap S_{ik}, \qquad A_{kj} = A_{ij} \cap S_{kj}.$$

Maka $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, sehingga

$$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1, \tag{9.3}$$

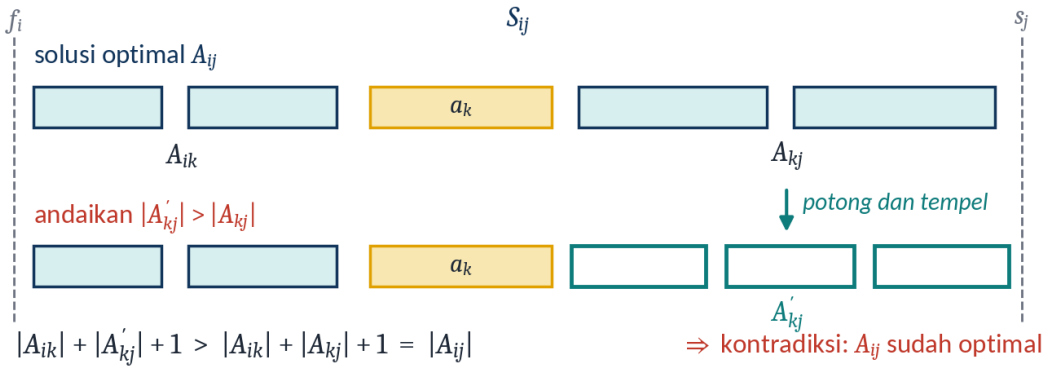
dan lebih jauh A_{ik} adalah himpunan selaras berukuran maksimum di S_{ik} dan A_{kj} adalah himpunan selaras berukuran maksimum di S_{kj}.

Bukti. Kita buktikan ketiga klaimnya berurutan.

Langkah 1: penguraiannya meliputi segalanya. Ambil sembarang aktivitas $a_m \in A_{ij}$ selain a_k. Karena A_{ij} adalah himpunan aktivitas yang saling selaras dan baik a_m maupun a_k ada di dalamnya, a_m dan a_k selaras. Jadi entah $f_m \leq s_k$ entah $f_k \leq s_m$.

Langkah 2: tiap unsurnya mendarat tepat di satu potongan. Andaikan $f_m \leq s_k$, yaitu a_m selesai sebelum a_k mulai. Kita juga tahu $a_m \in S_{ij}$, jadi $f_i \leq s_m$. Digabungkan, $f_i \leq s_m$ dan $f_m \leq s_k$, yang persis merupakan syarat bagi $a_m \in S_{ik}$. Karena itu $a_m \in A_{ij} \cap S_{ik} = A_{ik}$. Secara setangkup, jika $f_k \leq s_m$ maka $a_m \in S_{kj}$ sehingga $a_m \in A_{kj}$. Kedua kasusnya tak mungkin berlaku bersamaan, sebab $s_k < f_k$ akan memaksa $f_m \leq s_k < f_k \leq s_m < f_m$, suatu kemustahilan. Jadi setiap unsur A_{ij} selain a_k berada tepat di salah satu dari A_{ik}, A_{kj}, sehingga $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ sebagai gabungan yang saling lepas, dan karenanya cacah Persamaan (9.3) berlaku. ■ (dua klaim pertama)

Langkah 3: potongan-potongannya optimal — argumen potong-dan-tempel. Bagian ini adalah bukti dengan kontradiksi, dan **Gambar 9.2** menyatakannya secara utuh.



rekurensi:

$$c[i,j] = \begin{cases} 0 & \text{jika } S_{ij} = \varnothing \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{jika } S_{ij} \neq \varnothing \end{cases}$$

Gambar 9.2 Perampungan Teorema 9.1 lewat pengontradiksian, sekaligus menggambarkan rekurensi pemrograman dinamis bagi $c[i,j]$ yang mengikutinya. Andaikan ada himpunan A'_{kj} yang lebih baik, menyulihkannya ke dalam A_{ij} akan memberi lebih dari $|A_{ij}|$ aktivitas yang saling selaras — bertentangan dengan keoptimalan A_{ij}.

Inilah argumennya, ditulis satu langkah satu baris.

1. Andaikan, demi pengontradiksian, bahwa A_{kj} *bukan* himpunan selaras berukuran maksimum di S_{kj}.
2. Maka ada himpunan A'_{kj} berisi aktivitas yang saling selaras di S_{kj} dengan $|A'_{kj}| > |A_{kj}|$.
3. Bentuklah himpunan $B = A_{ik} \cup \{a_k\} \cup A'_{kj}$. Inilah **potong dan tempelnya**: kita memotong A_{kj} keluar dari A_{ij} lalu menempelkan A'_{kj} ke dalamnya.
4. **Apakah B sah?** Inilah langkah yang §7.5.3 peringatkan jangan pernah Anda lewati. Setiap aktivitas di A_{ik} selesai pada atau sebelum s_k; setiap aktivitas di A'_{kj} mulai pada atau sesudah f_k; dan $s_k < f_k$. Jadi setiap aktivitas A_{ik} selesai sebelum setiap aktivitas A'_{kj} mulai, dan a_k duduk di antaranya tanpa menyentuh keduanya. Aktivitas-aktivitas di dalam A_{ik} saling selaras karena $A_{ik} \subseteq A_{ij}$; aktivitas-aktivitas di dalam A'_{kj} saling selaras menurut pengandaiannya. Karena itu semua pasangan di B selaras: B sah.
5. **Seberapa besar B?** Gabungan pada langkah 3 saling lepas karena alasan yang baru saja disebut, jadi $|B| = |A_{ik}| + |A'_{kj}| + 1$.
6. Dengan memakai $|A'_{kj}| > |A_{kj}|$,

$$|B| = |A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|.$$

7. Tetapi B adalah himpunan aktivitas yang saling selaras yang semuanya terletak di S_{ij} — ia solusi sah bagi submasalah yang justru harus diselesaikan A_{ij} secara optimal — dan ia benar-benar lebih besar. Itu bertentangan dengan anggapan bahwa A_{ij} berukuran maksimum. 8. Karena itu A'_{kj} semacam itu tak ada, dan A_{kj} memang berukuran maksimum. Argumen setangkup, dengan menukar peran kedua sisinya, menunjukkan hal yang sama bagi A_{ik}. ■

Itulah bukti lengkap substruktur optimal, dan ia memakai persis perkakas §7.5.3: andaikan sebuah potongan tidak optimal, sulihkan potongan yang lebih baik, pastikan hasilnya sah, turunkan sebuah kontradiksi.

9.4.5 Program dinamis yang sebentar lagi kita buang

Teorema 9.1 langsung memberi sebuah program dinamis, yang rekurensinya sudah tampak di bagian bawah **Gambar 9.2**. Misalkan $c[i, j]$ adalah ukuran himpunan selaras berukuran maksimum di S_{ij}. Jika S_{ij} kosong, tak ada yang perlu dipilih dan $c[i, j] = 0$. Jika tidak, suatu aktivitas $a_k \in S_{ij}$ ada di dalam himpunan optimalnya, dan menurut Teorema 9.1 dan Persamaan (9.3) totalnya adalah $c[i, k] + c[k, j] + 1$ — tetapi kita tidak tahu a_k yang mana, jadi kita coba semuanya. Hasilnya adalah rekurensi Persamaan (9.4):

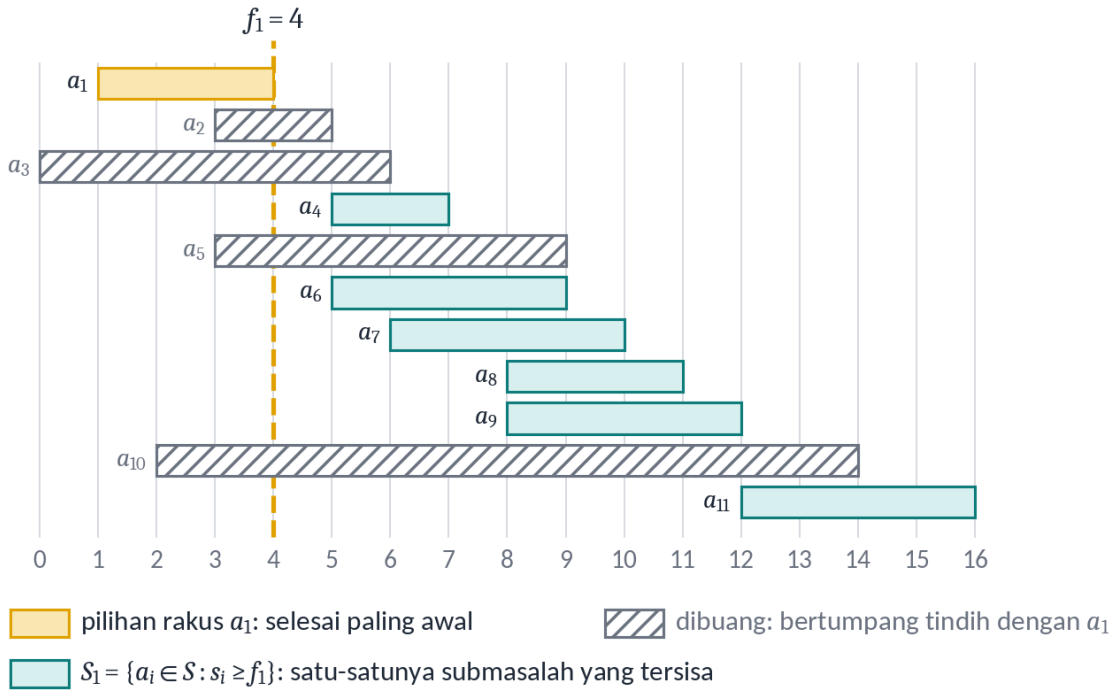
$$c[i, j] = \begin{cases} 0 & \text{jika } S_{ij} = \emptyset, \\ \max\limits_{a_k \in S_{ij}} \{\, c[i, k] + c[k, j] + 1 \,\} & \text{jika } S_{ij} \neq \emptyset. \end{cases} \tag{9.4}$$

Ini algoritma yang sungguh-sungguh baik. Ia juga, secara struktural, algoritma rantai matriks §7.6 yang muncul lagi: tabel berdimensi dua dengan pemaksimuman atas indeks dalam, karena itu kerjanya $\Theta(n^3)$ pada kasus terburuk. Anda dapat mem-*memoize*-nya (§7.7) atau mengisinya dari bawah ke atas (§7.4.2); bagaimanapun caranya, Anda akan membayar ruang $\Theta(n^2)$ dan kira-kira $n^3/6$ operasi.

Kita akan berbuat jauh lebih baik, sebab bagi persoalan *yang ini* pemaksimumannya tak perlu. Seluruh pokok bagian berikutnya adalah bahwa kita dapat mengenali a_k yang menang sekadar dengan memandangnya.

9.4.6 Pilihan rakusnya, dan mengapa hanya tersisa satu submasalah

Gambar 9.3 merangkum langkah ini.



solusi optimal $= \{a_1\} \cup$ **sebuah solusi optimal bagi** S_1

Gambar 9.3 Penyusutan menjadi satu submasalah: begitu pilihan rakus a_1 dijatuhkan, tersisalah satu submasalah S_1, dan solusi optimal bagi persoalan aslinya adalah a_1 bersama sebuah solusi optimal bagi S_1.

Ada dua hal yang sedang terjadi pada gambar itu, dan keduanya layak dipisahkan.

Pertama, penyederhanaan ruang submasalahnya. Alih-alih keluarga berindeks dua S_{ij}, kita mendefinisikan keluarga berindeks satu

$$S_k = \{\, a_i \in S \,:\; s_i \geq f_k \,\}, \tag{9.5}$$

yaitu himpunan aktivitas yang mulai setelah aktivitas a_k selesai. Mengapa satu indeks sudah cukup? Karena sebuah fakta tentang ujung *kanannya*. Jika kita selalu memilih aktivitas tersedia yang paling awal selesai, maka setelah memilih a_k kita tak pernah perlu membatasi ujung jauh selanya: setiap aktivitas tersisa yang mungkin kita pakai mulai setelah f_k, dan tak ada yang tersisih dari kanan. Secara formal, submasalah yang muncul dari satu jalannya algoritma rakus persis berupa $S_1 \subset S_{m_1} \subset \cdots$ — sebuah rantai, bukan sebuah tabel.

Kedua, penyusutan menjadi satu submasalah. Argumennya pendek dan lengkap, dan kita uraikan di sini satu langkah per baris.

1. Andaikan kita memilih a_1, aktivitas dengan waktu selesai terawal. (Pengurutannya menjamin $f_1 \leq f_i$ untuk semua i.)
2. Yang tersisa untuk diselesaikan adalah: carilah himpunan selaras berukuran maksimum di antara aktivitas yang selaras dengan a_1.
3. **Klaim:** setiap aktivitas yang selaras dengan a_1 mulai pada atau sesudah f_1 — artinya persoalan sisanya persis S_1 sebagaimana didefinisikan Persamaan (9.5), tanpa apa pun "sebelum a_1" yang perlu dirisaukan.
4. *Bukti klaimnya.* Misalkan a_i selaras dengan a_1. Menurut definisi keselarasan, entah $s_i \geq f_1$ entah $s_1 \geq f_i$. Andaikan yang kedua berlaku: $s_1 \geq f_i$. Karena $s_1 < f_1$ kita akan memperoleh $f_i \leq s_1 < f_1$, sehingga $f_i < f_1$ — sebuah aktivitas yang selesai benar-benar sebelum a_1. Tetapi f_1

adalah waktu selesai *minimum* atas seluruh S, jadi aktivitas semacam itu tak ada. Karena itu kasus kedua mustahil dan $s_i \geq f_1$. ■

5. Karena itu setelah memilih a_1 tersisalah tepat **satu** submasalah: S_1. Bukan dua, seperti pada Teorema 9.1; bukan n calon yang harus dicoba, seperti pada rekurensi §9.4.5. Satu.

> **DENGAN BAHASA SEDERHANA.** Karena a_1 adalah aktivitas yang paling pertama selesai di seluruh himpunannya, tak ada yang dapat menyelinap masuk *sebelumnya* — apa pun yang berakhir lebih awal akan bertentangan dengan a_1 sebagai yang pertama selesai. Jadi satu-satunya wilayah yang tersisa untuk diisi adalah waktu sesudah f_1, dan itu adalah satu contoh kasus lebih kecil dari persoalan yang persis sama.

9.4.7 Sifat pilihan rakus, dibuktikan dengan argumen penukaran

Kita sekarang punya bentuk sebuah algoritma rakus, tetapi belum punya hak untuk memakainya. Yang masih kurang adalah jaminan bahwa mengambil a_1 tidak merugikan kita — bahwa kita tidak sedang berada pada keadaan koin {1,3,4}, menyambar koin besar yang menarik lalu merusak segala yang menyusul.

> **TEOREMA 9.2 (sifat pilihan rakus bagi pemilihan aktivitas).** Tinjaulah sembarang submasalah tak kosong S_k, dan misalkan a_m sebuah aktivitas di S_k dengan **waktu selesai terawal**, yaitu $f_m = \min\{ f_i : a_i \in S_k \}$. Maka a_m termuat di dalam **suatu** himpunan bagian berukuran maksimum berisi aktivitas yang saling selaras dari S_k.

Bacalah pernyataannya baik-baik. Ia tidak mengatakan a_m ada di dalam *setiap* solusi optimal — §9.4.2 sudah memperlihatkan solusi optimal tak harus tunggal. Ia mengatakan ada *sekurang-kurangnya satu* solusi optimal yang memuat a_m. Itu persis sudah cukup: artinya memilih a_m tak mungkin mengunci kita di luar keoptimalan.

Bukti (argumen penukaran). Kita ambil sembarang solusi optimal lalu *mengubahnya* menjadi solusi yang memuat a_m, tanpa membuatnya menjadi lebih kecil. Karena kita selalu dapat melakukan hal itu, solusi optimal yang memuat a_m pastilah ada.

1. Misalkan A_k adalah **sembarang** himpunan bagian berukuran maksimum berisi aktivitas yang saling selaras dari S_k. (Ia ada karena S_k himpunan berhingga yang tak kosong, sehingga ia punya sekurang-kurangnya satu himpunan bagian yang sah, dan di antara berhingga banyak himpunan bagian yang sah itu ada yang terbesar.)
2. Urutkan aktivitas A_k menurut waktu selesai, dan misalkan a_j adalah yang **pertama** selesai, yaitu unsur A_k dengan waktu selesai terkecil.
3. **Kasus 1:** $a_j = a_m$**.** Maka A_k sendiri adalah solusi berukuran maksimum yang memuat a_m, dan kita selesai — tak ada yang perlu dibuktikan lagi.
4. **Kasus 2:** $a_j \neq a_m$**.** Bangunlah himpunan baru dengan menukar a_j dengan a_m:

$$A'_k = (A_k \setminus \{a_j\}) \cup \{a_m\}.$$

- Inilah **penukarannya**; segalanya sekarang bergantung pada dua pertanyaan tentang A'_k — apakah ia sah, dan seberapa besar ia?

5. **Seberapa besar** A'_k**?** Kita membuang tepat satu unsur dan menambahkan tepat satu unsur. Kita harus memastikan unsur yang ditambahkan belum ada di sana: $a_m \neq a_j$ menurut pengandaian Kasus 2, dan $a_m \notin A_k \setminus \{a_j\}$ — sebab andaikan a_m adalah suatu unsur lain dari A_k, maka f_m akan menjadi waktu selesai sebuah unsur A_k yang lebih besar daripada minimumnya f_j, sehingga $f_m > f_j$; padahal f_m adalah waktu selesai minimum atas seluruh $S_k \supseteq A_k$, jadi $f_m \leq f_j$, suatu kontradiksi. Karena itu

$$|A'_k| = |A_k| - 1 + 1 = |A_k|.$$

- Penukarannya **mempertahankan ukuran**. Inilah jantung tekniknya: kita tidak mengklaim pilihan rakusnya *lebih baik*, hanya bahwa ia *tidak lebih buruk*.

6. **Apakah A'_k sah?** Kita harus menunjukkan setiap pasangan di A'_k selaras.
   - Pasangan yang tidak melibatkan a_m: kedua unsurnya berasal dari A_k, yang saling selaras. ✓
   - Pasangan yang melibatkan a_m: misalkan a_i sembarang unsur dari $A_k \setminus \{a_j\}$. Kita harus menunjukkan a_m dan a_i selaras, dan kita akan menunjukkan pernyataan yang lebih kuat, yaitu $f_m \leq s_i$.
     - Karena a_j adalah aktivitas A_k yang pertama selesai, dan $a_i \in A_k$ dengan $a_i \neq a_j$, kita punya $f_j \leq f_i$; dan karena a_i dan a_j selaras (keduanya di A_k) dengan a_j selesai tidak lebih belakangan, satu-satunya kemungkinannya adalah $f_j \leq s_i$. (Alternatifnya, $f_i \leq s_j$, akan memberi $f_i \leq s_j < f_j \leq f_i$, yang mustahil.)
     - Karena a_m punya waktu selesai minimum di seluruh S_k, dan $a_j \in A_k \subseteq S_k$, kita punya $f_m \leq f_j$.
     - Merangkai keduanya: $f_m \leq f_j \leq s_i$, jadi $f_m \leq s_i$, yang merupakan syarat keselarasan $s_i \geq f_m$. ✓
   - Setiap pasangannya selaras, jadi A'_k adalah himpunan sah berisi aktivitas yang saling selaras, dan setiap unsurnya terletak di S_k.
7. Jadi A'_k adalah himpunan bagian sah dari S_k berisi aktivitas yang saling selaras dengan $|A'_k| = |A_k|$, yang merupakan ukuran maksimumnya. Karena itu A'_k sendiri adalah solusi berukuran maksimum, **dan ia memuat a_m**.
8. Pada kedua kasusnya, ada solusi berukuran maksimum yang memuat a_m. ■

> **ARGUMEN PENUKARAN, SEBAGAI CETAKAN.** Setiap bukti kebenaran algoritma rakus di buku ini berbentuk seperti yang baru Anda baca. Hafalkan keempat gerakannya. **(1) Ambil sembarang solusi optimal** A — bukan yang enak dipakai, sembarang saja. **(2) Temukan unsur** A **yang memainkan peran yang hendak dimainkan pilihan rakus Anda.** Di sini: aktivitas A yang pertama selesai. **(3) Tukarkan pilihan rakus Anda ke posisi itu**, lalu buktikan dua hal: hasilnya tetap **layak**, dan ia **tidak lebih kecil** (atau tidak lebih mahal). **(4) Simpulkan bahwa ada solusi optimal yang memuat pilihan rakusnya**, sehingga pilihan rakus itu *aman*. Lalu, secara terpisah, buktikan substruktur optimal supaya argumennya dapat diulangi pada apa yang tersisa. Keduanya bersama-sama, lewat induksi atas cacah pilihannya, membuktikan seluruh algoritmanya optimal.

9.4.8 Menyatukan kedua paruhnya

Teorema 9.1 dan 9.2 berpadu menjadi pernyataan yang benar-benar diperlukan algoritmanya.

> **AKIBAT 9.3.** Misalkan S_k sebuah submasalah tak kosong dan misalkan $a_m \in S_k$ punya waktu selesai terawal di S_k. Misalkan A^* sebuah himpunan bagian selaras berukuran maksimum dari S_m. Maka $\{a_m\} \cup A^*$ adalah himpunan bagian selaras berukuran maksimum dari S_k.

Bukti. Menurut Teorema 9.2 ada himpunan bagian selaras berukuran maksimum B dari S_k dengan $a_m \in B$. Menurut argumen §9.4.6 langkah 4, yang diterapkan di dalam S_k, setiap aktivitas B selain a_m mulai pada atau sesudah f_m, jadi $B \setminus \{a_m\}$ adalah himpunan bagian selaras dari S_m, sehingga $|B| - 1 \leq |A^*|$. Karena itu $|B| \leq |A^*| + 1 = |\{a_m\} \cup A^*|$. Sebaliknya $\{a_m\} \cup A^*$ selaras (setiap unsur $A^* \subseteq S_m$ mulai pada atau sesudah f_m) dan terletak di S_k, jadi ia solusi yang sah dan

$|\{a_m\} \cup A^*| \leq |B|$ menurut kemaksimuman B. Kedua ketaksamaan itu memberi kesamaan, jadi $\{a_m\} \cup A^*$ berukuran maksimum. ■

Akibat 9.3 adalah izin untuk berekursi. Ia berkata: ambil yang paling awal selesai, lalu selesaikan *satu* persoalan sejenis yang lebih kecil, lalu rekatkan. Lewat induksi atas cacah aktivitas yang tersisa — dengan kasus dasarnya submasalah kosong, yang solusi tunggal dan karenanya optimalnya adalah himpunan kosong — algoritma pada bagian berikutnya benar.

9.4.9 Algoritma rakus rekursifnya

Gambar 9.4 memberikan pseudokodenya.

```
    RECURSIVE-ACTIVITY-SELECTOR (s, f, k, n)
1   m = k + 1
2   while m ≤ n and s[m] < f[k]          // cari aktivitas pertama di S_k yang selesai
3       m = m + 1
4   if m ≤ n
5       return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR (s, f, m, n)
6   else return ∅
```

Gambar 9.4 Recursive-Activity-Selector dalam pseudokode. Gelung **while** pada baris 2 mencari aktivitas pertama di S_k yang selesai; pemanggilannya dilakukan dengan $k = 0$, memakai aktivitas fiktif a_0 dengan $f_0 = 0$.

Baris demi baris, dengan kata-kata:

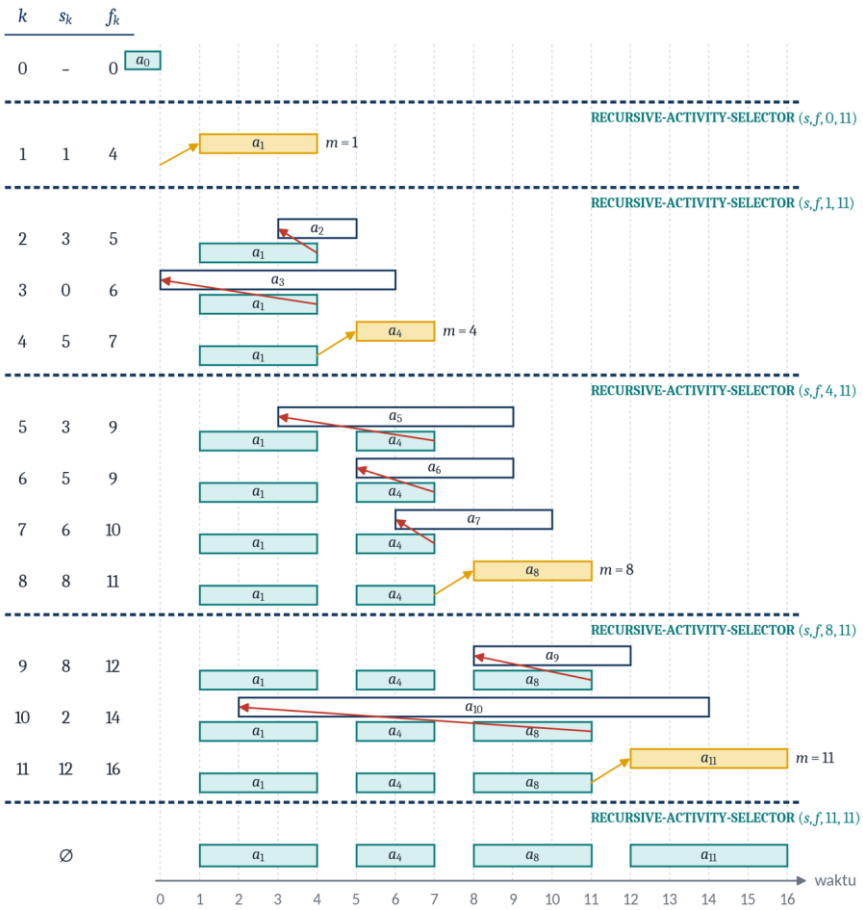
- **Baris 1**, `m = k + 1`. Mulailah memandang aktivitas tepat sesudah a_k. Karena lariknya terurut menurut waktu selesai, memindai maju dari $k + 1$ akan menyinggahi aktivitas dalam urutan waktu selesai yang menaik, sehingga yang *pertama* cocok yang kita jumpai adalah yang paling awal selesai.
- **Baris 2**, `while m ≤ n and s[m] < f[k]`. Teruslah maju selama aktivitas a_m *tidak* berada di S_k. Ingat bahwa S_k menuntut $s_m \geq f_k$; jadi $s_m < f_k$ berarti a_m mulai terlalu dini dan harus dilewati.
- **Baris 3**, `m = m + 1`. Maju.
- **Baris 4**, `if m ≤ n`. Apakah kita menemukannya sebelum melewati ujung lariknya?
- **Baris 5**, `return {a_m} ∪ Recursive-Activity-Selector(s, f, m, n)`. Ambil a_m — inilah pilihan rakusnya, yang dibenarkan Teorema 9.2 — lalu berekursilah pada satu submasalah yang tersisa, S_m, yang dibenarkan Akibat 9.3.
- **Baris 6**, `else return ∅`. S_k kosong; himpunan kosong adalah solusi optimalnya.

Pemanggilan awalnya adalah `Recursive-Activity-Selector(s, f, 0, n)`. Penjaga a_0 punya $f_0 = 0$, jadi uji `s[m] < f[0]` menjadi `s[m] < 0`, yang salah bagi setiap aktivitas yang sungguhan — gelungnya tak melewati apa pun dan m berakhir di 1, sehingga a_1 dengan tepat menjadi pilihan rakus yang pertama.

> **INI SEBENARNYA BUKAN REKURSI.** Perhatikan bahwa pemanggilan rekursif di baris 5 adalah hal **terakhir** yang dilakukan prosedurnya dengan kerjanya sendiri — ia sebuah *tail call* (panggilan ekor). Tak ada yang dihitung setelah panggilannya kembali kecuali merekatkan a_m di depan jawabannya. Rekursi berbentuk demikian adalah gelung yang menyamar, dan justru itulah sebabnya §9.4.11 dapat menuliskannya ulang sebagai gelung. Bandingkan dengan rekursi rantai matriks Bab 7, yang membuat dua panggilan lalu memadukan hasilnya; yang itu sungguh-sungguh rekursif dan tak dapat diratakan menjadi satu gelung.

9.4.10 Penelusuran pada contoh berjalan

Gambar 9.5 adalah penelusuran lengkap atas **Recursive-Activity-Selector** pada sebelas aktivitas **Gambar 9.1**. Inilah gambar yang paling banyak memberi keterangan dalam bab ini; pelajarilah sebelum membaca lebih jauh.



Gambar 9.5 Penelusuran atas **Recursive-Activity-Selector** pada sebelas aktivitas **Gambar 9.1**. Tiap baris adalah satu iterasi gelung **while**: batang atasnya adalah aktivitas a_k (amber bila diambil), batang-batang toska adalah aktivitas yang sudah terpilih, dan anak panahnya menandai pembandingan s_k terhadap f milik pilihan terakhir. Garis-garis putus-putus tebalnya memisahkan pemanggilan-pemanggilan rekursif, masing-masing diberi label argumennya; baris terbawahnya memperlihatkan jawaban $\{a_1, a_4, a_8, a_{11}\}$ yang menempati sumber dayanya dari waktu 1 sampai waktu 16.

Inilah cara membacanya. Tiga lajur paling kiri mengulang k, s_k, dan f_k dari tabel datanya. Tiap baris diagramnya bersesuaian dengan satu nilai m yang diperiksa gelung **while**. Batang atas pada tiap baris adalah aktivitas yang sedang diperiksa — putih bila dilewati, kuning amber bila diambil; batang-batang hijau toska di bawahnya adalah aktivitas yang sudah terpilih. Anak panah merah yang menunjuk ke kiri dari batang yang diperiksa kembali ke batang toska terakhir berarti "aktivitas ini mulai sebelum yang itu selesai, jadi lewati"; anak panah amber yang menunjuk ke kanan berarti aktivitasnya mulai sesudah pilihan terakhir selesai. Baris yang berlabel `m = …` adalah baris tempat gelungnya berhenti dan aktivitasnya diambil.

Sekarang penelusurannya dengan kata-kata, dengan setiap pembandingannya ditampakkan.

Panggilan 1: `Recursive-Activity-Selector(s, f, 0, 11)`. m mulai di 1. Uji `s[1] < f[0]`, yaitu $1 < 0$? **Tidak.** Badan gelungnya tak pernah berjalan. $m = 1 \leq 11$, jadi ambil a_1 lalu berekursi dengan $k = 1$.

Panggilan 2: `Recursive-Activity-Selector(s, f, 1, 11)`. Sekarang $f_k = f_1 = 4$. * $m = 2$: apakah $s_2 < 4$? $3 < 4$ — **ya**, lewati. (a_2 mulai pada 3, sementara a_1 masih berjalan.) * $m = 3$: apakah $s_3 < 4$? $0 < 4$ — **ya**, lewati. * $m = 4$: apakah $s_4 < 4$? $5 < 4$ — **tidak**. Berhenti. $m = 4$, ambil a_4, berekursi dengan $k = 4$.

Panggilan 3: `Recursive-Activity-Selector(s, f, 4, 11)`. Sekarang $f_k = f_4 = 7$. * $m = 5$: $s_5 = 3 < 7$ — lewati. * $m = 6$: $s_6 = 5 < 7$ — lewati. * $m = 7$: $s_7 = 6 < 7$ — lewati. * $m = 8$: $s_8 = 8 < 7$? **Tidak**. Berhenti. Ambil a_8, berekursi dengan $k = 8$.

Panggilan 4: `Recursive-Activity-Selector(s, f, 8, 11)`. Sekarang $f_k = f_8 = 11$. * $m = 9$: $s_9 = 8 < 11$ — lewati. * $m = 10$: $s_{10} = 2 < 11$ — lewati. * $m = 11$: $s_{11} = 12 < 11$? **Tidak**. Berhenti. Ambil a_{11}, berekursi dengan $k = 11$.

Panggilan 5: `Recursive-Activity-Selector(s, f, 11, 11)`. m mulai di 12. Syarat gelungnya menuntut $m \leq n = 11$, yang langsung gagal. Lalu `if m ≤ n` bernilai salah, jadi kembalikan $\varnothing$.

Menguraikannya balik: $\{a_{11}\} \cup \varnothing$, lalu $\{a_8, a_{11}\}$, lalu $\{a_4, a_8, a_{11}\}$, lalu $\{a_1, a_4, a_8, a_{11}\}$. Empat aktivitas, dan §9.4.2 sudah memberi tahu kita empat adalah maksimumnya. Mesinnya setuju, dari keluaran terverifikasi **Kode 9.2**:

```
=== RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11) ===
RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11)  -> m = 1, take a1
  RECURSIVE-ACTIVITY-SELECTOR(s, f, 1, 11)  -> m = 4, take a4
    RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 11)  -> m = 8, take a8
      RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 11)  -> m = 11, take a11
        RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)  -> no activity left, return {}
  result = {a1, a4, a8, a11}, size 4
```

Bentuk iteratifnya dan pemeriksaan tuntasnya menyusul pada jalannya yang sama, dan keduanya sepakat:

```
=== GREEDY-ACTIVITY-SELECTOR(s, f) ===
  result = {a1, a4, a8, a11}, size 4
  same answer as the recursive form: yes

=== Brute force over all 2^11 = 2048 subsets ===
  true maximum size = 4
```

> **HITUNGLAH KERJANYA.** Di sepanjang kelima panggilannya, peubah m mengambil nilai 1; 2,3,4; 5,6,7,8; 9,10,11; dan 12. Setiap indeks dari 1 sampai 11 diperiksa **tepat sekali**, ditambah satu uji terakhir yang melewati ujung lariknya. Itulah pengamatan kunci bagi waktu jalannya: meskipun prosedurnya rekursif, cacah total iterasi gelung atas seluruh panggilannya adalah n, sebab m tak pernah bergerak mundur.

9.4.11 Algoritma rakus iteratifnya

Langkah terakhir dalam mengembangkan sebuah solusi rakus adalah mengubah rekursinya menjadi gelung. **Gambar 9.6** adalah hasilnya.

```
GREEDY-ACTIVITY-SELECTOR (s,f)
1   n = s.length
2   A = {a_1}
3   k = 1
4   for m = 2 to n
5       if s[m] ≥ f[k]
6           A = A ∪ {a_m}
7           k = m
8   return A
```

Gambar 9.6 Greedy-Activity-Selector, bentuk iteratif dari **Gambar 9.4**. Peubah k selalu memegang indeks pilihan yang paling mutakhir, sehingga $f[k]$ adalah waktu saat sumber dayanya berikutnya menjadi bebas.

Baris demi baris:

- **Baris 1**, `n = s.length`. Ada berapa banyak aktivitasnya.
- **Baris 2**, `A = {a1}`. Ambil a_1 tanpa syarat. Ini sah karena lariknya terurut menurut waktu selesai, sehingga a_1 adalah yang paling awal selesai di seluruh S, dan Teorema 9.2 mengatakan ia aman.
- **Baris 3**, `k = 1`. Ingatlah bahwa a_1 adalah hal terakhir yang terpilih.
- **Baris 4**, `for m = 2 to n`. Berjalanlah maju melalui aktivitas yang tersisa dalam urutan waktu selesai yang menaik.
- **Baris 5**, `if s[m] ≥ f[k]`. Apakah a_m selaras dengan pilihan terakhir? Karena pilihan-pilihan sejauh ini terurut dan tak bertindih, keselarasan dengan pilihan *terakhir* menyiratkan keselarasan dengan semuanya.
- **Baris 6**, `A = A ∪ {a_m}`. Pilih ia.
- **Baris 7**, `k = m`. Sekarang ialah pilihan yang terakhir.
- **Baris 8**, `return A`.

Invarian gelungnya — teknik §2.3 — membuat kebenarannya menjadi persis.

> **INVARIAN GELUNG.** Pada awal tiap iterasi gelung **for** baris 4, A adalah himpunan bagian berukuran maksimum berisi aktivitas yang saling selaras dari $\{a_1, \ldots, a_{m-1}\}$, dan a_k adalah aktivitas di A dengan waktu selesai terakhir. **Inisialisasi.** Sebelum iterasi pertama, $m = 2$, jadi himpunannya $\{a_1\}$; himpunan bagian selaras berukuran maksimum dari himpunan berunsur satu adalah unsur itu sendiri, dan $k = 1$. ✓ **Pemeliharaan.** Tinjaulah iterasi m. Jika $s_m < f_k$ maka a_m bertindih dengan a_k; karena setiap aktivitas A selesai pada atau sebelum f_k dan a_m mulai sebelum f_k, menambahkan a_m mengharuskan kita membuang sesuatu, dan Teorema 9.2 yang diterapkan pada submasalah S_k mengatakan bahwa yang paling awal selesai di sana — yaitu aktivitas berikutnya yang *akan* kita terima — sudah aman, sehingga melewati a_m tidak merugikan apa pun. Jika $s_m \geq f_k$ maka a_m adalah anggota S_k yang paling awal selesai (segala yang di antara k dan m sudah ditolak, dan lariknya terurut menurut waktu selesai), sehingga menurut Akibat 9.3 mengambilnya menjaga A tetap berukuran maksimum. Bagaimanapun

> juga invariannya berlaku pada $m + 1$. ✓ **Terminasi.** Gelungnya berakhir dengan $m = n + 1$, jadi A adalah himpunan bagian selaras berukuran maksimum dari $\{a_1, \ldots, a_n\} = S$ — dan itulah yang kita inginkan. ✓

Mesinnya memastikan bahwa kedua bentuknya sepakat: blok yang dikutip di §9.4.10 memperlihatkan `same answer as the recursive form: yes`, dan sapuan *brute force* atas seluruh 2048 himpunan bagian pada jalannya yang sama meletakkan maksimum sejatinya pada 4.

9.4.12 Waktu jalannya

Kedua bentuk algoritmanya berjalan dalam waktu $\Theta(n)$ **jika masukannya sudah terurut**.

Bentuk iteratifnya. Baris 1–3 berwaktu tetapan. Gelung **for** baris 4 berjalan tepat $n - 1$ kali, dan badannya — satu pembandingan, mungkin satu penambahan dan satu penugasan — berwaktu tetapan. Totalnya: $\Theta(n)$.

Bentuk rekursifnya. Ini memerlukan pengamatan dari §9.4.10. Gelung **while** tiap panggilan memajukan m, dan m diteruskan ke panggilan berikutnya sebagai k yang baru, dan dari sana panggilan berikutnya melanjutkan pada $m + 1$. Jadi m menaik secara monoton dari 1 sampai paling banyak $n + 1$ di sepanjang seluruh eksekusinya, dan cacah total iterasi gelung atas semua panggilannya paling banyak n. Tiap iterasinya berbiaya $\Theta(1)$, dan cacah panggilannya paling banyak sebesar cacah pilihannya ditambah satu, yaitu paling banyak $n + 1$. Totalnya: $\Theta(n)$.

Jika masukannya *belum* terurut, kita harus mengurutkannya lebih dulu, yang berbiaya $O(n\lg n)$ dengan algoritma mana pun dari Bab 5. Biaya itu mendominasi, jadi:

> **TEOREMA 9.4.** Persoalan pemilihan aktivitas atas n aktivitas dapat diselesaikan dalam waktu $\Theta(n\lg n)$ dan ruang tambahan $\Theta(1)$ di luar keluarannya, dengan mengurutkan aktivitasnya menaik tak turun menurut waktu selesai lalu menjalankan **Greedy-Activity-Selector**.

Bandingkan dengan program dinamis §9.4.5: waktu $\Theta(n^3)$ dan ruang $\Theta(n^2)$. Pada $n = 1000$ itu selisih antara kira-kira sepuluh ribu operasi dan kira-kira 167 juta, serta antara beberapa bita dan empat megabita. Inilah yang dibelikan sebuah bukti untuk Anda.

Kode 9.2 mengukur bentuknya. Tabel yang dicetaknya adalah **Tabel 9.3**.

Tabel 9.3 Pengurutan ditambah satu sapuan rakus, diukur oleh **Kode 9.2** pada aktivitas acak. Lajur ketiganya adalah waktu terukur dibagi $n\lg n$; ia bertahan di sekitar 4 di sepanjang rentang n yang 32 kali lipat, dan itulah wajah $\Theta(n\lg n)$ pada mesin yang sungguhan.

n	Waktu (ms)	$t/(n\lg n)\times 10^6$	Aktivitas terpilih
250 000	18.07	4.03	122 426
500 000	36.84	3.89	177 718
1 000 000	91.81	4.61	252 379
2 000 000	163.89	3.91	357 471
4 000 000	373.26	4.25	504 635
8 000 000	762.48	4.16	713 102

Lajur ketiganya adalah alat diagnosisnya. Jika waktu jalannya sungguh-sungguh $c \cdot n\lg n$ untuk suatu tetapan c, maka membagi waktu terukur dengan $n\lg n$ pasti memberi bilangan yang kira-kira sama pada setiap ukuran — dan memang begitu: 4.03, 3.89, 4.61, 3.91, 4.25, 4.16, dengan goyangannya dapat dinisbahkan kepada gejala *cache* dan kepada sistem operasinya. Seandainya algoritmanya $\Theta(n^2)$, lajur itu akan berlipat dua pada setiap barisnya.

9.4.13 Program lengkapnya

Kode 9.2 adalah seluruh isi §9.4 dalam bentuk yang dapat dijalankan: kedua algoritmanya, pemeriksaan *brute force*-nya, keempat calon aturannya berikut contoh sanggahannya, dan satu jalannya pengukuran waktu.

Kode 9.2 Pemilihan aktivitas secara utuh: pemilih rakus rekursif dan iteratif, pemeriksaan *brute force*, keempat calon aturan berikut contoh sanggahan bagi ketiga yang keliru, dan satu jalannya pengukuran waktu. Kompilasi dengan `g++ -std=c++17 -O2 -o activity activity.cpp` lalu jalankan `./activity`.

```
// activity.cpp — Masalah pemilihan aktivitas.
//   * 11 aktivitas contoh berjalan, diselesaikan dengan pemilih rakus rekursif
//     dan dengan pemilih rakus iteratif;
//   * pemeriksaan brute force atas seluruh 2^n himpunan bagian, bahwa 4 memang
maksimumnya;
//   * tiga aturan rakus yang masuk akal tetapi TIDAK benar, masing-masing dengan
//     contoh sanggahan terkecil yang ditemukan lewat pencarian tuntas atas kasus kecil;
//   * satu jalannya pengukuran waktu yang membenarkan bentuk Theta(n lg n).
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <numeric>
#include <chrono>
#include <random>
#include <cmath>
using namespace std;

struct Act { int id, s, f; };

// ---------- contoh berjalan, sudah terurut menurut waktu selesai ----------
// indeks 0 adalah penjaga a0 dengan f0 = 0, sehingga "s[m] >= f[0]" selalu benar
static const vector<int> S = { 0, 1, 3, 0, 5, 3, 5, 6, 8, 8,  2, 12 };
static const vector<int> F = { 0, 4, 5, 6, 7, 9, 9,10,11,12, 14, 16 };

// ---------- RECURSIVE-ACTIVITY-SELECTOR, persis seperti di Subbab 9.4.9 ----
vector<int> recursiveSelector(const vector<int>& s, const vector<int>& f,
                              int k, int n, int depth = 0) {
    cout << string(2 * depth, ' ') << "RECURSIVE-ACTIVITY-SELECTOR(s, f, "
         << k << ", " << n << ")";
    int m = k + 1;
    while (m <= n && s[m] < f[k]) ++m;            // aktivitas pertama di S_k yang selesai
    if (m <= n) {
        cout << "  -> m = " << m << ", take a" << m << "\n";
        vector<int> rest = recursiveSelector(s, f, m, n, depth + 1);
        rest.insert(rest.begin(), m);
        return rest;
    }
    cout << "  -> no activity left, return {}\n";
    return {};
}
```

Kode 9.2 (Lanjutan) *Pemilihan aktivitas secara utuh: pemilih rakus rekursif dan iteratif, pemeriksaan brute force, keempat calon aturan berikut contoh sanggahan bagi ketiga yang keliru, dan satu jalannya pengukuran waktu. Kompilasi dengan g++ -std=c++17 -O2 -o activity activity.cpp lalu jalankan ./activity.*

```
// ---------- GREEDY-ACTIVITY-SELECTOR, bentuk iteratifnya ----------
vector<int> greedySelector(const vector<int>& s, const vector<int>& f, int n) {
    vector<int> A = { 1 };                        // A = {a1}
    int k = 1;
    for (int m = 2; m <= n; ++m)
        if (s[m] >= f[k]) { A.push_back(m); k = m; }
    return A;
}

// ---------- brute force atas tiap himpunan bagian: berapa SESUNGGUHNYA maksimumnya? -----
-----
int bruteForceMax(const vector<Act>& a) {
    int n = (int)a.size(), best = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        vector<int> chosen;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) chosen.push_back(i);
        bool ok = true;
        for (size_t i = 0; i + 1 < chosen.size() && ok; ++i)
            for (size_t j = i + 1; j < chosen.size() && ok; ++j) {
                const Act& x = a[chosen[i]]; const Act& y = a[chosen[j]];
                if (!(x.s >= y.f || y.s >= x.f)) ok = false;   // keduanya bertindih
            }
        if (ok) best = max(best, (int)chosen.size());
    }
    return best;
}

// ---------- keempat calon aturan rakus ----------
// Masing-masing berulang kali memilih aktivitas tersisa yang terbaik menurut
// patokannya sendiri, lalu membuang semua yang tak selaras dengannya.
typedef int (*Rule)(const vector<Act>&, const vector<char>&);

int ruleEarliestFinish(const vector<Act>& a, const vector<char>& alive) {
    int best = -1;
    for (size_t i = 0; i < a.size(); ++i) if (alive[i])
        if (best < 0 || a[i].f < a[best].f) best = (int)i;
    return best;
}
int ruleEarliestStart(const vector<Act>& a, const vector<char>& alive) {
    int best = -1;
    for (size_t i = 0; i < a.size(); ++i) if (alive[i])
        if (best < 0 || a[i].s < a[best].s) best = (int)i;
    return best;
}
int ruleShortest(const vector<Act>& a, const vector<char>& alive) {
    int best = -1;
    for (size_t i = 0; i < a.size(); ++i) if (alive[i])
```

Kode 9.2 (Lanjutan) *Pemilihan aktivitas secara utuh: pemilih rakus rekursif dan iteratif, pemeriksaan brute force, keempat calon aturan berikut contoh sanggahan bagi ketiga yang keliru, dan satu jalannya pengukuran waktu. Kompilasi dengan g++ -std=c++17 -O2 -o activity activity.cpp lalu jalankan ./activity.*

```
        if (best < 0 || (a[i].f - a[i].s) < (a[best].f - a[best].s)) best = (int)i;
    return best;
}
int ruleFewestConflicts(const vector<Act>& a, const vector<char>& alive) {
    int best = -1, bestc = 0;
    for (size_t i = 0; i < a.size(); ++i) if (alive[i]) {
        int c = 0;
        for (size_t j = 0; j < a.size(); ++j) if (alive[j] && j != i)
            if (!(a[i].s >= a[j].f || a[j].s >= a[i].f)) ++c;
        if (best < 0 || c < bestc) { best = (int)i; bestc = c; }
    }
    return best;
}

vector<int> runRule(const vector<Act>& a, Rule r) {
    vector<char> alive(a.size(), 1);
    vector<int> picked;
    for (;;) {
        int i = r(a, alive);
        if (i < 0) break;
        picked.push_back(i);
        for (size_t j = 0; j < a.size(); ++j)
            if (alive[j] && !(a[i].s >= a[j].f || a[j].s >= a[i].f)) alive[j] = 0;
        alive[i] = 0;
    }
    return picked;
}

// Ketiga aturan yang keliru, masing-masing pada contoh kasus terkecil (ditemukan
// lewat pencarian tuntas) yang membuatnya kalah dari optimum.
void showCounterexample(const char* name, Rule r, vector<Act> a) {
    vector<int> p = runRule(a, r);
    int opt = bruteForceMax(a);
    cout << "  " << name << " on an instance with n = " << a.size() << ":\n";
    cout << "      activity :";
    for (auto& x : a) cout << setw(9) << ("a" + to_string(x.id));
    cout << "\n      [s, f)   :";
    for (auto& x : a) cout << setw(9) << ("[" + to_string(x.s) + "," + to_string(x.f) +
")");
    cout << "\n      the rule picks " << p.size() << " activities {";
    sort(p.begin(), p.end());
    for (size_t i = 0; i < p.size(); ++i) { if (i) cout << ", "; cout << "a" << a[p[i]].id;
}
    cout << "}\n      brute force says the optimum is " << opt
         << "  ->  the rule is " << (p.size() < (size_t)opt ? "WRONG" : "ok") << "\n\n";
}
```

Kode 9.2 (Lanjutan) *Pemilihan aktivitas secara utuh: pemilih rakus rekursif dan iteratif, pemeriksaan brute force, keempat calon aturan berikut contoh sanggahan bagi ketiga yang keliru, dan satu jalannya pengukuran waktu. Kompilasi dengan g++ -std=c++17 -O2 -o activity activity.cpp lalu jalankan ./activity.*

```
int main() {
    int n = 11;
    vector<Act> example(n);
    for (int i = 1; i <= n; ++i) example[i - 1] = { i, S[i], F[i] };

    cout << "=== The 11 activities of the running example ===\n";
    cout << "  i :"; for (int i = 1; i <= n; ++i) cout << setw(4) << i;       cout << "\n";
    cout << " si :"; for (int i = 1; i <= n; ++i) cout << setw(4) << S[i];    cout << "\n";
    cout << " fi :"; for (int i = 1; i <= n; ++i) cout << setw(4) << F[i];    cout <<
"\n\n";

    cout << "=== RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11) ===\n";
    vector<int> rec = recursiveSelector(S, F, 0, n);
    cout << "  result = {";
    for (size_t i = 0; i < rec.size(); ++i) { if (i) cout << ", "; cout << "a" << rec[i]; }
    cout << "}, size " << rec.size() << "\n\n";

    cout << "=== GREEDY-ACTIVITY-SELECTOR(s, f) ===\n";
    vector<int> it = greedySelector(S, F, n);
    cout << "  result = {";
    for (size_t i = 0; i < it.size(); ++i) { if (i) cout << ", "; cout << "a" << it[i]; }
    cout << "}, size " << it.size() << "\n";
    cout << "  same answer as the recursive form: "
         << (rec == it ? "yes" : "NO") << "\n\n";

    cout << "=== Brute force over all 2^11 = 2048 subsets ===\n";
    cout << "  true maximum size = " << bruteForceMax(example) << "\n\n";

    cout << "=== Four greedy rules on the running example ===\n";
    struct { const char* name; Rule r; } rules[] = {
        { "earliest finish time ", ruleEarliestFinish },
        { "earliest start time  ", ruleEarliestStart },
        { "shortest duration    ", ruleShortest },
        { "fewest conflicts     ", ruleFewestConflicts },
    };
    for (auto& R : rules) {
        vector<int> p = runRule(example, R.r);
        sort(p.begin(), p.end());
        cout << "  " << R.name << " -> {";
        for (size_t i = 0; i < p.size(); ++i) { if (i) cout << ", "; cout << "a" <<
example[p[i]].id; }
        cout << "}, size " << p.size() << "\n";
    }
    cout << "\n=== Smallest counterexample for each incorrect rule ===\n";
    showCounterexample("earliest start time", ruleEarliestStart,
        { {1,0,8}, {2,1,4}, {3,0,1} });
    showCounterexample("shortest duration ", ruleShortest,
        { {1,5,8}, {2,1,5}, {3,4,6} });
```

Kode 9.2 (Lanjutan) *Pemilihan aktivitas secara utuh: pemilih rakus rekursif dan iteratif, pemeriksaan brute force, keempat calon aturan berikut contoh sanggahan bagi ketiga yang keliru, dan satu jalannya pengukuran waktu. Kompilasi dengan g++ -std=c++17 -O2 -o activity activity.cpp lalu jalankan ./activity.*

```
    showCounterexample("fewest conflicts  ", ruleFewestConflicts,
        { {1,4,7}, {2,9,13}, {3,2,5}, {4,1,4}, {5,1,2},
          {6,6,10}, {7,0,2}, {8,7,15}, {9,10,11} });

    cout << "=== Timing: sort dominates, so the shape is n lg n ===\n";
    cout << setw(12) << "n" << setw(16) << "time (ms)" << setw(16) << "t / (n lg n)" <<
"\n";
    mt19937 rng(7);
    for (long long N = 250000; N <= 8000000; N *= 2) {
        vector<Act> a((size_t)N);
        for (long long i = 0; i < N; ++i) {
            int x = (int)(rng() % 100000000), d = 1 + (int)(rng() % 1000);
            a[(size_t)i] = { (int)i, x, x + d };
        }
        auto t0 = chrono::steady_clock::now();
        sort(a.begin(), a.end(), [](const Act& p, const Act& q) { return p.f < q.f; });
        int cnt = 0, last = -1;
        for (auto& x : a) if (x.s >= last) { ++cnt; last = x.f; }
        auto t1 = chrono::steady_clock::now();
        double ms = chrono::duration<double, milli>(t1 - t0).count();
        double nlgn = (double)N * (log((double)N) / log(2.0));
        cout << setw(12) << N << setw(16) << fixed << setprecision(2) << ms
             << setw(16) << setprecision(4) << (ms * 1e6 / nlgn)
             << "   (selected " << cnt << ")\n";
    }
    return 0;
}
```

9.5 Unsur-Unsur Strategi Rakus

Kita kini sudah menyelesaikan satu persoalan secara rakus dan membuktikannya benar. Bagian ini menyarikan metodenya, supaya Anda dapat menerapkannya pada persoalan yang belum pernah Anda lihat.

9.5.1 Prosedur berlangkah enam

Prosedur itu paling baik dinyatakan sebagai daftar bernomor, sebab inilah daftar periksa yang harus dikerjakan bagi setiap calon algoritma rakus.

> **STRATEGI RAKUS DALAM ENAM LANGKAH. 1.** Tetapkan **substruktur optimal** persoalannya. **2.** Kembangkan **solusi rekursif**. **3.** Tunjukkan bahwa jika kita menjatuhkan pilihan rakusnya, maka **hanya tersisa satu submasalah**. **4.** Buktikan bahwa menjatuhkan pilihan rakus itu selalu **aman**. (Langkah 3 dan 4 boleh terjadi dalam urutan mana pun.) **5.** Kembangkan **algoritma rekursif** yang mewujudkan strategi rakusnya. **6. Ubahlah** algoritma rekursif itu menjadi algoritma iteratif.

Cocokkan keenamnya dengan §9.4 dan Anda akan menemukan semuanya:

Tabel 9.4 Enam langkah strategi rakus, dipetakan ke pekerjaan yang dikerjakan di §9.4. Tak ada yang dilewati dan tak ada yang ditambahkan.

Langkah	Tempatnya dikerjakan di §9.4	Apa yang dihasilkannya
1. substruktur optimal	§9.4.4, Teorema 9.1	$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, potongannya optimal
2. solusi rekursif	§9.4.5	$c[i,j] = \max_k\{c[i,k] + c[k,j] + 1\}$
3. hanya tersisa satu submasalah	§9.4.6	setelah mengambil a_1, tinggal S_1
4. pilihan rakusnya aman	§9.4.7, Teorema 9.2	argumen penukaran
5. algoritma rekursif	§9.4.9	**Recursive-Activity-Selector**
6. algoritma iteratif	§9.4.11	**Greedy-Activity-Selector**

Ada pula ringkasan tiga baris yang lebih pendek tentang apa saja yang harus *dibuktikan*. Inilah versi yang perlu Anda bawa di kepala ketika menyelesaikan persoalan baru dengan waktu terbatas, seperti yang dikerjakan Bab 12 untuk persoalan persis sejenis ini:

> **TIGA HAL YANG HARUS DIBUKTIKAN. 1. Bentuklah** persoalan optimasinya menjadi persoalan yang di dalamnya kita menjatuhkan sebuah pilihan lalu tertinggal dengan **satu** submasalah untuk diselesaikan. **2. Buktikan** bahwa selalu ada solusi optimal bagi persoalan aslinya yang menjatuhkan pilihan rakusnya — sehingga pilihan rakus itu selalu **aman**. **3. Peragakan substruktur optimalnya** dengan menunjukkan bahwa, setelah pilihan rakusnya dijatuhkan, yang tersisa adalah sebuah submasalah yang bersifat: memadukan solusi optimal submasalah itu dengan pilihan rakusnya menghasilkan solusi optimal bagi persoalan aslinya.

9.5.2 Bahan pertama: sifat pilihan rakus

> **DEFINISI.** Sebuah persoalan memiliki **sifat pilihan rakus** (*greedy-choice property*) jika sebuah solusi optimal global dapat dirakit dengan menjatuhkan pilihan-pilihan yang optimal secara lokal (rakus) — yaitu jika untuk setiap contoh kasus, pilihan yang dijatuhkan aturan rakusnya termuat di dalam **suatu** solusi optimal bagi contoh kasus itu.

Tiga hal tentang definisi ini layak diperhatikan.

"Suatu", bukan "setiap". Sebagaimana ditunjukkan §9.4.2, contoh berjalan itu punya dua solusi optimal yang berlainan, $\{a_1, a_4, a_8, a_{11}\}$ dan $\{a_2, a_4, a_9, a_{11}\}$. Pilihan rakus a_1 ada di yang pertama dan tidak ada di yang kedua. Itu tak apa-apa. Yang kita perlukan hanyalah bahwa *sekurang-kurangnya satu* solusi optimal sependapat dengan kita, sebab dengan begitu mengikuti arahan solusi tersebut tak mungkin merugikan.

"Tanpa menimbang hasil dari submasalahnya". Hal ini perlu ditekankan. Pilihannya boleh bergantung pada pilihan yang sudah dijatuhkan — pilihan-pilihan itu toh sudah mempersempit persoalannya — tetapi ia tak boleh bergantung pada pilihan mana pun di masa depan atau pada solusi submasalah mana pun. Jika aturan Anda perlu mengetahui jawaban sebuah submasalah, Anda sedang menulis program dinamis, bagaimanapun rupanya.

Arah geraknya. Algoritma rakus biasanya bergerak maju dengan cara **top-down**: ia menjatuhkan satu pilihan rakus demi satu pilihan rakus dan menyusutkan tiap contoh kasus persoalan yang diberikan menjadi yang lebih kecil. Pemrograman dinamis, sebaliknya, khasnya bekerja *bottom-up* (atau *top-down* dengan *memoization*, yang dalam hal itu *kodenya* berjalan *top-down* tetapi tiap submasalahnya tetap diselesaikan sebelum pilihan di atasnya dijatuhkan). Inilah perbedaan operasionalnya: **pemrograman dinamis menyelesaikan submasalahnya sebelum**

menjatuhkan pilihan pertama; algoritma rakus menjatuhkan pilihan pertamanya sebelum menyelesaikan satu submasalah pun.

9.5.3 Bahan kedua: substruktur optimal

Bahan yang kedua adalah bahan yang sudah kita kenal dari Bab 7.

> **DEFINISI (dinyatakan ulang dari §7.5.2).** Sebuah persoalan memperlihatkan **substruktur optimal** jika solusi optimal bagi persoalan itu memuat di dalam dirinya solusi-solusi optimal bagi submasalahnya.

Substruktur optimal adalah kunci dalam menaksir keterterapan pemrograman dinamis **dan** algoritma rakus — ia syarat bersama, bukan syarat pembeda. Tetapi cara kita *memakainya* berbeda. Bagi algoritma rakus kita menganggap kita sampai di submasalahnya setelah menjatuhkan pilihan rakus pada persoalan aslinya, lalu kita berargumen bahwa solusi optimal bagi submasalah itu, dipadukan dengan pilihan yang sudah dijatuhkan, menghasilkan solusi optimal bagi persoalan aslinya. Argumen itu secara tersirat memakai **induksi atas submasalahnya** untuk menunjukkan bahwa menjatuhkan pilihan rakus pada setiap langkah menghasilkan solusi optimal — dan persis begitulah Akibat 9.3 dipakai di akhir §9.4.8.

> **KEDUA BAHANNYA SAMA-SAMA DIPERLUKAN, DAN YANG SATU TIDAK MENYIRATKAN YANG LAIN.** Ransel 0-1 di §9.6 punya substruktur optimal tetapi **tidak** punya sifat pilihan rakus — dan itulah sebabnya ia memerlukan sebuah tabel. Persoalan lintasan sederhana terpanjang di §7.5.4 tak punya keduanya. Persoalan yang punya keduanya adalah persoalan yang dapat diselesaikan sebuah algoritma rakus.

9.5.4 Cara mengenali kedua bahannya: resep praktis

Inilah prosedur yang harus Anda ikuti ketika Anda berjumpa dengan persoalan optimasi baru dan menduga kerakusan mungkin berhasil.

1. **Tuliskan tujuannya sebagai satu kalimat yang dimulai dengan "maksimumkan" atau "minimumkan".** Jika Anda tak dapat melakukannya, Anda belum punya persoalan optimasi, dan tak satu pun dari kedua teknik itu berlaku.
2. **Daftarlah setiap aturan lokal yang masuk akal.** Di §9.4.3 ada empat. Jangan berhenti pada yang Anda gemari; jebakan §9.4.3 adalah bahwa dua aturan yang keliru memperoleh jawaban yang benar pada satu contoh kasus.
3. **Cobalah mematahkan tiap aturan pada contoh kasus kecil.** Tulislah *brute force*-nya. Carilah secara tuntas atas contoh kasus berukuran 3, lalu 4, lalu 5. Satu contoh sanggahan membunuh sebuah aturan dalam satu baris; tak ada yang lain yang bisa. Pencarian di **Kode 9.2** menemukan contoh sanggahan berisi tiga aktivitas bagi dua dari empat aturannya.
4. **Bagi tiap aturan yang selamat, cobalah argumen penukarannya.** Ambil sembarang solusi optimal; temukan unsur yang memainkan peran unsur rakusnya; tukarkan; periksa kelayakannya dan periksa bahwa ukuran atau biayanya tidak memburuk. Jika penukarannya merusak kelayakan, aturan itu mungkin keliru dan langkah 3 sekadar belum menemukan contoh sanggahannya.
5. **Buktikan substruktur optimalnya dengan potong-dan-tempel**, sambil memeriksa bahwa penempelannya sah (§7.5.3).
6. **Barulah sesudah itu tulis kodenya.** Dan simpanlah *brute force*-nya: ia adalah uji regresi Anda.

> **MENGAPA LANGKAH 3 SEBELUM LANGKAH 4.** Mencari contoh sanggahan itu murah, mekanis, dan dapat dikerjakan komputer. Mencari bukti itu mahal, menuntut daya cipta, dan

hanya dapat dikerjakan oleh Anda. Belanjakan waktu komputernya lebih dulu. Sudah sangat banyak jam terbuang untuk mencoba membuktikan hal-hal yang ternyata salah.

9.6 Rakus lawan Pemrograman Dinamis: Dua Masalah Ransel

Bagian ini menjajarkan kerakusan dan pemrograman dinamis dengan memakai sepasang persoalan yang begitu mirip sehingga keduanya hanya berbeda pada satu kata — dan satu kata itulah yang menentukan teknik mana yang berhasil.

KEKELIRUAN BAKU. Kedua strategi itu memanfaatkan substruktur optimal, jadi gampang sekali menulis solusi pemrograman dinamis bagi sebuah persoalan padahal solusi rakus sudah memadai atau, sebaliknya, mengira solusi rakus berhasil padahal sesungguhnya pemrograman dinamis diperlukan. Dua masalah ransel adalah pemeragaan terbersih dalam seluruh ilmu algoritma.

9.6.1 Kedua persoalannya

DEFINISI (masalah ransel 0-1). Seorang pencuri yang merampok sebuah toko menemukan n barang. Barang i bernilai v_i rupiah dan berbobot w_i kilogram, dengan v_i dan w_i bilangan bulat positif. Ransel si pencuri memuat paling banyak W kilogram. Barang mana yang harus diambil agar total nilai yang terbawa maksimum? Tiap barang harus diambil **utuh atau ditinggalkan** — karena itu *0-1*: barang i diambil nol kali atau satu kali.

DEFINISI (masalah ransel pecahan). Persis sama, kecuali bahwa barangnya **dapat dibagi**: si pencuri boleh mengambil sembarang pecahan $x_i \in [0,1]$ dari barang i, memperoleh nilai $x_i v_i$ pada bobot $x_i w_i$. Maksimumkan $\sum_i x_i\, v_i$ terhadap kendala $\sum_i x_i\, w_i \leq W$.

Perbedaan fisiknya adalah apakah rampasannya berupa batang emas (0-1: Anda tak dapat menggergaji sebatang menjadi dua tanpa merusaknya — atau, lebih jujurnya, apakah persoalannya *mengatakan* Anda tak dapat) atau serbuk emas (pecahan: sauklah sebanyak yang Anda suka).

Kedua persoalannya punya substruktur optimal, dan argumennya layak dilihat karena keduanya nyaris serupa.

0-1. Tinjaulah muatan paling bernilai yang berbobot paling banyak W. Jika kita membuang barang j dari muatan itu, sisanya pastilah muatan paling bernilai berbobot paling banyak $W - w_j$ yang dapat dibuat dari $n - 1$ barang lainnya. *Bukti:* seandainya ada muatan yang lebih baik bagi kapasitas yang tersusut dan himpunan barang yang tersusut itu, menyulihkannya lalu mengembalikan j akan memberi muatan sah yang lebih bernilai — potong dan tempel, kontradiksi. ■

Pecahan. Tinjaulah muatan paling bernilai yang berbobot paling banyak W. Jika kita membuang bobot ω dari barang j dari muatan itu, sisanya pastilah muatan paling bernilai berbobot paling banyak $W - \omega$ yang dapat dibuat dari $n - 1$ barang lainnya ditambah sisa $w_j - \omega$ kilogram barang j. Argumen potong-dan-tempel yang sama. ■

Jadi substruktur optimal tidak membedakan keduanya. Yang membedakan keduanya adalah sifat pilihan rakus.

9.6.2 Aturan rakusnya, dan contoh kasusnya

Aturan rakus yang wajar bagi salah satu dari keduanya adalah memeringkat barang menurut **nilai per kilogram**:

$$\text{kerapatan}(i) = \frac{v_i}{w_i}, \tag{9.6}$$

lalu mengambil yang terapat lebih dulu, yaitu barang dengan nilai Persamaan (9.6) terbesar. Contoh kasus yang menjadi tumpuan bagian ini adalah contoh baku berisi tiga barang, dan **Tabel 9.5** memaparkannya.

Tabel 9.5 Contoh kasus ransel berisi tiga barang, dengan kapasitas $W = 50$ kg. Barangnya sudah tersusun menurun menurut nilai per kilogram, jadi barang 1 adalah pilihan pertama algoritma rakusnya.

Barang i	Bobot w_i (kg)	Nilai v_i	Kerapatan v_i/w_i
1	10	60	6.00
2	20	100	5.00
3	30	120	4.00

Perhatikan bahwa $w_1 + w_2 + w_3 = 10 + 20 + 30 = 60 > 50 = W$, jadi tidak semuanya muat; sebuah pilihan harus dijatuhkan.

9.6.3 Pecahan: kerakusan berhasil

Jalankan aturannya. Kapasitas tersisa: 50.

- Barang 1 terapat pada 6.00 per kg. Bobotnya $10 \leq 50$, jadi ambil **seluruhnya**. Nilai yang diperoleh 60. Kapasitas tersisa $50 - 10 = 40$.
- Barang 2 berikutnya pada 5.00 per kg. Bobotnya $20 \leq 40$, jadi ambil **seluruhnya**. Nilai yang diperoleh 100. Kapasitas tersisa $40 - 20 = 20$.
- Barang 3 berikutnya pada 4.00 per kg. Bobotnya $30 > 20$, jadi ambil pecahan $x_3 = 20/30 = 2/3$. Nilai yang diperoleh $\frac{2}{3} \times 120 = 80$. Kapasitas tersisa 0.

Total nilainya: $60 + 100 + 80 = 240$. Ranselnya terisi tepat penuh: $10 + 20 + 20 = 50$ kg.

Sekarang buktinya bahwa ini optimal.

> **TEOREMA 9.5.** Bagi masalah ransel pecahan, algoritma rakus — urutkan barangnya menurun tak naik menurut v_i/w_i lalu ambil sebanyak yang diizinkan kapasitas tersisa dari masing-masing secara berurutan — menghasilkan solusi optimal.

Bukti (argumen penukaran). Berilah indeks pada barangnya sehingga $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$.

1. Mula-mula perhatikan bahwa **setiap** solusi optimal mengisi ranselnya sampai penuh, asalkan $\sum_i w_i \geq W$. Sebab jika $\sum_i x_i w_i < W$ dan ada barang yang belum terambil seutuhnya, kita dapat menaikkan pecahan barang itu sedikit dan memperoleh nilai yang positif — jadi solusinya tadi tidak optimal. Karena itu setiap solusi optimal punya $\sum_i x_i w_i = W$, dan demikian pula solusi rakusnya.
2. Misalkan $x = (x_1, \ldots, x_n)$ solusi rakusnya dan $y = (y_1, \ldots, y_n)$ **sembarang** solusi optimal. Jika $x = y$ kita selesai.
3. Jika tidak, misalkan j indeks **terkecil** tempat keduanya berbeda. Kita klaim $y_j < x_j$. Memang, menurut konstruksinya kerakusan mengambil pecahan terbesar dari barang j yang diizinkan kapasitas tersisa sesudah barang $1, \ldots, j-1$; karena y sependapat dengan x pada barang $1, \ldots, j-1$, kapasitas tersisanya pada langkah j sama, sehingga ia tak dapat mengambil barang j lebih banyak daripada yang diambil kerakusan. Karena itu $y_j \leq x_j$, dan karena keduanya berbeda, $y_j < x_j$.

4. Misalkan $\delta = (x_j - y_j)w_j > 0$: bobot barang j yang ditolak y dan diambil kerakusan. Karena y mengisi ranselnya penuh (langkah 1) dan mengambil barang j lebih sedikit, ia pasti mengambil bobot δ itu dari barang berindeks lebih besar daripada j. (Ia tak dapat mengambilnya dari barang $1, \ldots, j-1$, yang padanya y sudah sependapat dengan x.)
5. **Penukarannya.** Bangunlah y' dari y dengan menaikkan y_j sampai ia sama dengan x_j — yaitu menambahkan bobot δ dari barang j — dan membuang total bobot δ dari barang-barang berindeks lebih besar daripada j, dengan cara apa pun yang menjaga setiap $y'_i \in [0,1]$. Hal ini mungkin menurut langkah 4.
6. **Nilainya berubah sebesar**

$$\text{nilai}(y') - \text{nilai}(y) = \delta \cdot \frac{v_j}{w_j} - \sum_{i>j} \delta_i \cdot \frac{v_i}{w_i}, \tag{9.7}$$

dengan $\delta_i \geq 0$ bobot yang dibuang dari barang i dan $\sum_{i>j} \delta_i = \delta$. 7. Karena $i > j$ menyiratkan $v_i/w_i \leq v_j/w_j$,

$$\sum_{i>j} \delta_i \frac{v_i}{w_i} \leq \sum_{i>j} \delta_i \frac{v_j}{w_j} = \delta \cdot \frac{v_j}{w_j}.$$

8. Menyulihkannya ke Persamaan (9.7), $\text{nilai}(y') - \text{nilai}(y) \geq 0$: penukarannya tidak kehilangan nilai. Dan y' layak, sebab total bobotnya tak berubah pada W dan setiap pecahannya tetap di $[0,1]$.
9. Jadi y' juga optimal, dan ia sependapat dengan solusi rakusnya pada barang $1, \ldots, j$ — satu indeks lebih jauh daripada y tadi.
10. Mengulangi langkah 3–9 paling banyak n kali mengubah sembarang solusi optimal menjadi solusi rakusnya tanpa pernah kehilangan nilai. Karena itu solusi rakusnya optimal. ■

> **DI MANA BUKTINYA MEMAKAI KETERBAGIANNYA.** Di langkah 5. Kita perlu menambahkan bobot barang j *tepat* sebesar δ dan membuang *tepat* bobot δ dari barang-barang belakangan. Di dunia yang dapat dibagi kita selalu dapat melakukannya. Di dunia 0-1 kita tak dapat: barang datang dalam bongkahan, dan boleh jadi tak ada bongkahan yang ukurannya pas. Satu kalimat itulah seluruh perbedaan antara kedua persoalannya.

9.6.4 0-1: kerakusan gagal

Jalankan aturan yang sama, tetapi kini barangnya harus diambil utuh. Kapasitas tersisa: 50.

- Barang 1, kerapatan 6.00: bobotnya $10 \leq 50$. Ambil. Nilai 60. Kapasitas tersisa 40.
- Barang 2, kerapatan 5.00: bobotnya $20 \leq 40$. Ambil. Nilai 160 sejauh ini. Kapasitas tersisa 20.
- Barang 3, kerapatan 4.00: bobotnya $30 > 20$. **Tak dapat diambil.** Lewati.

Jawaban kerakusan: barang 1 dan 2, total bobot 30 kg, total nilai **160** — dengan **20 kg kapasitas tertinggal kosong dan tak terpakai**.

Sekarang cacahlah setiap himpunan bagiannya, yang dapat kita lakukan karena jumlahnya hanya $2^3 = 8$. **Tabel 9.6** adalah keluaran terverifikasi **Kode 9.3**.

Tabel 9.6 Setiap himpunan bagian contoh kasus di **Tabel 9.5**, lengkap dengan harganya. Optimumnya {2,3} bernilai 220; aturan rakusnya mengembalikan {1,2} bernilai 160.

Himpunan bagian	Bobot (kg)	Nilai	Muat di 50 kg?
{ }	0	0	ya
{1}	10	60	ya
{2}	20	100	ya
{1,2}	30	160	ya — inilah jawaban rakusnya
{3}	30	120	ya
{1,3}	40	180	ya
{2,3}	50	220	ya — **inilah optimumnya**
{1,2,3}	60	280	tidak

Optimumnya adalah {2,3}: bobot $20 + 30 = 50$, tepat penuh, nilai $100 + 120 = \mathbf{220}$. Kerakusan mengembalikan 160. Kekurangannya adalah

$$220 - 160 = 60, \qquad \frac{60}{220} = 0.2727\ldots \approx 27.3\% \text{ dari optimumnya.}$$

> **MENGAPA KERAKUSAN GAGAL, DENGAN TEPAT.** Aturan rakusnya mengambil barang 1 karena kerapatannya terbaik. Tetapi mengambil barang 1 menghabiskan 10 kg dan, yang lebih buruk, menyisakan *sisa* 40 kg yang ke dalamnya barang 2 dan 3 (berbobot 20 dan 30) tak mungkin keduanya muat. Solusi optimalnya *mengorbankan* barang yang terapat justru agar kapasitasnya terbagi rata. Kerakusan tak dapat melihat hal itu, sebab melihatnya menuntut pandangan atas apa yang sanggup ditampung kapasitas sisanya — dan itu sebuah submasalah, sedangkan aturan rakus dilarang memandang submasalah.

Perhatikan terutama bahwa jawaban kerakusan meninggalkan ranselnya **hanya terisi 60%**. Itulah tanda khas kegagalan rakus pada 0-1: kapasitas tak terpakai yang sebenarnya dapat diisi oleh pilihan lain yang secara satuan lebih buruk.

9.6.5 Seberapa parah, dan seberapa sering?

Satu contoh sanggahan membuktikan aturannya keliru. **Kode 9.3** mengukur seberapa keliru, atas banyak contoh kasus acak, dengan kapasitasnya ditetapkan separuh total bobotnya:

```
=== Stress test: how often is the greedy 0-1 answer wrong? ===
  n =   5: greedy was suboptimal in   4241 of 20000 random instances (21.2%),  mean
value/optimum = 0.9789,  worst = 0.4577
  n =  10: greedy was suboptimal in   7033 of 20000 random instances (35.2%),  mean
value/optimum = 0.9868,  worst = 0.7821
  n =  20: greedy was suboptimal in   9666 of 20000 random instances (48.3%),  mean
value/optimum = 0.9930,  worst = 0.9250

=== Fractional greedy is never wrong: same stress test ===
  n =   5: fractional greedy fell below the 0-1 optimum in 0 of 20000 instances
  n =  10: fractional greedy fell below the 0-1 optimum in 0 of 20000 instances
  n =  20: fractional greedy fell below the 0-1 optimum in 0 of 20000 instances
```

Dua pelajaran bersembunyi di balik angka-angka itu, dan keduanya penting.

Pertama, rakus 0-1 sering keliru. Pada $n = 20$ ia tidak optimal pada hampir separuh dari dua puluh ribu contoh kasus acak. Ini bukan kelainan yang langka.

Kedua — dan inilah yang halus — rakus 0-1 biasanya keliru hanya sedikit. Rata-rata nisbah nilai rakus terhadap nilai optimalnya adalah 0.9789, 0.9868, 0.9930: kerakusan lazimnya meleset satu atau dua persen saja. Justru itulah sebabnya kekeliruan ini bertahan hidup di dalam kode produksi. Ia memberi jawaban yang *tampak* baik-baik saja. Tetapi kasus terburuk pada $n = 5$ adalah 0.4577 — kurang dari **separuh** optimumnya. Jika Anda tak sanggup menanggung hal itu, Anda tak boleh memakai kerakusan.

> **"IA BERHASIL PADA DATA UJI SAYA" BUKANLAH BUKTI.** Algoritma rakus yang keliru dengan galat rata-rata 1% akan lolos dari rangkaian uji sambil lalu mana pun yang Anda tulis. Ia lalu akan kehilangan 54% nilainya pada satu contoh kasus yang justru penting. Inilah alasan praktis mengapa buktinya adalah seluruh pekerjaannya.

9.6.6 Kedua teknik itu, bersebelahan

Tabel 9.7 adalah ringkasan yang perlu dihafalkan.

Tabel 9.7 Rakus lawan pemrograman dinamis. Baris yang menentukan adalah yang kedua: sifat pilihan rakus adalah tuntutan tambahan kerakusan, dan justru itulah yang tak dimiliki ransel 0-1.

	Rakus	Pemrograman dinamis
Memerlukan substruktur optimal	ya	ya
Memerlukan sifat pilihan rakus	**ya**	tidak
Memerlukan submasalah yang bertumpang tindih	tidak	ya (§7.10.1)
Kapan pilihan pertamanya dijatuhkan?	**sebelum** submasalah mana pun diselesaikan	**sesudah** submasalahnya diselesaikan
Ada berapa submasalah sesudah sebuah pilihan?	tepat satu	umumnya beberapa, dengan beberapa calon pilihan
Arahnya	*top-down*, satu pilihan demi satu pilihan	*bottom-up*, atau *top-down* dengan *memoization*
Apakah ia meninjau ulang?	tak pernah	secara tersirat, dengan mengambil maks atau min atas semua pilihan
Biaya khasnya, pemilihan aktivitas	$\Theta(n \lg n)$	waktu $\Theta(n^3)$, ruang $\Theta(n^2)$
Ransel pecahan	**optimal** (Teorema 9.5)	berhasil tetapi tak perlu
Ransel 0-1	**keliru** — sampai 54% di bawah optimum	optimal, $\Theta(nW)$

> **PERTANYAAN DIAGNOSISNYA.** Tanyakanlah: *dapatkah saya menyebut pilihan yang benar tanpa menyelesaikan apa pun?* Bagi ransel pecahan jawabannya ya — ambil barang yang terapat, sebab penukaran terbagi apa pun dengan barang yang kurang rapat hanya dapat kehilangan nilai. Bagi ransel 0-1 jawabannya tidak: apakah barang 1 pantas berada di dalam muatannya bergantung pada bagaimana kapasitas sisanya terbagi di antara barang-barang lain, dan itu sebuah submasalah. Jika jawabannya tidak, bangunlah tabelnya.

9.6.7 Program lengkapnya

Kode 9.3 Dua masalah ransel: rakus menurut kerapatan (optimal bagi persoalan pecahan, keliru bagi persoalan 0-1), optimum 0-1 dengan pemrograman dinamis, *brute force* sebagai pemeriksa, dan uji beban atas 120 000 contoh kasus acak. Kompilasi dengan `g++ -std=c++17 -O2 -o knapsack knapsack.cpp` lalu jalankan `./knapsack`.

```
// knapsack.cpp — dua masalah ransel, bersebelahan.
//   * ransel pecahan, diselesaikan secara rakus menurut nilai per kilogram;
//   * ransel 0-1, tempat aturan rakus yang SAMA itu menjadi keliru;
//   * optimum 0-1 dengan pemrograman dinamis dan dengan brute force, sebagai pemeriksa;
//   * uji beban yang mengukur seberapa sering, dan seberapa jauh, rakus keliru.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <random>
using namespace std;

struct Item { int id, w, v; };

double density(const Item& it) { return (double)it.v / (double)it.w; }

// ---------- ransel pecahan: rakus menurut kerapatan, dan benar ----------
struct FracResult { double value; vector<double> frac; };
FracResult fractionalGreedy(vector<Item> a, int W) {
    int n = (int)a.size();
    vector<int> order(n);
    for (int i = 0; i < n; ++i) order[i] = i;
    sort(order.begin(), order.end(),
         [&](int p, int q) { return density(a[p]) > density(a[q]); });
    FracResult r{ 0.0, vector<double>(n, 0.0) };
    int left = W;
    for (int i : order) {
        if (left <= 0) break;
        if (a[i].w <= left) { r.frac[i] = 1.0; r.value += a[i].v; left -= a[i].w; }
        else { r.frac[i] = (double)left / a[i].w; r.value += r.frac[i] * a[i].v; left = 0;
}
    }
    return r;
}

// ---------- ransel 0-1: aturan rakus yang sama, atas barang utuh ----------
struct IntResult { int value, weight; vector<int> take; };
IntResult zeroOneGreedy(vector<Item> a, int W) {
    int n = (int)a.size();
    vector<int> order(n);
    for (int i = 0; i < n; ++i) order[i] = i;
    sort(order.begin(), order.end(),
         [&](int p, int q) { return density(a[p]) > density(a[q]); });
    IntResult r{ 0, 0, {} };
    int left = W;
    for (int i : order)
```

Kode 9.3 (Lanjutan) *Dua masalah ransel: rakus menurut kerapatan (optimal bagi persoalan pecahan, keliru bagi persoalan 0-1), optimum 0-1 dengan pemrograman dinamis, brute force sebagai pemeriksa, dan uji beban atas 120 000 contoh kasus acak. Kompilasi dengan g++ -std=c++17 -O2 -o knapsack knapsack.cpp lalu jalankan ./knapsack.*

```
        if (a[i].w <= left) { r.take.push_back(i); r.value += a[i].v; r.weight += a[i].w;
left -= a[i].w; }
    return r;
}

// ---------- ransel 0-1: optimum sejatinya, dengan pemrograman dinamis ----------
IntResult zeroOneDP(const vector<Item>& a, int W) {
    int n = (int)a.size();
    vector<vector<int>> best(n + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= n; ++i)
        for (int c = 0; c <= W; ++c) {
            best[i][c] = best[i - 1][c];                          // tinggalkan barang i
            if (a[i - 1].w <= c)
                best[i][c] = max(best[i][c], best[i - 1][c - a[i - 1].w] + a[i - 1].v);
        }
    IntResult r{ best[n][W], 0, {} };
    int c = W;
    for (int i = n; i >= 1; --i)
        if (best[i][c] != best[i - 1][c]) {                       // barang i terambil
            r.take.push_back(i - 1); r.weight += a[i - 1].w; c -= a[i - 1].w;
        }
    reverse(r.take.begin(), r.take.end());
    return r;
}

// ---------- ransel 0-1: brute force, untuk memeriksa pemrograman dinamisnya ----------
int zeroOneBrute(const vector<Item>& a, int W) {
    int n = (int)a.size(), best = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        int w = 0, v = 0;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) { w += a[i].w; v += a[i].v; }
        if (w <= W) best = max(best, v);
    }
    return best;
}

string names(const vector<int>& idx, const vector<Item>& a) {
    string s;
    for (size_t i = 0; i < idx.size(); ++i) { if (i) s += " + "; s += "item " +
to_string(a[idx[i]].id); }
    return s.empty() ? "(nothing)" : s;
}

int main() {
    // Contoh kasus tiga barang dari Subbab 9.6.2.
    vector<Item> a = { {1, 10, 60}, {2, 20, 100}, {3, 30, 120} };
    int W = 50;
```

Kode 9.3 (Lanjutan) *Dua masalah ransel: rakus menurut kerapatan (optimal bagi persoalan pecahan, keliru bagi persoalan 0-1), optimum 0-1 dengan pemrograman dinamis, brute force sebagai pemeriksa, dan uji beban atas 120 000 contoh kasus acak. Kompilasi dengan g++ -std=c++17 -O2 -o knapsack knapsack.cpp lalu jalankan ./knapsack.*

```
    cout << "=== The instance ===\n";
    cout << setw(8) << "item" << setw(10) << "weight" << setw(10) << "value"
         << setw(16) << "value per kg\n";
    for (auto& it : a)
        cout << setw(8) << it.id << setw(10) << it.w << setw(10) << it.v
             << setw(16) << fixed << setprecision(2) << density(it) << "\n";
    cout << "  knapsack capacity W = " << W << " kg\n\n";

    cout << "=== FRACTIONAL knapsack: greedy by value per kilogram ===\n";
    FracResult fr = fractionalGreedy(a, W);
    for (size_t i = 0; i < a.size(); ++i)
        cout << "  item " << a[i].id << ": take " << setprecision(4) << fr.frac[i]
             << " of it  ->  " << setprecision(2) << fr.frac[i] * a[i].w << " kg, "
             << fr.frac[i] * a[i].v << " in value\n";
    cout << "  TOTAL VALUE = " << setprecision(2) << fr.value << "   (this is
optimal)\n\n";

    cout << "=== 0-1 knapsack: the same greedy rule, whole items only ===\n";
    IntResult g = zeroOneGreedy(a, W);
    cout << "  greedy takes  " << names(g.take, a) << "\n";
    cout << "  weight " << g.weight << " kg of " << W << " (so " << W - g.weight
         << " kg of capacity is wasted), value " << g.value << "\n\n";

    cout << "=== 0-1 knapsack: the true optimum ===\n";
    IntResult d = zeroOneDP(a, W);
    cout << "  optimum takes " << names(d.take, a) << "\n";
    cout << "  weight " << d.weight << " kg of " << W << ", value " << d.value << "\n";
    cout << "  brute force over all 2^3 = 8 subsets agrees: " << zeroOneBrute(a, W) <<
"\n";
    cout << "  greedy loses " << d.value - g.value << " units of value ("
         << setprecision(1) << 100.0 * (d.value - g.value) / d.value << "% of the
optimum)\n\n";

    cout << "=== Every subset of the 0-1 instance, priced ===\n";
    cout << setw(22) << "subset" << setw(10) << "weight" << setw(10) << "value"
         << setw(10) << "fits?\n";
    for (int mask = 0; mask < 8; ++mask) {
        int w = 0, v = 0; string s;
        for (int i = 0; i < 3; ++i) if (mask >> i & 1) {
            w += a[i].w; v += a[i].v; if (!s.empty()) s += "+"; s += to_string(a[i].id);
        }
        if (s.empty()) s = "{}";
        cout << setw(22) << s << setw(10) << w << setw(10) << v
             << setw(10) << (w <= W ? "yes" : "no") << "\n";
    }
```

Kode 9.3 (Lanjutan) *Dua masalah ransel: rakus menurut kerapatan (optimal bagi persoalan pecahan, keliru bagi persoalan 0-1), optimum 0-1 dengan pemrograman dinamis, brute force sebagai pemeriksa, dan uji beban atas 120 000 contoh kasus acak. Kompilasi dengan g++ -std=c++17 -O2 -o knapsack knapsack.cpp lalu jalankan ./knapsack.*

```
    cout << "\n=== Stress test: how often is the greedy 0-1 answer wrong? ===\n";
    mt19937 rng(20260919);
    for (int n : {5, 10, 20}) {
        int trials = 20000, wrong = 0; double worst = 0.0, sumRatio = 0.0;
        for (int t = 0; t < trials; ++t) {
            vector<Item> b(n);
            int total = 0;
            for (int i = 0; i < n; ++i) {
                b[i] = { i + 1, 1 + (int)(rng() % 40), 1 + (int)(rng() % 100) };
                total += b[i].w;
            }
            int cap = total / 2;
            int gv = zeroOneGreedy(b, cap).value;
            int ov = zeroOneDP(b, cap).value;
            if (gv < ov) ++wrong;
            double ratio = ov ? (double)gv / ov : 1.0;
            sumRatio += ratio;
            worst = (t == 0) ? ratio : min(worst, ratio);
        }
        cout << "  n = " << setw(3) << n << ": greedy was suboptimal in " << setw(6) <<
wrong
             << " of " << trials << " random instances ("
             << setprecision(1) << 100.0 * wrong / trials << "%),"
             << "  mean value/optimum = " << setprecision(4) << sumRatio / trials
             << ",  worst = " << worst << "\n";
    }

    cout << "\n=== Fractional greedy is never wrong: same stress test ===\n";
    for (int n : {5, 10, 20}) {
        int trials = 20000, wrong = 0;
        for (int t = 0; t < trials; ++t) {
            vector<Item> b(n);
            int total = 0;
            for (int i = 0; i < n; ++i) {
                b[i] = { i + 1, 1 + (int)(rng() % 40), 1 + (int)(rng() % 100) };
                total += b[i].w;
            }
            int cap = total / 2;
            double fv = fractionalGreedy(b, cap).value;
            // optimum pecahan setidaknya sebesar optimum 0-1, dan rakus mencapainya
            if (fv + 1e-9 < zeroOneDP(b, cap).value) ++wrong;
        }
        cout << "  n = " << setw(3) << n << ": fractional greedy fell below the 0-1 optimum
in "
             << wrong << " of " << trials << " instances\n";
    }
    return 0;
}
```

Ketiga bagian pertamanya mencetak persis angka-angka yang kita kerjakan dengan tangan di atas:

```
=== FRACTIONAL knapsack: greedy by value per kilogram ===
  item 1: take 1.0000 of it  ->  10.00 kg, 60.00 in value
  item 2: take 1.0000 of it  ->  20.00 kg, 100.00 in value
  item 3: take 0.6667 of it  ->  20.00 kg, 80.00 in value
  TOTAL VALUE = 240.00   (this is optimal)

=== 0-1 knapsack: the same greedy rule, whole items only ===
  greedy takes  item 1 + item 2
  weight 30 kg of 50 (so 20 kg of capacity is wasted), value 160

=== 0-1 knapsack: the true optimum ===
  optimum takes item 2 + item 3
  weight 50 kg of 50, value 220
  brute force over all 2^3 = 8 subsets agrees: 220
  greedy loses 60 units of value (27.3% of the optimum)
```

Perhatikan akhirnya bahwa optimum pecahannya, 240, benar-benar lebih besar daripada optimum 0-1-nya, 220. Itu bukan kebetulan: mengizinkan pecahan hanya dapat menolong, jadi nilai pecahannya selalu merupakan batas atas bagi nilai 0-1. Batas itulah yang menjadi dasar algoritma *branch-and-bound* yang dipakai untuk menyelesaikan contoh kasus ransel 0-1 berukuran besar dalam praktik [3].

9.7 Kode Huffman

Pemampatan data adalah penerapan klasik metode rakus, dan kode Huffman memampatkan data dengan sangat efektif: penghematan 20% sampai 90% adalah hal yang lazim [4]. Bagian ini membangunnya secara tuntas: persoalannya, contoh berjalannya, pohonnya, algoritmanya, penelusuran dengan tangan, bukti kebenarannya, waktu jalannya, serta penyandi dan pengurai sandi yang benar-benar bekerja.

9.7.1 Persoalannya

Kita punya sebuah berkas yang tersusun dari karakter. Kita ingin menyimpannya dengan bit yang lebih sedikit daripada yang diperlukan cara yang jelas, dan kita ingin dapat memperoleh berkas aslinya kembali dengan persis — pemampatan **nirrugi** (*lossless*).

Cara yang jelas itu adalah **kode panjang tetap**: putuskan berapa bit yang akan dipakai tiap karakter, lalu berilah tiap karakter satu pola bit yang berlainan dengan panjang tersebut. Dengan $|C|$ karakter yang berbeda Anda memerlukan $\lceil \lg|C| \rceil$ bit untuk masing-masing, sebab k bit hanya dapat menamai 2^k hal yang berlainan.

Gagasan yang mengalahkannya adalah **kode panjang berubah**: berilah karakter yang *sering* muncul kata sandi yang pendek, dan karakter yang *jarang* muncul kata sandi yang panjang. Jika "a" menyusun 45% isi berkasnya, membelanjakan satu bit untuk tiap "a" alih-alih tiga bit menghemat dua bit sebanyak empat puluh lima ribu kali.

> **DEFINISI.** Misalkan C sebuah abjad berisi karakter, dan untuk tiap $c \in C$ misalkan $c.freq$ adalah cacah kemunculan c di dalam berkasnya. Sebuah **kode karakter biner** memberikan kepada tiap $c \in C$ sebuah **kata sandi** (*codeword*), yaitu untaian bit yang berhingga. Kodenya disebut **panjang tetap** jika semua kata sandinya sama panjang, dan **panjang berubah** jika tidak.

9.7.2 Contoh berjalan, dengan cacah bitnya dihitung

Gambar 9.7 adalah tabel frekuensi dan kata sandinya, dan itulah contoh kasus yang akan kita pakai di sepanjang sisa bagian ini.

	a	b	c	d	e	f
Frekuensi (dalam ribuan)	45	13	12	16	9	5
Kata kode panjang tetap	000	001	010	011	100	101
Kata kode panjang berubah	0	101	100	111	1101	1100

Gambar 9.7 Sebuah persoalan penyandian karakter. Sebuah berkas berisi 100 000 karakter hanya memuat a–f, dengan frekuensi sebagaimana ditampilkan (dalam ribuan). Kode panjang tetapnya berbiaya 300 000 bit; kode panjang berubah yang ditampilkan berbiaya 224 000.

Mari kita periksa sendiri kedua bilangan itu, sebab hitungannyalah yang menjadi pokoknya.

Abjadnya adalah $C = \{a, b, c, d, e, f\}$, jadi $|C| = 6$, dan frekuensinya (dalam ribuan) adalah 45,13,12,16,9,5. Jumlahnya

$$45 + 13 + 12 + 16 + 9 + 5 = 100,$$

jadi berkasnya berisi 100 ribu $=$100 000 karakter. Bagus — datanya taat asas.

Kode panjang tetapnya. Enam karakter memerlukan $\lceil \lg 6 \rceil$ bit. Nah, $\lg 4 = 2$ dan $\lg 8 = 3$, jadi $\lg 6$ terletak benar-benar di antara 2 dan 3, dan pembulatan ke atasnya adalah 3. Tiga bit memberi delapan pola, enam di antaranya kita pakai: 000,001,010,011,100,101. Setiap satu dari 100 000 karakternya berbiaya 3 bit, jadi berkasnya berbiaya

$$3 \times 100\,000 = 300\,000 \text{ bit.}$$

Kode panjang berubahnya. Tabel pada **Gambar 9.7** memberikan $a = 0$, $b = 101$, $c = 100$, $d = 111$, $e = 1101$, $f = 1100$. Kalikan tiap frekuensinya dengan panjang kata sandinya lalu jumlahkan, sebagaimana dilakukan **Tabel 9.8**.

Tabel 9.8 Kedua kode **Gambar 9.7**, dengan biaya masing-masing dihitung karakter demi karakter. Dalam ribuan bit: 300 bagi kode panjang tetap, 224 bagi yang panjang berubah.

Karakter	Frekuensi (ribuan)	Kata sandi tetap	Bit	Kata sandi berubah	Bit	Frekuensi × panjang
a	45	000	3	0	1	$45 \times 1 = 45$
b	13	001	3	101	3	$13 \times 3 = 39$
c	12	010	3	100	3	$12 \times 3 = 36$
d	16	011	3	111	3	$16 \times 3 = 48$
e	9	100	3	1101	4	$9 \times 4 = 36$
f	5	101	3	1100	4	$5 \times 4 = 20$
total	**100**		**300**			**224**

Menjumlahkan lajur terakhirnya:

$$45 + 39 + 36 + 48 + 36 + 20 = 224 \text{ ribu bit} = 224\,000 \text{ bit.}$$

Penghematannya

$$300\,000 - 224\,000 = 76\,000 \text{ bit}, \qquad \frac{76\,000}{300\,000} = 0.2533\ldots = 25.33\%.$$

Cara lain mengatakan hal yang sama: panjang kata sandi rata-ratanya adalah $224\,000/100\,000 = 2.24$ bit per karakter, melawan 3 bagi kode tetapnya.

DARI MANA PENGHEMATANNYA BERASAL. Pandanglah huruf "a" saja. Ia muncul 45 000 kali dan kata sandinya menyusut dari 3 bit menjadi 1, menghemat $2 \times 45\,000 = 90\,000$ bit hanya dengan dirinya sendiri. Huruf e dan f justru menjadi *lebih panjang* — dari 3 bit menjadi 4 — dengan biaya $1 \times 9\,000 + 1 \times 5\,000 = 14\,000$ bit. Penghematan bersihnya: $90\,000 - 14\,000 = 76\,000$. ✓ Pertukarannya adalah: buatlah yang sering itu murah lalu bayarlah harganya pada yang jarang.

9.7.3 Kode bebas awalan, dan mengapa ia penting

Kode panjang berubah memunculkan persoalan yang tak pernah dipunyai kode panjang tetap: **di mana satu kata sandi berhenti dan yang berikutnya mulai?** Dengan tiga bit masing-masing, Anda mencacah arusnya bertiga-tiga. Dengan panjang yang berubah-ubah tak ada apa pun yang menjadi dasar pencacahan.

Tinjaulah kode yang buruk $a = 0$, $b = 01$, $c = 1$. Sekarang uraikan sandi arus bit `001`. Apakah ia 0 | 0 | 1, yaitu "aac"? Ataukah ia 0 | 01, yaitu "ab"? Kedua pembacaannya sah. Kodenya **taksa** dan karena itu tak berguna: kita sudah memampatkan berkasnya menjadi sesuatu yang tak dapat kita baca kembali.

Kesulitannya adalah bahwa kata sandi bagi a, yaitu `0`, merupakan **awalan** kata sandi bagi b, yaitu `01`. Setelah membaca sebuah `0`, si pengurai sandi tak dapat membedakan apakah ia sudah menyelesaikan sebuah karakter atau sedang berada di tengah-tengah sebuah karakter.

DEFINISI. Sebuah kode disebut **bebas awalan** (*prefix-free*; pustaka yang lebih tua menyebutnya *prefix code*, kode awalan) jika tak ada kata sandi yang menjadi awalan kata sandi mana pun yang lain.

MENGAPA BEBAS AWALAN SUDAH CUKUP. Pada kode bebas awalan, penguraian sandinya tak taksa dan dapat dikerjakan secara rakus, dari kiri ke kanan, tanpa menengok ke depan sama sekali: bacalah bit sampai yang sudah Anda baca cocok dengan sebuah kata sandi, keluarkan karakter itu, lalu mulai lagi. Ia pasti cocok dengan tepat satu kata sandi, sebab seandainya bit yang sudah Anda baca cocok dengan dua kata sandi, yang satu pasti menjadi awalan yang lain. Itulah sebabnya perangkaian kata sandinya tak memerlukan pemisah, tak memerlukan panjang, dan tak memerlukan karakter pelarian.

Periksalah pada kode **Gambar 9.7** dan arus `0` `101` `100` `111` `1101` `1100`, yang dirangkai menjadi

$$0\,101\,100\,111\,1101\,1100 = 010110011111011100.$$

Uraikan dari kiri. `0` — apakah itu sebuah kata sandi? Ya, "a". Keluarkan "a". Berikutnya `1` — sebuah kata sandi? Tidak. `10` — tidak. `101` — ya, "b". Keluarkan "b". Berikutnya `1` — tidak. `10` — tidak. `100` — ya, "c". Dan seterusnya, memulihkan `abcdef` tanpa tebak-tebakan.

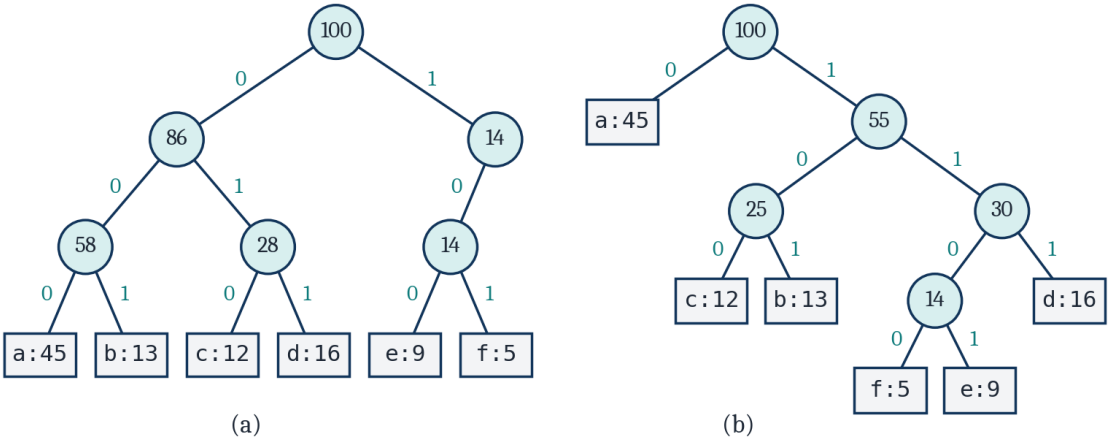
Perlu diketahui — meskipun tak akan kita buktikan — bahwa membatasi diri pada kode bebas awalan sama sekali tidak merugikan kita: bagi setiap kode yang dapat diuraikan secara tunggal, ada kode bebas awalan dengan panjang kata sandi yang persis sama. Jadi kode bebas awalan yang optimal adalah kode yang optimal, titik.

9.7.4 Kode sebagai pohon biner

Kode bebas awalan punya gambaran yang indah. Gambarlah sebuah pohon biner. Berilah label `0` pada setiap sisi kiri dan `1` pada setiap sisi kanan. Letakkan tiap karakter pada sebuah **daun**. Maka kata sandi bagi sebuah karakter adalah runtunan label sisi pada lintasan dari akar turun ke daunnya.

Mengapa kode semacam itu otomatis bebas awalan? Karena sebuah kata sandi menjadi awalan kata sandi lain tepat ketika satu daun menjadi leluhur daun yang lain — sedangkan daun tak punya keturunan. Pohonnya **memaksakan** sifat itu.

Gambar 9.8 menggambarkan kedua kode **Gambar 9.7** sebagai pohon.



Gambar 9.8 Pohon-pohon bagi kedua skema penyandian **Gambar 9.7**. Tiap daun membawa sebuah karakter dan frekuensinya; tiap simpul dalam membawa jumlah frekuensi di dalam upapohonnya. **(a)** kode panjang tetap $a = 000, \ldots, f = 101$; **(b)** kode awalan optimal $a = 0$, $b = 101$, ..., $f = 1100$.

Bacalah pohon (b) lalu pulihkan kata sandinya. Dari akar berlabel 100, sisi kiri (`0`) menuju langsung ke daun `a:45` — jadi $a = 0$, satu bit. Sisi kanan (`1`) menuju simpul dalam 55. Dari 55, kiri (`0`) menuju 25 dan kanan (`1`) menuju 30. Dari 25, kiri menuju `c:12` dan kanan menuju `b:13` — jadi $c = 100$ dan $b = 101$. Dari 30, kiri menuju 14 dan kanan menuju `d:16` — jadi $d = 111$. Dari 14, kiri menuju `f:5` dan kanan menuju `e:9` — jadi $f = 1100$ dan $e = 1101$. Setiap kata sandinya cocok dengan **Gambar 9.7**.

Dua pengamatan struktural tentang pohon (b), keduanya penting kelak.

Pertama, label simpul dalamnya. Tiap simpul dalam berlabel jumlah frekuensi daun-daun di bawahnya. Periksa akarnya: $45 + 55 = 100$ ✓, dan $55 = 25 + 30 = (12 + 13) + (14 + 16) = 25 + 30$ ✓, serta $14 = 5 + 9$ ✓. Jadi label akarnya selalu merupakan cacah seluruh karakter di dalam berkasnya.

Kedua, kedua pohonnya *penuh*. Sebuah pohon biner disebut **penuh** jika tiap simpul dalamnya punya tepat dua anak. Pohon (a) penuh, dan pohon (b) penuh.

> **LEMA 9.6.** Kode bebas awalan yang optimal bagi abjad C dengan $|C| \geq 2$ bersesuaian dengan sebuah pohon biner **penuh**, yang karena itu punya tepat $|C|$ daun dan tepat $|C| - 1$ simpul dalam.

Bukti. Andaikan ada simpul dalam x yang punya tepat satu anak y. Maka hapuslah x lalu pautkan y langsung ke induk x (atau jadikan y akarnya, jika x tadi akarnya). Setiap daun di subpohon y sekarang punya kata sandi yang **lebih pendek** satu bit, dan tak ada kata sandi lain yang berubah. Pohonnya masih merupakan kode bebas awalan yang sah, dan biaya totalnya benar-benar lebih kecil — jadi pohon aslinya tidak optimal. Karena itu pada pohon yang optimal setiap simpul dalam punya dua anak, dan pohonnya penuh. Pohon biner penuh dengan ℓ daun punya tepat $\ell - 1$ simpul dalam (buktinya: masing-masing dari $\ell - 1$ penggabungan yang membangunnya menciptakan satu simpul dalam; atau lewat induksi atas ℓ). ■

9.7.5 Biaya sebuah pohon

Kita sekarang dapat menuliskan dengan tepat hal yang ingin kita minimumkan.

DEFINISI. Bagi pohon T yang mewakili sebuah kode bebas awalan, misalkan $d_T(c)$ menyatakan **kedalaman** daun karakter c di dalam T — yaitu cacah sisi pada lintasan dari akar ke daun tersebut, yang persis merupakan cacah bit dalam kata sandi c. **Biaya** T adalah

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c),$$

(9.8)

yaitu cacah total bit yang diperlukan untuk menyandikan seluruh berkasnya. Pohon yang mencapai $B(T)$ terkecil yang mungkin disebut **kode optimal** bagi C.

Tiap lambangnya: C abjadnya; c berjalan atas karakter-karakternya; $c.freq$ adalah seberapa sering c muncul; $d_T(c)$ adalah berapa bit yang dipunyai kata sandi c pada pohon yang ini; hasil kalinya adalah berapa bit yang dihabiskan seluruh salinan c; jumlah atas c adalah seluruh berkasnya.

Terapkan Persamaan (9.8) pada pohon (b) **Gambar 9.8**: kedalamannya adalah $d(a) = 1$, $d(b) = 3$, $d(c) = 3$, $d(d) = 3$, $d(e) = 4$, $d(f) = 4$, jadi

$$\begin{aligned} B(T) &= 45(1) + 13(3) + 12(3) + 16(3) + 9(4) + 5(4) \\ &= 45 + 39 + 36 + 48 + 36 + 20 = 224, \end{aligned}$$

dalam ribuan bit — 224 000 yang sama dengan **Tabel 9.8**. Dan bagi pohon (a), setiap kedalamannya 3, jadi $B(T) = 3(45 + 13 + 12 + 16 + 9 + 5) = 3(100) = 300$.

RUMUS SETARA YANG MEMBUAT ALGORITMANYA MENJADI GAMBLANG. Ada cara kedua menghitung $B(T)$ yang menjadi kunci bagi segala yang menyusul. Alih-alih menjumlahkan atas daun, **jumlahkanlah label simpul dalamnya**. Bagi pohon (b) label-label itu adalah 100,55,25,30,14, dan

$$100 + 55 + 25 + 30 + 14 = 224.$$

Mengapa ini berhasil? Sebuah karakter c pada kedalaman $d_T(c)$ terletak di bawah tepat $d_T(c)$ simpul dalam — satu untuk tiap sisi pada lintasannya ke akar — dan ia menyumbang $c.freq$ kepada masing-masing label mereka. Jadi menjumlahkan semua label dalamnya mencacah $c.freq$ tepat $d_T(c)$ kali, yaitu $c.freq \cdot d_T(c)$. Menjumlahkannya atas c memberi $B(T)$. Dengan kata lain: **biaya pohonnya adalah total seluruh penggabungan yang membangunnya** — dan itu persis besaran yang diminimumkan algoritma rakusnya, satu penggabungan pada satu waktu.

9.7.6 Algoritmanya

Gambar 9.9 memberikan pseudokodenya. Panjangnya sembilan baris.

```
    HUFFMAN (C)
1   n = |C|
2   Q = C
3   for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN (Q)
6       z.right = y = EXTRACT-MIN (Q)
7       z.freq = x.freq + y.freq
8       INSERT (Q, z)
9   return EXTRACT-MIN (Q)          // kembalikan akar pohonnya
```

Gambar 9.9 Huffman(C) dalam pseudokode. Q adalah antrean prioritas-min yang berkunci frekuensi; gelungnya berjalan $|C| - 1$ kali, tiap kali menggantikan dua benda yang paling jarang muncul dengan simpul baru yang frekuensinya adalah jumlah keduanya.

Baris demi baris:

- **Baris 1**, `n = |C|`. Ada berapa banyak karakternya.
- **Baris 2**, `Q = C`. Bangunlah antrean prioritas-min berisi satu simpul daun per karakter, berkunci frekuensi. (Sebuah *antrean prioritas-min* adalah struktur data Bab 5 yang menyokong "berikan saya yang terkecil" dan "tambahkan yang ini" secara efisien; *heap* biner-min melakukan keduanya dalam $O(\lg n)$.)
- **Baris 3**, `for i = 1 to n-1`. Tepat $n - 1$ iterasi — menurut Lema 9.6, persis sebanyak simpul dalam yang harus kita ciptakan.
- **Baris 4**, `allocate a new node z`. Ini akan menjadi sebuah simpul dalam.
- **Baris 5**, `z.left = x = Extract-Min(Q)`. Keluarkan benda yang paling jarang muncul lalu jadikan ia anak kiri z.
- **Baris 6**, `z.right = y = Extract-Min(Q)`. Keluarkan benda yang (kini) paling jarang muncul lalu jadikan ia anak kanan z.
- **Baris 7**, `z.freq = x.freq + y.freq`. Frekuensi simpul barunya adalah jumlah keduanya.
- **Baris 8**, `Insert(Q, z)`. Kembalikan pohon gabungannya ke dalam antrean, tempat ia bersaing dengan syarat yang sama seperti daun-daunnya.
- **Baris 9**, `return Extract-Min(Q)`. Setelah $n - 1$ penggabungan, antreannya memuat tepat satu benda — akar pohon yang sudah jadi.

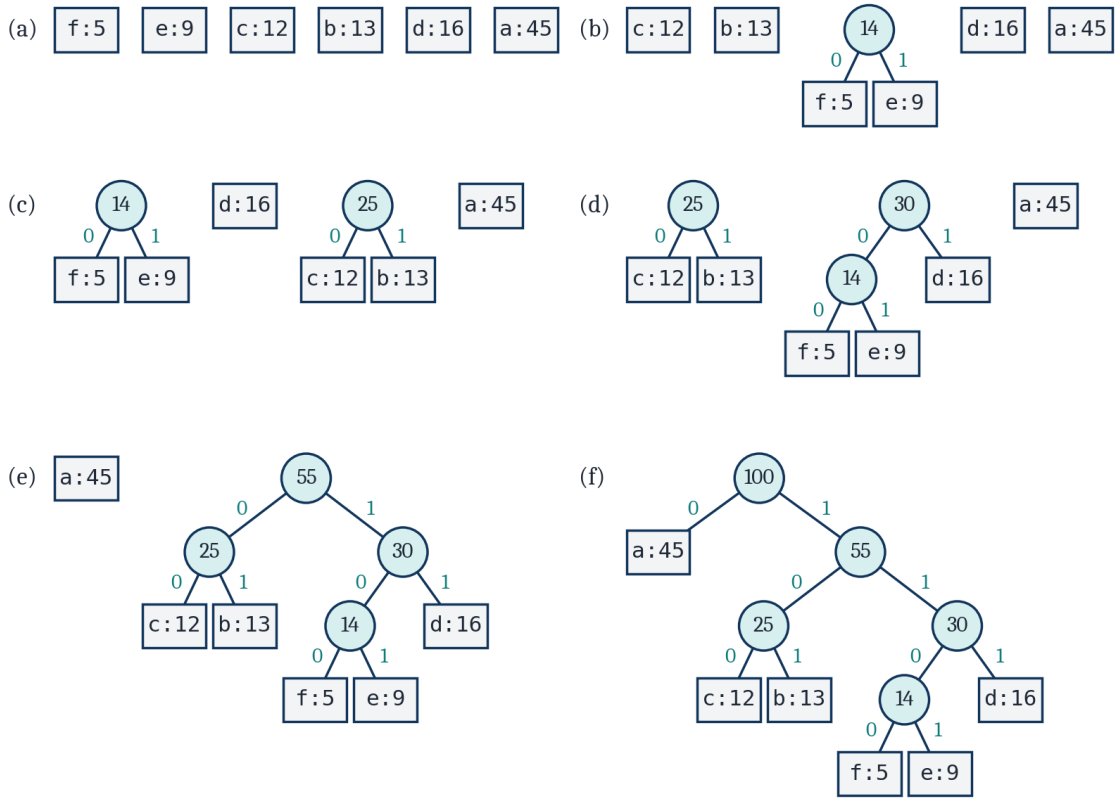
> **PILIHAN RAKUSNYA, DINYATAKAN.** *Gabungkan dua benda yang paling jarang muncul.* Itu seluruh aturannya. Perhatikan bagaimana ia terhubung dengan wawasan §9.7.5: biaya pohonnya adalah jumlah label dalamnya, penggabungan yang hendak kita lakukan menciptakan satu label dalam yang besarnya $x.freq + y.freq$, dan memilih dua yang terkecil membuat label baru itu sekecil mungkin. Rakus di sini berarti: lakukanlah penggabungan termurah yang tersedia, sekarang.

Perhatikan pula *mengapa* penggabungan mendorong benda ke bawah. Kapan pun x dan y menjadi anak sebuah simpul baru, setiap daun di subpohon mereka bertambah satu sisi jaraknya dari akar — jadi kata sandinya bertambah satu bit. Menggabungkan dua benda yang *paling jarang*

muncul karena itu memperpanjang kata sandi karakter-karakter yang paling tak sanggup diperpanjang, sesedikit mungkin kali. Itulah gerak hatinya; §9.7.8 mengubahnya menjadi bukti.

9.7.7 Penelusurannya, langkah demi langkah, pada contoh berjalan

Gambar 9.10 adalah penelusuran lengkapnya.



Gambar 9.10 Penelusuran enam panel atas **Huffman** pada frekuensi **Gambar 9.7**. Tiap panelnya memperlihatkan antreannya terurut menaik menurut frekuensi; pada tiap langkah dua pohon berfrekuensi terendah digabungkan. Daun berupa persegi panjang, simpul dalam berupa lingkaran. **(a)** enam daun awalnya; **(b)**–**(e)** tahap-tahap antaranya; **(f)** pohon akhirnya.

Sekarang penelusuran yang sama dengan kata-kata dan hitungan. Antreannya bermula, terurut, sebagai

f:5 e:9 c:12 b:13 d:16 a:45

yang merupakan panel (a).

Penggabungan 1. Dua yang terkecil adalah f: 5 dan e: 9. Keluarkan keduanya. Simpul dalam baru dengan frekuensi $5 + 9 = 14$, anak kiri f, anak kanan e. Sisipkan. Antreannya kini

c:12 b:13 (14) d:16 a:45

— panel (b). Perhatikan bahwa simpul barunya, pada 14, menyelip *di antara* b: 13 dan d: 16: pohon gabungan dan daun yang belum tersentuh bersaing murni menurut frekuensi.

Penggabungan 2. Dua yang terkecil adalah c: 12 dan b: 13. Simpul baru $12 + 13 = 25$, anak kiri c, anak kanan b. Antrean:

(14) d:16 (25) a:45

— panel (c).

Penggabungan 3. Dua yang terkecil adalah simpul (14) dan d: 16. Simpul baru $14 + 16 = 30$, anak kiri subpohon (14), anak kanan d. Antrean:

$$(25) \quad (30) \quad \text{a:45}$$

— panel (d). Di sini f dan e sudah terdorong ke bawah untuk kedua kalinya; kata sandi mereka sekarang sepanjang dua bit dan akan berakhir pada empat.

Penggabungan 4. Dua yang terkecil adalah (25) dan (30). Simpul baru $25 + 30 = 55$. Antrean:

$$\text{a:45} \quad (55)$$

— panel (e).

Penggabungan 5. Dua yang terakhir: a: 45 dan (55). Simpul baru $45 + 55 = 100$, anak kiri a, anak kanan subpohon (55). Antrean: (100) — panel (f), pohon yang sudah jadi, yang persis merupakan pohon (b) **Gambar 9.8**.

Lima penggabungan bagi enam karakter: $n - 1 = 5$ ✓, sebagaimana dituntut Lema 9.6.

Dan sekarang periksalah wawasan §9.7.5 terhadap penggabungan yang baru saja kita lakukan. Label dalam yang tercipta adalah 14,25,30,55,100, dan

$$14 + 25 + 30 + 55 + 100 = 224.$$

Biaya pohonnya adalah jumlah penggabungannya. **Tabel 9.9** menabelkan seluruh jalannya.

Tabel 9.9 Lima penggabungan **Huffman** atas frekuensi **Gambar 9.7**, dengan antreannya sesudah masing-masing. Jumlah berjalan label barunya adalah biaya $B(T)$ dalam ribuan bit, dan ia berakhir pada 224 — bilangan yang sama yang diperoleh **Tabel 9.8** dengan menjumlahkan atas daun.

Penggabungan	Benda yang dikeluarkan	Label baru	Antrean sesudahnya	Jumlah berjalan
—	—	—	f:5 e:9 c:12 b:13 d:16 a:45	0
1	f:5 dan e:9	14	c:12 b:13 (14) d:16 a:45	14
2	c:12 dan b:13	25	(14) d:16 (25) a:45	39
3	(14) dan d:16	30	(25) (30) a:45	69
4	(25) dan (30)	55	a:45 (55)	124
5	a:45 dan (55)	100	(100)	**224**

Program di **Kode 9.4** mencetak penelusuran ini dan ia sepakat baris demi baris:

```
=== Part 2: HUFFMAN(C), traced ===
  (a) start: f:5  e:9  c:12  b:13  d:16  a:45
  merge f:5 + e:9 = 14
        queue now: c:12  b:13  (14)  d:16  a:45
  merge c:12 + b:13 = 25
        queue now: (14)  d:16  (25)  a:45
  merge (14) + d:16 = 30
        queue now: (25)  (30)  a:45
  merge (25) + (30) = 55
        queue now: a:45  (55)
  merge a:45 + (55) = 100
        queue now: (100)

=== Part 3: the codebook read off the tree ===
    char      freq    codeword     depth freq x depth
       a        45           0         1            45
       b        13         101         3            39
       c        12         100         3            36
       d        16         111         3            48
       e         9        1101         4            36
       f         5        1100         4            20
  B(T) = 224 thousand bits = 224000 bits
  cost() walking the tree agrees: 224
```

Setiap kata sandinya cocok dengan tabel di **Gambar 9.7**.

9.7.8 Kebenaran, bagian satu: lema pilihan rakus

Sekarang kita buktikan algoritmanya optimal. Sebagaimana diajarkan §9.5, itu berarti dua lema: pilihan rakusnya aman, dan apa yang tersisa punya substruktur optimal.

> **LEMA 9.7 (sifat pilihan rakus bagi kode Huffman).** Misalkan C sebuah abjad yang tiap karakternya c punya frekuensi $c.freq$, dan misalkan x dan y dua karakter C yang frekuensinya **terendah**. Maka ada kode bebas awalan yang optimal bagi C yang di dalamnya kata sandi x dan y punya **panjang yang sama** dan **hanya berbeda pada bit terakhirnya** — setara dengan itu, yang di dalamnya x dan y menjadi daun bersaudara berkedalaman maksimum.

Bukti (argumen penukaran, dua kali). Gagasannya: ambil sembarang pohon optimal, lalu tukarkan x dan y masuk ke dua posisi bersaudara yang terdalam, dengan menunjukkan bahwa tiap penukarannya tak mungkin menaikkan biayanya.

1. Misalkan T **sembarang** pohon optimal bagi C. Menurut Lema 9.6, T penuh.
2. Misalkan a dan b dua daun yang merupakan **saudara berkedalaman maksimum** di T. Pasangan semacam itu ada: ambil sembarang daun terdalam; induknya adalah simpul dalam, jadi menurut kepenuhannya ia punya anak yang lain; anak itu pun sebuah daun (sebab jika tidak, ia akan punya keturunan yang lebih dalam daripada daun terdalamnya). Tulislah d bagi kedalaman bersama mereka, yang merupakan tinggi T.
3. Tanpa kehilangan keumuman, andaikan $x.freq \leq y.freq$ dan $a.freq \leq b.freq$. (Jika tidak, tukar namanya.)
4. Karena x dan y punya dua frekuensi terendah di C, dan a, b adalah karakter C, kita punya

$$x.freq \leq a.freq \qquad \text{dan} \qquad y.freq \leq b.freq.$$

(Untuk yang pertama: x punya frekuensi terendah di antara semuanya, jadi ia $\leq$ segalanya. Untuk yang kedua: y punya yang terendah kedua, jadi $y.freq \leq$ frekuensi karakter mana pun selain x; dan $b \neq x$ sebab andaikan b adalah x maka, karena $a.freq \leq b.freq = x.freq \leq a.freq$, kita akan punya $a.freq = x.freq$ dan dapat menamai ulang a sebagai x.) 5. **Penukaran pertama.** Bangunlah T' dari T dengan **menukar posisi** x **dan** a — yaitu mempertukarkan kedua daunnya, sambil menjaga tiap frekuensi tetap melekat pada karakternya. Hitunglah perubahan biayanya. Hanya kedua karakter yang ditukar itu yang berubah kedalaman, jadi

$$\begin{aligned} &B(T) - B(T') \\ &= [x.freq \cdot d_T(x) + a.freq \cdot d_T(a)] - [x.freq \cdot d_{T'}(x) + a.freq \\ &\qquad \cdot d_{T'}(a)]. \end{aligned}$$

6. Tetapi $d_{T'}(x) = d_T(a)$ dan $d_{T'}(a) = d_T(x)$, sebab keduanya sekadar bertukar tempat. Menyulihkan dan mengumpulkan sukunya,

$$\begin{aligned} &B(T) - B(T') \\ &= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x), \end{aligned}$$

dan dengan mengelompokkan suku-suku $a.freq$ dan suku-suku $x.freq$, bentuk ini terfaktorkan menjadi

$$B(T) - B(T') = (a.freq - x.freq)(d_T(a) - d_T(x)). \tag{9.9}$$

7. Sekarang periksalah kedua faktor Persamaan (9.9). Yang pertama, $a.freq - x.freq$, bernilai ≥ 0 menurut langkah 4. Yang kedua, $d_T(a) - d_T(x)$, bernilai ≥ 0 sebab a adalah daun berkedalaman **maksimum**, sehingga tak ada daun — termasuk x — yang lebih dalam. Hasil kali dua bilangan taknegatif bernilai taknegatif, jadi

$$B(T) - B(T') \geq 0, \qquad \text{yaitu,} \qquad B(T') \leq B(T).$$

8. Karena T optimal, $B(T') \geq B(T)$ pula; karena itu $B(T') = B(T)$ dan T' juga optimal. Penukarannya tidak merugikan kita sedikit pun.
9. **Penukaran kedua.** Bangunlah T'' dari T' dengan menukar y dan b. Perhitungan yang serupa, memakai $y.freq \leq b.freq$ dan kenyataan bahwa b masih berkedalaman maksimum di T' (menukar x dan a tidak mengubah *himpunan* kedalamannya, hanya karakter mana yang duduk di mana), memberi $B(T'') \leq B(T')$, dan karena itu $B(T'') = B(T') = B(T)$: T'' optimal.
10. Di dalam T'', karakter x dan y menempati posisi yang tadinya ditempati a dan b di T — daun bersaudara berkedalaman maksimum. Jadi kata sandi mereka sama panjang, yaitu d, dan hanya berbeda pada bit terakhirnya. ■

BACALAH LANGKAH 7 SEKALI LAGI. Ia seluruh buktinya dalam satu baris. *Karakter berfrekuensi rendah dan posisi yang dalam memang berjodoh.* Jika karakter yang murah itu belum berada di posisi yang dalam, menukar keduanya mengalikan selisih frekuensi yang taknegatif dengan selisih kedalaman yang taknegatif, dan biayanya hanya bisa turun. Itulah sebabnya menggabungkan dua lambang yang paling jarang lebih dulu — dan karenanya mendorong mereka ke dasar pohonnya — itu aman.

9.7.9 Kebenaran, bagian dua: lema substruktur optimal

LEMA 9.8 (substruktur optimal bagi kode Huffman). Misalkan C sebuah abjad, misalkan $x, y \in C$ karakter berfrekuensi minimum, dan bentuklah abjad baru

$$C' = (C \setminus \{x, y\}) \cup \{z\},$$

dengan z karakter baru yang $z.freq = x.freq + y.freq$. Misalkan T' **sembarang** pohon optimal bagi C'. Bentuklah T dari T' dengan menggantikan daun bagi z dengan sebuah simpul dalam yang beranak x dan y. Maka T adalah pohon optimal bagi C.

Bukti. Mula-mula kita hubungkan kedua biayanya; lalu kita berargumen lewat pengontradiksian.

1. **Hubungan biayanya.** Tiap karakter $c \in C \setminus \{x, y\}$ adalah daun pada kedua pohonnya pada kedalaman yang sama, jadi $d_T(c) = d_{T'}(c)$ dan c menyumbang secara identik kepada kedua biayanya.
2. Karakter x dan y duduk satu aras di bawah tempat z tadi duduk, jadi $d_T(x) = d_T(y) = d_{T'}(z) + 1$. Sumbangan gabungan mereka kepada $B(T)$ adalah

$$x.freq \cdot d_T(x) + y.freq \cdot d_T(y) = (x.freq + y.freq)(d_{T'}(z) + 1)$$

$$= z.freq \cdot d_{T'}(z) + (x.freq + y.freq).$$

3. Suku pertamanya, $z.freq \cdot d_{T'}(z)$, persis merupakan sumbangan z kepada $B(T')$. Jadi, dengan mengurangkannya,

$$B(T) = B(T') + x.freq + y.freq. \tag{9.10}$$

Dengan kata lain: **melipat pasangan** $\{x, y\}$ **menjadi satu lambang** z **menurunkan biayanya tepat sebesar** $x.freq + y.freq$**, apa pun pohonnya.** Inilah versi formal dari "biayanya adalah jumlah penggabungannya" pada §9.7.5. 4. **Pengontradiksiannya.** Andaikan T **tidak** optimal bagi C. Maka ada pohon T'' dengan $B(T'') < B(T)$. 5. Menurut Lema 9.7 kita boleh menganggap, tanpa kehilangan keumuman, bahwa T'' punya x dan y sebagai daun bersaudara — sebab Lema 9.7 menjamin adanya pohon optimal dengan sifat itu, dan T'' dapat diambil sebagai salah satu yang berbiaya minimum, karenanya optimal, karenanya (setelah penukaran-penukaran Lema 9.7, yang tidak mengubah biayanya) berbentuk demikian. 6. Sekarang tempuhlah arah sebaliknya: bentuklah T''' dari T'' dengan menghapus daun x dan y lalu mengubah induk mereka menjadi sebuah daun berlabel z dengan $z.freq = x.freq + y.freq$. Maka T''' adalah pohon yang sah bagi abjad C', dan menurut perhitungan yang persis sama dengan langkah 1–3,

$$B(T''') = B(T'') - x.freq - y.freq.$$

7. Menyulihkan pengandaian $B(T'') < B(T)$ lalu Persamaan (9.10),

$$B(T''') = B(T'') - x.freq - y.freq < B(T) - x.freq - y.freq = B(T').$$

8. Jadi T''' adalah pohon bagi C' yang biayanya benar-benar lebih kecil daripada T' — bertentangan dengan anggapan bahwa T' optimal bagi C'.
9. Karena itu T'' semacam itu tak ada, dan T optimal bagi C. ■

9.7.10 Teoremanya

TEOREMA 9.9. Prosedur **Huffman** pada **Gambar 9.9** menghasilkan kode bebas awalan yang optimal.

Bukti. Lewat induksi atas $n = |C|$.

Kasus dasar, $n = 2$. Prosedurnya melakukan satu penggabungan, menghasilkan pohon dengan dua daun pada kedalaman 1, jadi kedua kata sandinya sepanjang satu bit. Tak ada kode bebas

awalan bagi dua karakter yang dapat berbuat lebih baik daripada satu bit masing-masing, sebab kode yang punya kata sandi berpanjang nol tidaklah bebas awalan ketika karakternya ada dua. Jadi keluarannya optimal.

Langkah induktifnya. Anggaplah prosedurnya optimal pada setiap abjad berukuran $n-1$, dan misalkan $|C| = n \geq 3$. Baris 5–7 pada iterasi pertamanya mengeluarkan dua karakter x, y berfrekuensi minimum lalu menggantikannya dengan simpul z yang $z.freq = x.freq + y.freq$. Antreannya sekarang memuat persis abjad C' pada Lema 9.8, berukuran $n-1$, dan $n-2$ iterasi sisanya persis merupakan satu jalannya **Huffman** atas C'. Menurut hipotesis induktifnya, jalannya itu mengembalikan pohon optimal T' bagi C'. Keluaran prosedur selengkapnya adalah T' dengan daun z digantikan simpul dalam yang beranak x dan y — yang persis merupakan pohon T pada Lema 9.8. Lema 9.8 mengatakan T optimal bagi C. Lema 9.7-lah yang membuat hipotesis Lema 9.8 sah: ia menjamin bahwa memaksa x dan y bersaudara tidak menyingkirkan setiap pohon optimal. ■

9.7.11 Waktu jalannya

Hitunglah operasi pada **Gambar 9.9**.

- **Baris 2** membangun antrean prioritas berisi n unsur. Dengan **Build-Min-Heap** dari Bab 5 ini $O(n)$.
- **Baris 3** menjalankan gelungnya $n-1$ kali.
- **Baris 5 dan 6** masing-masing melakukan satu `Extract-Min`, berbiaya $O(\lg n)$ pada *heap* biner.
- **Baris 8** melakukan satu `Insert`, juga $O(\lg n)$.
- **Baris 4 dan 7** berbiaya $O(1)$.

Jadi gelungnya berbiaya $(n-1) \times O(\lg n) = O(n \lg n)$, dan pembangunannya berbiaya $O(n)$.

> **TEOREMA 9.10. Huffman** berjalan dalam waktu $O(n \lg n)$ pada abjad berisi n karakter, dengan memakai *heap* biner-min sebagai Q.

Dua penghalusan yang layak diketahui. Pertama, jika karakternya **sudah terurut menurut frekuensi**, seluruhnya dapat dikerjakan dalam waktu $O(n)$ dengan dua antrean alih-alih sebuah *heap* — satu memuat daun yang sudah terurut, satu memuat simpul dalam dalam urutan penciptaannya (yang otomatis tak menurun) — dan minimumnya selalu berada di depan salah satu dari keduanya. Itulah Soal Latihan 9.12. Kedua, antrean prioritas yang lebih cerdik tidak menolong secara umum: bila frekuensinya hanya dapat dibandingkan tanpa dimanfaatkan dengan cara lain, membangun kode optimal melibatkan kerja mengurutkan frekuensi itu, dan $\Theta(n \lg n)$ adalah yang terbaik yang dapat dicapai. Antrean prioritas bilangan bulat seperti pohon van Emde Boas hanya menolong bila frekuensinya bilangan bulat yang kecil.

9.7.12 Penyandi dan pengurai sandi lengkapnya

Kode 9.4 adalah pemampat yang benar-benar bekerja. Ia membangun pohonnya, mencetak penelusuran dan buku sandinya, menghitung $B(T)$ dengan dua cara sebagai pemeriksaan silang, menyandikan teks sungguhan menjadi arus bit yang benar-benar dipaket ke dalam bita, menguraikannya kembali, dan memastikan hasilnya identik dengan masukannya. Ia juga menangani kasus canggung satu lambang, yang tanpanya pohonnya akan punya kata sandi berpanjang nol.

Kode 9.4 Penyandi dan pengurai sandi Huffman yang lengkap: pembangunan yang ditelusuri atas frekuensi **Gambar 9.7**, buku sandinya, $B(T)$ yang dihitung dua cara yang saling bebas, perjalanan bolak-balik terpaket-bit yang sungguhan atas teks biasa, kasus tepi satu lambang, dan satu

jalannya pengukuran waktu. Kompilasi dengan `g++ -std=c++17 -O2 -o huffman huffman.cpp` lalu jalankan `./huffman`.

```
// huffman.cpp — penyandi dan pengurai sandi Huffman yang lengkap.
//   * membangun pohonnya untuk tabel frekuensi Subbab 9.7.2, mencetak antreannya
//     pada tiap penggabungan agar penelusurannya dapat dicocokkan dengan gambarnya;
//   * mencetak tabel kata sandi dan biaya B(T);
//   * membandingkannya dengan kode panjang tetap;
//   * lalu menyandikan dan menguraikan satu blok teks sungguhan, bit demi bit, serta
//     memeriksa bahwa yang keluar persis sama dengan yang masuk.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <queue>
#include <map>
#include <algorithm>
#include <cstdint>
#include <cmath>
#include <chrono>
#include <random>
using namespace std;

// ------------------------------------------------------------ pohonnya ---
struct Node {
    long long freq;
    char ch;            // bermakna hanya pada daun
    bool leaf;
    int serial;         // urutan pembuatan, hanya untuk memutus seri secara terulangkan
    Node *left, *right;
    Node(long long f, char c) : freq(f), ch(c), leaf(true),  serial(0), left(nullptr),
right(nullptr) {}
    Node(Node* l, Node* r)    : freq(l->freq + r->freq), ch(0), leaf(false), serial(0),
left(l), right(r) {}
};

// Antrean prioritasnya adalah antrean-MIN atas frekuensi. Seri dimenangkan simpul
// yang dibuat lebih dahulu, sehingga seluruh pembangunannya deterministik.
struct Later {
    bool operator()(const Node* a, const Node* b) const {
        if (a->freq != b->freq) return a->freq > b->freq;
        return a->serial > b->serial;
    }
};

string label(const Node* n) {
    if (n->leaf) return string(1, n->ch) + ":" + to_string(n->freq);
    return "(" + to_string(n->freq) + ")";
}
```

Kode 9.4 (Lanjutan) *Penyandi dan pengurai sandi Huffman yang lengkap: pembangunan yang ditelusuri atas frekuensi Gambar 9.7, buku sandinya, B(T) yang dihitung dua cara yang saling bebas, perjalanan bolak-balik terpaket-bit yang sungguhan atas teks biasa, kasus tepi satu lambang, dan satu jalannya pengukuran waktu. Kompilasi dengan g++ -std=c++17 -O2 -o huffman huffman.cpp lalu jalankan ./huffman.*

```
// Cetak seluruh antreannya dalam urutan menaik, tanpa mengusiknya.
void showQueue(priority_queue<Node*, vector<Node*>, Later> q, const string& tag) {
    cout << "  " << tag << " ";
    bool first = true;
    while (!q.empty()) { if (!first) cout << "  "; cout << label(q.top()); q.pop(); first =
false; }
    cout << "\n";
}

// HUFFMAN(C) — algoritma buku teks, dengan antreannya dicetak pada tiap langkah.
Node* huffman(const vector<pair<char, long long>>& C, bool trace) {
    priority_queue<Node*, vector<Node*>, Later> Q;
    int serial = 0;
    vector<pair<char, long long>> sorted = C;
    sort(sorted.begin(), sorted.end(),
         [](const pair<char, long long>& a, const pair<char, long long>& b) { return
a.second < b.second; });
    for (auto& p : sorted) { Node* n = new Node(p.second, p.first); n->serial = serial++;
Q.push(n); }
    if (Q.empty()) return nullptr;
    if (Q.size() == 1) {                 // satu lambang berbeda saja: beri ia satu bit
        Node* only = Q.top(); Q.pop();
        Node* root = new Node(only, new Node(0, '\0'));
        root->serial = serial++;
        return root;
    }
    int n = (int)Q.size();
    if (trace) showQueue(Q, "(a) start:");
    for (int i = 1; i <= n - 1; ++i) {
        Node* x = Q.top(); Q.pop();      // x = EXTRACT-MIN(Q)
        Node* y = Q.top(); Q.pop();      // y = EXTRACT-MIN(Q)
        Node* z = new Node(x, y);        // z.left = x, z.right = y, z.freq = x.freq +
y.freq
        z->serial = serial++;
        Q.push(z);                       // INSERT(Q, z)
        if (trace) {
            cout << "  merge " << label(x) << " + " << label(y)
                 << " = " << z->freq << "\n";
            showQueue(Q, "      queue now:");
        }
    }
    return Q.top();
}
```

Kode 9.4 (Lanjutan) *Penyandi dan pengurai sandi Huffman yang lengkap: pembangunan yang ditelusuri atas frekuensi Gambar 9.7, buku sandinya, B(T) yang dihitung dua cara yang saling bebas, perjalanan bolak-balik terpaket-bit yang sungguhan atas teks biasa, kasus tepi satu lambang, dan satu jalannya pengukuran waktu. Kompilasi dengan g++ -std=c++17 -O2 -o huffman huffman.cpp lalu jalankan ./huffman.*

```
// ------------------------------------------------------------ buku sandinya --
void collect(const Node* n, string code, map<char, string>& out) {
    if (!n) return;
    if (n->leaf) { out[n->ch] = code.empty() ? "0" : code; return; }
    collect(n->left,  code + "0", out);
    collect(n->right, code + "1", out);
}

// B(T) = jumlah atas karakter c dari c.freq * d_T(c)
long long cost(const Node* n, int depth = 0) {
    if (!n) return 0;
    if (n->leaf) return n->freq * depth;
    return cost(n->left, depth + 1) + cost(n->right, depth + 1);
}

// ------------------------------------------ penyandian / penguraian sandi ---
// Sandikan menjadi arus bit sungguhan: bita berikut cacah bit yang sahih.
struct BitStream { vector<uint8_t> bytes; size_t nbits = 0; };

void pushBit(BitStream& bs, int bit) {
    if (bs.nbits % 8 == 0) bs.bytes.push_back(0);
    if (bit) bs.bytes.back() |= (uint8_t)(1u << (7 - (bs.nbits % 8)));
    ++bs.nbits;
}

BitStream encode(const string& text, const map<char, string>& code) {
    BitStream bs;
    for (char c : text) {
        auto it = code.find(c);
        if (it == code.end()) { cerr << "symbol not in codebook\n"; exit(1); }
        for (char b : it->second) pushBit(bs, b == '1');
    }
    return bs;
}

// Penguraian sandi menyusuri pohonnya: 0 ke kiri, 1 ke kanan; begitu sampai di daun,
// keluarkan karakternya lalu lompat kembali ke akar. Tak perlu awalan panjang --
// persis itulah yang kita beli dengan sifat "bebas awalan".
string decode(const BitStream& bs, const Node* root, size_t nsymbols) {
    string out;
    const Node* cur = root;
    if (root && root->leaf) { return string(nsymbols, root->ch); }
    for (size_t i = 0; i < bs.nbits && out.size() < nsymbols; ++i) {
        int bit = (bs.bytes[i / 8] >> (7 - (i % 8))) & 1;
        cur = bit ? cur->right : cur->left;
        if (cur->leaf) { out += cur->ch; cur = root; }
```

Kode 9.4 (Lanjutan) *Penyandi dan pengurai sandi Huffman yang lengkap: pembangunan yang ditelusuri atas frekuensi Gambar 9.7, buku sandinya, B(T) yang dihitung dua cara yang saling bebas, perjalanan bolak-balik terpaket-bit yang sungguhan atas teks biasa, kasus tepi satu lambang, dan satu jalannya pengukuran waktu. Kompilasi dengan g++ -std=c++17 -O2 -o huffman huffman.cpp lalu jalankan ./huffman.*

```
    }
    return out;
}

int main() {
    cout << "=== Part 1: the frequency table of the running example ===\n";
    // frekuensi dalam ribuan; berkasnya memuat 100 000 karakter
    vector<pair<char, long long>> C = {
        {'a', 45}, {'b', 13}, {'c', 12}, {'d', 16}, {'e', 9}, {'f', 5}
    };
    long long total = 0; for (auto& p : C) total += p.second;
    cout << "  characters: ";
    for (auto& p : C) cout << p.first << ":" << p.second << "  ";
    cout << "\n  total = " << total << " thousand characters\n\n";

    cout << "=== Part 2: HUFFMAN(C), traced ===\n";
    Node* root = huffman(C, true);
    cout << "\n";

    cout << "=== Part 3: the codebook read off the tree ===\n";
    map<char, string> code;
    collect(root, "", code);
    cout << setw(8) << "char" << setw(10) << "freq" << setw(12) << "codeword"
         << setw(10) << "depth" << setw(14) << "freq x depth\n";
    long long B = 0;
    for (auto& p : C) {
        int d = (int)code[p.first].size();
        cout << setw(8) << p.first << setw(10) << p.second << setw(12) << code[p.first]
             << setw(10) << d << setw(14) << p.second * d << "\n";
        B += p.second * d;
    }
    cout << "  B(T) = " << B << " thousand bits = " << B * 1000 << " bits\n";
    cout << "  cost() walking the tree agrees: " << cost(root) << "\n\n";

    cout << "=== Part 4: fixed-length against Huffman ===\n";
    int fixedBits = (int)ceil(log2((double)C.size()));
    cout << "  " << C.size() << " distinct characters need ceil(lg " << C.size()
         << ") = " << fixedBits << " bits each\n";
    cout << "  fixed-length total  = " << fixedBits << " x " << total * 1000
         << " = " << fixedBits * total * 1000 << " bits\n";
    cout << "  Huffman total       = " << B * 1000 << " bits\n";
    cout << "  saving              = " << (fixedBits * total - B) * 1000 << " bits = "
         << fixed << setprecision(2)
         << 100.0 * (fixedBits * total - B) / (fixedBits * total) << "%\n";
    cout << "  average codeword length = " << (double)B / total << " bits per
character\n\n";
```

Kode 9.4 (Lanjutan) *Penyandi dan pengurai sandi Huffman yang lengkap: pembangunan yang ditelusuri atas frekuensi Gambar 9.7, buku sandinya, B(T) yang dihitung dua cara yang saling bebas, perjalanan bolak-balik terpaket-bit yang sungguhan atas teks biasa, kasus tepi satu lambang, dan satu jalannya pengukuran waktu. Kompilasi dengan g++ -std=c++17 -O2 -o huffman huffman.cpp lalu jalankan ./huffman.*

```
    cout << "=== Part 5: encode and decode real text ===\n";
    string text =
        "abracadabra abracadabra a laundry in surabaya washes bedsheets "
        "and towels and shirts and the dryer never stops running all day";
    map<char, long long> tally;
    for (char c : text) ++tally[c];
    vector<pair<char, long long>> C2(tally.begin(), tally.end());
    Node* root2 = huffman(C2, false);
    map<char, string> code2;
    collect(root2, "", code2);

    cout << "  input length      : " << text.size() << " characters\n";
    cout << "  distinct symbols  : " << C2.size() << "\n";
    cout << "  codebook (symbol -> codeword), shortest first:\n";
    vector<pair<char, string>> cb(code2.begin(), code2.end());
    sort(cb.begin(), cb.end(), [](const pair<char, string>& a, const pair<char, string>& b)
{
        if (a.second.size() != b.second.size()) return a.second.size() < b.second.size();
        return a.first < b.first;
    });
    int perLine = 0;
    cout << "    ";
    for (auto& p : cb) {
        cout << " '" << (p.first == ' ' ? '_' : p.first) << "'=" << p.second;
        if (++perLine % 6 == 0) cout << "\n    ";
    }
    cout << "\n";

    BitStream bs = encode(text, code2);
    string back = decode(bs, root2, text.size());
    cout << "  fixed-length cost : " << text.size() * 8 << " bits (8 bits per char)\n";
    cout << "  Huffman cost      : " << bs.nbits << " bits in "
         << bs.bytes.size() << " bytes\n";
    cout << "  compression       : " << setprecision(2)
         << 100.0 * (1.0 - (double)bs.nbits / (text.size() * 8)) << "% smaller\n";
    cout << "  round trip        : " << (back == text ? "IDENTICAL" : "DIFFERENT") << "\n";
    cout << "  first 64 bits     : ";
    for (size_t i = 0; i < 64 && i < bs.nbits; ++i)
        cout << (((bs.bytes[i / 8] >> (7 - (i % 8))) & 1) ? '1' : '0');
    cout << "\n  decoded prefix    : \"" << back.substr(0, 40) << "...\"\n\n";
```

Kode 9.4 (Lanjutan) *Penyandi dan pengurai sandi Huffman yang lengkap: pembangunan yang ditelusuri atas frekuensi Gambar 9.7, buku sandinya, B(T) yang dihitung dua cara yang saling bebas, perjalanan bolak-balik terpaket-bit yang sungguhan atas teks biasa, kasus tepi satu lambang, dan satu jalannya pengukuran waktu. Kompilasi dengan g++ -std=c++17 -O2 -o huffman huffman.cpp lalu jalankan ./huffman.*

```
    cout << "=== Part 6: the one-symbol edge case ===\n";
    string flat(1000, 'z');
    map<char, long long> t3; for (char c : flat) ++t3[c];
    vector<pair<char, long long>> C3(t3.begin(), t3.end());
    Node* root3 = huffman(C3, false);
    map<char, string> code3; collect(root3, "", code3);
    BitStream bs3 = encode(flat, code3);
    cout << "  1000 copies of 'z': codeword \"" << code3['z'] << "\", "
         << bs3.nbits << " bits, round trip "
         << (decode(bs3, root3, flat.size()) == flat ? "IDENTICAL" : "DIFFERENT") <<
"\n\n";

    cout << "=== Part 7: the running time is O(n lg n) ===\n";
    cout << setw(12) << "n symbols" << setw(16) << "build (ms)"
         << setw(18) << "t / (n lg n)\n";
    mt19937 rng(20260919);
    for (int n = 4000; n <= 512000; n *= 2) {
        vector<pair<char, long long>> big(n);
        for (int i = 0; i < n; ++i) big[i] = { (char)(i % 128), 1 + (long long)(rng() %
1000000) };
        auto t0 = chrono::steady_clock::now();
        Node* r = huffman(big, false);
        auto t1 = chrono::steady_clock::now();
        double ms = chrono::duration<double, milli>(t1 - t0).count();
        double nlgn = (double)n * log2((double)n);
        cout << setw(12) << n << setw(16) << setprecision(3) << ms
             << setw(18) << setprecision(5) << ms * 1e6 / nlgn
             << "   (root freq " << r->freq << ")\n";
    }
    return 0;
}
```

Bagian 4, 5, dan 6 mencetak:

```
=== Part 4: fixed-length against Huffman ===
  6 distinct characters need ceil(lg 6) = 3 bits each
  fixed-length total  = 3 x 100000 = 300000 bits
  Huffman total       = 224000 bits
  saving              = 76000 bits = 25.33%
  average codeword length = 2.24 bits per character

=== Part 5: encode and decode real text ===
  input length      : 126 characters
  distinct symbols  : 20
  codebook (symbol -> codeword), shortest first:
    '_'=101 'a'=111 'r'=001 'b'=0100 'd'=1100 'e'=1001
    'n'=1000 's'=1101 't'=0000 'h'=01101 'i'=00011 'l'=01110
    'u'=01010 'y'=01100 'c'=010111 'g'=000101 'o'=011111 'p'=010110
    'v'=000100 'w'=011110
  fixed-length cost : 1008 bits (8 bits per char)
  Huffman cost      : 489 bits in 62 bytes
  compression       : 51.49% smaller
  round trip        : IDENTICAL
  first 64 bits     : 1110100001111010111111100111010000111110111101000011110101111 11
  decoded prefix    : "abracadabra abracadabra a laundry in sur..."

=== Part 6: the one-symbol edge case ===
  1000 copies of 'z': codeword "0", 1000 bits, round trip IDENTICAL
```

Ada tiga hal yang perlu diperhatikan pada keluaran itu.

Penghematannya nyata dan besar. Kalimat sepanjang 126 karakter memakan 1008 bit jika disimpan apa adanya dan 489 bit jika disandikan Huffman — **51,5% lebih kecil** — lalu terurai kembali menjadi persis aslinya, karakter demi karakter. Ini masih berada di dalam rentang 20%–90% yang disebut di awal §9.7.

Buku sandinya persis seperti yang diramalkan teorinya. Karakter spasi dan huruf "a" — dua lambang yang paling sering muncul dalam kalimat itu — memperoleh kata sandi tiga bit; huruf "v" dan "w", yang masing-masing muncul sekali, memperoleh kata sandi enam bit. Lambang yang jarang duduk paling dalam, persis sebagaimana dituntut argumen penukaran Lema 9.7.

Kasus tepinya menggigit. Dengan hanya satu lambang yang berbeda, implementasi yang naif membangun pohon berupa satu daun tunggal pada kedalaman 0, sehingga kata sandinya adalah untaian kosong dan berkas tersandinya sepanjang nol bit — yang darinya tak ada apa pun yang dapat diuraikan. Programnya menanganinya dengan memasangkan lambang tunggalnya dengan sebuah boneka, memberinya kata sandi satu bit `0`, sehingga 1000 karakter berbiaya 1000 bit dan perjalanan bolak-baliknya berhasil. Setiap pemampat yang sungguhan memerlukan kasus khusus ini.

Bagian 7 mengukur waktu pembangunannya dari 4 000 sampai 512 000 lambang. Waktunya adalah 0.902, 1.821, 3.929, 10.200, 24.001, 65.980, 182.586, dan 461.363 milidetik. Membaginya dengan $n\lg n$ memberi 18.8,17.6,17.6,21.3,23.5,30.4,39.7,47.5 — sebuah lajur yang perlahan menaik, bukan yang mendatar. Batas asimtotik Teorema 9.10 tetap benar; yang diperlihatkan lajur yang menaik itu adalah bahwa implementasi ini mengalokasikan $2n-1$ simpul terpisah dengan `new` lalu mengejar penunjuk yang berserak di seluruh ingatan, sehingga *tetapannya* memburuk seiring himpunan kerjanya melampaui ukuran *cache*. Implementasi yang memakai satu larik datar berisi simpul dan indeks alih-alih penunjuk akan mendatarkan lajur itu. Ini peringatan yang berguna bahwa $O(n\lg n)$ adalah pernyataan tentang pertumbuhan, bukan janji tentang perilaku ingatan.

9.8 Matroid dan Metode Rakus

9.8.1 Pertanyaan yang dijawab bagian ini

Kita kini sudah membuktikan tiga algoritma rakus benar — pemilihan aktivitas, ransel pecahan, Huffman — dan tiap buktinya merupakan pekerjaan yang segar dari awal. Itu tidak memuaskan. Naluri seorang matematikawan akan bertanya: **adakah sebuah struktur tunggal yang membuat kerakusan otomatis benar setiap kali sebuah persoalan memiliki struktur itu?**

Ada, dan namanya **matroid**. Kita perlu berhati-hati tentang seberapa jauh klaimnya:

> **APA YANG DILIPUTI DAN TIDAK DILIPUTI TEORI MATROID.** Terus terang saja: teori matroid TIDAK meliputi semua kasus tempat metode rakus berlaku — khususnya **persoalan pemilihan aktivitas dan persoalan penyandian Huffman tidak terliput**. Tetapi ia memang meliputi banyak kasus yang penting secara praktis, termasuk persoalan pohon merentang minimum dan persoalan penjadwalan tugas di §9.8.6. Matroid adalah syarat cukup bagi kerakusan, bukan syarat perlu.

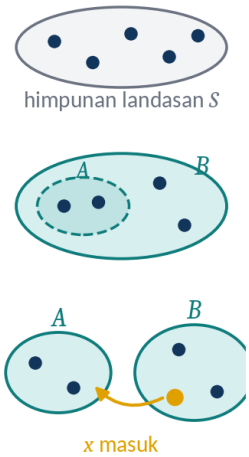
Jadi bagian ini bukan pengganti §9.4 dan §9.7; ia perkakas tambahan. Ketika persoalan Anda *memang* sebuah matroid, Anda memperoleh bukti kebenarannya secara cuma-cuma, dan itu tawar-menawar yang sangat menguntungkan.

9.8.2 Definisinya, dan apa arti tiap katanya

Gambar 9.11 menyatakan definisinya.

Matroid adalah pasangan $M = (S, \mathcal{I})$ yang memenuhi

1. S adalah himpunan berhingga — himpunan landasan.
2. $\mathcal{I}$ adalah keluarga tak kosong himpunan bagian S, yakni himpunan-himpunan bebas, dan bersifat turun-temurun: $B \in \mathcal{I}$ dan $A \subseteq B$ berakibat $A \in \mathcal{I}$ (jadi $\varnothing \in \mathcal{I}$).
3. Sifat penukaran: jika $A, B \in \mathcal{I}$ dan $|A| < |B|$, maka ada $x \in B - A$ dengan $A \cup \{x\} \in \mathcal{I}$.



Gambar 9.11 Definisi sebuah matroid: himpunan landasan berhingga S, keluarga turun-temurun $\mathcal{I}$ berisi himpunan bagian yang *bebas*, dan sifat penukarannya.

Itu tiga syarat atas sebuah pasangan $M = (S, \mathcal{I})$. Inilah ketiganya sekali lagi, masing-masing disusul terjemahannya ke dalam bahasa sehari-hari.

1. S adalah himpunan berhingga.

Terjemahan: ada sekarung benda. Sisi sebuah graf, tugas yang harus dijadwalkan, baris sebuah matriks — apa saja, asalkan jumlahnya berhingga. S disebut **himpunan landasan**.

2. $\mathcal{I}$ adalah keluarga tak kosong berisi himpunan bagian S, yang disebut himpunan bagian bebas, sedemikian sehingga jika $B \in \mathcal{I}$ dan $A \subseteq B$ maka $A \in \mathcal{I}$. Kita sebut $\mathcal{I}$ bersifat turun-temurun.

Terjemahan: sebagian kumpulan benda itu "beres" dan sisanya "rusak", dan **membuang benda tak pernah dapat merusakkan kumpulan yang beres**. Jika sehimpunan sisi tak memuat siklus, membuang satu sisi tetap menyisakan keadaan tanpa siklus. Jika sehimpunan tugas dapat dikerjakan tepat waktu semuanya, membuang satu tetap menyisakan himpunan yang seluruhnya dapat dikerjakan tepat waktu. Perhatikan satu akibat penting: karena $\mathcal{I}$ tak kosong ia memuat suatu himpunan B, dan $\varnothing \subseteq B$, sehingga **himpunan kosong selalu bebas**. Itu masuk akal: kumpulan yang kosong tak mungkin rusak.

Kata *bebas* dipinjam dari aljabar linear dan kata itulah yang menakutkan orang. Ia tidak berarti apa pun yang bersifat statistis. Bacalah ia sebagai **"taat asas ke dalam"**, atau sekadar **"sah"**.

3. Jika $A \in \mathcal{I}$, $B \in \mathcal{I}$, dan $|A| < |B|$, maka ada suatu unsur $x \in B - A$ sedemikian sehingga $A \cup \{x\} \in \mathcal{I}$. Kita katakan M memenuhi sifat penukaran.

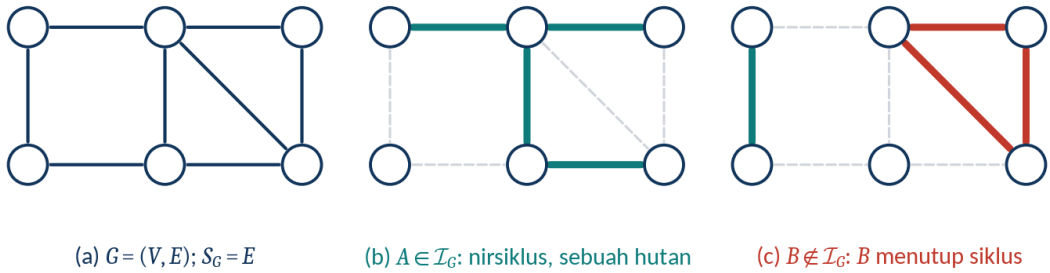
Terjemahan: **jika kumpulan sah Anda lebih kecil daripada kumpulan sah milik orang lain, Anda selalu dapat mencuri satu benda mereka dan tetap sah.** Bukan sembarang benda mereka — *suatu* benda. Inilah syarat yang mengerjakan seluruh pekerjaannya, dan inilah syarat yang akan gagal dipenuhi persoalan acak mana pun.

> **DENGAN BAHASA SEDERHANA.** Sebuah matroid adalah aturan untuk memutuskan kumpulan mana yang sah, yang memenuhi dua syarat keadilan. *Pertama*, kesahan tidak rusak oleh tindakan merelakan sesuatu. *Kedua*, tak seorang pun boleh terjebak: jika kumpulan sah

> Anda lebih kecil daripada kumpulan sah yang lain, selalu ada cara menumbuhkan kumpulan Anda satu langkah, dengan memakai sesuatu dari milik mereka. Syarat yang kedua itulah yang menghentikan algoritma rakus mengurung dirinya sendiri di sudut — ia menjamin bahwa kerakusan selalu dapat terus berjalan selama masih ada orang lain yang dapat berjalan.

9.8.3 Contoh yang dikerjakan: matroid grafik

Definisi yang abstrak menuntut contoh seketika. **Gambar 9.12** memberikan satu contoh.



Gambar 9.12 Matroid grafik $M_G = (S_G, \mathcal{I}_G)$ dari sebuah graf tak berarah $G = (V, E)$: himpunan landasannya adalah E, dan sehimpunan sisi bebas tepat ketika ia nirsiklus — yaitu ketika ia membentuk sebuah hutan. **(a)** Sebuah graf G; **(b)** himpunan bebas A berisi empat sisi tanpa siklus; **(c)** himpunan tak bebas B, yang sisi-sisi merahnya menutup sebuah siklus. M_G berkaitan erat dengan persoalan pohon merentang minimum, yang dibahas Bab 11 secara terperinci.

Ambillah graf tak berarah $G = (V, E)$, dengan V himpunan simpulnya (titik) dan E himpunan sisinya (garis yang menghubungkan titik). Tetapkan

- $S_G = E$, himpunan landasannya adalah sisi-sisinya;
- $A \in \mathcal{I}_G$ jika dan hanya jika A **nirsiklus** — yaitu subgraf $G_A = (V, A)$ tak memuat siklus, sehingga ia sebuah **hutan** (kumpulan pohon).

Periksalah ketiga syaratnya.

1. **Berhingga.** Sebuah graf punya berhingga banyak sisi. ✓
2. **Turun-temurun.** Jika sehimpunan sisi tak memuat siklus, dapatkah membuang sebuah sisi menciptakan siklus? Tidak — siklus di dalam himpunan bagiannya akan menjadi siklus di dalam keseluruhannya. ✓ (Dan $\varnothing$ nirsiklus. ✓)
3. **Penukaran.** Andaikan A dan B dua-duanya nirsiklus dan $|A| < |B|$. Hutan atas $|V|$ simpul dengan k sisi punya tepat $|V| - k$ komponen terhubung (tiap sisi yang ditambahkan ke graf kosong menyatukan dua komponen menjadi satu). Jadi G_A punya $|V| - |A|$ komponen dan G_B punya $|V| - |B|$ komponen; karena $|A| < |B|$, G_A punya komponen yang benar-benar **lebih banyak** daripada G_B. Nah, B, yang komponennya lebih sedikit, pasti memuat sebuah sisi (u, v) yang kedua ujungnya u dan v terletak di komponen G_A yang *berlainan* — sebab seandainya setiap sisi B menghubungkan dua simpul di dalam satu komponen G_A, maka setiap komponen G_B akan duduk di dalam sebuah komponen G_A, sehingga komponen G_B sekurang-kurangnya sebanyak komponen G_A. Menambahkan sisi (u, v) itu kepada A menyatukan dua komponen yang berlainan dan karena itu tak mungkin menciptakan siklus (siklus menuntut jalur kedua antara u dan v, dan tak ada jalur semacam itu yang menyeberangi komponen). Jadi $A \cup \{(u, v)\} \in \mathcal{I}_G$, dan $(u, v) \notin A$ sebab ia menyeberangi komponen-komponen G_A. ✓

Jadi matroid grafik memang sungguh-sungguh sebuah matroid, dan sifat penukarannya adalah pernyataan bahwa hutan yang kecil selalu dapat ditumbuhkan memakai sebuah sisi dari hutan yang lebih besar. Bab 11 akan memakai fakta ini persis untuk membuktikan algoritma Kruskal benar — Anda sekarang sudah melihat buktinya, lebih dahulu, dalam bahasa yang berbeda.

SEBUAH BUKAN-CONTOH, UNTUK MELIHAT APA YANG GAGAL. Misalkan S adalah aktivitas **Gambar 9.1** dan misalkan $\mathcal{I}$ adalah himpunan-himpunan aktivitas yang saling selaras. Syarat 1 dan 2 berlaku — membuang sebuah aktivitas tak mungkin menciptakan pertindihan. Tetapi syarat 3 gagal. Ambil $A = \{a_3\} = \{[0,6)\}$ dan $B = \{a_1, a_4\} = \{[1,4), [5,7)\}$; keduanya selaras, dan $|A| = 1 < 2 = |B|$. Dapatkah kita menambahkan sebuah unsur B kepada A dan tetap selaras? $a_1 = [1,4)$ bertindih dengan $[0,6)$; $a_4 = [5,7)$ juga bertindih dengan $[0,6)$. Tak satu pun berhasil. Sifat penukarannya gagal, dan persis inilah sebabnya pemilihan aktivitas **tidak** terliput teori matroid, meskipun kerakusan menyelesaikannya. Kebenarannya menuntut argumen yang dijahit khusus di §9.4.7.

9.8.4 Dua akibat sifat penukaran

Sifat penukaran punya dua akibat langsung yang menjelaskan *mengapa* matroid membuat kerakusan berhasil.

LEMA 9.11 (perluasan). Misalkan $M = (S, \mathcal{I})$ sebuah matroid dan $A \in \mathcal{I}$. Jika ada sembarang $B \in \mathcal{I}$ dengan $|B| > |A|$, maka A dapat diperluas: ada $x \in S - A$ dengan $A \cup \{x\} \in \mathcal{I}$.

Bukti. Ini sifat penukaran yang dinyatakan ulang; unsur pemerluasnya boleh diambil dari $B - A$. ■

LEMA 9.12. Semua himpunan bagian bebas yang **maksimal** dari sebuah matroid punya **ukuran yang sama**.

Bukti. Sebutlah A maksimal jika tak ada unsur yang dapat ditambahkan kepadanya sambil tetap bebas. Andaikan A dan B dua-duanya maksimal dan $|A| < |B|$. Menurut sifat penukaran ada $x \in B - A$ dengan $A \cup \{x\} \in \mathcal{I}$ — bertentangan dengan kemaksimalan A. Jadi $|A| \geq |B|$; menurut kesetangkupan $|B| \geq |A|$; karena itu $|A| = |B|$. ■

INILAH SELURUH ALASAN MENGAPA MATROID BERHASIL. Ingatlah pembedaan §9.4.2: *maksimal* (tak dapat diperluas) itu mudah, *maksimum* (tak ada yang lebih besar) itu sulit. Lema 9.12 mengatakan bahwa **di dalam matroid pembedaan itu runtuh** — setiap himpunan bebas yang maksimal otomatis berukuran maksimum. Algoritma rakus menghasilkan himpunan yang maksimal tanpa susah payah, menurut konstruksinya. Di dalam matroid, tanpa susah payah sudah cukup.

Bagi matroid grafik, Lema 9.12 mengatakan bahwa setiap hutan maksimal punya cacah sisi yang sama — yang, bagi graf terhubung, merupakan pernyataan yang sudah kita kenal bahwa setiap pohon merentang punya tepat $|V| - 1$ sisi.

9.8.5 Matroid berbobot dan algoritma rakusnya

Untuk memperoleh persoalan optimasi kita memerlukan bilangan.

DEFINISI. Sebuah matroid $M = (S, \mathcal{I})$ disebut **berbobot** jika ia disertai **fungsi bobot** w yang memberikan bobot positif tegas $w(x) > 0$ kepada tiap unsur $x \in S$. Bobot sebuah himpunan bagian adalah $w(A) = \sum_{x \in A} w(x)$. Himpunan bagian **optimal** dari sebuah matroid berbobot adalah himpunan bagian bebas yang **bobot totalnya maksimum**.

Karena bobotnya positif, himpunan bagian yang optimal selalu maksimal — menambahkan apa pun yang dapat Anda tambahkan akan menaikkan bobotnya — dan karena itu, menurut Lema 9.12, berukuran maksimum pula.

Pseudokode 9.1 memberikan algoritmanya; intinya hanya tiga baris.

Pseudokode 9.1 **Greedy**(M, w), algoritma rakus matroid berbobot. §9.8.8 memberikannya sebagai program utuh yang dapat dikompilasi, dengan orakel kebebasan yang abstrak.

```
GREEDY(M, w)
1  A = {}
2  sort M.S into monotonically decreasing order by weight w
3  for each x in M.S, taken in that order
4      if A union {x} is in M.I          // apakah himpunan yang diperbesar tetap bebas?
5          A = A union {x}
6  return A
```

Waktu jalannya mudah dibatasi. Mengurutkan $n = |S|$ unsur (baris 2) berbiaya $O(n\lg n)$. Gelung baris 3–5 berjalan n kali, dan tiap iterasinya melakukan satu uji kebebasan (baris 4). Jika sebuah uji berbiaya $f(n)$, totalnya adalah

$$O(n\lg n + n\,f(n)). \tag{9.11}$$

Bagi matroid grafik, sebuah uji adalah satu operasi *union-find*, yang pada dasarnya $O(1)$, sehingga Persamaan (9.11) menjadi $O(n\lg n)$ — dan algoritmanya *memang* algoritma Kruskal. Bagi matroid penjadwalan tugas di bawah, uji yang naif berbiaya $O(n)$, memberi $O(n^2)$, yang dapat disusutkan menjadi $O(n\lg n)$ dengan struktur himpunan-saling-lepas.

Mengapa ia benar? Kedua bahannya, seperti biasa.

LEMA 9.13 (matroid punya sifat pilihan rakus). Misalkan $M = (S, \mathcal{I})$ sebuah matroid berbobot dengan S terurut menurun monoton menurut bobot. Misalkan x adalah unsur **pertama** S yang membuat $\{x\}$ bebas, jika unsur semacam itu ada. Maka ada himpunan bagian optimal A dari S yang memuat x.

Bukti (argumen penukaran). Andaikan B sebuah himpunan bagian optimal yang tak kosong. Darinya kita bangun himpunan bagian optimal yang memuat x.

1. Jika $x \in B$ kita selesai, jadi anggaplah $x \notin B$.
2. Tak ada unsur B yang bobotnya lebih besar daripada $w(x)$. *Mengapa:* setiap $y \in B$ punya $\{y\} \subseteq B$ yang bebas menurut sifat turun-temurun, jadi y termasuk unsur yang singletonnya bebas; dan x adalah unsur *pertama* semacam itu dalam urutan bobot yang menurun, sehingga $w(y) \leq w(x)$.
3. Bangunlah himpunan A sebagai berikut. Mulailah dengan $A_0 = \{x\}$, yang bebas. Selama $|A_k| < |B|$, terapkan sifat penukaran pada A_k dan B untuk memperoleh unsur $B - A_k$ yang dapat ditambahkan; misalkan A_{k+1} hasilnya. Proses ini berakhir dengan himpunan A yang memenuhi $|A| = |B|$, $A \in \mathcal{I}$, dan $A \subseteq \{x\} \cup B$ — bahkan $A = \{x\} \cup B \setminus \{y\}$ untuk tepat satu unsur $y \in B$, sebab A memuat x ditambah $|B| - 1$ unsur B.
4. Sekarang bandingkan bobotnya:

$$w(A) = w(B) - w(y) + w(x).$$

5. Menurut langkah 2, $w(x) \geq w(y)$, jadi $w(A) \geq w(B)$.
6. Tetapi B optimal, jadi $w(A) \leq w(B)$. Karena itu $w(A) = w(B)$ dan A pun optimal — dan $x \in A$. ■

LEMA 9.14 (substruktur optimal). Misalkan x unsur pertama S yang dipilih **Greedy** bagi matroid berbobot $M = (S, \mathcal{I})$. Persoalan sisa, yaitu mencari himpunan bagian bebas berbobot maksimum yang memuat x, menyusut menjadi pencarian himpunan bagian bebas berbobot maksimum dari matroid berbobot $M' = (S', \mathcal{I}')$, dengan

$$S' = \{\, y \in S : \{x, y\} \in \mathcal{I} \,\}, \qquad \mathcal{I}' = \{\, B \subseteq S' : B \cup \{x\} \in \mathcal{I} \,\},$$

dengan bobot yang sama. (Pasangan M' disebut **kontraksi** M oleh x, dan ia sekali lagi sebuah matroid.)

Bukti. Jika A himpunan bagian bebas dari M yang memuat x, maka $A \setminus \{x\}$ bebas di M' menurut definisi $\mathcal{I}'$, dan $w(A) = w(A \setminus \{x\}) + w(x)$. Sebaliknya jika B bebas di M' maka $B \cup \{x\}$ bebas di M, dengan $w(B \cup \{x\}) = w(B) + w(x)$. Jadi nilai fungsi tujuan kedua persoalannya berselisih tetapan $w(x)$, dan pemaksimum yang satu adalah pemaksimum yang lain. ■

TEOREMA 9.15. Greedy(M, w) mengembalikan himpunan bagian optimal dari matroid berbobot M.

Bukti. Lewat induksi atas $|S|$. Jika tak ada singleton yang bebas, setiap himpunan bebas adalah himpunan kosong dan **Greedy** mengembalikan $\varnothing$, yang optimal. Jika tidak, misalkan x unsur pertama yang membuat $\{x\}$ bebas — unsur pertama yang diterima **Greedy**. Menurut Lema 9.13 suatu himpunan bagian optimal memuat x, jadi mengambil x aman. Menurut Lema 9.14 yang tersisa adalah persoalan yang sama pada kontraksi M', yang himpunan landasannya lebih kecil. Iterasi sisa **Greedy** persis merupakan satu jalannya **Greedy** atas M' dengan unsurnya masih dalam urutan bobot yang menurun, jadi menurut hipotesis induktifnya mereka mengembalikan himpunan bagian optimal dari M'. Menambahkan x memberi himpunan bagian optimal dari M. ■

BANDINGKAN JERIH PAYAHNYA. Di §9.4.7 kita membuktikan sifat pilihan rakus bagi satu persoalan yang khusus, dan buktinya sepanjang setengah halaman. Lema 9.13 membuktikannya **sekali, bagi setiap matroid berbobot yang akan pernah ada**. Itulah imbalan pengabstrakan: kenalilah persoalan Anda sebagai matroid dan bukti kebenarannya sudah tertulis.

9.8.6 Penerapannya: menjadwalkan tugas berwaktu-satuan dengan tenggat dan denda

DEFINISI. Sebuah **tugas berwaktu-satuan** adalah pekerjaan yang memakan tepat satu satuan waktu pada sebuah pemroses tunggal. Kita diberi n tugas $a_1, \ldots, a_n$; tugas a_i punya **tenggat** bilangan bulat d_i dengan $1 \le d_i \le n$, dan **denda** bilangan bulat $w_i > 0$ yang harus kita bayar jika tugas itu selesai **sesudah** tenggatnya. Sebuah **jadwal** adalah permutasi tugasnya; tugas pada posisi j berjalan selama $[j-1, j)$ dan selesai pada waktu j. Sebuah tugas disebut **terlambat** jika ia selesai sesudah tenggatnya, dan **awal** jika tidak. Persoalannya: carilah jadwal yang meminimumkan total denda tugas yang terlambat.

Dua penyederhanaan membuat persoalan ini tertangani, dan keduanya layak dilihat.

Penyederhanaan 1: hanya *himpunan* tugas awalnya yang penting. Diberi jadwal apa pun, kita dapat menata ulangnya menjadi **bentuk awal-dahulu**: letakkan semua tugas awal di depan, menaik tak turun menurut tenggat, lalu semua tugas terlambat dalam urutan apa pun. Hal ini tidak membuat satu pun tugas awal menjadi terlambat. *Mengapa:* andaikan tugas awal a tadinya di posisi j (jadi $j \le d_a$) dan sesudah penataan ulang ia berada di posisi j'. Dalam bentuk awal-dahulu, tugas-tugas pada posisi $1, \ldots, j'$ — yaitu a sendiri dan setiap tugas yang diletakkan sebelumnya — semuanya tugas awal yang tenggatnya $\le d_a$, sebab tugas-tugas awal diurutkan menurut tenggat. Sekarang kita mencacah. Keseluruhan j' tugas itu juga awal di dalam jadwal aslinya, sehingga di sana masing-masing selesai tidak lebih lambat daripada tenggatnya sendiri, yang paling besar d_a. Jadi di dalam jadwal aslinya ke-j' tugas itu menempati j' petak yang berlainan, semuanya bernomor $\le d_a$. Petak semacam itu hanya ada d_a buah, sehingga $j' \le d_a$: tugas a tetap selesai tepat waktu di posisinya yang baru. Jadi kita boleh membatasi perhatian pada jadwal awal-dahulu, dan jadwal semacam itu ditentukan sepenuhnya oleh himpunan tugas mana yang awal.

Penyederhanaan 2: meminimumkan denda tugas yang terlambat sama dengan memaksimumkan denda tugas yang awal. Total seluruh dendanya, $\sum_i w_i$, adalah tetapan. Jadi

$$\text{minimumkan} \sum_{\text{terlambat}} w_i \Leftrightarrow \text{maksimumkan} \sum_{\text{awal}} w_i. \tag{9.12}$$

Dengan Persamaan (9.12) kita sudah mengubah peminimuman menjadi pemaksimuman, dan itulah yang diinginkan sebuah matroid berbobot.

DEFINISI. Sehimpunan tugas A disebut **bebas** jika ada jadwal yang di dalamnya **tak satu pun tugas** A **terlambat**. Untuk $t = 0,1,\ldots,n$, misalkan

$$N_t(A) = |\{\, a_i \in A : d_i \le t \,\}| \tag{9.13}$$

adalah cacah tugas di A yang tenggatnya t atau lebih awal.

LEMA 9.16. Bagi sembarang himpunan A berisi tugas berwaktu-satuan, ketiga pernyataan berikut setara. **(1)** A bebas. **(2)** $N_t(A) \le t$ untuk setiap $t = 0,1,\ldots,n$. **(3)** Jika tugas-tugas A dijadwalkan menaik tak turun menurut tenggat, tak satu pun tugas A terlambat.

Bukti. Kita buktikan (1) ⇒ (2) ⇒ (3) ⇒ (1).

(1) ⇒ (2). Andaikan A bebas, jadi ada jadwal yang membuat tak satu pun tugas A terlambat. Tetapkan t. Setiap tugas di A yang tenggatnya $\le t$ harus selesai paling lambat pada waktu t di dalam jadwal itu. Hanya petak waktu $1,2,\ldots,t$ yang selesai pada atau sebelum waktu t, dan masing-masing memuat satu tugas berwaktu-satuan. Jadi paling banyak t tugas yang dapat selesai pada waktu t, sehingga $N_t(A) \le t$.

(2) ⇒ (3). Anggaplah $N_t(A) \le t$ untuk semua t, lalu jadwalkan A menaik tak turun menurut tenggat. Misalkan sebuah tugas a menempati posisi j, sehingga ia selesai pada waktu j. Sebanyak j tugas pada posisi $1,\ldots,j$ semuanya bertenggat $\le d_a$, sebab urutannya tak menurun. Karena itu ada sekurang-kurangnya j tugas di A yang tenggatnya $\le d_a$, yaitu $N_{d_a}(A) \ge j$. Menurut hipotesisnya $N_{d_a}(A) \le d_a$. Merangkainya: $j \le N_{d_a}(A) \le d_a$, jadi a selesai pada waktu $j \le d_a$ dan tidak terlambat.

(3) ⇒ (1). Pernyataan (3) memperagakan sebuah jadwal bagi A yang di dalamnya tak satu pun tugas A terlambat; perpanjanglah ia dengan menambahkan tugas yang bukan anggota A dalam urutan apa pun. Itulah jadwal yang menyaksikan kebebasan A. ■

Syarat (2), dengan $N_t(A)$ dicacah seperti pada Persamaan (9.13), adalah *ujinya*, dan itulah yang diwujudkan programnya: untuk memeriksa apakah sehimpunan tugas bebas, hitunglah untuk tiap t berapa banyak di antaranya yang jatuh tempo pada waktu t, lalu pastikan cacahnya tak pernah melebihi t.

TEOREMA 9.17. Misalkan S himpunan seluruh n tugas berwaktu-satuan dan misalkan $\mathcal{I}$ keluarga himpunan tugas yang bebas. Maka $M = (S, \mathcal{I})$ adalah sebuah matroid.

Bukti. Tiga syarat.

1. S **berhingga.** Ada n tugas. ✓
2. $\mathcal{I}$ **turun-temurun.** Misalkan $B \in \mathcal{I}$ dan $A \subseteq B$. Untuk setiap t, $N_t(A) \le N_t(B)$ sebab A tak punya tugas yang tak dipunyai B; dan $N_t(B) \le t$ menurut Lema 9.16(2). Jadi $N_t(A) \le t$ untuk semua t, sehingga $A \in \mathcal{I}$. ✓ (Dan $\varnothing$ bebas, sebab $N_t(\varnothing) = 0 \le t$.)
3. **Sifat penukarannya.** Misalkan $A, B \in \mathcal{I}$ dengan $|A| < |B|$. Kita harus menemukan $x \in B - A$ dengan $A \cup \{x\} \in \mathcal{I}$.

- Karena setiap tenggatnya paling banyak n, $N_n(A) = |A|$ dan $N_n(B) = |B|$, sehingga $N_n(A) < N_n(B)$.
- Juga $N_0(A) = N_0(B) = 0$, sebab tak ada tenggat yang ≤ 0.
- Misalkan k nilai t **terbesar** di $\{0,1,\dots,n\}$ yang membuat $N_t(A) \geq N_t(B)$. Nilai k semacam itu ada karena $t = 0$ memenuhinya, dan $k < n$ karena $t = n$ tidak. Menurut kemaksimalannya, $N_t(A) < N_t(B)$ untuk setiap $t > k$.
- Tinjaulah tugas yang tenggatnya **tepat** $k + 1$. Di A jumlahnya $N_{k+1}(A) - N_k(A)$; di B jumlahnya $N_{k+1}(B) - N_k(B)$. Nah, $N_{k+1}(B) > N_{k+1}(A)$ (sebab $k + 1 > k$) dan $N_k(B) \leq N_k(A)$ (menurut pemilihan k), jadi

$$N_{k+1}(B) - N_k(B) \;>\; N_{k+1}(A) - N_k(A).$$

- Jadi B memuat tugas bertenggat tepat $k + 1$ yang benar-benar lebih banyak daripada yang dimuat A. Karena itu himpunan tugas B yang bertenggat $k + 1$ bukan himpunan bagian dari milik A, dan kita boleh memilih tugas $x \in B$ dengan $d_x = k + 1$ dan $x \notin A$.
- **Periksalah** $A \cup \{x\}$ **bebas** dengan Lema 9.16(2). Untuk $t \leq k$: x bertenggat $k + 1 > t$, jadi $N_t(A \cup \{x\}) = N_t(A) \leq t$ ✓. Untuk $t \geq k + 1$: $N_t(A \cup \{x\}) = N_t(A) + 1 \leq N_t(B) \leq t$, dengan ketaksamaan pertamanya memakai $N_t(A) < N_t(B)$ untuk $t > k$ dan yang kedua adalah Lema 9.16(2) bagi B ✓.
- Karena itu $A \cup \{x\} \in \mathcal{I}$. ✓ ■

Menurut Teorema 9.15, karena itu, algoritma rakus — **ambillah tugas dalam urutan denda yang menurun, simpan tiap tugas jika himpunannya tetap bebas** — menyelesaikan persoalan penjadwalannya secara optimal. Tak ada bukti lain yang diperlukan. Itulah seluruh pokok matroid.

9.8.7 Dikerjakan dengan tangan pada sebuah contoh

Gambar 9.13 memberikan contoh kasus yang akan kita kerjakan dengan tangan.

Tugas

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Gambar 9.13 Sebuah contoh kasus bagi persoalan menjadwalkan tugas berwaktu-satuan dengan tenggat dan denda pada satu pemroses: tujuh tugas $a_1, \dots, a_7$ dengan tenggat d_i dan denda w_i.

Datanya: tenggat $d = \langle 4,2,4,3,1,4,6 \rangle$ dan denda $w = \langle 70,60,50,40,30,20,10 \rangle$. Totalnya

$$70 + 60 + 50 + 40 + 30 + 20 + 10 = 280,$$

yaitu yang akan kita bayar seandainya setiap tugas terlambat. Kebetulan tugasnya sudah terdaftar menurun menurut denda, jadi **Greedy** meninjaunya sebagai $a_1, a_2, \dots, a_7$.

Sekarang jalankan, dengan menguji kebebasannya memakai Lema 9.16(2) pada tiap langkah.

Tinjau a_1 ($d = 4$, $w = 70$). Himpunan cobanya $\{a_1\}$. Cacahnya: $N_1 = 0$, $N_2 = 0$, $N_3 = 0$, $N_4 = 1$, $N_5 = 1$, $N_6 = 1$, $N_7 = 1$. Apakah tiap $N_t \leq t$? $0 \leq 1$, $0 \leq 2$, $0 \leq 3$, $1 \leq 4$, … ✓ **Simpan.** $A = \{a_1\}$.

Tinjau a_2 ($d = 2$, $w = 60$). Himpunan cobanya $\{a_1, a_2\}$. $N_1 = 0 \leq 1$ ✓, $N_2 = 1 \leq 2$ ✓, $N_3 = 1 \leq 3$ ✓, $N_4 = 2 \leq 4$ ✓, sisanya beres. **Simpan.** $A = \{a_1, a_2\}$.

Tinjau a_3 ($d = 4$, $w = 50$). Himpunan cobanya $\{a_1, a_2, a_3\}$. $N_1 = 0$, $N_2 = 1 \leq 2$ ✓, $N_3 = 1 \leq 3$ ✓, $N_4 = 3 \leq 4$ ✓. **Simpan.** $A = \{a_1, a_2, a_3\}$.

Tinjau a_4 ($d = 3$, $w = 40$). Himpunan cobanya $\{a_1, a_2, a_3, a_4\}$. $N_1 = 0 \leq 1$ ✓, $N_2 = 1 \leq 2$ ✓, $N_3 = 2 \leq 3$ ✓, $N_4 = 4 \leq 4$ ✓ — tepat di batasnya. **Simpan.** $A = \{a_1, a_2, a_3, a_4\}$.

Tinjau a_5 ($d = 1$, $w = 30$). Himpunan cobanya $\{a_1, a_2, a_3, a_4, a_5\}$. $N_1 = 1 \leq 1$ ✓, $N_2 = 2 \leq 2$ ✓, $N_3 = 3 \leq 3$ ✓, $N_4 = \mathbf{5} > 4$ ✗. **Tolak** — lima tugas yang semuanya jatuh tempo pada waktu 4 tak mungkin semuanya dikerjakan dalam empat petak.

Tinjau a_6 ($d = 4$, $w = 20$). Himpunan cobanya $\{a_1, a_2, a_3, a_4, a_6\}$. $N_4 = 5 > 4$ ✗. **Tolak**, karena alasan yang sama.

Tinjau a_7 ($d = 6$, $w = 10$). Himpunan cobanya $\{a_1, a_2, a_3, a_4, a_7\}$. $N_4 = 4 \leq 4$ ✓, $N_5 = 4 \leq 5$ ✓, $N_6 = 5 \leq 6$ ✓, $N_7 = 5 \leq 7$ ✓. **Simpan.** $A = \{a_1, a_2, a_3, a_4, a_7\}$.

Jadi himpunan awalnya adalah $A = \{a_1, a_2, a_3, a_4, a_7\}$, berbobot

$$70 + 60 + 50 + 40 + 10 = 230,$$

dan himpunan terlambatnya $\{a_5, a_6\}$, berbobot

$$30 + 20 = 50.$$

Periksa: $230 + 50 = 280$ ✓. Total denda minimumnya adalah **50**.

Untuk menghasilkan jadwal yang sesungguhnya, letakkan tugas awalnya menaik tak turun menurut tenggat (Lema 9.16(3)) lalu tambahkan yang terlambat: $\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle$. Periksalah petak demi petak: a_2 di petak 1 dengan $d = 2 \geq 1$ ✓; a_4 di petak 2 dengan $d = 3 \geq 2$ ✓; a_1 di petak 3 dengan $d = 4 \geq 3$ ✓; a_3 di petak 4 dengan $d = 4 \geq 4$ ✓; a_7 di petak 5 dengan $d = 6 \geq 5$ ✓; a_5 di petak 6 dengan $d = 1 < 6$ — terlambat, sebagaimana diperkirakan; a_6 di petak 7 dengan $d = 4 < 7$ — terlambat. **Tabel 9.10** mencatat jalannya.

Tabel 9.10 Greedy pada tujuh tugas **Gambar 9.13**, satu baris per unsur yang ditinjau dalam urutan denda yang menurun. Bobot awal 230, denda yang dibayar 50.

Ditinjau	d	w	Uji kebebasan	Vonis	A Sesudahnya
a_1	4	70	$N_4 = 1 \leq 4$	simpan	$\{a_1\}$
a_2	2	60	$N_2 = 1 \leq 2, N_4 = 2 \leq 4$	simpan	$\{a_1, a_2\}$
a_3	4	50	$N_4 = 3 \leq 4$	simpan	$\{a_1, a_2, a_3\}$
a_4	3	40	$N_3 = 2 \leq 3, N_4 = 4 \leq 4$	simpan	$\{a_1, a_2, a_3, a_4\}$
a_5	1	30	$N_4 = 5 > 4$	**tolak**	$\{a_1, a_2, a_3, a_4\}$
a_6	4	20	$N_4 = 5 > 4$	**tolak**	$\{a_1, a_2, a_3, a_4\}$
a_7	6	10	$N_6 = 5 \leq 6$	simpan	$\{a_1, a_2, a_3, a_4, a_7\}$

Program di **Kode 9.5** memastikannya, dan memeriksanya terhadap *brute force* atas seluruh $2^7 = 128$ himpunan bagian:

```
early (on time) set A = {a1, a2, a3, a4, a7},  weight saved = 230
late set             = {a5, a6},  penalty paid = 50
schedule             : a2(slot 1, d=2, on time)
                       a4(slot 2, d=3, on time)
                       a1(slot 3, d=4, on time)
                       a3(slot 4, d=4, on time)
                       a7(slot 5, d=6, on time)
                       a5(slot 6, d=1, LATE)
                       a6(slot 7, d=4, LATE)

brute force over all 2^7 = 128 subsets:
  best independent set = {a1, a2, a3, a4, a7},  weight saved = 230,  minimum penalty = 50
  greedy matches the optimum: YES

stress test, 200000 random 8-task instances:
  greedy differed from the optimum in 0 of 200000 instances
```

Dua ratus ribu contoh kasus acak, dan kerakusan menyamai optimum tuntasnya pada setiap satu di antaranya. Memang ia tak mungkin berbuat lain: Teorema 9.17 mengatakan strukturnya sebuah matroid, dan Teorema 9.15 mengatakan kerakusan karenanya optimal. Uji bebannya bukan bukti bagi teoremanya; ia pemeriksaan bahwa *programnya* mewujudkan teorema itu.

9.8.8 Satu algoritma, dua persoalan

Kode 9.5 menulis **Greedy**(M, w) tepat sekali, terhadap orakel kebebasan yang abstrak, lalu menyuapinya dengan dua orakel yang berlainan. Dengan orakel penjadwalan tugas ia menyelesaikan persoalan di atas. Dengan orakel "apakah himpunan sisi ini nirsiklus?" ia menjelma menjadi **algoritma pohon merentang minimum Kruskal** — pokok bahasan Bab 11 — tanpa satu baris pun gelung rakusnya diubah.

Ada satu kerut: **Greedy** *memaksimumkan* bobot, sedangkan pohon merentang minimum *meminimumkannya*. Perbaikannya adalah siasat baku: jalankan algoritmanya pada bobot yang ditransformasikan $w'(e) = w_{\max} + 1 - w(e)$, yang membalikkan urutan sisinya tanpa mengubah himpunan mana yang bebas. Memaksimumkan w' atas sebuah hutan maksimal sama dengan meminimumkan w atasnya, sebab pohon merentang sebuah graf terhubung selalu punya tepat $|V| - 1$ sisi (Lema 9.12 lagi), sehingga $\sum w'(e) = (|V| - 1)(w_{\max} + 1) - \sum w(e)$ dan suku pertamanya adalah tetapan.

Kode 9.5 **Greedy**(M, w) ditulis sekali terhadap orakel kebebasan yang abstrak, lalu diterapkan pada matroid penjadwalan tugas (dengan pemeriksaan *brute force* dan uji beban atas 200 000 contoh kasus) dan pada matroid grafik, tempat ia menjelma menjadi algoritma Kruskal. Kompilasi dengan `g++ -std=c++17 -O2 -o matroid matroid.cpp` lalu jalankan `./matroid`.

```
// matroid.cpp — algoritma rakus matroid berbobot, ditulis sekali, dipakai dua kali.
//   * GREEDY(M, w) ditulis SEKALI saja terhadap orakel kebebasan yang abstrak;
//   * contoh kasus 1: penjadwalan tugas berwaktu-satuan dengan tenggat dan denda,
//     atas tujuh tugas Subbab 9.8.7, diperiksa dengan brute force;
//   * contoh kasus 2: matroid grafik, yang mengubah kode yang sama itu menjadi
//     algoritma pohon merentang minimum Kruskal (Bab 11).
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <functional>
#include <algorithm>
#include <numeric>
#include <random>
using namespace std;

// ------------------------------------------------------------ GREEDY(M,w) --
// S himpunan landasannya, diberikan sebagai indeks 0..n-1.  w[i] bobot unsur i.
// independent(A) memutuskan apakah himpunan bagian A termasuk keluarga I.
// Algoritmanya mengurutkan S secara MENURUN monoton menurut bobot lalu menambahkan
// tiap unsur bila penambahan itu menjaga himpunannya tetap bebas.
vector<int> matroidGreedy(int n, const vector<long long>& w,
                          const function<bool(const vector<int>&)>& independent,
                          bool trace, const vector<string>& name) {
    vector<int> order(n);
    iota(order.begin(), order.end(), 0);
    sort(order.begin(), order.end(), [&](int a, int b) {
        if (w[a] != w[b]) return w[a] > w[b];
        return a < b;                                  // pemutus seri yang terulangkan
    });
    vector<int> A;
    for (int x : order) {
        A.push_back(x);
        bool ok = independent(A);
        if (!ok) A.pop_back();
        if (trace) {
            cout << "    consider " << setw(4) << name[x] << " (weight " << setw(3) << w[x]
<< "): "
                 << (ok ? "keep   A = {" : "REJECT A = {");
            for (size_t i = 0; i < A.size(); ++i) { if (i) cout << ", "; cout <<
name[A[i]]; }
            cout << "}\n";
        }
    }
    return A;
}
```

Kode 9.5 (Lanjutan) *Greedy(M,w) ditulis sekali terhadap orakel kebebasan yang abstrak, lalu diterapkan pada matroid penjadwalan tugas (dengan pemeriksaan brute force dan uji beban atas 200 000 contoh kasus) dan pada matroid grafik, tempat ia menjelma menjadi algoritma Kruskal. Kompilasi dengan g++ -std=c++17 -O2 -o matroid matroid.cpp lalu jalankan ./matroid.*

```
// ------------- contoh kasus 1: penjadwalan tugas berwaktu-satuan -------
// Himpunan tugas A bebas jika dan hanya jika ada jadwal yang membuat tak satu pun
// tugas di A terlambat.  Ujinya: untuk tiap t, cacah tugas di A yang tenggatnya
// <= t haruslah paling banyak t.
static vector<int> DEADLINE;

bool taskIndependent(const vector<int>& A) {
    int n = (int)DEADLINE.size();
    for (int t = 1; t <= n; ++t) {
        int cnt = 0;
        for (int i : A) if (DEADLINE[i] <= t) ++cnt;
        if (cnt > t) return false;
    }
    return true;
}

int main() {
    cout << "=== Instance 1: the seven unit-time tasks of Section 9.8.7 ===\n";
    vector<string> tname = { "a1", "a2", "a3", "a4", "a5", "a6", "a7" };
    DEADLINE =            {    4,    2,    4,    3,    1,    4,    6  };
    vector<long long> pen = {  70,   60,   50,   40,   30,   20,   10  };
    int n = 7;

    cout << "  task     :"; for (auto& s : tname) cout << setw(6) << s;   cout << "\n";
    cout << "  deadline :"; for (int d : DEADLINE) cout << setw(6) << d;  cout << "\n";
    cout << "  penalty  :"; for (long long p : pen) cout << setw(6) << p; cout << "\n";
    long long totalPen = accumulate(pen.begin(), pen.end(), 0LL);
    cout << "  total penalty if every task were late = " << totalPen << "\n\n";

    cout << "  GREEDY, taking tasks in decreasing order of penalty:\n";
    vector<int> A = matroidGreedy(n, pen, taskIndependent, true, tname);

    cout << "\n  early (on time) set A = {";
    vector<int> sortedA = A; sort(sortedA.begin(), sortedA.end());
    for (size_t i = 0; i < sortedA.size(); ++i) { if (i) cout << ", "; cout <<
tname[sortedA[i]]; }
    long long saved = 0; for (int i : A) saved += pen[i];
    cout << "},  weight saved = " << saved << "\n";

    cout << "  late set             = {";
    vector<int> late; long long paid = 0;
    for (int i = 0; i < n; ++i)
        if (find(A.begin(), A.end(), i) == A.end()) { late.push_back(i); paid += pen[i]; }
    for (size_t i = 0; i < late.size(); ++i) { if (i) cout << ", "; cout << tname[late[i]];
}
    cout << "},  penalty paid = " << paid << "\n";
```

Kode 9.5 (Lanjutan) *Greedy(M,w) ditulis sekali terhadap orakel kebebasan yang abstrak, lalu diterapkan pada matroid penjadwalan tugas (dengan pemeriksaan brute force dan uji beban atas 200 000 contoh kasus) dan pada matroid grafik, tempat ia menjelma menjadi algoritma Kruskal. Kompilasi dengan g++ -std=c++17 -O2 -o matroid matroid.cpp lalu jalankan ./matroid.*

```
    // jadwal sesungguhnya: tugas awal menurut urutan tenggat, lalu yang terlambat
    vector<int> sched = A;
    sort(sched.begin(), sched.end(), [&](int a, int b) {
        if (DEADLINE[a] != DEADLINE[b]) return DEADLINE[a] < DEADLINE[b];
        return a < b;
    });
    for (int i : late) sched.push_back(i);
    cout << "  schedule             : ";
    for (size_t i = 0; i < sched.size(); ++i) {
        cout << tname[sched[i]] << "(slot " << i + 1 << ", d=" << DEADLINE[sched[i]]
             << (DEADLINE[sched[i]] >= (int)i + 1 ? ", on time" : ", LATE") << ")";
        if (i + 1 < sched.size()) cout << "\n                          ";
    }
    cout << "\n\n";

    cout << "  brute force over all 2^7 = 128 subsets:\n";
    long long bestSaved = -1; int bestMask = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        vector<int> B;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) B.push_back(i);
        if (!taskIndependent(B)) continue;
        long long s = 0; for (int i : B) s += pen[i];
        if (s > bestSaved) { bestSaved = s; bestMask = mask; }
    }
    cout << "    best independent set = {";
    bool first = true;
    for (int i = 0; i < n; ++i) if (bestMask >> i & 1) { if (!first) cout << ", "; cout <<
tname[i]; first = false; }
    cout << "},  weight saved = " << bestSaved
         << ",  minimum penalty = " << totalPen - bestSaved << "\n";
    cout << "    greedy matches the optimum: "
         << (bestSaved == saved ? "YES" : "NO") << "\n\n";

    cout << "  stress test, 200000 random 8-task instances:\n";
    mt19937 rng(20260919);
    int mismatches = 0;
    for (int t = 0; t < 200000; ++t) {
        int m = 8;
        DEADLINE.assign(m, 0);
        vector<long long> w(m);
        for (int i = 0; i < m; ++i) { DEADLINE[i] = 1 + (int)(rng() % m); w[i] = 1 + (long
long)(rng() % 100); }
        vector<string> nm(m, "x");
        vector<int> G = matroidGreedy(m, w, taskIndependent, false, nm);
        long long gs = 0; for (int i : G) gs += w[i];
        long long bs = -1;
        for (int mask = 0; mask < (1 << m); ++mask) {
            vector<int> B;
```

Kode 9.5 (Lanjutan) *Greedy(M,w) ditulis sekali terhadap orakel kebebasan yang abstrak, lalu diterapkan pada matroid penjadwalan tugas (dengan pemeriksaan brute force dan uji beban atas 200 000 contoh kasus) dan pada matroid grafik, tempat ia menjelma menjadi algoritma Kruskal. Kompilasi dengan g++ -std=c++17 -O2 -o matroid matroid.cpp lalu jalankan ./matroid.*

```
            for (int i = 0; i < m; ++i) if (mask >> i & 1) B.push_back(i);
            if (!taskIndependent(B)) continue;
            long long s = 0; for (int i : B) s += w[i];
            bs = max(bs, s);
        }
        if (gs != bs) ++mismatches;
    }
    cout << "    greedy differed from the optimum in " << mismatches << " of 200000
instances\n\n";

    // ------------------ contoh kasus 2: matroid grafik ---------------
    cout << "=== Instance 2: the same GREEDY on a graphic matroid = Kruskal ===\n";
    struct Edge { int u, v; long long w; string nm; };
    vector<Edge> E = {
        {0,1,4,"a-b"}, {0,7,8,"a-h"}, {1,2,8,"b-c"}, {1,7,11,"b-h"}, {2,3,7,"c-d"},
        {2,8,2,"c-i"}, {2,5,4,"c-f"}, {3,4,9,"d-e"}, {3,5,14,"d-f"}, {4,5,10,"e-f"},
        {5,6,2,"f-g"}, {6,7,1,"g-h"}, {6,8,6,"g-i"}, {7,8,7,"h-i"}
    };
    int V = 9;
    // kebebasan: himpunan sisi bebas jika dan hanya jika ia nirsiklus (sebuah hutan)
    auto acyclic = [&](const vector<int>& sel) {
        vector<int> parent(V); iota(parent.begin(), parent.end(), 0);
        function<int(int)> find_ = [&](int x) { while (parent[x] != x) x = parent[x] =
parent[parent[x]]; return x; };
        for (int i : sel) {
            int a = find_(E[i].u), b = find_(E[i].v);
            if (a == b) return false;                // sisi ini menutup sebuah siklus
            parent[a] = b;
        }
        return true;
    };
    // GREEDY memaksimumkan bobot, jadi untuk memperoleh pohon merentang MINIMUM kita
    // suapkan w'(e) = Wmax + 1 - w(e), yang membalik urutannya tanpa mengubah matroidnya
    long long Wmax = 0; for (auto& e : E) Wmax = max(Wmax, e.w);
    vector<long long> wp(E.size());
    vector<string> enm(E.size());
    for (size_t i = 0; i < E.size(); ++i) { wp[i] = Wmax + 1 - E[i].w; enm[i] = E[i].nm; }
    vector<int> T = matroidGreedy((int)E.size(), wp, acyclic, false, enm);
    long long treeWeight = 0; for (int i : T) treeWeight += E[i].w;
    sort(T.begin(), T.end(), [&](int a, int b) { return E[a].w < E[b].w; });
    cout << "  minimum spanning tree edges (in the order the weights fall):\n   ";
    for (int i : T) cout << " " << E[i].nm << "(" << E[i].w << ")";
    cout << "\n  total weight = " << treeWeight << ",  edges chosen = " << T.size()
         << " (a spanning tree of " << V << " vertices needs " << V - 1 << ")\n";
    return 0;
}
```

Contoh kasus keduanya mencetak:

```
=== Instance 2: the same GREEDY on a graphic matroid = Kruskal ===
  minimum spanning tree edges (in the order the weights fall):
    g-h(1) c-i(2) f-g(2) a-b(4) c-f(4) c-d(7) a-h(8) d-e(9)
  total weight = 37,  edges chosen = 8 (a spanning tree of 9 vertices needs 8)
```

Delapan sisi bagi sembilan simpul, bobot total 37. Anda akan berjumpa lagi dengan graf ini di Bab 11, tempat jawaban yang sama diperoleh dengan algoritma Kruskal yang ditulis dalam bentuknya yang biasa — dan tempat bukti kebenarannya, pada dasarnya, adalah Teorema 9.15 bagi matroid grafik.

9.9 Katalog Algoritma Rakus yang Masih Akan Datang

Metode rakus punya banyak penerapan lanjut, dan semuanya layak dipaparkan sekarang supaya Anda mengenalinya saat mereka tiba. **Tabel 9.11** adalah petanya.

Tabel 9.11 Algoritma rakus dalam buku ini. Lajur ketiganya adalah aturan lokalnya; lajur keempatnya adalah kewajiban pembuktiannya. Perhatikan betapa banyak entri di lajur keempat yang berbunyi "sebuah argumen penukaran".

Algoritma	Persoalan	Pilihan rakusnya	Mengapa ia aman	Bab
Greedy-Activity-Selector	himpunan aktivitas selaras terbesar	aktivitas yang paling awal selesai	Teorema 9.2, argumen penukaran	9 (di sini)
Fractional-Knapsack	muatan terbagi yang paling bernilai	barang dengan nilai per kg tertinggi	Teorema 9.5, argumen penukaran	9 (di sini)
Huffman	kode bebas awalan yang optimal	gabungkan dua lambang yang paling jarang	Lema 9.7 dan 9.8	9 (di sini)
Greedy(M, w)	himpunan bebas berbobot maksimum	unsur terberat yang menjaga kebebasan	Teorema 9.15	9 (di sini)
Penjadwalan tugas	total denda terkecil	tugas berdenda tertinggi yang menjaga kebebasan	Teorema 9.17 + 9.15	9 (di sini)
Kruskal	pohon merentang minimum	sisi teringan yang tak menutup siklus	matroid grafik, Teorema 9.15	11
Prim	pohon merentang minimum	sisi teringan yang meninggalkan pohon yang sudah terbangun	sifat potongan, sebuah argumen penukaran	11
Dijkstra	lintasan terpendek sumber tunggal, bobot taknegatif	simpul belum-dikunjungi dengan jarak sementara terkecil	argumen penukaran atas lintasan	13
Pengurutan topologis (Kahn)	urutan linear yang menghormati sebuah DAG	sembarang simpul yang tak lagi punya pendahulu	setiap DAG punya simpul semacam itu	11

Tiga di antaranya pantas disinggung sekarang.

Algoritma Kruskal (Bab 11) mengurutkan sisinya menurut bobot lalu menambahkan tiap sisi yang tidak menciptakan siklus. §9.8.8 sudah memperlihatkan kepada Anda bahwa ini secara harfiah adalah **Greedy**(M, w) pada matroid grafik, sehingga kebenarannya adalah Teorema 9.15 dan tak memerlukan argumen baru.

Algoritma Prim (Bab 11) menumbuhkan satu pohon tunggal, berulang kali menambahkan sisi teringan yang meninggalkannya. Ini *bukan* algoritma rakus matroid — ia tidak memproses sisinya dalam urutan bobot global — dan ia memerlukan buktinya sendiri. Bukti itu adalah argumen penukaran: diberi sembarang pohon merentang minimum, jika ia tidak memuat sisi teringan yang menyeberangi potongan yang sekarang, tambahkanlah sisi itu (sehingga tercipta sebuah siklus), hapuslah sisi siklus yang lebih berat yang menyeberangi potongan yang sama, lalu perhatikanlah bahwa bobotnya tidak naik. Bandingkan itu dengan langkah 5 Teorema 9.2 dan Anda akan melihat keempat gerakan yang sama.

Algoritma Dijkstra (Bab 13) berulang kali memfinalkan simpul belum-dikunjungi yang jarak sementaranya terkecil. Sekali lagi sebuah argumen penukaran: seandainya ada lintasan yang lebih pendek ke simpul itu, ia pasti meninggalkan wilayah yang sudah final pada suatu titik, dan simpul tempat ia keluar akan punya jarak sementara yang lebih kecil — kontradiksi.

Perhatikanlah bahwa argumen ini memakai **ketaknegatifan bobot sisinya** pada tepat satu tempat, dan itulah sebabnya Dijkstra gagal pada sisi bernilai negatif sehingga Bellman-Ford (sebuah program dinamis) diperlukan sebagai gantinya. Itulah keputusan rakus-lawan-pemrograman-dinamis §9.6, muncul lagi dalam kostum yang baru.

9.10 Profitina dalam Praktik

Profitina adalah platform visibilitas AI — sebuah produk **GEO** (*Generative Engine Optimization*), yang mengudara di `app.profitina.com` — yang mengukur seberapa siap situs web sebuah merek untuk *dikutip oleh mesin penjawab AI*, dari isyarat yang nyata seperti keterjangkauan isi, data terstruktur, dan otoritas teknis, lalu membantu memperbaiki apa yang ditemukannya. Tiga bagian produk itu merupakan persoalan rakus persis sejenis yang dibicarakan bab ini. Kita paparkan *bentuk* tiap persoalannya dan *kelas* solusinya; isi perut penilaiannya bersifat hak milik dan bukan pokoknya di sini.

1. Penjadwal pemindaian ulangnya adalah persoalan pemilihan aktivitas. Tiap merek di platform itu dipindai ulang secara otomatis pada irama yang bergantung pada tingkat langganannya. Sebuah pemindaian menempati satu pekerja selama selang yang terbatas, dan satu pekerja hanya dapat menjalankan satu pemindaian pada satu waktu — persis "penggunaan sumber daya bersama secara eksklusif" pada §9.4.1. Pada malam yang sibuk, pemindaian yang diminta melebihi pekerja yang tersedia, sehingga sesuatu harus dibuang. Mengurutkan pemindaian yang tertunda menurut waktu masing-masing akan *selesai* lalu menyapunya secara rakus memaksimumkan cacah merek yang disegarkan, dalam $\Theta(n\lg n)$, dengan bukti Teorema 9.4 di belakangnya. Mengurutkannya menurut waktu masing-masing akan *mulai* — yaitu "siapa datang dahulu, dilayani dahulu" yang naif — adalah aturan mulai-terawal §9.4.3, dan ia terbukti lebih buruk.

2. Anggaran perayapan di dalam satu pemindaian adalah masalah ransel. Satu pemindaian punya anggaran halaman. Tiap halaman calon memakan sejumlah waktu ambil-dan-urai serta menyumbang sejumlah nilai diagnostik. Halaman tak dapat diambil setengah-setengah, jadi ini adalah ransel **0-1** §9.6.4 dan kerakusan *keliru* — dan justru itulah jebakannya: memeringkat halaman menurut nilai per milidetik lalu mengisi anggarannya memberi jawaban yang biasanya meleset satu-dua persen dan kadang-kadang jauh lebih buruk, sebagaimana diperlihatkan **Tabel 9.6** dan uji beban §9.6.5. Pilihan rekayasa yang jujur adalah sebuah program dinamis ketika anggarannya kecil, atau jawaban rakusnya *ditambah* batas pecahan Teorema 9.5 sebagai ukuran tersurat atas berapa banyak nilai yang mungkin terlewat.

3. Lumbung analitik historisnya adalah persoalan pemampatan. Papan pantaunya menyimpan tren skor mingguan per merek dan menyediakan ekspor CSV, sehingga platform itu menumpuk deretan panjang kode status dan kode peristiwa yang sangat berulang. Itu sebaran frekuensi yang condong, dan di situlah penyandian Huffman memperoleh penghematan 20%–90% yang disebut di §9.7. §9.7 adalah seluruh metodenya, dan perjalanan bolak-balik yang diverifikasi di **Kode 9.4** adalah sifat yang penting: log termampat yang tak dapat diuraikan dengan persis tidak ada gunanya.

> **PRODUKNYA MENASIHATI; PENGGUNANYA MEMUTUSKAN.** Penjadwal yang membuang pemindaian, dan anggaran perayapan yang melewati halaman, dua-duanya mengubah apa yang dilihat pemilik merek. Tak satu pun boleh disajikan sebagai jaminan kelengkapan. Dan setiap pemindaian atas situs yang menyentuh data pribadi harus menghormati UU PDP 27/2022 — yang dalam bahasa penjadwalan berarti bahwa permintaan subjek data punya

> **tenggat**, dan melewatkannya membawa **denda**: matroid penjadwalan tugas §9.8.6, dengan dendanya bersifat hukum alih-alih uang.

9.10.1 Studi kasus yang dikerjakan: Profitina Laundry

Contoh yang dapat dijalankan selalu **Profitina Laundry**, sebuah usaha dua mesin di Surabaya. **Kode 9.6** menggarap satu hari kerja di tempat itu, dan hari itu memuat ketiga persoalannya.

Bagian A — pengeringnya. Tokonya punya satu pengering industri dan sebelas permintaan pemesanan. Pemesanan mana yang harus diterimanya? Ini persis pemilihan aktivitas. Programnya menjalankan aturan yang sudah terbukti optimal dan dua aturan yang mungkin saja lebih disukai pemiliknya:

```
  earliest finish (greedy, proved optimal) -> 4 bookings served
      Pak Anto 07:30-09:00, Warung Bu Tin 09:30-11:30, Salon Ayu 12:00-14:00, Klinik Sehat
14:00-16:00
  first come, first served                 -> 3 bookings served
      Hotel Melati 07:00-10:00, Bu Wati 10:00-12:30, Pak Hendra 12:30-14:30
  longest booking first                    -> 3 bookings served
      Bu Sri 08:00-11:00, Kos Merdeka 11:00-13:30, Klinik Sehat 14:00-16:00

  brute force over all 2^11 = 2048 subsets: the maximum is 4 bookings
```

Empat pelanggan alih-alih tiga — kenaikan 33% pada cacah yang terlayani, hanya dengan mengubah satu baris pembanding pengurutannya. “Siapa datang dahulu, dilayani dahulu” terasa adil dan kehilangan satu pelanggan setiap hari. “Pemesanan terpanjang dahulu” terasa menguntungkan — muatan lebih lama, tagihan lebih besar — dan juga kehilangan satu pelanggan.

Bagian B — log statusnya. Pengeringnya memancarkan satu karakter status per menit: `R` berjalan, `I` menganggur, `W` menunggu operator, `E` galat, `C` pendinginan, `S` pemerasan, `D` pembuangan air. Sehari itu berarti 9 000 karakter, dan sambungan naik tokonya lambat. Sandikan dengan Huffman:

```
  9000 status characters, 7 distinct codes
    code       count    codeword      bits count x bits
       R        7580           1         1          7580
       S         356         011         3          1068
       C         355         001         3          1065
       I         355         010         3          1065
       D         118       00010         5           590
       E         118       00011         5           590
       W         118        0000         4           472
  B(T) = 12430 bits
  plain 8-bit log      : 72000 bits
  Huffman-coded log    : 12430 bits
  saving               : 82.74%
  round trip           : IDENTICAL
```

Penghematan 82,7%, dan lognya terurai kembali bita demi bita identik. Perhatikan mengapa penghematannya sebesar itu: `R` menyumbang 7580 dari 9000 karakternya, dan ia memperoleh kata sandi **satu bit**. Sebarannya amat condong, dan kecondonganlah yang diubah Huffman menjadi pemampatan. Perhatikan pula bahwa `W`, `D`, dan `E` semuanya muncul 118 kali, namun `W` memperoleh empat bit sedangkan `D` dan `E` memperoleh lima — seri antara frekuensi yang sama itu diputuskan oleh urutan penggabungannya, dan pohon mana pun di antaranya sama optimalnya.

Bagian C — van pengantarannya. Cucian yang sudah selesai berangkat dengan satu van berbatas 60 kg. Lima paket sedang menunggu, masing-masing dengan biaya antarnya. Paket tak dapat dibelah. Ini ransel 0-1, jadi aturan kerapatannya *keliru*:

```
greedy by fee per kg : P1 + P2 + P4   = 615
true optimum (DP)    : P1 + P2 + P5   = 683
greedy leaves 68 on the table (10.0% of the optimum)
if the parcels were bulk detergent that CAN be split, greedy would
be optimal and would earn 758.00
```

Kerakusan mengambil P4 karena ia seri dengan P3 pada biaya per kilogram di 11.0 dan lebih ringan; optimumnya justru mengambil P5, yang kerapatannya *terburuk* di antara kelimanya pada 8.0, karena P5 berbobot tepat 25 kg yang ditinggalkan bebas oleh P1 dan P2. Kerakusan kehilangan 10% pendapatan antaran hari itu. Dan baris terakhirnya adalah pelajaran §9.6 dalam satu bilangan: kelima muatan yang sama, seandainya dapat dibagi, akan menghasilkan 758 di bawah aturan rakus yang sama — dan 758 itu adalah batas atas yang sah bagi jawaban tak-terbagi 683.

Kode 9.6 Profitina Laundry: pemilihan aktivitas atas pemesanan pengeringnya (dengan pemeriksaan *brute force*), pemampatan Huffman atas log status hari itu dengan perjalanan bolak-balik yang terverifikasi, dan van pengantarannya sebagai ransel 0-1 tempat kerakusan kehilangan 10% pendapatannya. Kompilasi dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp` lalu jalankan `./laundry`.

```cpp
// laundry.cpp — Profitina Laundry: tiga keputusan rakus dalam satu hari kerja.
//   Bagian A  satu pengering industri, banyak permintaan pemesanan: pemesanan mana
//             yang diambil (pemilihan aktivitas -- rakus terbukti optimal).
//   Bagian B  log status mesin sehari, dimampatkan dengan Huffman sebelum dikirim
//             lewat sambungan naik toko yang lambat.
//   Bagian C  van pengantaran, dengan paket utuh yang tak dapat dibelah dua
//             (ransel 0-1 -- rakus TIDAK optimal, dan programnya menunjukkannya).
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <queue>
#include <map>
#include <algorithm>
#include <cstdint>
using namespace std;

static string hhmm(int minutesFrom7am) {
    int t = 7 * 60 + minutesFrom7am;
    char buf[16];
    snprintf(buf, sizeof buf, "%02d:%02d", t / 60, t % 60);
    return string(buf);
}

// ------------------------------------------------ Bagian A: pengeringnya ---
struct Booking { string customer; int s, f; };   // menit setelah pukul 07:00

vector<int> earliestFinish(vector<Booking> b) {
    int n = (int)b.size();
    vector<int> idx(n);
    for (int i = 0; i < n; ++i) idx[i] = i;
    sort(idx.begin(), idx.end(), [&](int p, int q) {
        if (b[p].f != b[q].f) return b[p].f < b[q].f;
        return p < q;
    });
    vector<int> chosen;
    int freeAt = -1;
    for (int i : idx)
        if (b[i].s >= freeAt) { chosen.push_back(i); freeAt = b[i].f; }
    return chosen;
}
```

Kode 9.6 (Lanjutan) *Profitina Laundry: pemilihan aktivitas atas pemesanan pengeringnya (dengan pemeriksaan brute force), pemampatan Huffman atas log status hari itu dengan perjalanan bolak-balik yang terverifikasi, dan van pengantarannya sebagai ransel 0-1 tempat kerakusan kehilangan 10% pendapatannya. Kompilasi dengan g++ -std=c++17 -O2 -o laundry laundry.cpp lalu jalankan ./laundry.*

```
vector<int> firstComeFirstServed(vector<Booking> b) {
    int n = (int)b.size();
    vector<int> idx(n);
    for (int i = 0; i < n; ++i) idx[i] = i;
    sort(idx.begin(), idx.end(), [&](int p, int q) { return b[p].s < b[q].s; });
    vector<int> chosen; int freeAt = -1;
    for (int i : idx) if (b[i].s >= freeAt) { chosen.push_back(i); freeAt = b[i].f; }
    return chosen;
}

vector<int> longestFirst(vector<Booking> b) {
    int n = (int)b.size();
    vector<int> idx(n);
    for (int i = 0; i < n; ++i) idx[i] = i;
    sort(idx.begin(), idx.end(), [&](int p, int q) {
        return (b[p].f - b[p].s) > (b[q].f - b[q].s);
    });
    vector<int> chosen;
    for (int i : idx) {
        bool clash = false;
        for (int j : chosen) if (!(b[i].s >= b[j].f || b[j].s >= b[i].f)) clash = true;
        if (!clash) chosen.push_back(i);
    }
    return chosen;
}

int bruteForceDryer(const vector<Booking>& b) {
    int n = (int)b.size(), best = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        vector<int> c;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) c.push_back(i);
        bool ok = true;
        for (size_t i = 0; i + 1 < c.size() && ok; ++i)
            for (size_t j = i + 1; j < c.size() && ok; ++j)
                if (!(b[c[i]].s >= b[c[j]].f || b[c[j]].s >= b[c[i]].f)) ok = false;
        if (ok) best = max(best, (int)c.size());
    }
    return best;
}
```

Kode 9.6 (Lanjutan) *Profitina Laundry: pemilihan aktivitas atas pemesanan pengeringnya (dengan pemeriksaan brute force), pemampatan Huffman atas log status hari itu dengan perjalanan bolak-balik yang terverifikasi, dan van pengantarannya sebagai ransel 0-1 tempat kerakusan kehilangan 10% pendapatannya. Kompilasi dengan g++ -std=c++17 -O2 -o laundry laundry.cpp lalu jalankan ./laundry.*

```
// ------------------------------------------------ Bagian B: berkas lognya ---
struct Node {
    long long freq; char ch; bool leaf; int serial; Node *l, *r;
    Node(long long f, char c) : freq(f), ch(c), leaf(true),  serial(0), l(nullptr),
r(nullptr) {}
    Node(Node* a, Node* b)    : freq(a->freq + b->freq), ch(0), leaf(false), serial(0),
l(a), r(b) {}
};
struct Later {
    bool operator()(const Node* a, const Node* b) const {
        if (a->freq != b->freq) return a->freq > b->freq;
        return a->serial > b->serial;
    }
};
Node* build(const map<char, long long>& f) {
    priority_queue<Node*, vector<Node*>, Later> Q;
    int serial = 0;
    vector<pair<char, long long>> v(f.begin(), f.end());
    sort(v.begin(), v.end(), [](const pair<char,long long>& a, const pair<char,long long>&
b){ return a.second < b.second; });
    for (auto& p : v) { Node* n = new Node(p.second, p.first); n->serial = serial++;
Q.push(n); }
    if (Q.size() == 1) { Node* o = Q.top(); Q.pop(); Node* z = new Node(o, new
Node(0,'\0')); return z; }
    while (Q.size() > 1) {
        Node* x = Q.top(); Q.pop();
        Node* y = Q.top(); Q.pop();
        Node* z = new Node(x, y); z->serial = serial++; Q.push(z);
    }
    return Q.top();
}
void codes(const Node* n, string c, map<char, string>& out) {
    if (!n) return;
    if (n->leaf) { out[n->ch] = c.empty() ? "0" : c; return; }
    codes(n->l, c + "0", out); codes(n->r, c + "1", out);
}

// ------------------------------------------------------ Bagian C: vannya -----
struct Parcel { string id; int kg, fee; };

int vanGreedy(vector<Parcel> p, int cap, vector<string>& picked) {
    sort(p.begin(), p.end(), [](const Parcel& a, const Parcel& b) {
        return (double)a.fee / a.kg > (double)b.fee / b.kg;
    });
    int left = cap, got = 0;
    for (auto& x : p) if (x.kg <= left) { left -= x.kg; got += x.fee;
picked.push_back(x.id); }
    return got;
}
```

Kode 9.6 (Lanjutan) *Profitina Laundry: pemilihan aktivitas atas pemesanan pengeringnya (dengan pemeriksaan brute force), pemampatan Huffman atas log status hari itu dengan perjalanan bolak-balik yang terverifikasi, dan van pengantarannya sebagai ransel 0-1 tempat kerakusan kehilangan 10% pendapatannya. Kompilasi dengan g++ -std=c++17 -O2 -o laundry laundry.cpp lalu jalankan ./laundry.*

```
int vanOptimal(const vector<Parcel>& p, int cap, vector<string>& picked) {
    int n = (int)p.size();
    vector<vector<int>> best(n + 1, vector<int>(cap + 1, 0));
    for (int i = 1; i <= n; ++i)
        for (int c = 0; c <= cap; ++c) {
            best[i][c] = best[i - 1][c];
            if (p[i - 1].kg <= c)
                best[i][c] = max(best[i][c], best[i - 1][c - p[i - 1].kg] + p[i - 1].fee);
        }
    int c = cap;
    for (int i = n; i >= 1; --i)
        if (best[i][c] != best[i - 1][c]) { picked.push_back(p[i - 1].id); c -= p[i -
1].kg; }
    reverse(picked.begin(), picked.end());
    return best[n][cap];
}

double vanFractional(vector<Parcel> p, int cap) {
    sort(p.begin(), p.end(), [](const Parcel& a, const Parcel& b) {
        return (double)a.fee / a.kg > (double)b.fee / b.kg;
    });
    double got = 0; int left = cap;
    for (auto& x : p) {
        if (left <= 0) break;
        if (x.kg <= left) { got += x.fee; left -= x.kg; }
        else { got += (double)x.fee * left / x.kg; left = 0; }
    }
    return got;
}

int main() {
    cout << "======== PART A: one industrial dryer, eleven booking requests ========\n";
    vector<Booking> b = {
        {"Bu Sri",    60, 240}, {"Pak Anto",  30, 120}, {"Hotel Melati", 0, 180},
        {"Warung Bu Tin", 150, 270}, {"Pak Yudi", 120, 300}, {"Bu Wati", 180, 330},
        {"Kos Merdeka", 240, 390}, {"Salon Ayu", 300, 420}, {"Pak Hendra", 330, 450},
        {"Bu Lastri", 390, 480}, {"Klinik Sehat", 420, 540}
    };
    int n = (int)b.size();
    cout << setw(16) << "customer" << setw(10) << "start" << setw(10) << "finish"
         << setw(12) << "minutes\n";
    for (auto& x : b)
        cout << setw(16) << x.customer << setw(10) << hhmm(x.s) << setw(10) << hhmm(x.f)
             << setw(12) << x.f - x.s << "\n";
```

Kode 9.6 (Lanjutan) *Profitina Laundry: pemilihan aktivitas atas pemesanan pengeringnya (dengan pemeriksaan brute force), pemampatan Huffman atas log status hari itu dengan perjalanan bolak-balik yang terverifikasi, dan van pengantarannya sebagai ransel 0-1 tempat kerakusan kehilangan 10% pendapatannya. Kompilasi dengan g++ -std=c++17 -O2 -o laundry laundry.cpp lalu jalankan ./laundry.*

```
    struct { const char* name; vector<int> (*fn)(vector<Booking>); } rules[] = {
        { "earliest finish (greedy, proved optimal)", earliestFinish },
        { "first come, first served                ", firstComeFirstServed },
        { "longest booking first                   ", longestFirst },
    };
    cout << "\n";
    for (auto& R : rules) {
        vector<int> c = R.fn(b);
        sort(c.begin(), c.end(), [&](int p, int q) { return b[p].f < b[q].f; });
        cout << "  " << R.name << " -> " << c.size() << " bookings served\n      ";
        for (size_t i = 0; i < c.size(); ++i) {
            cout << b[c[i]].customer << " " << hhmm(b[c[i]].s) << "-" << hhmm(b[c[i]].f);
            if (i + 1 < c.size()) cout << ", ";
        }
        cout << "\n";
    }
    cout << "\n  brute force over all 2^" << n << " = " << (1 << n)
         << " subsets: the maximum is " << bruteForceDryer(b) << " bookings\n";

    cout << "\n======== PART B: compressing the day's status log ========\n";
    // Satu karakter per menit-mesin. R = berjalan, I = menganggur, W = menunggu
    // operator, E = galat, C = pendinginan, S = pemerasan, D = pembuangan air.
    string log;
    const char* pattern =
"RRRRRRRRSRRRRRRRRCIRRRRRRRRSRRRRRRRRCIRRRRRRRRDRRRRRRRRSWRRRRRRRRCIRRRRRRRRE";
    while (log.size() < 9000) log += pattern;
    log.resize(9000);
    map<char, long long> freq;
    for (char c : log) ++freq[c];
    Node* root = build(freq);
    map<char, string> code; codes(root, "", code);

    cout << "  " << log.size() << " status characters, " << freq.size() << " distinct
codes\n";
    cout << setw(8) << "code" << setw(12) << "count" << setw(12) << "codeword"
         << setw(10) << "bits" << setw(14) << "count x bits\n";
    long long bits = 0;
    vector<pair<char,long long>> fv(freq.begin(), freq.end());
    sort(fv.begin(), fv.end(), [](const pair<char,long long>&a, const pair<char,long
long>&b){ return a.second > b.second; });
    for (auto& p : fv) {
        int d = (int)code[p.first].size();
        cout << setw(8) << p.first << setw(12) << p.second << setw(12) << code[p.first]
             << setw(10) << d << setw(14) << p.second * d << "\n";
        bits += p.second * d;
    }
    long long flat = (long long)log.size() * 8;
    cout << "  B(T) = " << bits << " bits\n";
    cout << "  plain 8-bit log      : " << flat << " bits\n";
```

Kode 9.6 (Lanjutan) *Profitina Laundry: pemilihan aktivitas atas pemesanan pengeringnya (dengan pemeriksaan brute force), pemampatan Huffman atas log status hari itu dengan perjalanan bolak-balik yang terverifikasi, dan van pengantarannya sebagai ransel 0-1 tempat kerakusan kehilangan 10% pendapatannya. Kompilasi dengan g++ -std=c++17 -O2 -o laundry laundry.cpp lalu jalankan ./laundry.*

```
    cout << "  Huffman-coded log    : " << bits << " bits\n";
    cout << "  saving               : " << fixed << setprecision(2)
         << 100.0 * (flat - bits) / flat << "%\n";
    // perjalanan bolak-balik
    string bitstr;
    for (char c : log) bitstr += code[c];
    string back; const Node* cur = root;
    for (char x : bitstr) {
        cur = (x == '1') ? cur->r : cur->l;
        if (cur->leaf) { back += cur->ch; cur = root; }
    }
    cout << "  round trip           : " << (back == log ? "IDENTICAL" : "DIFFERENT") <<
"\n";

    cout << "\n======== PART C: the delivery van, whole parcels only ========\n";
    vector<Parcel> p = {
        {"P1", 14, 210}, {"P2", 21, 273}, {"P3", 30, 330},
        {"P4", 12, 132}, {"P5", 25, 200}
    };
    int cap = 60;
    cout << setw(8) << "parcel" << setw(8) << "kg" << setw(10) << "fee"
         << setw(16) << "fee per kg\n";
    for (auto& x : p)
        cout << setw(8) << x.id << setw(8) << x.kg << setw(10) << x.fee
             << setw(16) << setprecision(3) << (double)x.fee / x.kg << "\n";
    cout << "  van capacity = " << cap << " kg\n\n";
    vector<string> g, o;
    int gv = vanGreedy(p, cap, g);
    int ov = vanOptimal(p, cap, o);
    double fv2 = vanFractional(p, cap);
    cout << "  greedy by fee per kg : ";
    for (size_t i = 0; i < g.size(); ++i) { if (i) cout << " + "; cout << g[i]; }
    cout << "   = " << gv << "\n";
    cout << "  true optimum (DP)    : ";
    for (size_t i = 0; i < o.size(); ++i) { if (i) cout << " + "; cout << o[i]; }
    cout << "   = " << ov << "\n";
    cout << "  greedy leaves " << ov - gv << " on the table ("
         << setprecision(1) << 100.0 * (ov - gv) / ov << "% of the optimum)\n";
    cout << "  if the parcels were bulk detergent that CAN be split, greedy would\n";
    cout << "  be optimal and would earn " << setprecision(2) << fv2 << "\n";
    return 0;
}
```

9.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. Menulis algoritma rakus lalu menyebutnya benar. Kesalahan yang paling sering dengan selisih jauh. Kerakusan adalah harapan; argumen penukaranlah yang menjadikannya algoritma (§9.2.2). Jika Anda tak dapat menghasilkan argumennya, katakanlah begitu di dalam komentar dan simpanlah *brute force*-nya.

2. Menguji pada satu contoh kasus lalu menyimpulkan. Di §9.4.3, dua dari tiga aturan yang *keliru* memperoleh jawaban yang benar pada sebelas aktivitas contoh berjalan. Satu contoh kasus yang lolos bukanlah bukti yang lemah — ia sama sekali bukan bukti.

3. Mengurutkan menurut kunci yang keliru. Pemilihan aktivitas mengurutkan menurut waktu **selesai**. Mengurutkan menurut waktu mulai memberi 3 alih-alih 4 pada contoh berjalan;

mengurutkan menurut durasi gagal pada contoh kasus berisi tiga aktivitas. Kuncinya tidak dapat dipertukarkan dan tak ada gerak hati yang dapat menggantikan buktinya.

4. Memakai selang tertutup alih-alih setengah terbuka. Dengan $[s_i, f_i]$, aktivitas yang berakhir pukul 10:00 dan aktivitas yang mulai pukul 10:00 "bertindih" pada satu titik waktu dan algoritma Anda diam-diam menolak sepasang yang sebenarnya selaras. Tulislah $[s_i, f_i)$ lalu ujilah `s[m] >= f[k]`, jangan pernah `s[m] > f[k]`.

5. Menerapkan aturan kerapatan pada ransel 0-1. §9.6.4: ia mengembalikan 160 padahal 220 tersedia, dan dalam 20 000 contoh kasus acak ia tidak optimal 48% kalinya pada $n = 20$. Tanda khasnya adalah **kapasitas yang tak terpakai** di dalam jawaban rakusnya.

6. Mengira "cukup dekat" itu sebuah batas. Rata-rata nisbah rakus terhadap optimum di §9.6.5 adalah 0.993; yang terburuk 0.458. Rata-rata bukanlah jaminan. Jika Anda terpaksa mengirimkan kerakusan, kirimkanlah batas pecahannya bersama-sama.

7. Membangun kode panjang berubah yang tidak bebas awalan. Kode $a = 0$, $b = 01$, $c = 1$ memampatkan dengan indah dan tak dapat diuraikan (§9.7.3). Bangkitkanlah kata sandi selalu dari sebuah pohon, sehingga sifatnya bersifat struktural alih-alih sesuatu yang harus Anda ingat-ingat.

8. Melupakan kasus satu lambang pada implementasi Huffman. Berkas berisi bita yang serba sama menghasilkan satu daun tunggal pada kedalaman 0, kata sandi kosong, dan keluaran nol bit yang terurai menjadi tak ada apa pun. **Kode 9.4** memasangkan lambang tunggalnya dengan sebuah boneka; setiap pemampat yang sungguhan harus melakukan sesuatu yang setara.

9. Lupa menyimpan pohonnya. Bit yang tersandi tak bermakna tanpa buku sandinya. Berkas termampat harus membawa tabel frekuensinya atau bentuk pohonnya di dalam tajuknya, dan tajuk itu terhitung ke dalam ukuran termampatnya. 224 000 bit di §9.7.2 hanyalah muatannya saja.

10. Salah mengurutkan antrean prioritasnya. Huffman memerlukan antrean **min**. Di C++, `std::priority_queue` secara bawaan adalah *heap*-**maks**, jadi pembanding yang mengembalikan `a->freq < b->freq` memberi Anda dua lambang yang *paling sering* muncul dan sebuah kode yang seburuk-buruknya. **Kode 9.4** menulis `a->freq > b->freq` persis karena alasan ini.

11. Menganggap persoalan Anda sebuah matroid. Sifat turun-temurun biasanya mudah dan **sifat penukaran**-lah yang biasanya menjadi tempat matinya — sebagaimana terjadi pada pemilihan aktivitas (§9.8.3). Ujilah sifat penukarannya pada dua himpunan bebas kecil yang berbeda ukuran sebelum memanggil Teorema 9.15.

12. Memakai algoritma rakus matroid untuk meminimumkan. **Greedy**(M, w) memaksimumkan, dan definisinya menuntut bobot yang **positif tegas**. Untuk meminimumkan, transformasikan dengan $w'(x) = w_{\max} + 1 - w(x)$ sebagaimana dilakukan §9.8.8, lalu periksalah bahwa transformasinya sah — di sini ia sah hanya karena setiap himpunan bebas maksimal punya ukuran yang sama (Lema 9.12).

13. Mengacaukan maksimal dengan maksimum. $\{a_3, a_9, a_{11}\}$ tak dapat diperluas dan masih kurang satu dari optimumnya (§9.4.2). "Algoritma saya tak dapat menambahkan apa pun lagi" tidak membuktikan apa-apa di luar matroid.

14. Mengharapkan kerakusan bertahan ketika bobot ditambahkan. Memaksimumkan **cacah** aktivitas dapat diselesaikan secara rakus; memaksimumkan total **nilai**-nya tidak (Soal Latihan

9.15, tempat kerakusan mengembalikan 10 melawan optimum 100). Periksalah apa sebenarnya fungsi tujuannya.

9.12 Ringkasan Bab

Sebuah **algoritma rakus** menjatuhkan, pada tiap langkah, pilihan yang tampak terbaik pada saat itu, dinilai dengan aturan lokal yang tidak menengok masa depan maupun solusi submasalahnya, dan tak pernah meninjaunya ulang (§9.2.1). Karena itu ia cepat — khasnya $\Theta(n\lg n)$, didominasi sebuah pengurutan — dan khasnya **keliru**: tak ada apa pun dalam bentuk sebuah algoritma rakus yang membuatnya benar, sehingga **buktinya adalah seluruh pekerjaannya** (§9.2.2).

Memberi uang kembalian memperlihatkan pokoknya dalam bentuk mungil. Aturan "koin terbesar yang muat" keliru pada {1,3,4} — ia membayar 6 sebagai $4 + 1 + 1$ padahal $3 + 3$ sudah cukup, dan ia keliru pada 49 dari 200 nilai pertama — dan keliru pada {1,10,25}, tempat 30 berbiaya enam keping alih-alih tiga. Pada {1,5,10,25} dan pada pecahan Indonesia ia optimal pada setiap nilai (**Tabel 9.1**, **Kode 9.1**). Aturannya tak pernah berubah; hanya strukturnya yang berubah.

Pemilihan aktivitas (**Gambar 9.1**) meminta himpunan berukuran maksimum berisi aktivitas yang saling selaras pada satu sumber daya. Dari empat aturan yang masuk akal, mulai-terawal dan durasi-terpendek gagal pada contoh kasus berisi tiga aktivitas dan bentrokan-tersedikit gagal pada sembilan (**Kode 9.2**); **waktu selesai terawal** benar. Buktinya berparuh dua. **Substruktur optimal** (Teorema 9.1): sebuah A_{ij} optimal terbelah menjadi $A_{ik} \cup \{a_k\} \cup A_{kj}$ dengan $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$, dan kedua potongannya optimal, lewat potong-dan-tempel (**Gambar 9.2**) — yang menghasilkan program dinamis $\Theta(n^3)$ Persamaan (9.4) berupa $c[i,j] = \max_{a_k \in S_{ij}}\{c[i,k] + c[k,j] + 1\}$. **Sifat pilihan rakus** (Teorema 9.2): yang paling awal selesai di S_k terletak di dalam *suatu* solusi berukuran maksimum, dibuktikan dengan **argumen penukaran** — ambil sembarang A_k yang optimal, tukarkan yang pertama selesai dengan a_m, lalu pastikan hasilnya layak dan berukuran sama. Bersama pengamatan bahwa hanya tersisa satu submasalah (**Gambar 9.3**) keduanya memberi Akibat 9.3 dan karenanya **Recursive-Activity-Selector** (**Gambar 9.4**), yang penelusurannya pada contoh berjalan (**Gambar 9.5**) menghasilkan $\{a_1, a_4, a_8, a_{11}\}$, serta **Greedy-Activity-Selector** yang iteratif (**Gambar 9.6**). Keduanya $\Theta(n)$ pada masukan terurut, sehingga $\Theta(n\lg n)$ seluruhnya (**Tabel 9.3**).

Unsur-unsur strategi rakus adalah enam langkah pada **Tabel 9.4** dan dua bahan: **sifat pilihan rakus** dan **substruktur optimal**. Resep praktisnya adalah: nyatakan fungsi tujuannya, daftarlah setiap aturan lokal yang masuk akal, cobalah mematahkan masing-masing lewat pencarian tuntas pada contoh kasus kecil, cobalah argumen penukaran pada yang selamat, buktikan substruktur optimalnya dengan potong-dan-tempel, dan baru sesudah itu tulislah kodenya.

Dua masalah ransel memisahkan kedua tekniknya (**Tabel 9.7**). Keduanya punya substruktur optimal. Ransel **pecahan** punya sifat pilihan rakus, dan aturan kerapatannya optimal (Teorema 9.5) — pada **Tabel 9.5** ia memperoleh 240. Ransel **0-1** tidak: aturan yang sama memperoleh 160 padahal 220 tersedia (**Tabel 9.6**), kekurangan 27,3%, dan ia tidak optimal pada 48% contoh kasus acak berisi 20 barang (**Kode 9.3**). Bukti Teorema 9.5 memakai keterbagiannya pada tepat satu langkah, dan itulah seluruh perbedaannya.

Kode Huffman (**Gambar 9.7**) memampatkan dengan memberi karakter yang sering muncul kata sandi yang pendek. Berkas contoh yang berisi 100 000 karakter berbiaya 300 000 bit dengan panjang tetap dan 224 000 bit dengan panjang berubah, penghematan 25,33% (**Tabel 9.8**). Kode

harus **bebas awalan** atau ia tak dapat diuraikan, dan menggambarkannya sebagai **pohon** (**Gambar 9.8**) membuat sifat itu bersifat struktural. Pohon yang optimal bersifat **penuh** (Lema 9.6), dan biayanya adalah $B(T) = \sum_{c \in C} c\,.\,freq \cdot d_T(c)$ (Persamaan (9.8)), yang juga sama dengan jumlah label simpul dalamnya — yaitu total seluruh penggabungannya. **Huffman**(C) (**Gambar 9.9**) berulang kali menggabungkan dua benda yang paling jarang muncul; ditelusuri pada contoh berjalan (**Gambar 9.10**, **Tabel 9.9**) penggabungannya adalah 14,25,30,55,100, yang berjumlah 224. Kebenarannya adalah **Lema 9.7** (frekuensi yang rendah dan posisi yang dalam memang berjodoh: menukar keduanya mengubah biayanya sebesar $(a.freq - x.freq)(d(a) - d(x)) \geq 0$, seperti pada Persamaan (9.9)) dan **Lema 9.8** (melipat $\{x, y\}$ menjadi z menurunkan biayanya tepat sebesar $x.freq + y.freq$, seperti pada Persamaan (9.10)), yang memberi Teorema 9.9. Waktu jalannya $O(n\lg n)$ dengan *heap* biner-min (Teorema 9.10). **Kode 9.4** menyandikan dan menguraikan teks sungguhan 51,5% lebih kecil dengan perjalanan bolak-balik yang identik.

Matroid (**Gambar 9.11**) menjawab pertanyaan "kapan kerakusan otomatis benar?". Sebuah matroid adalah himpunan landasan berhingga beserta keluarga **turun-temurun** berisi himpunan bagian yang *bebas* yang memenuhi **sifat penukaran**: himpunan bebas yang lebih kecil selalu dapat ditumbuhkan memakai sebuah unsur dari himpunan bebas yang lebih besar. Akibat yang penting adalah **Lema 9.12** — di dalam matroid semua himpunan bebas yang maksimal berukuran sama, sehingga *maksimal* dan *maksimum* berimpit, dan kemaksimalan algoritma rakus yang diperoleh tanpa susah payah itu sudah memadai. **Greedy**(M, w) (**Pseudokode 9.1**) mengambil unsurnya dalam urutan bobot yang menurun, menyimpan tiap unsur yang menjaga kebebasannya, dan ia optimal bagi setiap matroid berbobot (Lema 9.13, 9.14, Teorema 9.15) dalam $O(n\lg n + nf(n))$. **Matroid grafik** (**Gambar 9.12**) menjadikannya algoritma Kruskal; **penjadwalan tugas berwaktu-satuan dengan tenggat dan denda** (**Gambar 9.13**) adalah sebuah matroid menurut Teorema 9.17, lewat uji $N_t(A) \leq t$ pada Lema 9.16, dan kerakusan mengembalikan himpunan awal $\{a_1, a_2, a_3, a_4, a_7\}$ untuk denda minimum 50 (**Tabel 9.10**, **Kode 9.5**). Teori matroid meliputi banyak kasus praktis tetapi **tidak** meliputi pemilihan aktivitas maupun penyandian Huffman, yang sifat penukarannya gagal.

Katalognya (**Tabel 9.11**) mendaftar apa yang akan datang: Kruskal dan Prim di Bab 11, Dijkstra di Bab 13, masing-masing dibuktikan dengan argumen penukaran yang kini sudah tiga kali Anda lihat.

9.12.1 Menengok ke depan

Bab 10 meninggalkan optimasi untuk sementara dan membangun kosakata **graf**: bagaimana merepresentasikannya, dan bagaimana berjalan di atasnya dengan pencarian melebar dan pencarian mendalam. Kedua penelusuran itu adalah alas yang menopang Bab 11 dan Bab 13.

Bab 11 lalu kembali kepada materi bab ini dengan bunga. **Pohon merentang minimum** adalah persoalan matroid yang paling kanonis, dan §9.8.8 sudah menjalankan algoritma Kruskal pada graf yang dipakai Bab 11 — jawabannya, bobot 37, sudah tersedia di sana untuk Anda cocokkan. **Algoritma Prim** adalah algoritma rakus yang *bukan* algoritma rakus matroid, dan menyaksikan bukti kebenarannya terkuak sebagai argumen penukaran akan menjadi saat §9.4.7 membayar dirinya sendiri. **Dijkstra** di Bab 13 melengkapi polanya, dan kegagalannya pada sisi bernilai negatif adalah peragaan terakhir sekaligus tertajam bagi §9.6: ketika sifat pilihan rakus pergi, tabelnya kembali.

9.13 Soal Latihan

9.1 Tinjaulah sistem koin $\{1,4,6\}$. Carilah, dengan tangan, nilai terkecil yang membuat aturan rakus uang kembalian tidak optimal. Berikan jawaban rakusnya dan jawaban optimalnya, lengkap dengan hitungannya, lalu jelaskan dalam satu kalimat langkah rakus mana yang melakukan kerusakannya.

9.2 (a) Buktikan bahwa jika 1 termasuk salah satu pecahannya, aturan rakus uang kembalian selalu berakhir dengan membayar nilainya secara tepat. (b) Berikan sebuah sistem tanpa 1 dan sebuah nilai yang membuat aturan rakusnya macet dengan uang masih terutang, padahal kembalian yang pas sebenarnya mungkin.

9.3 Jalankan aturan **waktu mulai terawal** dengan tangan pada sebelas aktivitas contoh berjalan (**Gambar 9.1**), satu langkah satu baris, dengan menampakkan pada tiap langkah aktivitas mana yang diambil dan mana yang dibuang. Pastikan ia memilih tiga aktivitas, lalu sebutkan satu keputusan buruknya.

9.4 Telusuri aturan **bentrokan tersedikit** dengan tangan pada contoh sanggahan berisi sembilan aktivitas yang dicetak **Kode 9.2**, yaitu $a_1 = [4,7)$, $a_2 = [9,13)$, $a_3 = [2,5)$, $a_4 = [1,4)$, $a_5 = [1,2)$, $a_6 = [6,10)$, $a_7 = [0,2)$, $a_8 = [7,15)$, $a_9 = [10,11)$. Hitunglah bentrokan tiap aktivitas pada tiap tahapnya, dan tunjukkan bahwa aturannya berakhir dengan tiga. Lalu carilah himpunan berisi empat dengan aturan selesai-terawal.

9.5 Pastikan bahwa $\{a_2, a_4, a_9, a_{11}\}$ dari **Gambar 9.1** adalah himpunan berisi aktivitas yang saling selaras berukuran 4, dengan memeriksa keenam pasangannya satu per satu. Lalu jelaskan dengan tepat mengapa **Greedy-Activity-Selector** justru mengembalikan $\{a_1, a_4, a_8, a_{11}\}$, dan mengapa hal itu tidak bertentangan dengan Teorema 9.2.

9.6 Buktikan bahwa aturan "berulang kali ambil aktivitas selaras dengan **waktu mulai terakhir**" juga optimal bagi persoalan pemilihan aktivitas. *Petunjuk:* jangan mengulang argumen penukarannya; cerminkan sumbu waktunya.

9.7 Satu pemroses harus menjalankan n pekerjaan, pekerjaan i memakan waktu p_i satuan, satu demi satu tanpa sela. **Waktu selesai** sebuah pekerjaan adalah saat ia rampung. Buktikan, dengan argumen penukaran atas dua pekerjaan yang bersebelahan, bahwa menjadwalkan pekerjaannya menaik tak turun menurut p_i (**pekerjaan terpendek dahulu**) meminimumkan jumlah waktu selesainya. Lalu hitunglah urutan optimalnya dan jumlahnya untuk $p = \langle 7,2,5,1,4,3 \rangle$.

9.8 Bangunlah sebuah keluarga contoh kasus ransel 0-1 berisi **dua barang** yang padanya rakus-menurut-kerapatan memperoleh pecahan optimum yang dapat dibuat sekecil-kecilnya. Nyatakan nisbahnya sebagai fungsi parameter Anda.

9.9 Buktikan bahwa masalah ransel pecahan selalu punya solusi optimal yang di dalamnya **paling banyak satu** barang diambil secara pecahan. Lalu jelaskan apa yang dikatakan hal ini tentang hubungan antara optimum pecahan dan optimum 0-1.

9.10 Bangunlah kode Huffman bagi delapan frekuensi $\langle 1,1,2,3,5,8,13,21 \rangle$ — bilangan Fibonacci. Daftarlah penggabungannya, berikan kedalaman tiap karakternya, hitunglah $B(T)$, dan bandingkan dengan kode panjang tetap. Perikan bentuk pohonnya dalam satu kalimat, dan sebutkan sifat barisan Fibonacci apa yang menyebabkannya.

9.11 Buktikan bahwa pada pohon yang dihasilkan **Huffman**, kedua karakter berfrekuensi terendah punya kata sandi berpanjang maksimum. Apakah kebalikannya benar — haruskah kedua karakter berkata sandi terpanjang merupakan dua yang frekuensinya terendah?

9.12 Andaikan karakternya diserahkan kepada Anda dalam keadaan **sudah terurut** menurut frekuensi. Tunjukkan cara mewujudkan **Huffman** dalam waktu $O(n)$ dengan dua antrean FIFO biasa dan tanpa *heap*. Nyatakan dan buktikan invarian yang membuatnya berhasil.

9.13 Rampatkan penyandian Huffman ke kata sandi atas abjad **terner** $\{0,1,2\}$. Pilihan rakusnya menjadi apa? Untuk nilai $n = |C|$ yang mana konstruksinya berhasil tanpa perubahan, dan apa yang harus dilakukan ketika tidak? Terapkan jawaban Anda pada frekuensi $\langle 5,9,12,13,16,45 \rangle$ dari **Gambar 9.7**.

9.14 Buktikan bahwa pohon biner penuh dengan ℓ daun punya tepat $\ell - 1$ simpul dalam. Berikan bukti induksi dan bukti pencacahan sekaligus.

9.15 Tiap aktivitas pada **Gambar 9.1** sekarang membawa sebuah **nilai**, dan kita menginginkan himpunan selaras yang total nilainya terbesar alih-alih yang ukurannya terbesar. Tunjukkan dengan contoh berisi tiga aktivitas bahwa aturan selesai-terawal tidak lagi optimal, lalu berikan program dinamis yang optimal. Berapa waktu jalannya?

9.16 Untuk himpunan berhingga S dengan $|S| = n$ dan bilangan bulat $0 \le k \le n$, misalkan $\mathcal{I} = \{A \subseteq S : |A| \le k\}$. Buktikan bahwa $M = (S, \mathcal{I})$ adalah sebuah matroid — ia disebut **matroid seragam** $U_{k,n}$. Apa yang dilakukan **Greedy**(M, w) padanya, dan apa yang kemudian dikatakan Teorema 9.15?

9.17 Pada contoh kasus penjadwalan tugas **Gambar 9.13**, ubahlah denda a_5 dari 30 menjadi 100, dengan segala yang lain tetap. Jalankan ulang **Greedy** dengan tangan, dengan memberikan uji kebebasannya pada tiap langkah. Nyatakan himpunan awal yang baru, jadwal yang baru, dan denda minimum yang baru.

9.18 *(Tugas perancangan.)* Profitina Laundry membeli pengering kedua. Pemesanannya tak berubah, tetapi kini dua di antaranya dapat berjalan bersamaan. Rancanglah sebuah algoritma rakus bagi cacah pemesanan terbanyak yang dapat dilayani dua mesin yang identik, nyatakan dengan tepat aturan untuk memutuskan sebuah pemesanan masuk ke mesin yang *mana*, lalu tulislah programnya beserta pemeriksaan *brute force*-nya. Apakah aturan selesai-terawal masih berhasil?

9.14 Penyelesaian Soal Latihan

9.1 Ujilah nilainya berurutan. Untuk 1,2,3 kerakusan membayar dengan keping 1 dan demikian pula optimumnya. $4 \to 4$ (satu keping) ✓. $5 \to 4 + 1$ (dua) dan tak ada keping tunggal bernilai 5 ✓. $6 \to 6$ (satu) ✓. $7 \to 6 + 1$ (dua) ✓. Sekarang 8:

- Rakus: koin terbesar ≤ 8 adalah 6; terutang $8 - 6 = 2$; terbesar ≤ 2 adalah 1; terutang 1; lalu 1; terutang 0. Koin yang dipakai: $6 + 1 + 1 = 8$, **tiga keping**.
- Optimum: $4 + 4 = 8$, **dua keping**.

Jadi nilai gagal yang terkecil adalah **8**. **Kode 9.1** memastikannya jika baris `audit("G", {1, 4, 6}, 200);` ditambahkan ke `main`: program melaporkan `smallest counterexample: amount = 8`, dengan jawaban rakus `6 + 1 + 1` (3 keping) melawan jawaban optimal `4 + 4` (2 keping), dan kerakusan keliru pada 65 dari 200 nilai. Langkah yang merusaknya adalah mengambil 6: ia menyisakan utang 2, yang pada sistem ini hanya dapat dibayar dengan dua keping 1, sedangkan mengambil koin yang lebih kecil, yaitu 4, menyisakan 4, yang terbayar dengan satu keping.

9.2 *(a)* Misalkan A nilai yang terutang. Pada tiap langkah, pecahan terbesar yang tidak melebihi sisa utangnya selalu ada, sebab 1 adalah salah satu pecahannya dan $1 \le$ sembarang sisa utang

yang positif. Tiap langkah menyusutkan sisa utangnya sekurang-kurangnya 1, dan sisa utangnya tak pernah menjadi negatif sebab kita hanya menyerahkan koin yang tidak melebihinya. Jadi setelah paling banyak A langkah sisa utangnya 0, dan pembayarannya tepat. ■

(b) Ambillah pecahan $\{3,5\}$ dan nilai $A = 9$. Kerakusan mengambil 5, menyisakan 4; koin terbesar ≤ 4 adalah 3, menyisakan 1; sekarang tak ada koin yang muat dan kerakusan macet dengan 1 masih terutang. Padahal $9 = 3 + 3 + 3$ dengan tepat. (Ini kegagalan pemaketan kantong Soal Latihan 7.18 dalam kostum yang lain.)

9.3 Datanya ada di **Tabel 9.2**.

- **Langkah 1.** Mulai terawal di antara kesebelasnya: a_3 dengan $s_3 = 0$. Ambil $a_3 = [0,6)$. Buang semua yang bertindih dengannya: $a_1 = [1,4)$ ✗, $a_2 = [3,5)$ ✗, $a_4 = [5,7)$ ✗ (karena $5 < 6$), $a_5 = [3,9)$ ✗, $a_6 = [5,9)$ ✗, $a_{10} = [2,14)$ ✗. Yang selamat: $a_7 = [6,10)$ (karena $6 \geq 6$), $a_8 = [8,11)$, $a_9 = [8,12)$, $a_{11} = [12,16)$.
- **Langkah 2.** Mulai terawal di antara yang selamat: a_7 dengan $s_7 = 6$. Ambil $a_7 = [6,10)$. Buang a_8 $(8 < 10)$ dan a_9 $(8 < 10)$. Yang selamat: a_{11}.
- **Langkah 3.** Ambil $a_{11} = [12,16)$. Tak ada yang tersisa.

Hasilnya $\{a_3, a_7, a_{11}\}$, **tiga** aktivitas, cocok dengan **Kode 9.2**. Keputusan buruknya adalah langkah 1: a_3 mulai paling awal tetapi menempati enam satuan waktu, memblokir enam aktivitas lain. Mulai lebih awal tidaklah sama dengan selesai lebih awal.

9.4 Mula-mula hitunglah bentrokan tiap aktivitasnya (dua aktivitas bentrok ketika keduanya bertindih); **Tabel 9.12** adalah hasilnya.

Tabel 9.12 Cacah bentrokan bagi sembilan aktivitas Soal Latihan 9.4, sebelum pemilihan apa pun dilakukan. Lima aktivitas seri pada minimum dua.

Aktivitas	Selang	Bentrok dengan	Cacah
a_1	[4,7)	a_3, a_6	2
a_2	[9,13)	a_6, a_8, a_9	3
a_3	[2,5)	a_1, a_4	2
a_4	[1,4)	a_3, a_5, a_7	3
a_5	[1,2)	a_4, a_7	2
a_6	[6,10)	a_1, a_2, a_8	3
a_7	[0,2)	a_4, a_5	2
a_8	[7,15)	a_2, a_6, a_9	3
a_9	[10,11)	a_2, a_8	2

- **Tahap 1.** Cacah minimumnya 2, dicapai oleh a_1, a_3, a_5, a_7, a_9. Implementasinya mengambil yang pertama menurut urutan indeks: $a_1 = [4,7)$. Buang a_1 dan bentrokannya a_3, a_6.
- **Tahap 2.** Yang selamat $a_2, a_4, a_5, a_7, a_8, a_9$. Hitung ulang: a_2 bentrok dengan $a_8, a_9 \to 2$; a_4 dengan $a_5, a_7 \to 2$; a_5 dengan $a_4, a_7 \to 2$; a_7 dengan $a_4, a_5 \to 2$; a_8 dengan $a_2, a_9 \to 2$; a_9 dengan $a_2, a_8 \to 2$. Semuanya seri pada 2; ambil $a_2 = [9,13)$. Buang a_8, a_9.
- **Tahap 3.** Yang selamat a_4, a_5, a_7, masing-masing bentrok dengan dua lainnya $\to 2$ masing-masing. Ambil $a_4 = [1,4)$. Buang a_5, a_7.
- Tak ada yang tersisa. Hasilnya $\{a_1, a_2, a_4\}$, **tiga**.

Sekarang aturan selesai-terawal. Terurut menurut waktu selesai: $a_5 = [1,2)$, $a_7 = [0,2)$, $a_4 = [1,4)$, $a_3 = [2,5)$, $a_1 = [4,7)$, $a_6 = [6,10)$, $a_9 = [10,11)$, $a_2 = [9,13)$, $a_8 = [7,15)$, dengan waktu selesai 2,2,4,5,7,10,11,13,15. Ambil a_5 $(f_5 = 2)$; berikutnya yang $s \geq 2$ adalah $a_3 = [2,5)$ $(f_3 = 5)$;

berikutnya yang $s \geq 5$ adalah $a_6 = [6,10)$ ($f_6 = 10$); berikutnya yang $s \geq 10$ adalah $a_9 = [10,11)$. Hasilnya $\{a_5, a_3, a_6, a_9\}$, **empat**. Konstruksi contoh sanggahannya sejenis yang diberikan di [5].

9.5 Keenam pasangannya. $a_2 = [3,5)$, $a_4 = [5,7)$, $a_9 = [8,12)$, $a_{11} = [12,16)$.

(a_2, a_4): $s_4 = 5 \geq f_2 = 5$ ✓. (a_2, a_9): $8 \geq 5$ ✓. (a_2, a_{11}): $12 \geq 5$ ✓. (a_4, a_9): $8 \geq 7$ ✓. (a_4, a_{11}): $12 \geq 7$ ✓. (a_9, a_{11}): $12 \geq 12$ ✓. Keenamnya selaras, berukuran 4 — dan 4 adalah maksimumnya, diverifikasi dengan *brute force* di **Kode 9.2**.

Greedy-Activity-Selector mengembalikan $\{a_1, a_4, a_8, a_{11}\}$ karena ia mengambil a_1 (yang paling awal selesai, $f_1 = 4$) alih-alih a_2 ($f_2 = 5$), dan sejak itu jalannya terpaksa. Tak ada pertentangan dengan Teorema 9.2: teoremanya mengatakan yang paling awal selesai terletak di dalam **suatu** solusi berukuran maksimum, dan memang a_1 terletak di dalam $\{a_1, a_4, a_8, a_{11}\}$. Ia tak pernah mengklaim bahwa setiap solusi berukuran maksimum memuat pilihan rakusnya, dan $\{a_2, a_4, a_9, a_{11}\}$ adalah pengingat tetap mengapa pernyataan yang lebih lemah itulah yang tepat.

9.6 Cerminkan sumbu waktunya. Petakan tiap aktivitas $[s_i, f_i)$ ke $(-f_i, -s_i]$, atau setara dengan itu, pilihlah tetapan M yang lebih besar daripada setiap waktu selesai lalu petakan $a_i = [s_i, f_i)$ ke $\hat{a}_i = [M - f_i,\ M - s_i)$. Tiga pengamatan melengkapi buktinya.

1. Pemetaan itu bijeksi dari contoh kasus ke contoh kasus, dan ia inversnya sendiri bila M yang sama dipakai.
2. Ia mempertahankan keselarasan. Dua aktivitas selaras jika dan hanya jika selangnya saling lepas; pencerminan adalah gerak kaku pada garis (sebuah pembalikan disusul pergeseran), dan gerak kaku mempertahankan kesalinglepasan. (Selang setengah terbuka terpetakan ke selang setengah terbuka dengan ketangan yang berlawanan, dan kasus batasnya beres sebab $s_i \geq f_j \Leftrightarrow M - s_i \leq M - f_j$.)
3. Ia mempertukarkan mulai dan selesai: awal $\hat{a}_i$ adalah $M - f_i$, jadi aktivitas dengan f_i **terbesar** punya awal **terkecil** sesudah pencerminan; dan selesai $\hat{a}_i$ adalah $M - s_i$, jadi aktivitas dengan s_i **terbesar** punya selesai **terkecil** sesudah pencerminan.

Karena itu "ambil aktivitas selaras dengan waktu mulai terakhir" pada contoh kasus aslinya persis sama dengan "ambil aktivitas selaras dengan waktu selesai terawal" pada contoh kasus yang tercermin, yang menurut Teorema 9.4 menghasilkan himpunan selaras berukuran maksimum di sana; dan menurut pengamatan 1 dan 2 himpunan yang bersesuaian pada contoh kasus aslinya bersifat selaras dan berukuran sama, karenanya maksimum. ■

9.7 *Penukarannya.* Misalkan σ sembarang jadwal dan andaikan dua pekerjaan yang **bersebelahan** tertukar urutannya: pekerjaan i berjalan tepat sebelum pekerjaan j dan $p_i > p_j$. Andaikan pekerjaan-pekerjaan sebelumnya selesai pada waktu t. Maka

- sebelum penukarannya: $C_i = t + p_i$ dan $C_j = t + p_i + p_j$, menyumbang $2t + 2p_i + p_j$;
- sesudah penukarannya: $C_j = t + p_j$ dan $C_i = t + p_j + p_i$, menyumbang $2t + 2p_j + p_i$.

Waktu selesai pekerjaan lain tak berubah, sebab kedua pekerjaan itu bersama-sama tetap menempati selang $[t, t + p_i + p_j)$. Perubahan totalnya adalah

$$(2t + 2p_j + p_i) - (2t + 2p_i + p_j) = p_j - p_i < 0,$$

yaitu penurunan yang tegas. Jadi setiap jadwal yang punya inversi bersebelahan dapat diperbaiki secara tegas, dan karena itu jadwal yang optimal tak punya inversi bersebelahan — yang berarti ia terurut menaik tak turun menurut p_i. ■

(Bukti kedua: jika pekerjaan $\sigma(k)$ adalah yang ke-k berjalan, waktu selesainya $\sum_{r \le k} p_{\sigma(r)}$, sehingga totalnya $\sum_k (n-k+1)\ p_{\sigma(k)}$: panjang pekerjaan pertama dicacah n kali, panjang yang kedua $n-1$ kali, dan seterusnya. Untuk meminimumkan jumlah hasil kali dua barisan, pasangkanlah pengali terbesar dengan panjang terkecil — dan itulah pekerjaan-terpendek-dahulu.)

Perhitungannya. Dengan $p = \langle 7,2,5,1,4,3 \rangle$ urutan terurutnya adalah $j_4(1), j_2(2), j_6(3), j_5(4), j_3(5), j_1(7)$, memberi waktu selesai 1,3,6,10,15,22 dan total $1+3+6+10+15+22 = 57$, rata-rata $57/6 = 9.5$. **Kode 9.7** memeriksanya terhadap seluruh $6! = 720$ urutan:

```
=== E1: one machine, minimize the sum of completion times ===
  job      :   j1   j2   j3   j4   j5   j6
  length   :    7    2    5    1    4    3
  SJF order:   j4   j2   j6   j5   j3   j1
    completion times:    1    3    6   10   15   22
    sum = 57,  average = 9.5000
  all 720 orders examined: best = 57, worst = 97
  SJF attains the best: YES
```

9.8 Misalkan $K \ge 2$ sebuah parameter. Ambil kapasitas $W = K$ dan dua barang:

- barang 1: $w_1 = 1$, $v_1 = 2$, kerapatan 2;
- barang 2: $w_2 = K$, $v_2 = K$, kerapatan 1.

Kerakusan mengambil barang 1 (kerapatannya lebih tinggi), menyisakan kapasitas $K - 1 < K$, sehingga barang 2 tak lagi muat; nilai kerakusannya 2. Optimumnya mengambil barang 2 sendirian, bernilai K. Nisbahnya

$$\frac{\text{rakus}}{\text{optimum}} = \frac{2}{K} \ \to\ 0.$$

Jadi rakus-menurut-kerapatan **tidak** punya jaminan aproksimasi yang tetap pada ransel 0-1. (Perbaikan bakunya — ambil mana yang lebih baik antara solusi rakusnya dan barang tunggal paling bernilai yang muat — memang memberi jaminan 1/2; lihat [3].)

9.9 Urutkan barangnya menurut kerapatan. Andaikan sebuah solusi optimal mengambil dua barang i dan j secara pecahan tegas, $0 < x_i < 1$ dan $0 < x_j < 1$, dan tanpa kehilangan keumuman $v_i/w_i \ge v_j/w_j$. Pilihlah

$$\varepsilon = \min\{\,(1-x_i)w_i,\ x_j w_j\,\} > 0,$$

lalu pindahkan ε satuan bobot dari barang j ke barang i: naikkan x_i sebesar ε/w_i dan turunkan x_j sebesar ε/w_j. Bobot totalnya tak berubah, kedua pecahannya tetap di [0,1], dan nilainya berubah sebesar

$$\varepsilon\left(\frac{v_i}{w_i} - \frac{v_j}{w_j}\right) \ge 0.$$

Jadi solusi barunya tetap optimal, dan menurut pemilihan ε sekurang-kurangnya satu dari x_i, x_j telah menjadi 0 atau 1 — cacah barang yang pecahan tegas berkurang sekurang-kurangnya satu. Mengulanginya paling banyak n kali menyisakan paling banyak satu barang pecahan. ■

Akibatnya. Optimum pecahannya berselisih dari optimum 0-1-nya paling banyak sebesar nilai satu barang yang terambil sebagian itu, dan ia selalu ≥ optimum 0-1-nya sebab setiap solusi 0-1 merupakan solusi pecahan yang sah. Jadi nilai pecahan yang dihitung dalam $O(n\lg n)$ adalah batas atas bagi optimum 0-1-nya, ketat sampai dalam satu barang — dan persis itulah batas yang dipakai algoritma *branch-and-bound* untuk memangkas [3].

9.10 Penggabungannya, dari **Kode 9.7**:

```
=== E2: Huffman with Fibonacci frequencies ===
  frequencies:    1    1    2    3    5    8   13   21
    merge   1 +   1 =   2
    merge   2 +   2 =   4
    merge   3 +   4 =   7
    merge   5 +   7 =  12
    merge   8 +  12 =  20
    merge  13 +  20 =  33
    merge  21 +  33 =  54
  character :    c1    c2    c3    c4    c5    c6    c7    c8
  frequency :     1     1     2     3     5     8    13    21
  depth     :     7     7     6     5     4     3     2     1
  B(T) = 132 bits for 54 characters, i.e. 2.4444 bits per character
  a fixed-length code would need 3 bits each, total 162 bits
```

Periksalah $B(T)$ dengan tangan: $1(7) + 1(7) + 2(6) + 3(5) + 5(4) + 8(3) + 13(2) + 21(1) = 7 + 7 + 12 + 15 + 20 + 24 + 26 + 21 = 132$ ✓. Penghematannya atas kode panjang tetap adalah $162 - 132 = 30$ bit, atau 18.5%.

Pohonnya berbentuk **ulat bulu**: pada tiap arasnya akar punya satu anak daun dan satu anak dalam, sehingga kedalamannya berjalan 1,2,3,4,5,6,7,7 — sencong yang paling mungkin bagi sebuah pohon biner. Penyebabnya adalah sifat yang mendefinisikan bilangan Fibonacci, $F_k = F_{k-1} + F_{k-2}$, beserta identitas yang diturunkannya, $F_1 + F_2 + \cdots + F_k = F_{k+2} - 1$. Setelah k karakter pertama tergabung menjadi satu simpul, frekuensi simpul itu adalah $F_{k+2} - 1$. Dua daun berikutnya yang menunggu di antrean adalah F_{k+1} dan F_{k+2}, dan frekuensi simpul itu terletak di antara keduanya: $F_{k+1} \leq F_{k+2} - 1 < F_{k+2}$. Jadi dua benda terkecil selalu simpul itu dan daun berikutnya, dan simpul itu langsung digabungkan lagi. Periksa pada penggabungan di atas: label barunya, 2,4,7,12,20,33,54, tepat sama dengan $F_{k+2} - 1$ untuk $k = 2, \ldots, 8$. Frekuensi Fibonacci adalah kasus ekstrem bagi kedalaman pohon Huffman.

9.11 Huffman menggabungkan x dan y pada iterasi pertamanya, sehingga keduanya menjadi daun bersaudara. Tinggal ditunjukkan bahwa tak ada daun yang lebih dalam daripada mereka. Kita buktikan pernyataan yang lebih kuat: *simpul yang dikeluarkan dari antrean lebih awal letaknya sekurang-kurangnya sama dalam di pohon akhir dengan simpul yang dikeluarkan belakangan.* (Kita menganggap, seperti **Kode 9.4**, bahwa frekuensi yang seri dimenangkan oleh simpul yang diciptakan lebih dulu.)

1. Frekuensi yang dikeluarkan dari antrean tak pernah menurun sepanjang jalannya algoritma. Pada tiap langkah dua benda terkecil dikeluarkan, dan satu-satunya benda yang ditambahkan adalah jumlah keduanya, yang sekurang-kurangnya sebesar masing-masing; jadi segala yang dikeluarkan belakangan sekurang-kurangnya sebesar segala yang dikeluarkan lebih awal. (Inilah invarian yang dibuktikan di Soal Latihan 9.12.)
2. Misalkan u dikeluarkan pada penggabungan ke-i dan v pada penggabungan yang lebih belakang, ke-$j > i$, dan misalkan p_u serta p_v adalah simpul dalam yang diciptakan kedua penggabungan itu — induk u dan induk v. Maka p_u diciptakan sebelum p_v dan, menurut langkah 1, $p_u.freq \leq p_v.freq$. Dengan aturan pemecah seri di atas, p_u karenanya dikeluarkan sebelum p_v (atau p_u dikeluarkan dan p_v adalah akarnya, yang tak pernah dikeluarkan).

3. Sekarang berargumenlah dengan induksi mundur dari akarnya. Kedalaman setiap simpul yang dikeluarkan sama dengan kedalaman induknya ditambah satu. Menurut langkah 2 induk-induknya berurutan dengan cara yang sama seperti anak-anaknya, jadi $\text{depth}(p_u) \geq \text{depth}(p_v)$, dan menambahkan satu pada kedua ruas memberi $\text{depth}(u) \geq \text{depth}(v)$.

Karena x dan y adalah dua benda pertama yang dikeluarkan, tak ada daun yang lebih dalam daripada mereka, dan kata sandi mereka berpanjang maksimum. ■

Kebalikannya salah. Pada **Gambar 9.8**(b), kata sandi terpanjangnya dimiliki e (frekuensi 9) dan f (frekuensi 5) — dan itu memang dua yang terendah. Tetapi tinjaulah frekuensi $\langle 1,1,1,1 \rangle$: **keempat** karakternya memperoleh kedalaman 2, sehingga karakter berkata sandi terpanjang adalah semuanya, bukan hanya dua. Lebih tajam lagi, dengan $\langle 1,1,2,2 \rangle$ **Huffman** menghasilkan pohon seimbang yang di dalamnya kedua karakter berfrekuensi 2 juga berkedalaman maksimum. Jadi kedalaman maksimum tidak menunjuk khusus kepada dua frekuensi terkecil.

9.12 Pakailah dua antrean FIFO.

- Q_1 memuat n daunnya dalam urutan frekuensi yang menaik tak turun — sudah diserahkan kepada kita dalam keadaan terurut.
- Q_2 bermula kosong lalu menerima tiap simpul dalam yang baru tercipta di **belakangnya**.

`Extract-Min` menjadi: bandingkan bagian depan Q_1 dan Q_2 (dengan memperlakukan antrean kosong sebagai $+\infty$) lalu keluarkan yang lebih kecil. Masing-masing dari $2(n-1)$ pengeluaran dan $n-1$ penyisipannya berbiaya $O(1)$, jadi seluruh pembangunannya $O(n)$.

Invarian yang membuatnya berhasil: Q_2 **selalu terurut menaik tak turun menurut frekuensi.** Bukti. Misalkan $z_1, z_2, \ldots$ adalah simpul dalamnya dalam urutan penciptaan, dengan frekuensi $g_1, g_2, \ldots$. Kita tunjukkan $g_{k+1} \geq g_k$ untuk setiap k.

- Pada langkah k dua benda terkecil yang tersedia, katakanlah berfrekuensi $u \leq v$, dikeluarkan, sehingga $g_k = u + v$. Setiap benda yang tertinggal berfrekuensi sekurang-kurangnya v, sebab u dan v adalah dua yang terkecil.
- Pada langkah $k+1$ dua benda dikeluarkan, katakanlah berfrekuensi $u' \leq v'$. Masing-masing adalah benda yang tertinggal dari langkah k (frekuensi $\geq v$) atau simpul baru z_k itu sendiri (frekuensi $u + v \geq v$). Jadi $u' \geq v$ dan $v' \geq v$.
- *Kasus 1: tak satu pun adalah z_k.* Maka $g_{k+1} = u' + v' \geq v + v \geq u + v = g_k$.
- *Kasus 2: salah satunya adalah z_k.* Maka $g_{k+1} = g_k + (\text{yang lain}) \geq g_k$, sebab frekuensi tidak negatif.

Bagaimanapun juga $g_{k+1} \geq g_k$. ■

Karena Q_1 terurut menurut hipotesisnya dan Q_2 terurut menurut invariannya, minimum globalnya selalu berada di depan salah satu dari keduanya. Inilah implementasi baku $O(n)$ bagi konstruksi Huffman [6].

9.13 *Pilihan rakusnya menjadi: gabungkan **tiga** benda yang paling jarang muncul menjadi satu simpul beranak tiga*, berlabel 0, 1, dan 2. Rumus biayanya tak berubah, $B(T) = \sum_c c.freq \cdot d_T(c)$, tetapi $d_T(c)$ kini mencacah digit terner ("trit") alih-alih bit.

Kapan ia berhasil tanpa perubahan. Pohon terner penuh punya 3 anak per simpul dalam. Tiap penggabungan membuang 3 benda dan menambahkan 1, kehilangan bersih 2, dan kita harus berakhir dengan tepat 1 benda. Jadi kita memerlukan $n - 2m = 1$ untuk suatu cacah penggabungan bilangan bulat m, yaitu n harus **ganjil**. Secara umum, bagi abjad berukuran k syaratnya adalah $n \equiv 1 \pmod{k-1}$.

Apa yang harus dilakukan ketika tidak. Tambahkan karakter boneka berfrekuensi 0 sampai n ganjil (secara umum, sampai $n \equiv 1 \pmod{k-1}$). Karakter berfrekuensi nol tidak menyumbang apa pun kepada $B(T)$, sehingga kode bagi karakter yang sungguhan tetap optimal; kata sandi bonekanya sekadar dibuang sesudahnya. Yang genting, bonekanya harus ditambahkan **sebelum** penggabungan pertama, supaya ia terpakai habis oleh penggabungan yang pertama (yang termurah) dan tidak menyimpangkan penggabungan berikutnya.

Diterapkan pada $\langle 5,9,12,13,16,45 \rangle$. Di sini $n = 6$, yang genap, jadi tambahkan satu boneka berfrekuensi 0, memberi $\langle 0,5,9,12,13,16,45 \rangle$ dengan $n = 7$. * Gabungkan tiga yang terkecil, $0 + 5 + 9 = 14$. Antrean: 12,13,14,16,45. * Gabungkan $12 + 13 + 14 = 39$. Antrean: 16,39,45. * Gabungkan $16 + 39 + 45 = 100$. Selesai. Kedalamannya: karakter 45 dan 16 berada pada kedalaman 1; 12 dan 13 pada kedalaman 2; 5 dan 9 pada kedalaman 3 (dan bonekanya dibuang). Jadi

$$\begin{aligned} B(T) &= 45(1) + 16(1) + 12(2) + 13(2) + 5(3) + 9(3) \\ &= 45 + 16 + 24 + 26 + 15 + 27 = 153 \text{ trit.} \end{aligned}$$

Bandingkan dengan $\lceil \log_3 6 \rceil = 2$ trit panjang tetap, yang berbiaya $2 \times 100 = 200$ trit: penghematan 23.5%. (Seandainya kita melupakan bonekanya lalu menggabungkan $5 + 9 + 12 = 26$ lebih dulu, kita akan berakhir dengan $26,13,16,45 \to 26 + 13 + 16 = 55$, lalu $55 + 45$ — sebuah simpul beranak hanya **dua**, yang memberi $B = 45(1) + 16(2) + 13(2) + 5(3) + 9(3) + 12(3) = 45 + 32 + 26 + 15 + 27 + 36 = 181$, yang lebih buruk sebesar 28 trit. Bonekanya bukan urusan teknis belaka.)

9.14 *Lewat induksi atas* ℓ. **Dasar:** pohon biner penuh dengan $\ell = 1$ daun adalah satu simpul tunggal, yang merupakan daun, sehingga simpul dalamnya $0 = \ell - 1$ ✓. **Langkah:** misalkan T penuh dengan $\ell \geq 2$ daun. Akarnya adalah simpul dalam (sebab jika tidak $\ell = 1$), jadi menurut kepenuhannya ia punya dua subpohon T_L dan T_R, keduanya penuh, dengan $\ell_L + \ell_R = \ell$ dan $\ell_L, \ell_R \geq 1$. Menurut hipotesis induktifnya keduanya punya $\ell_L - 1$ dan $\ell_R - 1$ simpul dalam. Dengan menambahkan akarnya, T punya $(\ell_L - 1) + (\ell_R - 1) + 1 = \ell_L + \ell_R - 1 = \ell - 1$ ✓. ■

Lewat pencacahan sisi. Misalkan i cacah simpul dalam dan ℓ cacah daunnya, sehingga pohonnya punya $n = i + \ell$ simpul dan, karena ia pohon, $n - 1$ sisi. Setiap sisi adalah sisi dari suatu simpul ke salah satu anaknya, dan tiap simpul dalam menyumbang tepat 2 sisi semacam itu sedangkan daun menyumbang 0. Jadi cacah sisinya adalah $2i$. Menyamakannya, $2i = i + \ell - 1$, sehingga $i = \ell - 1$ ✓. ■

9.15 Ambil tiga aktivitas: $a_1 = [0,10)$ bernilai 100, $a_2 = [0,3)$ bernilai 5, $a_3 = [4,9)$ bernilai 5. Aturan selesai-terawal mengurutkannya menurut waktu selesai, mengambil a_2 ($f = 3$), lalu a_3 ($s = 4 \geq 3$), untuk total nilai 10. Padahal a_1 sendirian bernilai 100. Dari **Kode 9.7**:

```
=== E3: weighted activity selection -- greedy now FAILS ===
  activity :        a1        a2        a3
  [s, f)   :    [0,10)     [0,3)     [4,9)
  value    :       100         5         5
  greedy by earliest finish takes a2+a3, value 10
  dynamic programming finds value 100
  brute force confirms 100
  greedy loses 90 of 100 -- adding weights destroys the greedy-choice property
```

Program dinamisnya. Urutkan aktivitasnya menurut waktu selesai. Untuk $1 \leq i \leq n$ definisikan $q(i)$ sebagai indeks terbesar $j < i$ dengan $f_j \leq s_i$, atau 0 jika tak ada — aktivitas terakhir yang

dapat mendahului a_i. Misalkan best$[i]$ adalah total nilai terbesar yang dapat diperoleh dengan hanya memakai aktivitas $a_1, \dots, a_i$. Maka

$$\text{best}[0] = 0, \qquad \text{best}[i] = \max\{\,\text{best}[i-1],\ \ v_i + \text{best}[q(i)]\,\}.$$

Kedua sukunya adalah dua pilihannya: tinggalkan a_i di luar, atau ambil ia (yang dalam hal itu segala yang selaras dengannya dan selesai lebih awal termuat di dalam $a_1, \dots, a_{q(i)}$). Ini persis pola §7.8. Pengurutannya $O(n\lg n)$; menghitung semua $q(i)$ dengan pencarian biner atas waktu selesai yang terurut adalah $O(n\lg n)$; rekurensinya $O(n)$. Totalnya: $\Theta(n\lg n)$.

Mengapa kerakusan mati. Sifat pilihan rakusnya gagal. Yang paling awal selesai, yaitu a_2, tidak berada di dalam solusi optimal mana pun, sehingga tak ada argumen penukaran yang dapat menyelamatkannya: langkah 5 Teorema 9.2 mempertahankan ukurannya tetap sama, tetapi dengan adanya nilai, penukarannya dapat kehilangan nilai, dan di sini ia kehilangan 95 dari 100.

9.16 Tiga syarat.

1. S berhingga. ✓
2. **Turun-temurun.** Jika $B \in \mathcal{I}$ sehingga $|B| \le k$, dan $A \subseteq B$, maka $|A| \le |B| \le k$, jadi $A \in \mathcal{I}$. ✓ ($\varnothing \in \mathcal{I}$ sebab $0 \le k$.)
3. **Penukaran.** Misalkan $A, B \in \mathcal{I}$ dengan $|A| < |B|$. Maka $|A| < |B| \le k$, jadi $|A| \le k-1$. Karena $|B| > |A|$, himpunan $B - A$ tak kosong; pilihlah sembarang x di dalamnya. Maka $|A \cup \{x\}| = |A| + 1 \le k$, jadi $A \cup \{x\} \in \mathcal{I}$. ✓

Jadi $U_{k,n}$ adalah sebuah matroid. **Greedy**(M, w) mengurutkan S menurun menurut bobot lalu menambahkan unsurnya selama ukurannya tetap $\le k$, sehingga ia mengembalikan k **unsur terberat** (semuanya yang n itu jika $k \ge n$). Teorema 9.15 lalu mengatakan — dengan benar tetapi tanpa keterkejutan — bahwa k unsur terberat membentuk himpunan bagian terberat berukuran paling banyak k. Matroid seragam adalah kasus yang remeh, dan ia berguna justru karena ia memperlihatkan bahwa teoremanya tidak kosong: mengurutkan-lalu-mengambil memang benar-benar yang dilakukan algoritma rakus matroid. Matroid diperkenalkan Whitney [7]; hubungan tepatnya dengan algoritma rakus berasal dari Edmonds [8], dan Oxley [9] adalah acuan modern yang baku.

9.17 Tenggatnya tak berubah pada $d = \langle 4,2,4,3,1,4,6 \rangle$; dendanya kini $w = \langle 70,60,50,40,100,20,10 \rangle$, bertotal 350. Dalam urutan denda yang menurun, tugasnya adalah $a_5(100), a_1(70), a_2(60), a_3(50), a_4(40), a_6(20), a_7(10)$.

- a_5 $(d = 1)$: $\{a_5\}$ punya $N_1 = 1 \le 1$ ✓. **Simpan.**
- a_1 $(d = 4)$: $\{a_5, a_1\}$ punya $N_1 = 1 \le 1, N_2 = 1 \le 2, N_3 = 1 \le 3, N_4 = 2 \le 4$ ✓. **Simpan.**
- a_2 $(d = 2)$: $\{a_5, a_1, a_2\}$ punya $N_1 = 1 \le 1, N_2 = 2 \le 2$ ✓, $N_3 = 2 \le 3, N_4 = 3 \le 4$ ✓. **Simpan.**
- a_3 $(d = 4)$: $\{a_5, a_1, a_2, a_3\}$ punya $N_2 = 2 \le 2, N_4 = 4 \le 4$ ✓. **Simpan.**
- a_4 $(d = 3)$: menambahkannya memberi $N_3 = 3 \le 3$ ✓ tetapi $N_4 = 5 > 4$ ✗. **Tolak.**
- a_6 $(d = 4)$: $N_4 = 5 > 4$ ✗. **Tolak.**
- a_7 $(d = 6)$: $N_4 = 4 \le 4, N_5 = 4 \le 5, N_6 = 5 \le 6$ ✓. **Simpan.**

Himpunan awalnya $A = \{a_1, a_2, a_3, a_5, a_7\}$ berbobot $70 + 60 + 50 + 100 + 10 = 290$; himpunan terlambatnya $\{a_4, a_6\}$ dengan denda $40 + 20 =$ **60**. Periksa: $290 + 60 = 350$ ✓. *Brute force* atas seluruh 128 himpunan bagiannya memastikan 290 adalah bobot awal terbaik yang dapat dicapai, sehingga 60 adalah denda minimumnya. Jadwalnya adalah tugas awal menaik tak turun menurut

tenggat disusul yang terlambat: $\langle a_5, a_2, a_1, a_3, a_7, a_4, a_6 \rangle$ — petak 1 a_5 ($d = 1$ ✓), petak 2 a_2 ($d = 2$ ✓), petak 3 a_1 ($d = 4$ ✓), petak 4 a_3 ($d = 4$ ✓), petak 5 a_7 ($d = 6$ ✓), lalu a_4 dan a_6 terlambat.

Perhatikan apa yang berubah dan apa yang tidak: menaikkan denda a_5 dari 30 menjadi 100 menariknya ke depan urutan sehingga ia disimpan, tetapi uji kebebasannya tetap hanya mengizinkan empat tugas bertenggat ≤ 4, jadi **sesuatu** harus dibuang — dan yang dibuang adalah a_4, yang kini termurah di antara para pesaingnya. Total dendanya naik dari 50 menjadi 60 meskipun kita hanya mengubah sebuah bilangan yang sebelumnya memang kita bayar.

9.18 *Aturannya.* Urutkan pemesanannya menurut **waktu selesai**, seperti sebelumnya. Simpanlah larik freeAt$[1..k]$ berisi waktu saat tiap mesin berikutnya menjadi bebas (mula-mula $-\infty$). Untuk tiap pemesanan berurutan, tinjaulah mesin yang freeAt$[m] \leq s$; jika tak ada, tolaklah pemesanannya; jika ada, tugaskan ia kepada mesin yang freeAt-nya **terbesar** di antara mereka — yaitu "mesin selaras yang tersibuk". Perbaruilah freeAt mesin itu menjadi f.

Mengapa "selaras yang tersibuk", bukan "sembarang"? Menugaskannya kepada mesin yang paling lama menganggur menyia-nyiakan ketersediaan mesin lain yang lebih sempit. Mengambil freeAt selaras yang terbesar meninggalkan mesin-mesin sisanya sebebas mungkin, yang merupakan naluri yang sama dengan aturan selesai-terawal itu sendiri: jagalah sumber dayanya setersedia mungkin.

Apakah selesai-terawal masih berhasil? Ya. Sehimpunan selang dapat dijalankan pada k mesin yang identik jika dan hanya jika tak ada titik waktu yang tertutupi oleh lebih dari k di antaranya, sehingga persoalannya menjadi "pilihlah himpunan bagian terbesar yang kedalamannya paling banyak k" — dan sapuan terurut di atas menyelesaikannya. Secara empiris, 300 000 contoh kasus acak dengan n antara 2 dan 9 serta $k = 2$ diperiksa terhadap pencarian *brute force* atas seluruh himpunan bagiannya (dengan menguji kedalaman pertindihan maksimum tiap himpunan bagian), dan jawaban rakusnya menyamai optimumnya pada **setiap satu** di antaranya. Programnya dan pemeriksanya adalah perluasan langsung Bagian A **Kode 9.6**: gantilah skalar tunggal `freeAt` dengan `vector<int> freeAt(k)` dan uji `s >= freeAt` dengan pemilihan mesin-selaras-tersibuk di atas.

Bagi Profitina Laundry. Dijalankan pada sebelas pemesanan §9.10.1, algoritmanya melayani **7** pemesanan dengan dua pengering melawan 4 dengan satu pengering — pengering 1 mengambil Pak Anto, Warung Bu Tin, Salon Ayu, dan Klinik Sehat persis seperti sebelumnya, dan pengering 2 mengambil Hotel Melati, Bu Wati, dan Pak Hendra. *Brute force* atas seluruh 2^{11} himpunan bagiannya (dengan menerima sebuah himpunan bagian bila tak ada saat yang tertutupi lebih dari dua pemesanan) memastikan bahwa 7 adalah maksimumnya. Pengering ketiga akan mencapai 10 dari 11. Itulah alasan untuk membeli mesin yang kedua, dan itu alasan yang dapat dikemukakan sebuah algoritma rakus dalam $\Theta(n \lg n)$.

Kode 9.7 Program-program bagi Soal Latihan 9.7, 9.10, dan 9.15: pekerjaan-terpendek-dahulu yang diperiksa terhadap seluruh $n!$ urutan, Huffman atas frekuensi Fibonacci, dan pemilihan aktivitas berbobot tempat aturan rakusnya gagal. Kompilasi dengan `g++ -std=c++17 -O2 -o exercises exercises.cpp` lalu jalankan `./exercises`.

```
// exercises.cpp — ketiga program yang diminta dalam soal latihan.
//   E1  meminimumkan rata-rata waktu selesai pada satu mesin: pekerjaan terpendek
//       dahulu, diperiksa terhadap setiap satu dari n! urutan untuk n kecil.
//   E2  Huffman atas frekuensi Fibonacci: pohon paling condong yang mungkin bagi
//       sebuah pohon biner.
//   E3  pemilihan aktivitas BERBOBOT: rakus menurut selesai terawal kini KELIRU,
//       dan pemrograman dinamis diperlukan.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <queue>
#include <map>
#include <algorithm>
#include <numeric>
using namespace std;

// ------------------------------------- E1: pekerjaan terpendek dahulu --
long long totalCompletion(const vector<int>& p, const vector<int>& order) {
    long long clock = 0, sum = 0;
    for (int i : order) { clock += p[i]; sum += clock; }
    return sum;
}

void e1() {
    cout << "=== E1: one machine, minimize the sum of completion times ===\n";
    vector<int> p = { 7, 2, 5, 1, 4, 3 };
    int n = (int)p.size();
    cout << "  job      :"; for (int i = 0; i < n; ++i) cout << setw(5) << ("j" +
to_string(i + 1)); cout << "\n";
    cout << "  length   :"; for (int x : p) cout << setw(5) << x; cout << "\n";

    vector<int> sjf(n); iota(sjf.begin(), sjf.end(), 0);
    sort(sjf.begin(), sjf.end(), [&](int a, int b) { return p[a] < p[b]; });
    cout << "  SJF order:"; for (int i : sjf) cout << setw(5) << ("j" + to_string(i + 1));
cout << "\n";
    long long sjfCost = totalCompletion(p, sjf);
    cout << "    completion times:";
    { long long c = 0; for (int i : sjf) { c += p[i]; cout << setw(5) << c; } }
    cout << "\n    sum = " << sjfCost << ",  average = " << fixed << setprecision(4)
         << (double)sjfCost / n << "\n";
```

Kode 9.7 (Lanjutan) *Program-program bagi Soal Latihan 9.7, 9.10, dan 9.15: pekerjaan-terpendek-dahulu yang diperiksa terhadap seluruh n! urutan, Huffman atas frekuensi Fibonacci, dan pemilihan aktivitas berbobot tempat aturan rakusnya gagal. Kompilasi dengan g++ -std=c++17 -O2 -o exercises exercises.cpp lalu jalankan ./exercises.*

```
    vector<int> order(n); iota(order.begin(), order.end(), 0);
    long long best = -1, worst = -1;
    do {
        long long c = totalCompletion(p, order);
        if (best < 0 || c < best) best = c;
        if (worst < 0 || c > worst) worst = c;
    } while (next_permutation(order.begin(), order.end()));
    cout << "  all " << [&]{ long long f = 1; for (int i = 2; i <= n; ++i) f *= i; return
f; }()
         << " orders examined: best = " << best << ", worst = " << worst << "\n";
    cout << "  SJF attains the best: " << (sjfCost == best ? "YES" : "NO") << "\n\n";
}

// ------------------------------------ E2: Huffman atas Fibonacci -----
struct Node {
    long long freq; string name; bool leaf; int serial; Node *l, *r;
    Node(long long f, string nm) : freq(f), name(nm), leaf(true),  serial(0), l(nullptr),
r(nullptr) {}
    Node(Node* a, Node* b)       : freq(a->freq + b->freq), name(""), leaf(false),
serial(0), l(a), r(b) {}
};
struct Later {
    bool operator()(const Node* a, const Node* b) const {
        if (a->freq != b->freq) return a->freq > b->freq;
        return a->serial > b->serial;
    }
};
void walk(const Node* n, int d, map<string,int>& depth) {
    if (!n) return;
    if (n->leaf) { depth[n->name] = d; return; }
    walk(n->l, d + 1, depth); walk(n->r, d + 1, depth);
}

void e2() {
    cout << "=== E2: Huffman with Fibonacci frequencies ===\n";
    vector<long long> fib = { 1, 1, 2, 3, 5, 8, 13, 21 };
    vector<string> nm = { "c1", "c2", "c3", "c4", "c5", "c6", "c7", "c8" };
    priority_queue<Node*, vector<Node*>, Later> Q;
    int serial = 0;
    for (size_t i = 0; i < fib.size(); ++i) { Node* x = new Node(fib[i], nm[i]); x->serial
= serial++; Q.push(x); }
    cout << "  frequencies:"; for (long long f : fib) cout << setw(5) << f; cout << "\n";
    while (Q.size() > 1) {
        Node* x = Q.top(); Q.pop();
        Node* y = Q.top(); Q.pop();
        Node* z = new Node(x, y); z->serial = serial++;
        cout << "    merge " << setw(3) << x->freq << " + " << setw(3) << y->freq
             << " = " << setw(3) << z->freq << "\n";
        Q.push(z);
    }
```

Kode 9.7 (Lanjutan) *Program-program bagi Soal Latihan 9.7, 9.10, dan 9.15: pekerjaan-terpendek-dahulu yang diperiksa terhadap seluruh n! urutan, Huffman atas frekuensi Fibonacci, dan pemilihan aktivitas berbobot tempat aturan rakusnya gagal. Kompilasi dengan g++ -std=c++17 -O2 -o exercises exercises.cpp lalu jalankan ./exercises.*

```
    Node* root = Q.top();
    map<string,int> depth; walk(root, 0, depth);
    long long B = 0;
    cout << "  character :"; for (auto& s : nm) cout << setw(6) << s; cout << "\n";
    cout << "  frequency :"; for (long long f : fib) cout << setw(6) << f; cout << "\n";
    cout << "  depth     :"; for (auto& s : nm) cout << setw(6) << depth[s]; cout << "\n";
    for (size_t i = 0; i < fib.size(); ++i) B += fib[i] * depth[nm[i]];
    cout << "  B(T) = " << B << " bits for " << accumulate(fib.begin(), fib.end(), 0LL)
         << " characters, i.e. " << fixed << setprecision(4)
         << (double)B / accumulate(fib.begin(), fib.end(), 0LL) << " bits per character\n";
    cout << "  a fixed-length code would need 3 bits each, total "
         << 3 * accumulate(fib.begin(), fib.end(), 0LL) << " bits\n\n";
}

// ----------------------- E3: pemilihan aktivitas BERBOBOT ------
struct WAct { int s, f, v; };

void e3() {
    cout << "=== E3: weighted activity selection -- greedy now FAILS ===\n";
    vector<WAct> a = { {0, 10, 100}, {0, 3, 5}, {4, 9, 5} };
    int n = (int)a.size();
    cout << "  activity :"; for (int i = 0; i < n; ++i) cout << setw(10) << ("a" +
to_string(i + 1)); cout << "\n";
    cout << "  [s, f)   :"; for (auto& x : a) cout << setw(10) << ("[" + to_string(x.s) +
"," + to_string(x.f) + ")"); cout << "\n";
    cout << "  value    :"; for (auto& x : a) cout << setw(10) << x.v; cout << "\n";

    // rakus menurut selesai terawal, menjumlahkan nilai yang kebetulan terkumpul
    vector<int> idx(n); iota(idx.begin(), idx.end(), 0);
    sort(idx.begin(), idx.end(), [&](int p, int q) { return a[p].f < a[q].f; });
    int gv = 0, freeAt = -1; string gp;
    for (int i : idx) if (a[i].s >= freeAt) { gv += a[i].v; freeAt = a[i].f; gp +=
(gp.empty() ? "" : "+") + string("a") + to_string(i + 1); }
    cout << "  greedy by earliest finish takes " << gp << ", value " << gv << "\n";

    // pemrograman dinamis: urutkan menurut selesai, lalu best[i] = max(best[i-1], v_i +
best[j])
    vector<WAct> b = a;
    sort(b.begin(), b.end(), [](const WAct& p, const WAct& q) { return p.f < q.f; });
    vector<int> best(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        int j = 0;                                  // aktivitas terakhir yang selesai
pada atau sebelum b[i-1].s
        for (int k = i - 1; k >= 1; --k) if (b[k - 1].f <= b[i - 1].s) { j = k; break; }
        best[i] = max(best[i - 1], b[i - 1].v + best[j]);
    }
    cout << "  dynamic programming finds value " << best[n] << "\n";
```

Kode 9.7 (Lanjutan) *Program-program bagi Soal Latihan 9.7, 9.10, dan 9.15: pekerjaan-terpendek-dahulu yang diperiksa terhadap seluruh n! urutan, Huffman atas frekuensi Fibonacci, dan pemilihan aktivitas berbobot tempat aturan rakusnya gagal. Kompilasi dengan g++ -std=c++17 -O2 -o exercises exercises.cpp lalu jalankan ./exercises.*

```
    int brute = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        vector<int> c; int v = 0;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) { c.push_back(i); v += a[i].v; }
        bool ok = true;
        for (size_t i = 0; i + 1 < c.size() && ok; ++i)
            for (size_t j = i + 1; j < c.size() && ok; ++j)
                if (!(a[c[i]].s >= a[c[j]].f || a[c[j]].s >= a[c[i]].f)) ok = false;
        if (ok) brute = max(brute, v);
    }
    cout << "  brute force confirms " << brute << "\n";
    cout << "  greedy loses " << brute - gv << " of " << brute
         << " -- adding weights destroys the greedy-choice property\n";
}

int main() { e1(); e2(); e3(); return 0; }
```

Untuk latihan lebih lanjut atas persoalan rakus persis sejenis ini — penjadwalan selang, sistem koin, ragam Huffman — arsip soal yang dikatalogkan di [10] adalah tempat yang tepat untuk dituju, dan pembahasan di [4] membawa soal latihan tambahan tentang matroid serta tentang hubungan antara algoritma rakus dan pemrograman linear.

Daftar Pustaka

[1] Bank Indonesia, “Gambar Uang Rupiah” (katalog uang kertas dan uang logam rupiah yang beredar: uang kertas emisi 2022 dan uang logam emisi 2016). [Daring]. Tersedia: https://www.bi.go.id/id/rupiah/gambar-uang/default.aspx (diakses 24 Sep. 2026).

[2] D. Pearson, “A polynomial-time algorithm for the change-making problem,” *Operations Research Letters*, vol. 33, no. 3, pp. 231–234, May 2005.

[3] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. Chichester, U.K.: Wiley, 1990.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[5] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[6] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[7] H. Whitney, “On the abstract properties of linear dependence,” *American Journal of Mathematics*, vol. 57, no. 3, pp. 509–533, Jul. 1935.

[8] J. Edmonds, “Matroids and the greedy algorithm,” *Mathematical Programming*, vol. 1, no. 1, pp. 127–136, Dec. 1971.

[9] J. Oxley, *Matroid Theory*, 2nd ed. Oxford, U.K.: Oxford Univ. Press, 2011.

[10] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, 4th ed. Raleigh, NC, USA: Lulu Press, 2020.

Bab 10
Algoritma Graf Dasar: Penelusuran Melebar dan Penelusuran Mendalam

Di akhir bab ini Anda akan sanggup menatap hampir setiap kekusutan benda-dan-hubungan — peta jalan, papan rangkaian, silsilah keluarga, daftar tugas beserta urutan pengerjaannya, halaman-halaman sebuah situs web beserta tautan di antaranya — lalu menuliskannya sebagai graf, dengan kata yang tepat untuk setiap bagiannya. Anda akan sanggup menyimpan graf itu di dalam komputer dengan tiga cara yang berbeda, menyebutkan dengan tepat berapa memori yang dihabiskan tiap cara, menyebutkan dengan tepat berapa lama masing-masing dari empat pertanyaan dasar dijawab pada tiap cara, dan memilih di antara ketiganya dengan alasan, bukan dengan kebiasaan. Anda akan sanggup menjalankan *breadth-first search* (penelusuran melebar) dan *depth-first search* (penelusuran mendalam) dengan tangan, selangkah demi selangkah, sambil menuliskan isi antrean atau tumpukannya sesudah setiap langkah, dan memperoleh jawaban yang sama dengan jawaban mesin. Anda akan sanggup *membuktikan* bahwa penelusuran melebar menemukan lintasan terpendek pada graf yang sisi-sisinya tidak berpanjang, bukan sekadar memercayainya. Anda akan sanggup menurunkan $O(V + E)$ untuk kedua penelusuran itu dari dasar. Dan Anda akan sanggup memakai kedua penelusuran itu untuk menjawab enam pertanyaan terpisah yang sekilas sama sekali tidak berhubungan dengan penelusuran: graf ini terdiri atas berapa keping, apakah ia memuat siklus, dapatkah simpul-simpulnya dibelah menjadi dua kubu tanpa satu pun sisi di dalam sebuah kubu, apa rute terpendek dari sini ke sana, apakah himpunan prasyarat ini bertentangan dengan dirinya sendiri, dan dalam urutan apa tugas-tugas ini boleh dikerjakan.

10.1 Dari Mana Kita Melanjutkan

Sembilan bab di belakang kita punya satu kesamaan: masukannya selalu berupa **barisan**. Di Bab 2 kita mengurutkan larik $A[1..n]$ dan mencari di dalamnya. Di Bab 3 kita belajar menghitung biaya pengerjaannya, dan menyelesaikan rekurensi yang dihasilkan perhitungan itu. Di Bab 5, 6, dan 7 masukannya bertambah struktur — *heap* adalah larik yang dibaca sebagai pohon, pohon pencarian biner adalah himpunan kunci yang ditata sedemikian rupa sehingga satu pembandingan memangkas separuh pencarian, tabel *dynamic programming* adalah kisi submasalah — tetapi dalam setiap kasus itu *strukturnya kitalah yang memilih*. Kita yang memaksakannya. Lariknya tidak datang dengan bentuk; kitalah yang memberinya bentuk.

Bab 9 mengubah gaya argumennya, bukan bentuk datanya. Algoritma *greedy* (rakus) membuat pilihan terbaik secara lokal dan tidak pernah meninjaunya kembali, dan kerja bab itu adalah membuktikan bahwa untuk persoalan-persoalan tertentu — pemilihan kegiatan, pengodean Huffman — pilihan terbaik secara lokal juga aman secara global.

Bab ini mengubah bentuk datanya, dan itulah perubahan terbesar semacam itu di dalam buku ini. Dari sini sampai akhir, masukannya **datang** dengan struktur yang sudah ada di dalamnya, struktur yang bukan kita yang memilih dan tidak dapat kita tata ulang. Jaringan jalan adalah fakta tentang Surabaya. Tautan antarhalaman web adalah fakta tentang web. Prasyarat di antara tugas-

tugas sebuah sistem *build* adalah fakta tentang perangkat lunaknya. Tugas kita bukan lagi memaksakan sebuah urutan, melainkan *menemukan* urutan apa yang sudah ada di sana.

Objek matematis untuk data semacam itu adalah **graf**, dan dua algoritma yang menjadi bahan pembangun segala hal lain dalam teori graf adalah **penelusuran melebar** dan **penelusuran mendalam**. Bab 11 sampai 15 — pohon merentang minimum, pengurutan topologis yang dipakai sungguhan, lintasan terpendek satu sumber, lintasan terpendek semua pasangan — seluruhnya adalah penghalusan, hiasan, atau perumuman berbobot dari kedua penelusuran dalam bab ini. Jika Anda memahami bab ini sepenuhnya, sisa materi graf dalam buku ini hanyalah rangkaian langkah kecil. Jika tidak, tak satu pun di antaranya akan masuk akal. Berjalan pelan-pelan sungguh berharga di sini.

10.1.1 Apa yang baru, dan apa yang terbawa dari sebelumnya

Ada tiga hal yang terbawa tanpa berubah, dan ketiganya pantas disebut namanya supaya Anda tahu bahwa Anda tidak memulai dari nol.

Notasi asimtotik Bab 3 adalah notasi yang sama. Ketika kita mengatakan sebuah penelusuran berongkos $O(V + E)$, maksudnya persis sama dengan ketika kita mengatakan *insertion sort* berongkos $O(n^2)$: ada konstanta c dan n_0 sedemikian sehingga untuk semua masukan berukuran paling sedikit n_0, waktu jalannya paling banyak c kali fungsi yang disebut itu. Satu-satunya kebaruan adalah bahwa "ukuran" masukannya kini memuat *dua* bilangan, bukan satu, sebab sebuah graf dapat menjadi besar dengan dua cara yang saling bebas — banyak simpul, atau banyak sisi.

Antrean dan tumpukan Bab 2 adalah antrean dan tumpukan yang sama. Penelusuran melebar adalah antrean dengan graf yang ditempelkan padanya; penelusuran mendalam adalah tumpukan dengan graf yang ditempelkan padanya. Kalimat itu nyaris merangkum seluruh bab ini.

Disiplin invarian gelung Bab 2 adalah disiplin yang sama. Ketika kita membuktikan bahwa penelusuran melebar menghitung jarak terpendek, pembuktiannya adalah induksi atas langkah-langkah gelung `while`, persis seperti pembuktian *insertion sort* adalah induksi atas langkah-langkah gelung `for`-nya.

Yang sungguh-sungguh baru adalah ini: sebuah algoritma graf dapat mengunjungi simpul yang sama dua kali, dan jika itu terjadi, ia mungkin tidak pernah berhenti. Karena itu setiap algoritma dalam bab ini membawa sebuah *penanda* — sebuah warna — yang satu-satunya tugasnya adalah mengingat apa yang sudah pernah dilihat. Melupakan penanda itu adalah cara paling umum untuk menulis program graf yang menggantung selamanya, dan §10.11 akan menyebutnya sebagai yang pertama dari daftar kesalahan yang sering terjadi.

10.2 Apa Itu Graf

10.2.1 Mulailah dari sesuatu yang bisa Anda lihat

Sebelum lambang apa pun, sebuah gambar yang sudah Anda kenal.

Gambarlah lima rumah di sebuah kampung. Di antara sebagian pasangan rumah ada jalan setapak; di antara yang lain tidak ada, sebab ada tembok atau selokan yang menghalangi. Sekarang berhentilah memikirkan rumah dan jalan setapak itu sebagai rumah dan jalan setapak. Simpan hanya dua fakta:

1. ada lima *benda*;

2. *pasangan-pasangan* tertentu dari benda-benda itu terhubung.

Segala sesuatu yang lain — seberapa besar rumahnya, seberapa panjang jalan setapaknya, siapa tinggal di mana — sudah dibuang. Yang tersisa adalah sebuah graf. Rumah-rumahnya disebut **simpul** (satu di antaranya adalah sebuah *simpul*; sebagian buku menyebutnya *node*), dan jalan-jalan setapaknya disebut **sisi** (sebagian buku menyebutnya *arc* atau *link*).

Itulah seluruh gagasannya, dan pantas kita berhenti sejenak untuk menimbang betapa kejamnya penyederhanaan itu. Sebuah graf tidak tahu di mana simpul-simpulnya berada. Ketika Anda menggambar graf yang sama dua kali, sekali dengan simpul-simpulnya berderet dan sekali dengan simpul-simpulnya melingkar, Anda telah menggambar *graf yang sama*. Gambarnya adalah kemudahan bagi mata manusia; grafnya hanyalah daftar bendanya dan daftar pasangan yang terhubung.

> **WAWASAN KUNCI:** Gambar sebuah graf bukanlah grafnya. Dua gambar yang tampak sama sekali berbeda bisa jadi merupakan graf yang sama, dan algoritma-algoritma dalam bab ini tidak dapat membedakan keduanya, sebab yang pernah mereka tanyakan hanyalah "apakah pasangan ini terhubung?"

10.2.2 Menuliskannya

Sekarang kita perlu lambang, dan setiap lambang akan kita perkenalkan dengan kata-kata sebelum kita pakai.

Sebutlah himpunan bendanya V. Huruf V berasal dari *vertices*. Himpunan hanyalah kumpulan tanpa pengulangan dan tanpa urutan, jadi menuliskan $V = \{a, b, c, d, e\}$ berarti "ada lima benda, dan namanya a sampai e" — tidak lebih.

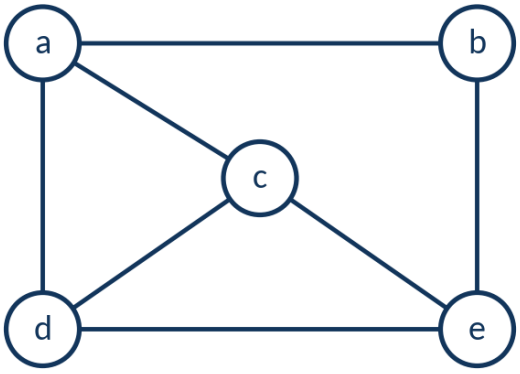
Sebutlah himpunan pasangan yang terhubung itu E. Huruf E berasal dari *edges*. Setiap anggota E adalah sepasang anggota V. Jika benda a dan b terhubung, maka pasangan (a, b) adalah anggota E, yang kita tulis $(a, b) \in E$; lambang $\in$ dibaca "adalah anggota dari" atau, lebih singkat, "ada di dalam".

Banyaknya simpul ditulis $|V|$, dibaca "ukuran V" atau "banyaknya simpul". Begitu pula $|E|$ adalah banyaknya sisi. Di dalam notasi asimtotik hampir semua orang, termasuk buku ini, membuang garis-garisnya dan menulis $O(V + E)$ alih-alih $O(|V| + |E|)$; itu adalah penyalahgunaan notasi, dipahami secara universal, dan artinya $O(|V| + |E|)$.

10.2.3 Definisi formalnya

Dengan ketiga lambang itu di tangan, definisinya cukup satu baris, dan **Gambar 10.1** menyatakannya, lengkap dengan contoh yang akan kita pakai sepanjang sisa bab ini.

> **DEFINISI.** Sebuah **graf** $G = (V, E)$ terdiri atas himpunan V berisi **simpul** dan himpunan $E \subseteq V \times V$ berisi **sisi**, dengan setiap sisi $e = (u, v)$ berupa sepasang simpul. Jika pasangannya *terurut* — artinya, jika (u, v) dan (v, u) bermakna berbeda — maka G adalah **graf berarah**. Jika pasangannya tak terurut, sehingga (u, v) dan (v, u) adalah sisi yang sama, maka G adalah **graf tak berarah**.



$$V = \{a, b, c, d, e\}$$

$$E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$$

Gambar 10.1 Definisi graf, beserta graf contoh yang dipakai sepanjang bab ini: $V = \{a, b, c, d, e\}$ dan $E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$. Lima simpul, tujuh sisi. Perhatikan bahwa gambarnya dan kedua daftar himpunan itu membawa keterangan yang persis sama; gambarnya hanya lebih enak dipandang.

Ada dua bagian definisi yang perlu dibongkar.

$V \times V$, dibaca "V silang V", adalah himpunan *semua* pasangan yang bisa Anda bentuk dari V — untuk lima simpul kita itu berarti $5 \times 5 = 25$ pasangan, termasuk (a, a) dan menghitung (a, b) dan (b, a) secara terpisah. Karena itu menuliskan $E \subseteq V \times V$ ("E adalah himpunan bagian dari V silang V") hanya mengatakan ini: sisi-sisinya adalah *sebagian* dari pasangan yang mungkin. Yang mana, definisinya tidak mengatakan — itulah datanya.

Kata **terurut** adalah seluruh beda antara kedua jenis graf, dan ia pantas diberi contoh yang konkret, bukan yang abstrak. Di jalan dua arah, "ada jalan dari pasar ke masjid" dan "ada jalan dari masjid ke pasar" adalah fakta yang sama: satu sisi, tak terurut. Di jalan satu arah keduanya fakta yang berbeda: Anda mungkin memerlukan dua sisi, atau satu, dan ke mana arahnya menunjuk sangatlah penting. Graf jalan dua arah bersifat tak berarah; graf jalan satu arah bersifat berarah. Ketika menggambar graf berarah kita memberi kepala panah pada tiap sisi; ketika menggambar graf tak berarah kita tidak.

Marilah kita baca **Gambar 10.1** dengan suara keras, pelan-pelan, sebab segala hal berikutnya bergantung pada kesanggupan melakukan ini tanpa susah payah.

- "V sama dengan himpunan yang memuat a, b, c, d, dan e." Lima simpul.
- "E sama dengan himpunan yang memuat pasangan a–b, pasangan a–c, pasangan a–d, pasangan b–e, pasangan c–d, pasangan c–e, dan pasangan d–e." Tujuh sisi.
- Karena tidak ada yang menyatakan sebaliknya, pasangannya tak terurut: ini graf tak berarah. Pasangan (a, b) dan pasangan (b, a) adalah sisi yang sama, dan gambarnya melukiskannya sekali saja.
- Pasangan mana yang *hilang*? (a, e) dan (b, c) dan (b, d). Dari $\binom{5}{2} = 10$ pasangan tak terurut yang tersedia, tujuh ada dan tiga tidak ada.

Baris terakhir itulah yang paling sering dilewati mahasiswa, dan justru itulah yang penting: sebuah graf sama-sama merupakan pernyataan tentang pasangan mana yang *tidak* terhubung dan pasangan mana yang terhubung.

10.2.4 Dua kelaziman kecil

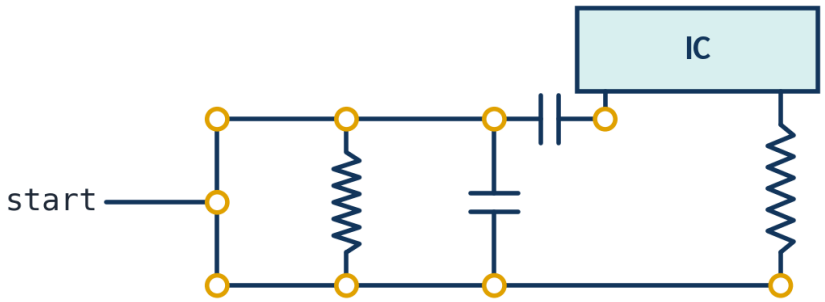
Gelung-diri (*self-loop*) adalah sisi dari sebuah simpul ke dirinya sendiri, (v, v). Gelung-diri dilarang di kebanyakan keadaan tak berarah dan diizinkan di keadaan berarah; contoh

penelusuran mendalam di §10.8 memuat tepat satu, yaitu busur $z \to z$, dan kita akan melihat bahwa algoritmanya menanganinya tanpa kasus khusus apa pun.

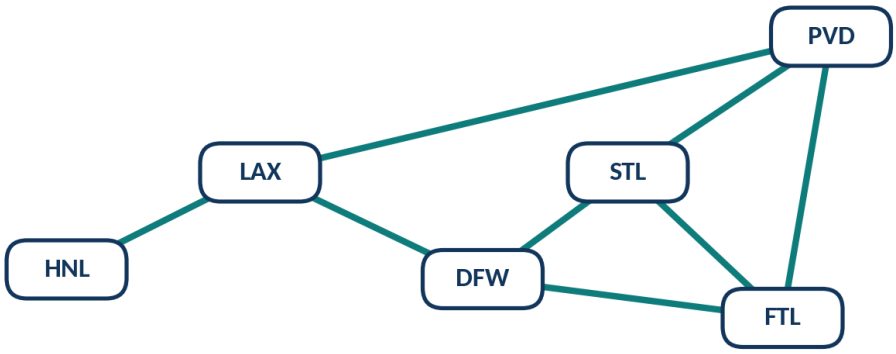
Graf tanpa gelung-diri dan tanpa sisi berulang disebut **sederhana**. Setiap graf tak berarah dalam bab ini bersifat sederhana kecuali dinyatakan lain. Ketika sebuah graf memang mengizinkan dua sisi berbeda di antara pasangan simpul yang sama, ia disebut **multigraf**; multigraf adalah yang Anda peroleh jika memodelkan dua jembatan sejajar di atas sungai yang sama sebagai dua sisi, bukan satu.

10.3 Untuk Apa Graf Itu

Kita mulai dengan penerapan, sebab itulah urutan yang benar: tidak seorang pun sepatutnya mempelajari sebuah notasi sebelum mengetahui notasi itu *untuk* apa. **Gambar 10.2** dan **Gambar 10.3** memperlihatkan dua di antaranya. Kita akan menyusuri daftarnya satu per satu dan menyebutkan, untuk masing-masing, apa simpulnya, apa sisinya, dan pertanyaan apa yang akan Anda ajukan.



Gambar 10.2 Sebuah rangkaian elektronik sebagai graf. Simpulnya adalah titik-titik solder tempat komponen bertemu; sisinya adalah komponennya — resistor, kapasitor — yang menghubungkan titik-titik itu. Pertanyaan "berapa lintasan dengan hambatan terkecil dari `start` ke blok bertanda IC?" adalah pertanyaan lintasan terpendek pada graf ini.



Gambar 10.3 Sebuah jaringan penerbangan sebagai graf: simpulnya adalah bandara (HNL, LAX, DFW, STL, FTL, PVD), sisinya adalah rute yang diterbangi di antaranya. "Bisakah saya pergi dari Honolulu ke Providence, dan dengan berapa kali ganti pesawat?" persis merupakan pertanyaan yang dijawab penelusuran melebar di §10.7.

Rangkaian elektronik. Simpulnya adalah simpang tempat kawat dan komponen bertemu; sisinya adalah komponennya. Karena sebuah resistor punya hambatan, sisi di sini membawa sebuah bilangan, dan grafnya *berbobot* (§10.4.14). Pertanyaan yang diajukan pada **Gambar 10.2** — carilah lintasan dengan hambatan terkecil — adalah persoalan lintasan terpendek, dan Bab 13 dikhususkan untuknya. Tetapi dua pertanyaan yang lebih sederhana dari bab ini juga diajukan pada rangkaian setiap hari: *apakah rangkaiannya terhubung*, artinya adakah rute bagi arus dari

sumber ke tiap komponen, dan *apakah ia memuat siklus*, yang bagi sebuah rangkaian adalah beda antara rantai seri dan gelung.

Jaringan: jalan, penerbangan, komunikasi. Simpulnya adalah tempat, sisinya adalah hubungan di antaranya. Inilah contoh purwarupanya dan inilah yang patut Anda simpan di kepala. Simpul sebuah jaringan jalan adalah simpang; sisinya adalah ruas jalan di antara simpang. Untuk jaringan penerbangan, seperti pada **Gambar 10.3**, simpulnya adalah bandara dan sisinya adalah rute. Untuk jaringan komunikasi simpulnya adalah perute dan sisinya adalah kabel atau tautan radio di antaranya. Pertanyaannya adalah: *dapatkah yang ini mencapai yang itu*, *dengan lompatan paling sedikit*, *dengan jarak terpendek*, *apa yang rusak jika tautan ini putus*.

Perhatikan bahwa kedua pertanyaan itu tidak sama, dan bedanya menata sisa buku ini. "Lompatan paling sedikit" menghitung sisi dan mengabaikan panjangnya — itulah bab ini, dan penelusuran melebar menjawabnya. "Jarak terpendek" menjumlahkan panjangnya — itulah Bab 13, dan itu memerlukan algoritma Dijkstra.

Kompilator. Ketika sebuah kompilator harus menentukan urutan mengevaluasi bagian-bagian sebuah ungkapan, atau sebuah sistem *build* harus menentukan urutan mengompilasi berkas-berkas sebuah program, simpulnya adalah tugas dan tiap sisi $u \to v$ berarti "u harus selesai sebelum v boleh dimulai". Grafnya berarah. Pertanyaannya adalah: *berikan saya sebuah urutan semua tugas yang menghormati setiap sisi*, dan jawabannya adalah **pengurutan topologis** di §10.9.6. Pertanyaan kedua sama pentingnya: *apakah himpunan prasyaratnya bahkan dapat dipenuhi?* Jika tugas u harus mendahului v, dan v harus mendahului w, dan w harus mendahului u, maka tidak ada urutan yang mungkin, dan kompilatornya harus mengatakannya alih-alih berputar selamanya. Mendeteksi hal itu adalah deteksi siklus, §10.9.2.

Grafika. Sebuah jala poligon adalah graf: simpulnya adalah sudut-sudut segitiganya dan sisinya adalah sisi-sisi segitiga itu. Merender, menghaluskan, menyederhanakan, dan memberi tekstur pada jala adalah penelusuran graf, semuanya. *Flood fill* — perkakas ember cat di program gambar mana pun — secara harfiah adalah penelusuran melebar atas sebuah graf yang simpulnya adalah piksel dan sisinya menghubungkan piksel yang bersentuhan.

Menyelesaikan labirin. Simpulnya adalah simpang dan jalan buntu di dalam labirin; sisinya adalah lorongnya. Penelusuran mendalam adalah algoritma yang Anda pakai tanpa perlu diberi tahu: berjalanlah maju sampai tidak bisa lagi, lalu mundurlah ke simpang terakhir yang masih punya lorong belum terjelajah dan cobalah lorong itu. §10.8 membuat hal ini menjadi tepat, dan gambaran klasiknya — seutas benang dan sekaleng cat — adalah salah satu yang terbaik dalam pokok bahasan ini.

Pemetaan. Aplikasi peta menyimpan jaringan jalan sebagai graf dan menjawab kueri perutean di atasnya. Tetapi ia juga menjawab pertanyaan yang sanggup ditangani bab ini sendiri: alamat mana yang sama sekali terjangkau, wilayah peta mana yang merupakan satu keping terhubung, seberapa jauh dalam satuan *belokan*, bukan kilometer, satu tempat dari tempat lain.

Perutean, pencarian, penggerombolan di dalam jaringan. Perutean: carilah sebuah lintasan, atau lintasan terbaik. Pencarian: berangkat dari satu mesin, kunjungilah setiap mesin yang terjangkau darinya — sebuah penelusuran graf, persis. Penggerombolan: pecahlah grafnya menjadi kelompok simpul yang rapat terhubung di dalamnya dan jarang terhubung di antaranya. Penggerombolan yang paling kasar sekaligus paling berguna adalah *komponen terhubungnya*, dan §10.9.1 menghitungnya dengan penelusuran melebar dan tidak lebih.

CONTOH: Pemindai visibilitas AI milik Profitina sendiri berjumpa dengan graf pada setiap pekerjaan yang ia jalankan. Ketika memindai sebuah situs web ia harus mengunjungi setiap halaman yang terjangkau dari halaman muka dengan mengikuti tautan. Simpulnya adalah halaman; sisinya adalah pranala, dan sisi itu *berarah*, sebab tautan dari halaman p ke halaman q tidak memberi Anda tautan kembali. "Kunjungi setiap halaman yang terjangkau dari halaman muka, dan jangan kunjungi satu halaman pun dua kali" adalah penelusuran graf dan bukan yang lain; §10.10 menceritakan apa yang dilakukan Profitina tentangnya.

Pola pada kesembilan hal itu pantas dinyatakan tersendiri.

WAWASAN KUNCI: Untuk memodelkan sebuah persoalan sebagai graf, jawablah tiga pertanyaan dalam urutan ini. **Apa bendanya?** Benda-benda itu menjadi V. **Kapan dua benda terhubung?** Aturan itu menjadi E. **Apakah hubungannya punya arah?** Jika ya, grafnya berarah. Hanya sesudah ketiganya terjawab barulah Anda boleh bertanya algoritma mana yang akan dijalankan — dan jika ketiga jawaban itu salah, tidak ada algoritma yang akan menyelamatkan Anda.

10.4 Kosakata Graf

Teori graf tidak lazim di antara pokok bahasan buku ini karena hampir seluruh kesulitannya bagi pemula adalah *kosakata*: begitu Anda tahu arti kata-katanya, teoremanya pendek-pendek. Jadi kita akan melakukannya dengan benar — setiap istilah didefinisikan, setiap istilah digambarkan pada graf bab ini sendiri, dan sebuah contoh yang dikerjakan langsung sesudah tiap definisi.

Di sepanjang bagian ini, G adalah graf **Gambar 10.1**: $V = \{a, b, c, d, e\}$, $E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$.

10.4.1 Bertetangga, bersisian, tetangga

DEFINISI. Dua simpul **bertetangga** (*adjacent*) jika ada sisi yang menghubungkan keduanya. Sebuah sisi **bersisian** (*incident*) dengan masing-masing dari dua simpul di ujungnya. Simpul-simpul yang bertetangga dengan v adalah **tetangga** v, dan himpunannya ditulis $\mathrm{Adj}[v]$.

Jadi *bertetangga* adalah hubungan antara dua simpul, dan *bersisian* adalah hubungan antara sebuah sisi dan sebuah simpul. Mahasiswa mempertukarkan keduanya selama kira-kira sepekan lalu tidak pernah lagi.

Pada G: a dan b bertetangga, sebab $(a, b) \in E$. Simpul a dan e **tidak** bertetangga, sebab tidak ada pasangan (a, e) yang muncul di E — carilah di **Gambar 10.1** dan Anda tidak akan menemukannya. Sisi (a, b) bersisian dengan a dan bersisian dengan b, dan tidak dengan yang lain. Dan

$$\mathrm{Adj}[a] = \{b, c, d\}, \quad \mathrm{Adj}[b] = \{a, e\}, \quad \mathrm{Adj}[c] = \{a, d, e\}, \quad \mathrm{Adj}[d] = \{a, c, e\}, \quad \mathrm{Adj}[e] = \{b, c, d\}.$$

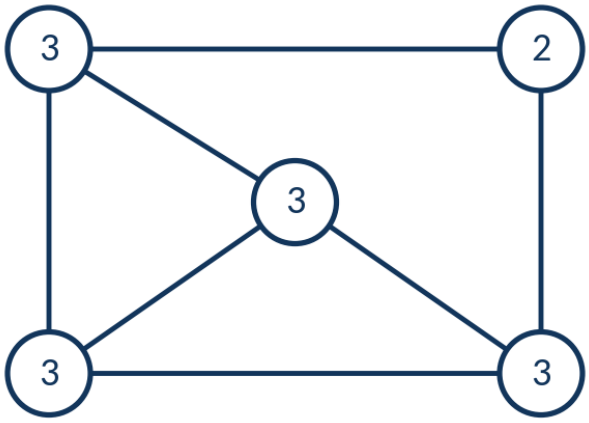
Bacalah yang pertama dengan suara keras: "tetangga a adalah b, c, dan d." Cocokkan dengan gambarnya. Lalu cocokkan keempat yang lain. Ini bukan pekerjaan sia-sia: $\mathrm{Adj}[v]$ adalah *satu-satunya* hal yang pernah ditanyakan setiap algoritma dalam bab ini kepada grafnya, jadi kelima baris di atas, dalam arti yang sesungguhnya, adalah seluruh masukan bagi sisa bab ini.

Pada graf berarah, "bertetangga" memperoleh arah. Kita katakan v bertetangga *dengan* u ketika busur (u, v) ada, dan kemudian $\text{Adj}[u]$ memuat kepala dari busur-busur yang meninggalkan u. Perhatikan bahwa pada graf berarah $v \in \text{Adj}[u]$ **tidak** berarti $u \in \text{Adj}[v]$; pada graf tak berarah hal itu selalu berlaku, dan ketaksetangkupan itulah sumber separuh kutu di dalam kode graf.

10.4.2 Derajat

DEFINISI. Derajat sebuah simpul v pada graf tak berarah, ditulis $\deg(v)$, adalah banyaknya sisi yang bersisian dengan v — setara dengan itu, pada graf sederhana, banyaknya tetangga yang ia punya: $\deg(v) = |\text{Adj}[v]|$.

Gambar 10.4 menggambarkan derajat: setiap simpul G diberi label derajatnya sendiri alih-alih namanya.



Gambar 10.4 Derajat. Setiap simpul graf bab ini diberi label derajatnya alih-alih namanya. Dibaca berlawanan arah jarum jam dari kiri atas, labelnya adalah 3,2,3,3,3; jumlahnya 14, yaitu dua kali banyaknya sisi.

Marilah kita hitung kelima derajat itu dengan tangan, dengan mencacah ujung-sisi pada tiap simpul.

- $\deg(a)$: sisi (a, b), (a, c), (a, d) menyentuh a. Jadi 3.
- $\deg(b)$: sisi (a, b), (b, e) menyentuh b. Jadi 2.
- $\deg(c)$: sisi (a, c), (c, d), (c, e) menyentuh c. Jadi 3.
- $\deg(d)$: sisi (a, d), (c, d), (d, e) menyentuh d. Jadi 3.
- $\deg(e)$: sisi (b, e), (c, e), (d, e) menyentuh e. Jadi 3.

Menjumlahkannya: $3 + 2 + 3 + 3 + 3 = 14$. Dan $|E| = 7$, jadi jumlah derajatnya adalah $2|E|$. Itu bukan kebetulan, dan sebabnya dapat dinyatakan dengan tepat. Marilah kita buktikan.

10.4.3 Lema jabat tangan, dibuktikan

LEMA JABAT TANGAN. Pada setiap graf tak berarah $G = (V, E)$,

$$\sum_{v \in V} \deg(v) = 2\,|E|. \tag{10.1}$$

Bukti. Kita mencacah kumpulan objek yang sama dengan dua cara berbeda lalu menyamakan kedua cacahannya. Objeknya adalah **ujung-sisi**: sebuah ujung-sisi adalah pasangan yang terdiri

atas sebuah sisi bersama salah satu dari dua simpul di ujungnya. Karena itu sisi (a, b) menyumbang dua ujung-sisi, yaitu $\langle (a, b), a \rangle$ dan $\langle (a, b), b \rangle$.

Cacahan pertama, menurut sisi. Setiap sisi punya tepat dua ujung, tidak lebih dan tidak kurang, sebab menurut definisinya sebuah sisi adalah sepasang simpul. Ada $|E|$ sisi. Jadi banyaknya ujung-sisi tepat $2|E|$.

Cacahan kedua, menurut simpul. Tetapkan sebuah simpul v. Ujung-sisi yang simpulnya v berkorespondensi satu-satu dengan sisi yang bersisian dengan v, dan banyaknya $\deg(v)$, menurut definisi derajat. Setiap ujung-sisi punya tepat satu simpul, jadi tidak ada ujung-sisi yang tercacah pada dua simpul berbeda dan tidak ada yang terlewat. Dijumlahkan atas semua simpul, banyaknya ujung-sisi adalah $\sum_{v \in V} \deg(v)$.

Kedua cacahan itu mencacah himpunan yang sama, jadi keduanya sama:

$$\sum_{v \in V} \deg(v) = 2\,|E|. \qquad \blacksquare$$

Dengan bahasa sehari-hari, Persamaan (10.1) berkata: *karena simpul yang bertetangga masing-masing mencacah sisi yang menghubungkannya, setiap sisi tercacah dua kali.* Susurilah sisi (c, e): ia tercacah sekali ketika Anda mendata tetangga c dan sekali lagi ketika Anda mendata tetangga e. Tujuh sisi, masing-masing tercacah dua kali, empat belas.

Lema ini punya tiga akibat yang akan kita pakai.

Akibat 1 — banyaknya simpul berderajat ganjil selalu genap. Belahlah jumlahnya menjadi simpul berderajat genap dan simpul berderajat ganjil. Bagian yang genap bernilai genap. Seluruh jumlahnya adalah $2|E|$, yang genap. Jadi bagian yang ganjil pun harus genap; dan jumlah bilangan-bilangan ganjil bernilai genap hanya bila *banyaknya* genap. (Inilah sebabnya, di sebuah pesta, banyaknya orang yang berjabat tangan sejumlah ganjil kali selalu genap — dari situlah nama lema ini.)

Akibat 2 — panjang total semua daftar ketetanggaan adalah $2|E|$. Ini pernyataan yang sama dengan topi seorang pemrogram, dan inilah fakta tunggal yang paling penting bagi analisis waktu jalan di bagian akhir bab ini. Jika grafnya kita simpan sebagai $|V|$ daftar, satu per simpul, dengan $\mathrm{Adj}[v]$ memuat tetangga v, maka daftar v punya $\deg(v)$ entri, sehingga semua daftar itu bersama-sama memuat $\sum_v \deg(v) = 2|E|$ entri. Karena itu algoritma yang menyusuri setiap daftar ketetanggaan tepat sekali melakukan kerja $\Theta(E)$, bukan $\Theta(V \cdot E)$. Penelusuran melebar maupun mendalam melakukan persis itu, dan dari situlah $O(V + E)$ mereka berasal.

Akibat 3 — derajat rata-ratanya adalah $2|E|/|V|$. Bagilah kedua ruas dengan $|V|$. Untuk graf kita itu adalah $14/5 = 2.8$.

10.4.4 Derajat masuk dan derajat keluar

Pada graf berarah, busur yang meninggalkan v dan busur yang memasuki v adalah dua hal yang berbeda, jadi derajatnya terbelah dua.

> **DEFINISI.** Pada graf berarah, **derajat keluar** v, ditulis $\deg^+(v)$, adalah banyaknya busur yang *ekornya* v — artinya, banyaknya busur berbentuk $(v, \cdot)$. **Derajat masuk**, ditulis $\deg^-(v)$, adalah banyaknya busur yang *kepalanya* v — busur berbentuk $(\cdot, v)$. **Derajat** v adalah $\deg^+(v) + \deg^-(v)$.

Padanan berarah dari lema jabat tangan lebih sederhana, dan datang dengan dua paruh:

$$\sum_{v \in V} \deg^{+}(v) = |E|, \qquad \sum_{v \in V} \deg^{-}(v) = |E|, \qquad \text{sehingga} \qquad \sum_{v \in V} (\deg^{+}(v) + \deg^{-}(v)) = 2|E|. \tag{10.2}$$

Bukti. Setiap busur punya tepat satu ekor, jadi menjumlahkan derajat keluar mencacah setiap busur tepat sekali. Setiap busur punya tepat satu kepala, jadi menjumlahkan derajat masuk juga mencacah setiap busur tepat sekali. ■

Contoh yang dikerjakan, pada graf berarah contoh penelusuran mendalam §10.8, yang busur-busurnya adalah $u \to v, u \to x, v \to y, w \to y, w \to z, x \to v, y \to x$, dan $z \to z$.

Tabel 10.1 Derajat masuk dan derajat keluar graf berarah yang dipakai di §10.8. Kedua lajurnya berjumlah $8 = |E|$, sebagaimana dituntut lemanya. Gelung-diri $z \to z$ menyumbang satu pada derajat keluar z dan satu pada derajat masuknya.

Simpul	deg⁺	Busur yang keluar	deg⁻	Busur yang masuk
u	2	$u \to v, u \to x$	0	—
v	1	$v \to y$	2	$u \to v, x \to v$
w	2	$w \to y, w \to z$	0	—
x	1	$x \to v$	2	$u \to x, y \to x$
y	1	$y \to x$	2	$v \to y, w \to y$
z	1	$z \to z$	1	$z \to z$

Derajat keluar: $2 + 1 + 2 + 1 + 1 + 1 = 8$. Derajat masuk: $0 + 2 + 0 + 2 + 2 + 1 = 8$. Dan $|E| = 8$. Hitungannya cocok, persis seperti yang diramalkan Persamaan (10.2).

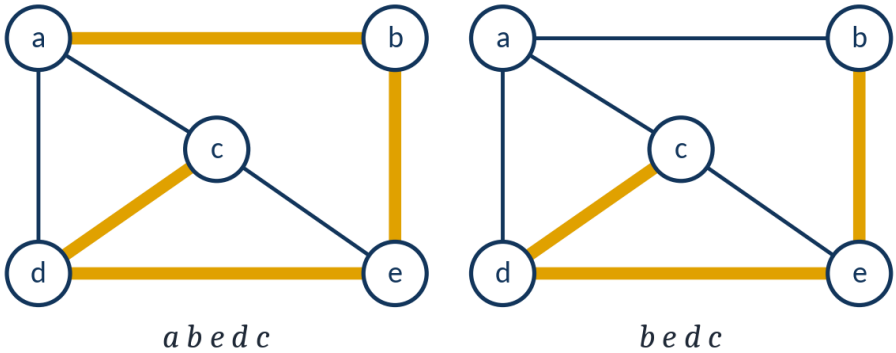
Simpul dengan derajat masuk 0 disebut **sumber**; simpul dengan derajat keluar 0 disebut **muara**. Pada **Tabel 10.1**, u dan w adalah sumber dan tidak ada muara. Sumber itu penting: §10.9.6 menunjukkan bahwa pengurutan topologis sebuah graf berarah tak bersiklus dapat dihasilkan dengan membuang sumber berulang-ulang, dan graf berarah tak bersiklus selalu punya sekurang-kurangnya satu.

10.4.5 Jalan, lintasan, dan lintasan sederhana

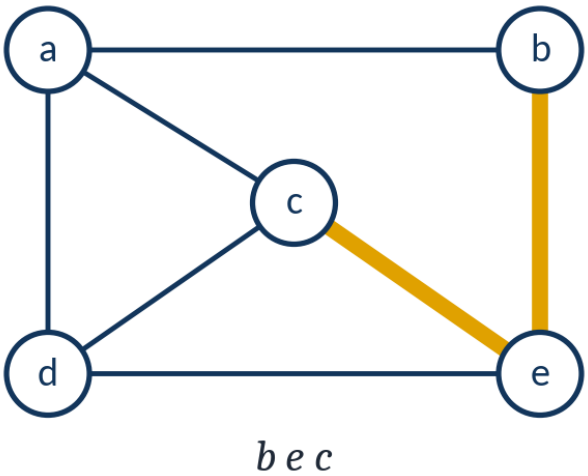
Di sini peristilahannya pantas diberi catatan, sebab buku-buku berbeda dan bedanya membuat orang tersandung.

> **DEFINISI.** Sebuah **lintasan** (*path*; sebagian buku menyebutnya *walk*, jalan) dari v_1 ke v_k adalah barisan simpul $v_1, v_2, \ldots, v_k$ yang setiap pasangan berurutannya, v_i dan v_{i+1}, bertetangga. **Panjangnya** adalah banyaknya sisi yang ia pakai, yaitu $k - 1$ — bukan banyaknya simpul. Sebuah lintasan disebut **sederhana** jika tidak ada simpul yang berulang.

Ringkasnya: *lintasan* adalah barisan simpul $v_1, v_2, \ldots, v_k$ sedemikian sehingga simpul-simpul berurutan v_i dan v_{i+1} bertetangga; *lintasan sederhana* tidak memuat simpul yang berulang. **Gambar 10.5** memperlihatkan dua contohnya dan **Gambar 10.6** contoh ketiga.



Gambar 10.5 Dua lintasan pada graf bab ini, disorot. Di sebelah kiri, $a\ b\ e\ d\ c$, berpanjang 4; di sebelah kanan, $b\ e\ d\ c$, berpanjang 3. Keduanya sederhana: tidak ada simpul yang dipakai dua kali.



Gambar 10.6 Lintasan ketiga, $b\ e\ c$, berpanjang 2. Inilah lintasan terpendek dari b ke c pada graf ini — tidak ada sisi (b, c), jadi tidak ada lintasan berpanjang 1.

Marilah kita periksa lintasan sebelah kiri pada **Gambar 10.5** sisi demi sisi, sebab inilah persis pemeriksaan yang dilakukan setiap algoritma dalam bab ini.

- Apakah a bertetangga dengan b? Ya: $(a, b) \in E$. Langkah diambil, panjang sejauh ini 1.
- Apakah b bertetangga dengan e? Ya: $(b, e) \in E$. Panjang sejauh ini 2.
- Apakah e bertetangga dengan d? Ya: $(d, e) \in E$; pasangannya tak terurut, jadi (e, d) adalah sisi yang sama. Panjang sejauh ini 3.
- Apakah d bertetangga dengan c? Ya: $(c, d) \in E$. Panjang sejauh ini 4.
- Adakah simpul yang berulang pada a, b, e, d, c? Tidak. Jadi lintasannya sederhana.

Sekarang sebuah lintasan yang *tidak* sederhana: a, b, e, c, a, d. Setiap pasangan berurutannya bertetangga — periksalah: (a, b), (b, e), (c, e), (a, c), (a, d), semuanya di E — jadi ini lintasan berpanjang 5. Tetapi a muncul dua kali, jadi ia tidak sederhana. Dan perhatikan bahwa ia juga *sia-sia*: lintasan yang lebih pendek a, d punya kedua ujung yang sama dan berpanjang 1. Pengamatan itu merambat menjadi sebuah lema yang akan kita pakai dalam pembuktian lintasan terpendek:

> **WAWASAN KUNCI:** Jika ada lintasan dari u ke v sama sekali, maka ada lintasan *sederhana* dari u ke v, dan lintasan itu tidak lebih panjang. Alasannya: jika suatu simpul w muncul dua kali pada lintasannya, hapuslah segala sesuatu yang berada tepat di antara kedua kemunculannya, berikut salah satu kemunculannya. Hasilnya tetap sebuah lintasan — simpul sebelum kemunculan pertama masih bertetangga dengan w, dan w masih bertetangga dengan simpul sesudah kemunculan kedua — dan hasilnya lebih pendek secara tegas. Ulangi sampai

> tidak ada simpul yang muncul dua kali; prosesnya pasti berhenti, sebab panjangnya berkurang setiap kali dan tidak bisa turun di bawah nol.

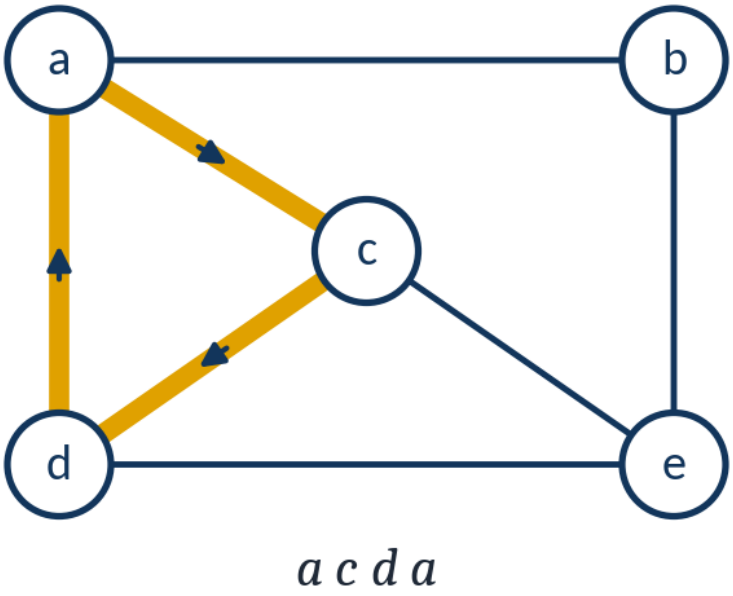
Akibat yang akan kita sandari adalah bahwa lintasan sederhana pada graf dengan $|V|$ simpul berpanjang paling banyak $|V| - 1$, sebab ia tidak boleh memakai satu simpul dua kali. Jadi tidak ada lintasan terpendek yang pernah lebih panjang daripada $|V| - 1$ sisi — fakta yang pantas diingat ketika Anda menentukan ukuran sebuah larik.

Dua patah kata lagi. Simpul v disebut **terjangkau** dari u jika ada lintasan dari u ke v; dan kita menulis $\delta(u, v)$, "delta dari u dan v", untuk **jarak lintasan terpendek** dari u ke v, artinya panjang terkecil di antara semua lintasan dari u ke v, atau ∞ jika v tidak terjangkau dari u. Setiap simpul terjangkau dari dirinya sendiri lewat lintasan berpanjang 0 yang hanya terdiri atas simpul itu, jadi $\delta(v, v) = 0$ selalu.

10.4.6 Siklus dan graf tak bersiklus

> **DEFINISI. Siklus** adalah lintasan $v_1, v_2, \ldots, v_k, v_1$ yang sederhana kecuali bahwa simpul terakhirnya sama dengan simpul pertamanya. Graf tanpa siklus disebut **tak bersiklus** (*acyclic*).

Ringkasnya: *siklus* adalah lintasan sederhana, kecuali bahwa simpul terakhirnya sama dengan simpul pertamanya. **Gambar 10.7** menggambarkan satu siklus; anak-anak panah kecilnya menelusuri jalannya, yang berakhir tepat di tempat ia bermula.



Gambar 10.7 Sebuah siklus pada graf bab ini, disorot: $a\ c\ d\ a$. Berangkatlah dari a, berjalanlah ke c, berjalanlah ke d, berjalanlah kembali ke a — tiga sisi, dan Anda berada di tempat Anda memulai tanpa pernah mengulang satu simpul pun di perjalanan.

Periksalah: $(a, c) \in E$, $(c, d) \in E$, $(a, d) \in E$. Tiga sisi, tiga simpul, dan satu-satunya pengulangan adalah simpul pertama yang muncul lagi di akhir. Itulah siklus berpanjang 3, yang juga disebut **segitiga**.

Graf kita punya beberapa siklus lain: $a\ b\ e\ c\ a$ (panjang 4), $a\ b\ e\ d\ a$ (panjang 4), $c\ d\ e\ c$ (panjang 3), dan siklus berpanjang 5 yaitu $a\ b\ e\ c\ d\ a$. Jadi graf ini sungguh-sungguh *tidak* tak bersiklus.

Pada graf sederhana tak berarah, siklus terpendek yang mungkin berpanjang 3: panjang 1 berarti gelung-diri, yang dilarang, dan panjang 2 menuntut dua sisi berbeda di antara pasangan yang sama, yang juga dilarang graf sederhana. Sebaliknya pada graf **berarah**, keduanya mungkin:

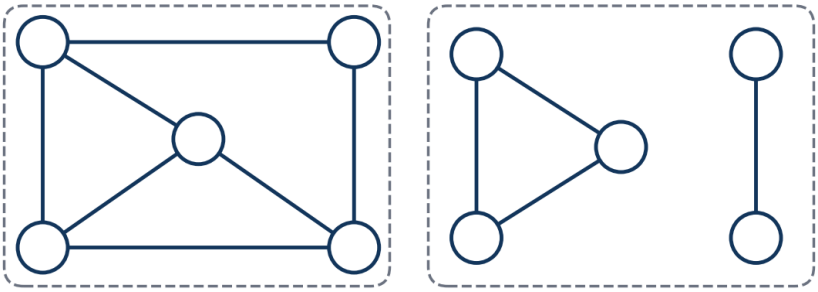
gelung-diri $z \to z$ adalah siklus berarah berpanjang 1, dan sepasang busur $u \to v$, $v \to u$ adalah siklus berarah berpanjang 2. Ini penting di §10.9.2, tempat uji siklus ternyata sungguh-sungguh berbeda pada kedua keadaan itu.

Graf berarah tanpa siklus berarah begitu penting sehingga ia punya nama sendiri dan akronim sendiri: **graf berarah tak bersiklus**, atau **DAG**. §10.9.5 membahasnya.

10.4.7 Graf terhubung

> **DEFINISI.** Graf tak berarah disebut **terhubung** jika untuk setiap pasangan simpul u dan v ada lintasan dari u ke v. Graf yang tidak terhubung disebut **tak terhubung**.

Ringkasnya: sebuah graf *terhubung* bila sembarang dua simpulnya dihubungkan oleh suatu lintasan. **Gambar 10.8** membandingkan kedua keadaan itu secara berdampingan.



terhubung tidak terhubung

Gambar 10.8 Terhubung dan tidak terhubung. Di sebelah kiri, setiap simpul dapat dicapai dari setiap simpul lain. Di sebelah kanan simpul-simpul yang sama digambar dengan lebih sedikit sisi, dan dua simpul di ujung kanan terputus dari segitiga di sebelah kiri: tidak ada lintasan yang menghubungkannya, jadi grafnya tak terhubung.

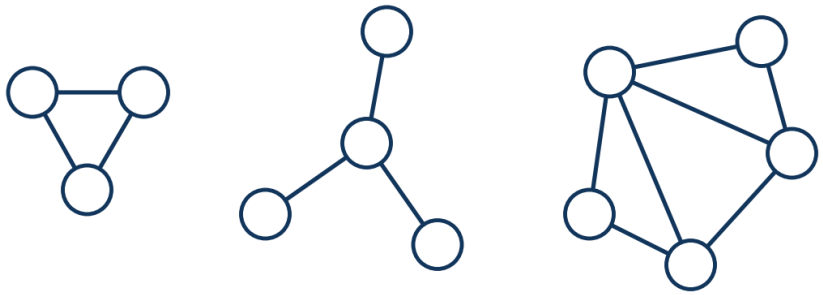
Graf G kita terhubung. Untuk memastikannya Anda pada dasarnya harus memeriksa seluruh $\binom{5}{2} = 10$ pasangan, tetapi ada jalan pintas yang akan kita jadikan algoritma di §10.9.1: pilihlah satu simpul mana saja dan lihatlah apakah Anda dapat mencapai semua yang lain darinya. Dari a kita punya lintasan a, $a\,b$, $a\,c$, $a\,d$, dan $a\,b\,e$; itu kelima simpulnya, jadi G terhubung. Jalan pintas itu berhasil karena keterjangkauan pada graf tak berarah bersifat *setangkup* — jika Anda dapat pergi dari a ke u, Anda dapat pergi dari u ke a dengan menyusuri sisi yang sama secara terbalik — dan *menghantar*: untuk pergi dari u ke v, pergilah dari u ke a lalu dari a ke v.

10.4.8 Komponen terhubung

> **DEFINISI. Komponen terhubung** sebuah graf tak berarah adalah subgraf terhubung yang maksimal: sekumpulan simpul, beserta semua sisi di antaranya, yang terhubung, dan yang kepadanya tidak ada lagi simpul graf itu yang dapat ditambahkan tanpa merusak keterhubungannya.

Kata **maksimal** mengerjakan seluruh tugasnya, jadi marilah kita sangat jelas tentangnya. Pada gambar sebelah kanan **Gambar 10.8**, simpul tunggal di kanan atas membentuk subgraf terhubung seorang diri — satu simpul, tanpa sisi, dan secara sepele setiap pasangan simpulnya dihubungkan oleh sebuah lintasan. Tetapi ia *bukan* komponen terhubung, sebab kita dapat menambahkan simpul di bawahnya, berikut sisi di antara keduanya, dan tetap memperoleh sesuatu yang terhubung. Komponen adalah keping yang tidak dapat Anda besarkan lagi.

Gambar 10.9 adalah sebuah contoh; jawabannya, yang akan kita pastikan dengan program di §10.9.1, adalah tiga komponen.



Gambar 10.9 Sebuah graf dengan tiga komponen terhubung: sebuah segitiga di sebelah kiri, sebuah bintang di tengah, dan sebuah keping berisi lima simpul di sebelah kanan. Tidak ada sisi yang menyeberang di antara ketiganya, dan di dalam masing-masingnya setiap simpul mencapai setiap simpul lain.

Tiga fakta tentang komponen, masing-masing akan kita perlukan:

1. **Setiap simpul terletak di tepat satu komponen.** Keterjangkauan membelah V menjadi kelompok-kelompok: tempatkan u dan v di kelompok yang sama tepat ketika ada lintasan di antara keduanya. Karena keterjangkauan bersifat refleksif, setangkup, dan menghantar — sebuah *relasi ekuivalensi* — kelompok-kelompoknya tidak bertindihan dan bersama-sama menutupi V. Tiap kelompok, beserta sisi-sisi internalnya, adalah sebuah komponen.
2. **Sebuah graf terhubung tepat ketika ia punya satu komponen.** Itulah definisinya, dibaca dari ujung yang lain.
3. **Tidak ada sisi yang menghubungkan dua komponen berbeda.** Jika ada sisi yang menghubungkan u di satu komponen ke v di komponen lain, maka u dapat mencapai v, sehingga keduanya berada di kelompok yang sama. Inilah sebabnya penelusuran yang dimulai di dalam sebuah komponen tidak pernah dapat melarikan diri darinya, dan itulah seluruh alasan mengapa algoritma §10.9.1 berhasil.

10.4.9 Komponen terhubung kuat, untuk graf berarah

Pada graf berarah, keterjangkauan tidak lagi setangkup — jalan satu arah hanya berjalan satu arah — sehingga "terhubung" harus dinyatakan dengan lebih hati-hati.

> **DEFINISI.** Graf berarah disebut **terhubung kuat** jika untuk setiap pasangan terurut simpul u, v ada lintasan berarah dari u ke v *dan* lintasan berarah dari v ke u. **Komponen terhubung kuat** adalah subgraf terhubung kuat yang maksimal.

Perhatikan graf berarah bertiga busur $u \to v$, $v \to w$, $w \to u$. Setiap simpul mencapai setiap simpul lain dengan mengelilingi cincinnya, jadi ia terhubung kuat: satu komponen terhubung kuat yang memuat ketiganya. Sekarang hapuslah busur $w \to u$. Lintasan $u \to v \to w$ masih ada, tetapi sama sekali tidak ada yang mengantar kembali dari w ke u. Grafnya tidak lagi terhubung kuat, dan kini ia punya tiga komponen terhubung kuat, masing-masing berisi satu simpul.

Graf berarah di §10.8 — dengan busur $u \to v, u \to x, v \to y, w \to y, w \to z, x \to v, y \to x, z \to z$ — punya empat komponen terhubung kuat: $\{v, x, y\}$, sebab $v \to y \to x \to v$ adalah cincin berarah; serta ketiga komponen tunggal $\{u\}$, $\{w\}$, dan $\{z\}$. Periksalah $\{u\}$: u mencapai v, tetapi tidak ada busur yang sama sekali memasuki u, jadi tidak ada yang mencapai u, jadi u tidak dapat berbagi komponen dengan siapa pun.

Menghitung komponen terhubung kuat memerlukan *dua* lintasan penelusuran mendalam dan merupakan penerapan klasik dari waktu selesai di §10.8.3. Kita tidak akan mengerjakannya di bab ini, tetapi segala sesuatu yang ia perlukan dibangun di sini.

10.4.10 Subgraf dan subgraf merentang

> **DEFINISI. Subgraf** dari $G = (V, E)$ adalah graf $G' = (V', E')$ dengan $V' \subseteq V$ dan $E' \subseteq E$, sedemikian sehingga setiap sisi di E' punya kedua ujungnya di dalam V'. Sebuah subgraf disebut **merentang** jika $V' = V$ — artinya, jika ia mempertahankan setiap simpul dan hanya membuang sisi.

Bentuk singkat definisinya adalah *subgraf: himpunan bagian simpul dan sisi yang membentuk sebuah graf*, dan anak kalimat "yang membentuk sebuah graf" adalah syarat yang baru saja dinyatakan: Anda tidak boleh mempertahankan sisi yang ujungnya telah Anda buang, sebab sisi itu akan menggantung.

Tiga contoh yang dikerjakan pada G.

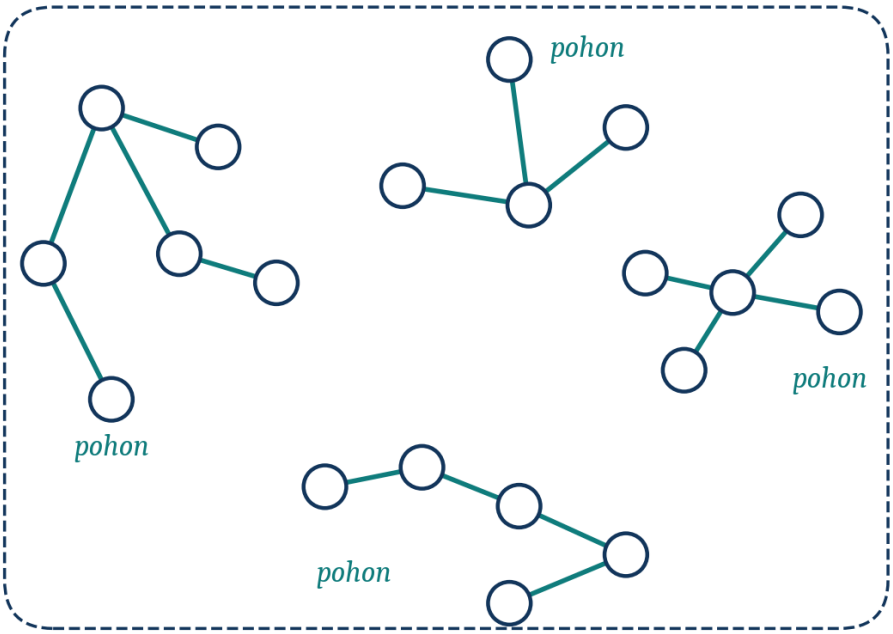
- $V' = \{a, b, e\}$, $E' = \{(a, b), (b, e)\}$. Sah: kedua ujung kedua sisi itu bertahan. Ini subgraf, dan ia berupa lintasan.
- $V' = \{a, b, e\}$, $E' = \{(a, b), (b, e), (c, e)\}$. **Tidak sah**: sisi (c, e) memerlukan c, dan c sudah dibuang.
- $V' = \{a, b, c, d, e\}$, $E' = \{(a, b), (a, c), (a, d), (b, e)\}$. Sah, dan merentang, sebab setiap simpul hadir. Empat dari tujuh sisinya telah dibuang.

Subgraf merentanglah yang kita pedulikan, sebab kedua penelusuran bab ini menghasilkannya. Penelusuran melebar dari s mengembalikan subgraf merentang dari komponen yang memuat s (§10.7.10); penelusuran mendalam mengembalikan subgraf merentang dari seluruh grafnya (§10.8.5). Bab 11 seluruhnya membahas pencarian subgraf merentang *termurah* yang tetap terhubung — pohon merentang minimum.

10.4.11 Pohon dan hutan

> **DEFINISI. Pohon (bebas)** adalah graf tak berarah yang terhubung dan tak bersiklus. **Hutan** adalah graf tak berarah yang tak bersiklus — setara dengan itu, kumpulan pohon, satu per komponen terhubung.

Ringkasnya: *pohon (bebas)* adalah graf terhubung tanpa siklus; *hutan* adalah kumpulan pohon. **Gambar 10.10** menggambarkan keduanya: empat pohon, dan bingkai putus-putus yang menghimpun keempatnya menjadi satu hutan.



hutan: keempat pohon bersama-sama

Gambar 10.10 Pohon dan hutan. Masing-masing dari keempat graf itu sendiri-sendiri adalah sebuah *pohon*: terhubung, dan tanpa siklus di mana pun di dalamnya. Keempatnya bersama-sama membentuk satu graf yang tak bersiklus tetapi tidak terhubung — sebuah *hutan*, dengan empat komponen.

Kata **bebas** dalam "pohon bebas" membedakannya dari pohon *berakar* di Bab 6. Pohon pencarian biner punya akar, dan setiap simpulnya punya induk serta paling banyak dua anak; peran-peran itu adalah bagian dari datanya. Pohon bebas tidak punya semua itu. Ia hanyalah graf yang kebetulan terhubung dan tak bersiklus; tidak ada simpul yang istimewa. Anda *boleh* memilih sebuah simpul dan menyebutnya akar, lalu setiap simpul lain memperoleh induk — tetangga yang selangkah lebih dekat ke akar — dan itulah persis operasi yang dilakukan kedua penelusuran kita.

Pohon bebas punya kekakuan yang indah, dan ketiga fakta berikut dipakai terus-menerus. Untuk graf tak berarah G dengan $|V|$ simpul, sembarang dua di antara ketiga hal ini mengakibatkan yang ketiga:

1. G terhubung;
2. G tak bersiklus;
3. $|E| = |V| - 1$.

Marilah kita setidaknya melihat *mengapa* sebuah pohon punya tepat $|V| - 1$ sisi, sebab argumennya pendek dan faktanya dipakai di §10.7.10. Berinduksilah atas $|V|$. Pohon dengan satu simpul tidak punya sisi: $0 = 1 - 1$. Sekarang ambillah pohon T dengan $|V| \geq 2$ simpul. Ia pasti punya simpul berderajat 1 — sebuah **daun**. (Mengapa? Mulailah dari mana saja dan teruslah berjalan melalui sisi yang belum terpakai. Anda tidak akan pernah dapat kembali ke simpul yang sudah Anda kunjungi, sebab itu akan menutup sebuah siklus, dan pohon tidak punya siklus. Karena simpulnya berhingga banyaknya, akhirnya Anda pasti sampai pada simpul yang tidak punya sisi tak terpakai yang meninggalkannya, dan simpul itu berderajat 1.) Hapuslah daun itu beserta satu sisinya. Yang tersisa tetap tak bersiklus — menghapus sesuatu tidak mungkin menciptakan siklus — dan tetap terhubung, sebab daun itu tidak berada pada lintasan siapa pun di antara dua simpul lain. Jadi ia adalah pohon dengan $|V| - 1$ simpul, yang menurut hipotesis induksi punya $|V| - 2$ sisi. Kembalikan daunnya: $|V| - 1$ sisi. ■

Dua akibat yang perlu diingat: sebuah pohon punya tepat satu lintasan sederhana di antara sembarang dua simpulnya (seandainya ada dua yang berbeda, menggabungkannya akan membentuk siklus); dan menambahkan satu sisi baru mana pun ke sebuah pohon menciptakan tepat satu siklus.

10.4.12 Graf lengkap

> **DEFINISI. Graf lengkap** dengan n simpul, ditulis K_n, adalah graf sederhana tak berarah yang di dalamnya *setiap* pasangan simpul berbeda bertetangga.

Berapa sisi yang dipunyai K_n? Masing-masing dari n simpulnya bertetangga dengan $n-1$ simpul lainnya, jadi derajatnya berjumlah $n(n-1)$; menurut lema jabat tangan jumlah itu adalah $2|E|$, sehingga

$$|E| = \binom{n}{2} = \frac{n(n-1)}{2}. \tag{10.3}$$

Untuk $n = 5$, Persamaan (10.3) memberi 10. Graf G kita punya 7 dari 10 sisi itu, jadi ia dekat dengan lengkap tetapi tidak lengkap: (a, e), (b, c), dan (b, d) tidak ada.

K_n penting karena ia adalah *kasus terburuk untuk penyimpanan*. Ia memberi tahu kita sebesar apa $|E|$ dapat menjadi untuk $|V|$ tertentu, dan karena itu seberapa jauh ketiga representasi §10.5 dapat berjarak satu sama lain.

10.4.13 Graf bipartit

> **DEFINISI.** Sebuah graf disebut **bipartit** jika simpul-simpulnya dapat dibelah menjadi dua himpunan saling lepas L dan R — dua *sisi kubu* — sedemikian sehingga setiap sisi punya satu ujung di L dan ujung lainnya di R. Setara dengan itu, tidak ada sisi yang menghubungkan dua simpul pada kubu yang sama.

Graf bipartit memodelkan setiap keadaan yang punya dua jenis benda dan sebuah hubungan di antara kedua jenis itu: mahasiswa dan mata kuliah, pekerja dan giliran kerja, pelanggan dan pesanan, mesin dan pekerjaan. "Setiap sisi menyeberang" persis merupakan pernyataan bahwa tidak ada hubungan *di dalam* satu jenis.

Contoh yang dikerjakan.

- **Lintasan** p, q, r, s bersifat bipartit: letakkan p dan r di kiri, q dan s di kanan. Setiap sisi — (p, q), (q, r), (r, s) — menyeberang.
- **Siklus genap** p, q, r, s, p bersifat bipartit: kiri $\{p, r\}$, kanan $\{q, s\}$.
- **Siklus ganjil** tidak. Cobalah segitiga p, q, r: letakkan p di kiri, lalu q harus ke kanan karena sisi (p, q), lalu r harus ke kiri karena (q, r) — dan kini sisi (r, p) menghubungkan dua simpul di kiri. Tidak ada penempatan yang berhasil.
- Graf G kita **tidak** bipartit, dan segitiga $a\,c\,d$ pada **Gambar 10.7** adalah sebabnya: tiga simpul yang saling bertetangga tidak dapat dibelah ke dua kubu, sebab dua di antara ketiganya pasti mendarat di kubu yang sama.
- $K_{2,3}$ — dua simpul di kiri, tiga di kanan, keenam sisi penyeberangnya hadir — bersifat bipartit menurut konstruksinya, dan merupakan graf bipartit *lengkap*.

Pencirian yang membuat kebipartitan dapat diuji adalah ini, dan §10.9.3 mengubahnya menjadi sebelas baris C++:

WAWASAN KUNCI: Sebuah graf bersifat bipartit **jika dan hanya jika** ia tidak memuat siklus berpanjang ganjil. Arah yang mudah: mengelilingi sebuah siklus berganti kubu pada setiap langkah, sehingga kembali ke simpul awal menuntut langkah berjumlah genap. Arah yang lain adalah algoritmanya: cobalah mewarnai grafnya dengan dua warna lewat penelusuran melebar, dan entah Anda berhasil — yang memperlihatkan kedua kubunya — atau Anda menemukan sisi di antara dua simpul berwarna sama, yang memperlihatkan siklus ganjil.

10.4.14 Graf berbobot

DEFINISI. Graf berbobot adalah graf beserta sebuah fungsi $w: E \to \mathbb{R}$ yang memberikan sebuah bilangan, yaitu **bobot**, kepada tiap sisi.

Bobotnya adalah apa pun yang dikatakan penerapannya: kilometer pada jalan, ohm pada resistor di **Gambar 10.2**, menit pada satu ruas penerbangan, rupiah pada sebuah pengantaran, kapasitas pada sebuah pipa.

Segala sesuatu dalam bab *ini* tidak berbobot, dan itu pilihan yang disengaja, bukan kelalaian. Ketika tidak ada bobot, "lintasan terpendek" hanya dapat berarti "sisi paling sedikit", dan kemudian antrean saja sudah cukup: penelusuran melebar menyelesaikan persoalannya dalam $O(V + E)$. Begitu sisinya membawa bobot, antrean tidak lagi cukup — rute dengan lompatan paling sedikit belum tentu rute dengan kilometer paling sedikit — dan Anda memerlukan antrean berprioritas serta algoritma Dijkstra (Bab 13), atau, jika sebagian bobotnya negatif, Bellman-Ford. Kita perlu tepat tentang hal ini, sebab inilah sumber jawaban salah yang paling sering muncul dalam praktik:

PERINGATAN: Penelusuran melebar meminimumkan **banyaknya sisi**. Jika sisi Anda punya panjang dan panjangnya tidak semuanya sama, BFS akan dengan yakin mengembalikan lintasan yang bukan terpendek. Rute tiga lompatan bisa saja lebih pendek dalam kilometer daripada rute dua lompatan. Pakailah BFS ketika setiap sisi berongkos sama, dan pakailah Bab 13 selain itu.

10.4.15 Sebanyak apa sisinya bisa jadi? Jarang dan padat

Kita kini cukup tahu untuk membatasi $|E|$ dalam kaitannya dengan $|V|$, dan batas itu mengatur setiap keputusan perancangan di §10.5. Tulislah $n = |V|$ dan $m = |E|$.

Graf sederhana tak berarah. Sisi terbanyak yang mungkin adalah sisi graf lengkap K_n, yang diberikan Persamaan (10.3):

$$0 \le m \le \binom{n}{2} = \frac{n(n-1)}{2} = O(n^2).$$

Graf berarah, tanpa gelung-diri. Tiap pasangan terurut boleh membawa sebuah busur, jadi

$$0 \le m \le n(n-1) = O(n^2).$$

Graf berarah, gelung-diri diizinkan. Setiap satu dari n^2 pasangan terurut, termasuk (v, v), boleh membawa sebuah busur:

$$0 \le m \le n^2.$$

Jadi dalam setiap kasus $m = O(n^2)$, dan ujung atas rentangnya bersifat kuadratik. Tetapi ada pula batas *bawah* yang pantas diketahui: jika grafnya terhubung maka $m \ge n - 1$, sebab graf

terhubung memuat sebuah pohon merentang dan pohon dengan n simpul punya $n-1$ sisi (§10.4.11).

Rentang inilah — dari kira-kira n sampai kira-kira n^2 — yang dinamai kata *jarang* dan *padat*.

> **DEFINISI.** Sebuah graf disebut **jarang** (*sparse*) ketika $|E|$ jauh lebih dekat ke $|V|$ daripada ke $|V|^2$ — secara tak resmi, $|E| = O(|V|)$ atau $O(|V|\log|V|)$. Ia disebut **padat** (*dense*) ketika $|E|$ dekat dengan maksimumnya, $\Theta(|V|^2)$.

Istilahnya tak resmi tetapi bedanya nyata dan biasanya ditentukan oleh penerapannya, bukan oleh algoritmanya. **Tabel 10.2** memaparkan hitungannya untuk beberapa ukuran, dan lajur terakhirnyalah yang menentukan bagaimana Anda menyimpan grafnya.

Tabel 10.2 Seberapa jauh sesungguhnya jarak antara jarang dan padat. Untuk sejuta simpul, sebuah matriks memuat 10^{12} entri — satu terabita bahkan pada satu bita per entri — sementara graf jarang dengan simpul yang sama mungkin hanya punya sejuta sisi. Tabel tunggal inilah alasan §10.5 menghabiskan waktu begitu lama pada representasi.

$\|V\|$	$\|E\|$ Jika pohon (V−1)	$\|E\|$ Jika lengkap	Entri dalam matriks $V\times V$	Nisbah, lengkap : pohon
10	9	45	100	5
100	99	4 950	10 000	50
1 000	999	499 500	1 000 000	500
10 000	9 999	49 995 000	100 000 000	5 000
1 000 000	999 999	$\approx 5\times 10^{11}$	10^{12}	$\approx 5\times 10^{5}$

Kasus mana yang nyata? Hampir semua graf besar yang muncul dalam praktik bersifat jarang, dan alasannya sederhana: benda nyata punya jumlah hubungan yang terbatas. Sebuah simpang jalan paling banyak punya segelintir jalan; seseorang punya ratusan teman, bukan jutaan; sebuah halaman web punya puluhan tautan, bukan miliaran. Graf padat memang ada — matriks jarak antara setiap pasangan kota, matriks kemiripan antara setiap pasangan dokumen — tetapi itu perkecualian, dan graf itu kecil, justru karena n^2 tumbuh begitu cepat.

10.4.16 Seluruh kosakatanya dalam satu halaman

Tabel 10.3 mengumpulkan setiap istilah bagian ini, beserta maknanya dalam satu baris dan sebuah contohnya di dalam graf G pada **Gambar 10.1**. Kembalilah ke sini setiap kali ada kata dalam sisa bab ini yang terasa asing.

Tabel 10.3 Kosakata lengkap §10.4, beserta contoh tiap istilah di dalam graf **Gambar 10.1**. Setiap satu dari kata-kata ini dipakai di suatu tempat dalam sisa bab ini.

Istilah	Makna dalam satu baris	Contoh di G
simpul	salah satu bendanya	a
sisi	sepasang benda yang terhubung	(a, b)
bertetangga	terhubung oleh sebuah sisi	a dan c ya; a dan e tidak
bersisian	dikatakan tentang sisi dan simpul di ujungnya	(b, e) bersisian dengan b dan dengan e
$\mathrm{Adj}[v]$	tetangga dari v	$\mathrm{Adj}[c] = \{a, d, e\}$
$\deg(v)$	banyaknya sisi pada v	$\deg(b) = 2$
$\deg^+, \deg^-$	derajat keluar, derajat masuk (hanya graf berarah)	—
lintasan	barisan simpul, yang berurutan bertetangga	$a\ b\ e\ d\ c$
panjang lintasan	banyaknya sisi di dalamnya	$a\ b\ e\ d\ c$ berpanjang 4
lintasan sederhana	lintasan tanpa simpul berulang	$b\ e\ c$
$\delta(u, v)$	sisi paling sedikit pada lintasan u ke v	$\delta(b, c) = 2$
siklus	lintasan sederhana yang simpul terakhirnya simpul pertamanya	$a\ c\ d\ a$
tak bersiklus	tidak punya siklus	G tidak
terhubung	setiap pasangan terhubung oleh suatu lintasan	G ya
komponen terhubung	subgraf terhubung yang maksimal	G punya 1
terhubung kuat	berarah: setiap pasangan terhubung dua arah	hanya graf berarah
subgraf	himpunan bagian simpul dengan sebagian sisi di antaranya	$\{a, b, e\}$ dengan $(a, b), (b, e)$
subgraf merentang	subgraf yang mempertahankan setiap simpul	$\{a, b, c, d, e\}$ dengan 4 dari 7 sisi
pohon	terhubung dan tak bersiklus	sembarang pohon merentang G
hutan	tak bersiklus, boleh tak terhubung	pohon adalah hutan berkomponen satu
daun	simpul berderajat 1 di dalam pohon	—
graf lengkap K_n	setiap pasangan bertetangga	G bukan: 3 pasangan hilang
bipartit	simpul terbelah dua kubu, setiap sisi menyeberang	G bukan — ia punya segitiga
berbobot	tiap sisi membawa sebuah bilangan	G bukan

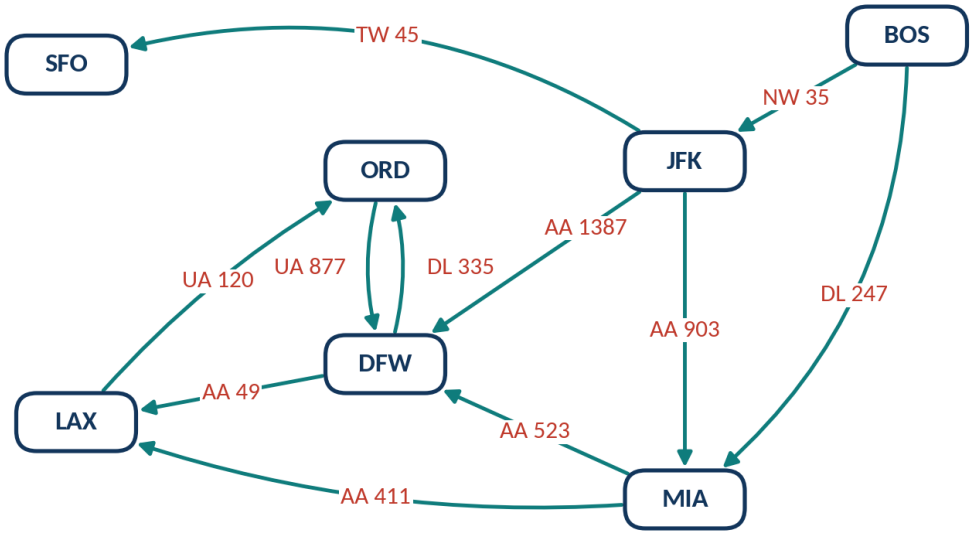
Tabel 10.3 (Lanjutan) *Kosakata lengkap §10.4, beserta contoh tiap istilah di dalam graf Gambar 10.1. Setiap satu dari kata-kata ini dipakai di suatu tempat dalam sisa bab ini.*

Istilah	Makna dalam satu baris	Contoh di G
jarang / padat	$\lvert E\rvert$ dekat $\lvert V\rvert$ / dekat $\lvert V\rvert^2$	G punya 7 dari 10 yang mungkin: padat
sumber / muara	derajat masuk 0 / derajat keluar 0	hanya graf berarah
gelung-diri	sisi dari sebuah simpul ke dirinya sendiri	tidak ada di G
DAG	graf berarah tanpa siklus berarah	§10.9.5

10.5 Tiga Cara Menyimpan Graf

10.5.1 Apa yang harus dikerjakan sebuah representasi

Pertanyaannya sederhana: *bagaimana kita dapat merepresentasikan sebuah graf?* Jawaban pertama yang diberikan siapa pun adalah: *untuk memulai, kita dapat menyimpan simpul dan sisinya ke dalam dua wadah, dan pada tiap objek sisi kita simpan rujukan ke simpul awal dan simpul akhirnya.* **Gambar 10.11** adalah graf yang kita pakai untuk membuat hal ini konkret — graf berarah berisi penerbangan, dengan tiap busur membawa nomor penerbangan — dan **Gambar 10.12** memperlihatkan struktur dua wadah yang dibangun di atasnya.



Gambar 10.11 Contoh berjalan untuk penyimpanan graf: graf berarah berisi penerbangan antara tujuh bandara (BOS, JFK, ORD, SFO, LAX, DFW, MIA), dengan tiap busur diberi label nomor penerbangan. Ada sebelas busur. Karena busurnya berarah, penerbangan AA 49 dari DFW ke LAX adalah sisi yang berbeda dari penerbangan DL 335 dari DFW ke ORD, dan tak satu pun di antaranya menyiratkan adanya penerbangan ke arah sebaliknya.

Sebelum membandingkan representasi, kita harus bersepakat tentang atas dasar apa kita membandingkannya. Setiap algoritma dalam bab ini, dan hampir setiap algoritma graf di mana pun, dibangun dari hanya empat pertanyaan:

P1. Adakah sisi dari u ke v? Sebutlah ini *uji ketetanggaan*. Algoritma bercita rasa matriks menanyakannya terus-menerus.

P2. Berapa $\deg(v)$**?** Atau, pada graf berarah, $\deg^+(v)$.

P3. Daftarkan tetangga v. Artinya, cacahlah $\mathrm{Adj}[v]$. **Inilah yang paling penting**: kedua penelusuran dalam bab ini tidak mengerjakan yang lain, dan mengerjakannya sekali per simpul itulah yang membuat keduanya linear.

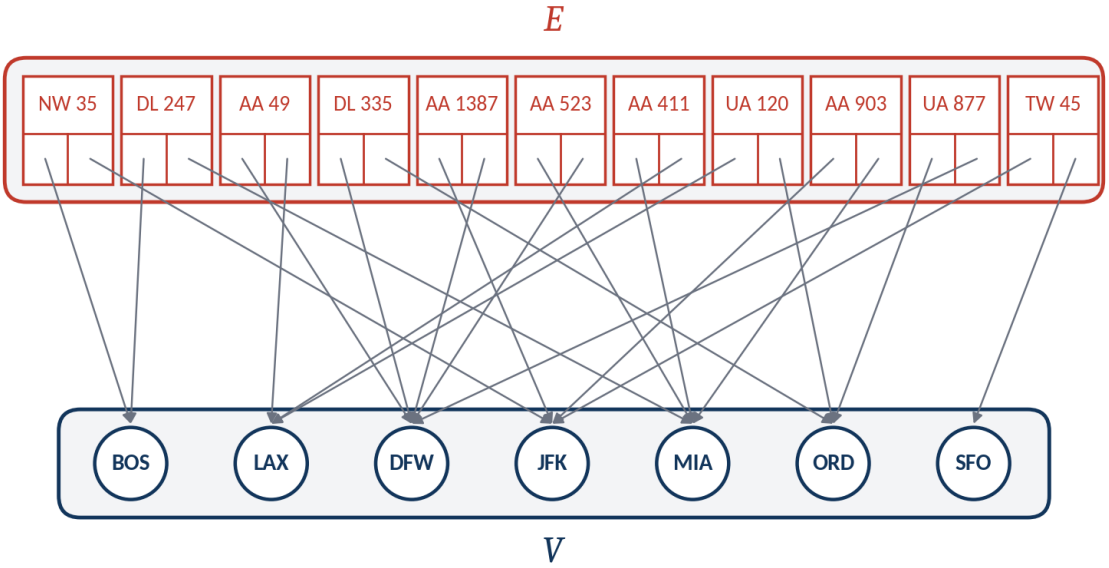
P4. Daftarkan semua sisinya. Diperlukan oleh algoritma Kruskal di Bab 11, di antara yang lain.

Dan satu ukuran lagi, yang bukan pertanyaan melainkan kendala: **berapa banyak memori yang ditempati representasinya?**

Dengan kelima tolok ukur itu disepakati, inilah ketiga representasinya, masing-masing digambarkan dalam sebuah gambar.

10.5.2 Daftar sisi

DEFINISI. Pada representasi **daftar sisi** (*edge list*), grafnya disimpan sebagai sebuah wadah berisi objek simpul bersama sebuah wadah berisi objek sisi, dengan tiap objek sisi memuat rujukan ke kedua ujungnya.



Gambar 10.12 Daftar sisi, dibangun pada graf penerbangan **Gambar 10.11**. Wadah bawah memuat ketujuh simpulnya; wadah atas memuat kesebelas sisinya, dan setiap objek sisi menunjuk ke bawah pada kedua ujungnya. Untuk menemukan penerbangan yang meninggalkan DFW, Anda harus memeriksa setiap satu dari sebelas objek sisi itu, sebab tidak ada apa pun di dalam strukturnya yang mengelompokkannya menurut bandara.

Dalam C++ seluruh representasinya hanya dua baris:

```
int n;                          // simpulnya adalah 0, 1, ..., n-1
std::vector<std::pair<int,int>> E;   // satu entri per sisi
```

Untuk graf **Gambar 10.1**, wadahnya memuat

$$E = [(a,b),\ (a,c),\ (a,d),\ (b,e),\ (c,d),\ (c,e),\ (d,e)].$$

Ruang. Dua rujukan simpul per sisi, ditambah simpul-simpulnya sendiri: $\Theta(V+E)$. Dengan bilangan bulat menggantikan rujukan, itu berarti $2|E|$ kata mesin untuk sisinya — untuk graf kita, 14.

P1, uji ketetanggaan. Tidak ada yang dapat dilakukan selain memindai seluruh wadahnya mencari pasangan itu. $\Theta(E)$ pada kasus terburuk.

P2, derajat. Pindai seluruh wadahnya sambil mencacah entri yang menyebut v. $\Theta(E)$.

P3, daftarkan tetangganya. Pindai seluruh wadahnya sambil mengumpulkan entri yang menyebut v. $\Theta(E)$ — dan inilah cacat sesungguhnya representasi ini: *menemukan sisi yang bersisian dengan sebuah simpul tertentu tidak efisien sebab ia menuntut pemeriksaan seluruh barisan sisinya.*

P4, daftarkan semua sisinya. Bacakan isi wadahnya. $\Theta(E)$, dan itu optimal — tidak ada cara lebih cepat untuk mencetak $|E|$ benda.

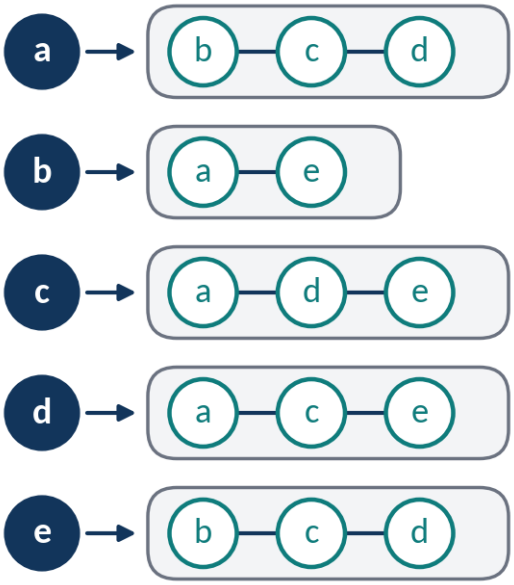
Jadi daftar sisi adalah yang terbaik di antara ketiganya pada tepat satu pekerjaan, P4, dan yang terburuk pada tiga yang lain. Itu sama sekali tidak membuatnya tak berguna: algoritma Kruskal untuk pohon merentang minimum (Bab 11) bermula dengan mengurutkan *semua* sisinya lalu

tidak pernah menanyakan P1, P2, atau P3, sehingga daftar sisi persis merupakan struktur yang tepat baginya. Dalam praktik, ini juga format tempat graf biasanya tiba — sebuah berkas dengan satu sisi per baris — sehingga program sungguhan biasanya membaca daftar sisi lalu mengubahnya menjadi sesuatu yang lain. Pengubahan itu sembilan baris dan muncul di **Kode 10.1**.

10.5.3 Daftar ketetanggaan

> **DEFINISI. Daftar ketetanggaan** sebuah simpul v adalah barisan yang memuat simpul-simpul yang bertetangga dengan v. Pada representasi **daftar ketetanggaan** (*adjacency list*), grafnya disimpan sebagai larik berisi $|V|$ barisan semacam itu, satu per simpul.

Ringkasnya: *daftar ketetanggaan sebuah simpul v adalah barisan simpul yang bertetangga dengan v; grafnya direpresentasikan dengan daftar ketetanggaan semua simpulnya.* **Gambar 10.13** menggambarkannya untuk graf bab ini.



Gambar 10.13 Daftar ketetanggaan graf bab ini. Lima daftar, satu per simpul. Daftar simpul a memuat b, c, d; daftar simpul b memuat a, e; dan seterusnya. Daftar-daftar itu bersama-sama memuat 14 entri, yaitu $2|E|$ — lema jabat tangan, dibuat kasatmata.

Dalam C++:

```
int n;
std::vector<std::vector<int>> adj;   // adj[v] memuat tetangga dari v
```

Untuk graf kita kelima daftarnya adalah

$$\mathrm{Adj}[a] = [b, c, d], \quad \mathrm{Adj}[b] = [a, e], \quad \mathrm{Adj}[c] = [a, d, e], \quad \mathrm{Adj}[d] = [a, c, e], \quad \mathrm{Adj}[e] = [b, c, d].$$

Perhatikan baik-baik bahwa pada graf **tak berarah** setiap sisi muncul **dua kali**: sisi (a, b) muncul sebagai b di $\mathrm{Adj}[a]$ dan muncul lagi sebagai a di $\mathrm{Adj}[b]$. Itu bukan kemubaziran yang perlu dioptimalkan agar hilang; itulah yang membuat P3 cepat dari kedua ujungnya. Pada graf **berarah** tiap busur muncul sekali, di daftar milik ekornya.

Ruang. $|V|$ kepala daftar ditambah satu entri per slot sisi berarah. Tak berarah: $\Theta(V + 2E) = \Theta(V + E)$. Berarah: $\Theta(V + E)$. Untuk graf kita: 5 kepala ditambah 14 entri, 19 kata.

P1, uji ketetanggaan. Susurilah Adj[u] mencari v: $O(\deg(u))$. Itu cepat ketika derajatnya kecil dan lambat ketika tidak — pada kasus terburuk $O(V)$.

P2, derajat. Panjang daftarnya, yang sudah diketahui sebuah `std::vector`: $\Theta(1)$.

P3, daftarkan tetangganya. Bacalah Adj[v]: $\Theta(\deg(v))$, dan itu optimal, sebab Anda tidak dapat mendaftarkan $\deg(v)$ benda lebih cepat daripada itu.

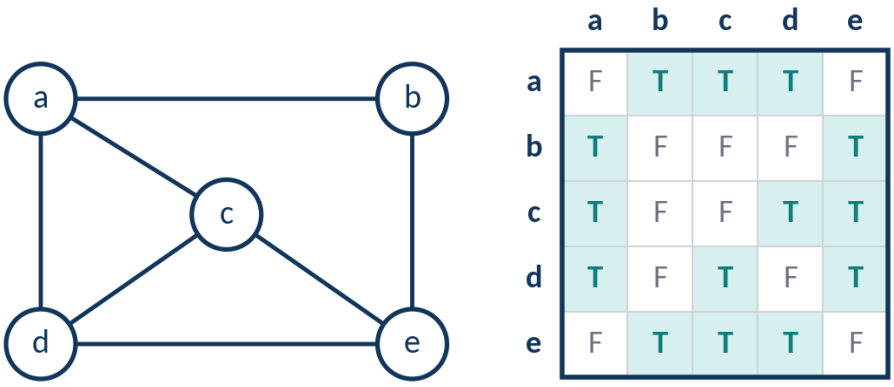
P4, daftarkan semua sisinya. Susurilah setiap daftar: $\Theta(V + E)$. Sedikit lebih buruk daripada $\Theta(E)$ milik daftar sisi, sebab Anda harus menyinggahi $|V|$ kepala daftarnya sekalipun sebagian daftarnya kosong, dan tiap sisi tak berarah dijumpai dua kali sehingga Anda harus ingat untuk memancarkannya sekali saja.

> **WAWASAN KUNCI:** Daftar ketetanggaan adalah pilihan bawaan. Inilah representasi yang diandaikan kedua penelusuran bab ini — BFS maupun DFS dianalisis untuk representasi daftar ketetanggaan — sebab P3 adalah satu-satunya pertanyaan yang diajukan sebuah penelusuran, dan daftar ketetanggaan menjawab P3 secara optimal sambil hanya memakai ruang $\Theta(V + E)$. Kecuali Anda punya alasan khusus untuk berbuat lain, simpanlah graf Anda dengan cara ini.

10.5.4 Matriks ketetanggaan

> **DEFINISI. Matriks ketetanggaan** sebuah graf dengan n simpul adalah larik $n \times n$ bernama M yang $M[i][j]$-nya bernilai **benar** jika ada sisi dari simpul i ke simpul j, dan **salah** jika tidak ada.

Gambar 10.14 menggambarkannya: grafnya di sebelah kiri, matriks berisi T dan F di sebelah kanan.



Gambar 10.14 Matriks ketetanggaan graf bab ini, digambar berdampingan dengan grafnya sendiri. Baris c terbaca T, F, F, T, T: simpul c bertetangga dengan a, d, dan e tetapi tidak dengan b, dan entri diagonalnya F sebab tidak ada gelung-diri. Matriksnya setangkup terhadap diagonalnya sebab grafnya tak berarah, dan ia punya 25 entri untuk graf yang hanya punya 7 sisi.

Menuliskan matriksnya dengan 1 untuk benar dan 0 untuk salah, serta membaca **Gambar 10.14** baris demi baris:

$$M = \begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 1 & 0 \\ b & 1 & 0 & 0 & 0 & 1 \\ c & 1 & 0 & 0 & 1 & 1 \\ d & 1 & 0 & 1 & 0 & 1 \\ e & 0 & 1 & 1 & 1 & 0 \end{array}$$

Cocokkan satu baris dengan E: baris c punya 1 di kolom a, d, e, dan $(a, c), (c, d), (c, e)$ memang ketiga sisi pada c. Cocokkan satu ketiadaan: $M[a][e] = 0$, dan $(a, e) \notin E$. Bagus.

Tiga fakta struktural, masing-masing dengan alasannya:

- **Matriksnya setangkup** — $M[i][j] = M[j][i]$ — persis ketika grafnya tak berarah, sebab pada graf semacam itu sisi (i, j) dan sisi (j, i) adalah sisi yang sama. Untuk graf berarah matriksnya pada umumnya tidak setangkup, dan ketaksetangkupannya *itulah* keterangan arahnya.
- **Diagonalnya seluruhnya nol** tepat ketika tidak ada gelung-diri. Gelung-diri pada v menaruh 1 di $M[v][v]$.
- **Jumlah barisnya adalah derajatnya.** Pada baris v, banyaknya 1 adalah banyaknya tetangga v. Pada graf berarah, jumlah barisnya adalah $\deg^+(v)$ dan jumlah kolomnya adalah $\deg^-(v)$.

Ruang. $\Theta(V^2)$, selalu — 25 kata untuk graf lima simpul kita, berapa pun $|E|$-nya — $O(n^2)$ untuk n simpul. Inilah seluruh persoalan representasi ini: graf dengan sejuta simpul memerlukan 10^{12} entri, yaitu satu terabita pada satu bita per entri, dan tidak mungkin dialokasikan, sesedikit apa pun sisi yang dipunyai grafnya.

P1, uji ketetanggaan. Satu pembacaan larik: $\Theta(1)$. Inilah seluruh keunggulan representasi ini, dan keunggulan itu nyata.

P2, derajat. Jumlahkan sebuah baris: $\Theta(V)$.

P3, daftarkan tetangganya. Susurilah sebuah baris sambil memungut angka 1-nya: $\Theta(V)$ — dan perhatikan bahwa ini $\Theta(V)$ *bahkan ketika v sama sekali tidak punya tetangga*, sebab Anda tetap harus memeriksa $|V|$ sel untuk mengetahuinya. Itulah persis sebabnya penelusuran berbasis matriks berongkos $\Theta(V^2)$, bukan $O(V + E)$, dan §10.9.8 mengukur bedanya.

P4, daftarkan semua sisinya. Pindai seluruh matriksnya: $\Theta(V^2)$.

Satu catatan terakhir tentang implementasi. `std::vector<bool>` dalam C++ adalah spesialisasi yang dikemas per bit, yang membuat sebuah matriks delapan kali lebih kecil dengan bayaran akses elemen yang lebih lambat dan sedikit perilaku yang mengejutkan (`operator[]`-nya tidak mengembalikan `bool&` yang sesungguhnya). Demi kejelasan, program di bawah memakai `std::vector<char>`; untuk matriks padat yang sungguh-sungguh besar, pengemasan per bit adalah imbal-tukar yang tepat dan `std::vector<bool>` atau himpunan bit rakitan sendiri adalah perkakasnya.

10.5.5 Ruang, berdampingan

Tabel 10.4 Biaya ruang ketiga representasi. Untuk graf mungil **Gambar 10.1**, ketiganya berjarak tidak sampai dua kali lipat satu sama lain, dan itulah sebabnya contoh kecil menyesatkan. Pada sejuta simpul, matriksnya sama sekali tidak bisa dipertimbangkan.

Representasi	Ruang, secara umum	Ruang untuk (V=5, E=7)	Ruang untuk V=10^6, E=4×10^6
Daftar sisi	$\Theta(V + E)$	14 kata	$\approx 8 \times 10^6$ kata
Daftar ketetanggaan	$\Theta(V + E)$	19 kata	$\approx 9 \times 10^6$ kata
Matriks ketetanggaan	$\Theta(V^2)$	25 kata	10^{12} entri — mustahil

Pelajaran dari **Tabel 10.4** adalah bahwa contoh kecil menyembunyikan bedanya. Pada lima simpul, matriksnya berongkos 25 kata melawan 19 kata milik daftar ketetanggaan — tidak ada yang akan peduli. Titik impasnya berada di tempat $V^2 \approx V + 2E$, yaitu kira-kira $E \approx V^2/2$: matriks menang soal ruang hanya untuk graf yang sungguh-sungguh mendekati lengkap.

10.5.6 Keempat pertanyaan, berdampingan

Tabel 10.5 Biaya setiap operasi dasar pada masing-masing dari ketiga representasi. Tidak ada representasi yang menang di setiap baris, dan itulah sebabnya ketiganya bertahan hidup. Baris terakhirlah yang menentukan isi bab ini: penelusuran atas daftar ketetanggaan bersifat linear; penelusuran yang sama atas matriks bersifat kuadratik.

Pertanyaan	Daftar sisi	Daftar ketetanggaan	Matriks ketetanggaan
P1: apakah (u, v) sebuah sisi?	$\Theta(E)$	$O(\deg u)$, terburuk $O(V)$	$\Theta(1)$
P2: berapa $\deg(v)$?	$\Theta(E)$	$\Theta(1)$	$\Theta(V)$
P3: daftarkan $\mathrm{Adj}[v]$	$\Theta(E)$	$\Theta(\deg v)$	$\Theta(V)$
P4: daftarkan semua sisi	$\Theta(E)$	$\Theta(V + E)$	$\Theta(V^2)$
Tambahkan sebuah sisi	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Hapus sisi (u, v)	$\Theta(E)$	$O(\deg u + \deg v)$	$\Theta(1)$
Ruang	$\Theta(V + E)$	$\Theta(V + E)$	$\Theta(V^2)$
Biaya satu BFS atau DFS penuh	—	$\Theta(V + E)$	$\Theta(V^2)$

10.5.7 Memilih

Pilihannya mengalir dari dua pertanyaan tentang penerapan Anda.

Pertama: apakah grafnya jarang atau padat? Jika $|E|$ jauh lebih kecil daripada $|V|^2$ — dan itulah kasus yang lazim — matriksnya memboroskan banyak sekali memori, dan yang lebih buruk, ia membuat setiap penelusuran menjadi kuadratik. Pakailah daftar ketetanggaan. Jika grafnya sungguh-sungguh padat, matriksnya tidak menghabiskan memori lebih banyak daripada daftarnya dan lebih cepat di segala hal kecuali P4.

Kedua: antara P1 dan P3, mana yang mendominasi? Jika gelung terdalam algoritma Anda adalah "apakah (u, v) sebuah sisi?", jawaban $\Theta(1)$ milik matriks layak dibayar; algoritma Floyd-Warshall di Bab 15 dibangun dengan cara ini dan memakai matriks tanpa perlu meminta maaf. Jika gelung terdalamnya adalah "berikan saya tetangga v" — dan itulah setiap penelusuran, setiap pencarian, Prim, Dijkstra, BFS, DFS — pakailah daftar ketetanggaan.

Tabel 10.6 Kapan meraih representasi yang mana, dan alasannya untuk tiap kasus. Jika ragu, pilihlah daftar ketetanggaan; ia jauh lebih sering menjadi jawaban yang benar daripada tidak.

Keadaan	Pakai	Mengapa
Sebuah penelusuran, penjelajahan, BFS, DFS, Prim, Dijkstra	Daftar ketetanggaan	P3 satu-satunya pertanyaan yang diajukan, dan daftarnya menjawabnya secara optimal
Graf jarang, $E = O(V)$ atau $O(V \log V)$	Daftar ketetanggaan	Ruang $\Theta(V + E)$, bukan $\Theta(V^2)$
Graf padat, $E = \Theta(V^2)$, dengan uji sisi berwaktu tetap yang diperlukan	Matriks ketetanggaan	Ruang $\Theta(V^2)$ tidak menambah biaya apa pun, dan P1 menjadi $\Theta(1)$
Lintasan terpendek semua pasangan, ketertutupan transitif, algoritma bergaya matriks	Matriks ketetanggaan	Algoritmanya *adalah* aritmetika matriks (Bab 15)
Algoritma Kruskal; membaca graf dari berkas	Daftar sisi	Satu-satunya pertanyaan yang diajukan adalah P4, dan daftar sisi menjawabnya secara optimal
Sisi disisipkan dan dihapus terus-menerus, dengan uji sisi di antaranya	Matriks ketetanggaan, atau daftar ketetanggaan berisi himpunan hash	Penyisipan, penghapusan, dan pengujian $\Theta(1)$

10.5.8 Ketiganya, dalam satu program

Kode 10.1 membangun graf **Gambar 10.1** tiga kali, dalam ketiga representasi, lalu menjawab keempat pertanyaannya dari masing-masing. Menjalankan ketiganya berdampingan adalah cara tercepat untuk meyakinkan diri Anda bahwa ketiganya memuat keterangan yang sama — derajatnya harus cocok, uji ketetanggaannya harus cocok — dan bahwa biaya pemeriksaannya berbeda-beda.

Kode 10.1 Graf bab ini disimpan dengan tiga cara sekaligus, dengan keempat kueri dasarnya dijawab dari masing-masing representasi. Kompilasilah dengan `g++ -std=c++17 -O2`.

```
// Tiga representasi satu graf, berdampingan, pada graf Bab 10:
//   V = {a,b,c,d,e},  E = {(a,b),(a,c),(a,d),(b,e),(c,d),(c,e),(d,e)}
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

// ---------- Representasi 1: daftar sisi ----------
struct EdgeList {
    int n;                                  // banyaknya simpul
    vector<pair<int,int>> E;                // satu entri per sisi
    explicit EdgeList(int n_) : n(n_) {}
    void addEdge(int u, int v) { E.push_back({u, v}); }

    bool areAdjacent(int u, int v) const {          // Theta(E)
        for (const auto& e : E)
            if ((e.first == u && e.second == v) || (e.first == v && e.second == u))
                return true;
        return false;
    }
    int degree(int u) const {                       // Theta(E)
        int d = 0;
        for (const auto& e : E) if (e.first == u || e.second == u) ++d;
        return d;
    }
    vector<int> neighbors(int u) const {           // Theta(E)
        vector<int> out;
        for (const auto& e : E) {
            if (e.first  == u) out.push_back(e.second);
            if (e.second == u) out.push_back(e.first);
        }
        sort(out.begin(), out.end());
        return out;
    }
    size_t words() const { return 2 * E.size(); }   // dua int disimpan per sisi
};

// ---------- Representasi 2: daftar ketetanggaan ----------
struct AdjList {
    int n;
    vector<vector<int>> adj;
    explicit AdjList(int n_) : n(n_), adj(n_) {}
    void addEdge(int u, int v) { adj[u].push_back(v); adj[v].push_back(u); }
```

Kode 10.1 (Lanjutan) *Graf bab ini disimpan dengan tiga cara sekaligus, dengan keempat kueri dasarnya dijawab dari masing-masing representasi. Kompilasilah dengan g++ -std=c++17 -O2.*

```
    bool areAdjacent(int u, int v) const {          // O(deg(u))
        for (int w : adj[u]) if (w == v) return true;
        return false;
    }
    int degree(int u) const { return (int)adj[u].size(); }       // O(1)
    const vector<int>& neighbors(int u) const { return adj[u]; } // O(1) untuk mencapainya
    size_t words() const {                          // n kepala + 2E entri
        size_t t = n;
        for (const auto& l : adj) t += l.size();
        return t;
    }
};

// ---------- Representasi 3: matriks ketetanggaan ----------
struct AdjMatrix {
    int n;
    vector<vector<char>> M;                 // char, bukan bool: tata letaknya terduga
    explicit AdjMatrix(int n_) : n(n_), M(n_, vector<char>(n_, 0)) {}
    void addEdge(int u, int v) { M[u][v] = 1; M[v][u] = 1; }

    bool areAdjacent(int u, int v) const { return M[u][v] != 0; }  // Theta(1)
    int degree(int u) const {                                       // Theta(V)
        int d = 0;
        for (int v = 0; v < n; ++v) if (M[u][v]) ++d;
        return d;
    }
    vector<int> neighbors(int u) const {                           // Theta(V)
        vector<int> out;
        for (int v = 0; v < n; ++v) if (M[u][v]) out.push_back(v);
        return out;
    }
    size_t words() const { return (size_t)n * n; }
};

static const string NAME = "abcde";
static void printList(const vector<int>& v) {
    cout << "{";
    for (size_t i = 0; i < v.size(); ++i) cout << (i ? "," : "") << NAME[v[i]];
    cout << "}";
}

int main() {
    const int n = 5;
    const vector<pair<int,int>> edges = {{0,1},{0,2},{0,3},{1,4},{2,3},{2,4},{3,4}};
```

Kode 10.1 (Lanjutan) *Graf bab ini disimpan dengan tiga cara sekaligus, dengan keempat kueri dasarnya dijawab dari masing-masing representasi. Kompilasilah dengan g++ -std=c++17 -O2.*

```
    EdgeList  el(n);
    AdjList   al(n);
    AdjMatrix am(n);
    for (auto e : edges) { el.addEdge(e.first,e.second); al.addEdge(e.first,e.second);
am.addEdge(e.first,e.second); }

    cout << "|V| = " << n << ", |E| = " << edges.size() << "\n\n";

    cout << "adjacency matrix (1 = edge present)\n    ";
    for (int j = 0; j < n; ++j) cout << NAME[j] << ' ';
    cout << "\n";
    for (int i = 0; i < n; ++i) {
        cout << "  " << NAME[i] << ' ';
        for (int j = 0; j < n; ++j) cout << (am.areAdjacent(i,j) ? '1' : '0') << ' ';
        cout << "\n";
    }

    cout << "\nadjacency lists\n";
    for (int u = 0; u < n; ++u) {
        cout << "  " << NAME[u] << " -> "; printList(al.neighbors(u)); cout << "\n";
    }

    cout << "\nedge list\n  ";
    for (auto e : el.E) cout << "(" << NAME[e.first] << "," << NAME[e.second] << ") ";
    cout << "\n";

    cout << "\ndegrees (all three representations must agree)\n";
    int sum = 0;
    for (int u = 0; u < n; ++u) {
        int a = el.degree(u), b = al.degree(u), c = am.degree(u);
        cout << "  deg(" << NAME[u] << ") = " << a << " " << b << " " << c
             << (a==b && b==c ? "   agree" : "   MISMATCH") << "\n";
        sum += b;
    }
    cout << "  sum of degrees = " << sum << " = 2|E| = " << 2*edges.size() << "\n";

    cout << "\nadjacency queries\n";
    const int pairs[4][2] = {{0,1},{0,4},{2,4},{1,3}};
    for (auto& p : pairs)
        cout << "  " << NAME[p[0]] << "~" << NAME[p[1]] << " : "
             << (el.areAdjacent(p[0],p[1]) ? "yes" : "no ") << " "
             << (al.areAdjacent(p[0],p[1]) ? "yes" : "no ") << " "
             << (am.areAdjacent(p[0],p[1]) ? "yes" : "no ") << "\n";
```

Kode 10.1 (Lanjutan) *Graf bab ini disimpan dengan tiga cara sekaligus, dengan keempat kueri dasarnya dijawab dari masing-masing representasi. Kompilasilah dengan g++ -std=c++17 -O2.*

```
    cout << "\nstorage, counted in machine words\n";
    cout << "  edge list       " << el.words() << "\n";
    cout << "  adjacency list  " << al.words() << "\n";
    cout << "  adjacency matrix " << am.words() << "\n";
    return 0;
}
```

Keluarannya:

```
|V| = 5, |E| = 7

adjacency matrix (1 = edge present)
    a b c d e
  a 0 1 1 1 0
  b 1 0 0 0 1
  c 1 0 0 1 1
  d 1 0 1 0 1
  e 0 1 1 1 0

adjacency lists
  a -> {b,c,d}
  b -> {a,e}
  c -> {a,d,e}
  d -> {a,c,e}
  e -> {b,c,d}

edge list
  (a,b) (a,c) (a,d) (b,e) (c,d) (c,e) (d,e)

degrees (all three representations must agree)
  deg(a) = 3 3 3   agree
  deg(b) = 2 2 2   agree
  deg(c) = 3 3 3   agree
  deg(d) = 3 3 3   agree
  deg(e) = 3 3 3   agree
  sum of degrees = 14 = 2|E| = 14

adjacency queries
  a~b : yes yes yes
  a~e : no  no  no
  c~e : yes yes yes
  b~d : no  no  no

storage, counted in machine words
  edge list       14
  adjacency list  19
  adjacency matrix 25
```

Ada empat hal yang patut diperhatikan pada keluaran itu, sebab masing-masing menegaskan sesuatu yang telah kita buktikan dengan tangan.

Matriks yang tercetak persis merupakan matriks pada **Gambar 10.14** dengan 1 untuk T dan 0 untuk F. Daftar ketetanggaan yang tercetak persis merupakan kelima daftar **Gambar 10.13**. Derajatnya cocok pada ketiga representasi dan berjumlah $14 = 2|E|$, yaitu lema jabat tangan §10.4.3 yang ditegaskan oleh mesin. Dan ketiga angka penyimpanannya, 14, 19, dan 25, adalah lajur tengah **Tabel 10.4**.

10.6 Menelusuri Sebuah Graf

Segala sesuatu sejauh ini bersifat memerikan. Sekarang kita menghitung.

Tugasnya dapat dinyatakan dalam satu baris: *penelusuran sistematis atas setiap sisi dan simpul grafnya*. Dua kata dalam baris itu memikul bebannya.

Sistematis berarti: menurut sebuah aturan, bukan karena untung-untungan. Aturannya harus menjamin bahwa ketika penelusurannya berhenti, setiap simpul yang seharusnya ia kunjungi telah dikunjungi, dan tidak ada simpul yang dikunjungi dua kali. Mengunjungi dua kali bukan sekadar boros; pada graf yang punya siklus hal itu fatal, sebab kunjungan kedua menuntun ke kunjungan ketiga dan programnya tidak pernah berhenti.

Setiap sisi dan simpul berarti penelusurannya harus tuntas atas bagian graf yang dapat ia jangkau — dan perhatikan bahwa "dapat dijangkau" adalah pembatasan yang sungguhan. Penelusuran yang dimulai di s mengunjungi persis simpul-simpul yang terjangkau dari s, yang pada graf tak terhubung bukan seluruhnya. Itu adalah fitur, dan §10.9.1 membuat sebuah algoritma darinya.

Hanya ada dua aturan yang wajar, dan keduanya berbeda dalam satu keputusan: **ketika Anda punya beberapa tempat belum terjelajah untuk dituju, mana yang Anda tuju lebih dulu?**

- Tujulah yang **paling awal** Anda ketahui. Itulah **antrean** — masuk pertama, keluar pertama — dan hasilnya adalah **penelusuran melebar**.
- Tujulah yang **paling baru** Anda ketahui. Itulah **tumpukan** — masuk terakhir, keluar pertama — dan hasilnya adalah **penelusuran mendalam**.

Itulah seluruh bedanya. Dua algoritma, sama persis kecuali wadah di tengahnya, namun keduanya menghitung hal-hal yang sangat berbeda: yang satu menemukan lintasan terpendek, yang lain menemukan struktur siklus dan prasyarat. Peganglah kalimat itu sepanjang dua bagian berikutnya; kalimat itu akan menjaga Anda tetap berorientasi.

Kedua algoritma memerlukan cara untuk mengingat apa yang sudah dilihat, dan keduanya memakai perangkat yang sama: sebuah **warna** pada tiap simpul.

> **DEFINISI.** Selama penelusuran, setiap simpul berada pada tepat satu dari tiga warna. **Putih** berarti *belum ditemukan*: penelusurannya belum sampai kepadanya. **Kelabu** berarti *sudah ditemukan tetapi belum selesai*: penelusurannya sudah sampai kepadanya, tetapi belum melihat semua sisinya. **Hitam** berarti *selesai*: penelusurannya sudah sampai kepadanya dan telah memeriksa setiap sisi yang meninggalkannya.

Setiap simpul bermula putih, berubah kelabu tepat sekali, dan berubah hitam tepat sekali; warnanya hanya pernah bergerak putih → kelabu → hitam, tidak pernah mundur. Sifat searah itulah yang menjamin terminasi, dan pantas disebutkan sebabnya sebelum salah satu algoritmanya muncul: sebuah simpul dimasukkan ke antrean atau tumpukan hanya pada saat ia berubah kelabu, ia berubah kelabu hanya sekali, jadi ia masuk ke wadahnya paling banyak sekali. Ada $|V|$ simpul, jadi ada paling banyak $|V|$ penyisipan ke wadah, jadi gelung utamanya berjalan paling banyak $|V|$ kali. Tidak ada gelung tak berujung, tidak ada program menggantung.

> **WAWASAN KUNCI:** Simpul kelabu adalah **perbatasan** penelusurannya: pembatas antara bagian graf yang sudah dijelajahi dan bagian yang belum terlihat. Penelusuran melebar dan penelusuran mendalam adalah dua disiplin berbeda untuk mengelola perbatasan itu. BFS menyimpannya sebagai antrean sehingga perbatasannya menyebar keluar dalam cincin-cincin berjarak sama. DFS menyimpannya sebagai tumpukan sehingga perbatasannya berupa satu rantai panjang yang menghunjam menjauhi titik awalnya: *simpul kelabu membentuk rantai linear*, sebagaimana akan diperlihatkan §10.8.1.

Sepatah catatan tentang nama warnanya. Buku ini menulis *white*, *gray*, *black* (putih, kelabu, hitam), mengikuti Cormen, Leiserson, Rivest, dan Stein [1]. Buku lain menyebutnya *unvisited*, *visiting*, *visited*, atau *undiscovered*, *in progress*, *done*. Ketiganya adalah keadaan yang sama. Dan jika Anda hanya perlu tahu *apakah* sebuah simpul sudah pernah dilihat, dua keadaan sudah cukup — larik `bool visited[]` biasa — dan itulah sebabnya banyak kode pemrograman kompetitif memakai larik boolean dan sama sekali tanpa warna. Skema tiga warna memperoleh manfaat dari keadaan tambahannya di §10.8, tempat beda antara kelabu dan hitam persis merupakan pembeda antara sisi balik dan sisi silang, dan karena itu persis merupakan pendeteksi siklus.

10.7 Penelusuran Melebar

10.7.1 Labirin dan seutas benang

Gambaran klasik untuk penelusuran melebar, dari Goodrich dan Tamassia [2], pantas dikutip utuh, sebab gambaran itu lebih baik daripada algoritmanya dalam menjelaskan algoritmanya.

BFS pada graf tak berarah G bagaikan mengembara di dalam labirin dengan seutas benang. Simpul awal s diberi jarak 0. Pada babak pertama, benangnya diulur sepanjang satu sisi, dan semua sisi yang hanya berjarak satu sisi dari jangkarnya dikunjungi (ditemukan), dan diberi jarak 1. Pada babak kedua, semua sisi baru yang dapat dicapai dengan mengulur benangnya sepanjang 2 sisi dikunjungi dan diberi jarak 2. Ini berlanjut sampai setiap simpul memperoleh sebuah aras. Label sembarang simpul v bersesuaian dengan panjang lintasan terpendek (dalam satuan sisi) dari s ke v.

Anggaplah kiasan itu dengan sungguh-sungguh. Anda berdiri di s, memegang ujung segulung benang, dan ujung yang lain sudah Anda ikatkan ke s. Anda tidak boleh berlari menyusuri lorong; Anda harus mengulur benangnya **semeter demi semeter**, dan pada tiap panjang Anda harus menyusuri seluruh lingkaran tempat yang dijangkau meter itu sebelum mengulur lebih jauh. Sesudah satu meter Anda sudah berdiri di setiap ruangan yang berjarak satu lorong dari s. Sesudah dua meter, di setiap ruangan yang berjarak dua lorong — dan yang penting, *hanya* ruangan yang belum pernah Anda datangi, sebab ruangan yang Anda capai pada satu meter tidak menjadi berjarak dua meter hanya karena ada pula jalan yang lebih panjang menuju ke sana.

Anak kalimat terakhir itulah seluruh alasan mengapa algoritmanya benar, dan itu pula alasan mengapa ia memerlukan warnanya. Begitu sebuah ruangan dicapai, jaraknya tertetapkan selamanya, sebab ia dicapai pada uluran paling awal yang mungkin. Rute mana pun yang datang belakangan pasti setidaknya sama panjangnya. Jadi:

> **WAWASAN KUNCI:** Pada penelusuran melebar, sebuah simpul ditemukan pada saat *paling awal* ia dapat ditemukan, dan jaraknya dituliskan saat itu lalu tidak pernah ditinjau ulang. Sifat "sentuhan pertama menang" inilah yang membuat BFS menjadi algoritma lintasan terpendek, dan inilah persis yang runtuh jika Anda lupa memeriksa warnanya sebelum memasukkan simpul ke antrean.

Penelusurannya berjalan dalam **cincin**: mula-mula s sendirian, lalu segala sesuatu berjarak 1, lalu segala sesuatu berjarak 2, dan seterusnya. Warnanya membuat cincin itu kasatmata. Ketika penelusurannya sudah selesai dengan jarak k dan hendak memulai jarak $k+1$, simpul hitam adalah simpul berjarak $\leq k-1$, simpul kelabu persis merupakan cincin berjarak k, dan segala sesuatu berjarak $>k$ masih putih.

10.7.2 Tiga hal yang dihitung BFS

Penelusuran melebar dari simpul sumber s memelihara tiga larik, dan menghasilkan ketiganya sebagai jawabannya.

`color[v]` — putih, kelabu, atau hitam, seperti di §10.6. Ini pembukuan; ia dibuang di akhir. Satu-satunya tugasnya adalah memastikan tidak ada simpul yang ditemukan dua kali.

`d[v]` — **jarak** dari s ke v: banyaknya sisi pada lintasan terpendek. Ia bermula pada ∞ untuk setiap simpul selain s, yang berarti "belum diketahui terjangkau", dan ditetapkan tepat sekali, pada saat v ditemukan. Lambang ∞ dipakai alih-alih -1 sebab ia membuat hitungannya jujur: memang benar bahwa belum ada lintasan berpanjang berhingga yang ditemukan. Di dalam

program, `INT_MAX` atau nilai apa pun yang lebih besar daripada $|V|$ sudah memadai, sebab tidak ada lintasan terpendek yang melampaui $|V| - 1$ sisi (§10.4.5).

`π[v]` — **pendahulu** v, dibaca "pi dari v": simpul tempat v ditemukan. Ia bermula pada NIL untuk semua orang dan ditetapkan tepat sekali, pada saat yang sama dengan `d[v]`. Larik π inilah yang memungkinkan Anda memulihkan *rute*-nya sendiri, bukan hanya panjangnya: untuk pergi dari s ke v, carilah $\pi[v]$, lalu $\pi[\pi[v]]$, dan teruskan sampai Anda sampai di s; lalu baliklah apa yang telah Anda tuliskan. §10.7.10 menunjukkan bahwa larik π adalah sebuah pohon, dan §10.7.11 membuktikan bahwa lintasan yang ia sandikan adalah lintasan terpendek.

10.7.3 Algoritmanya

Gambar 10.15 memberikan BFS, diberi kode warna menurut apa yang dikerjakan tiap blok barisnya.

```
BFS (G, s)
01  for each vertex u ∈ V[G] − {s}
02      color[u] ← WHITE
03      d[u] ← ∞
04      π[u] ← NIL
05  color[s] ← GRAY
06  d[s] ← 0
07  π[s] ← NIL
08  Q ← {s}
09  while Q ≠ ∅ do
10      u ← head[Q]
11      for each v ∈ Adj[u] do
12          if color[v] = WHITE then
13              color[v] ← GRAY
14              d[v] ← d[u] + 1
15              π[v] ← u
16              ENQUEUE (Q, v)
17      DEQUEUE (Q)
18      color[u] ← BLACK
```

Inisialisasi setiap simpul kecuali s

Inisialisasi sumber dan antreannya

Tangani semua anak u sebelum menangani anak dari anak mana pun

Gambar 10.15 Penelusuran melebar. Blok amber (baris 01–04) menginisialisasi setiap simpul selain sumbernya; blok toska (05–08) menginisialisasi sumbernya dan antreannya; blok abu-abu (09–18) adalah penelusurannya sendiri, dan catatan di sebelah kanan menyatakan apa yang ia capai — *tangani semua anak u sebelum menangani anak mana pun dari anak-anaknya*, dan itulah definisi melebar.

Marilah kita baca baris demi baris, dengan kata-kata.

- **Baris 01–04.** Untuk setiap simpul u selain sumbernya: catlah putih, tetapkan jaraknya ∞, dan tetapkan pendahulunya NIL. Belum ada yang diketahui tentang apa pun. Gelung ini berjalan $|V| - 1$ kali dan mengerjakan kerja tetap setiap kali, jadi ongkosnya $\Theta(V)$.
- **Baris 05.** Catlah sumbernya kelabu. Ia sudah ditemukan — kita sedang berdiri di atasnya — tetapi kita belum melihat sisi-sisinya.
- **Baris 06.** Jarak sumbernya ke dirinya sendiri adalah 0.

- **Baris 07.** Sumbernya tidak punya pendahulu; tidak ada yang menemukannya.
- **Baris 08.** Masukkan sumbernya ke antrean. Antreannya kini memuat seluruh perbatasannya, yang berisi satu simpul.
- **Baris 09.** Selama antreannya tidak kosong — artinya, selama masih ada perbatasan yang harus dikerjakan.
- **Baris 10.** Misalkan u adalah simpul di kepala antrean. Karena wadahnya *antrean*, inilah simpul yang paling lama menunggu, karena itu simpul yang paling awal ditemukan, karena itu salah satu yang paling dekat ke s yang belum selesai.
- **Baris 11.** Untuk setiap tetangga v dari u — artinya, susurilah $\text{Adj}[u]$ sekali.
- **Baris 12.** Jika v putih, ia belum pernah dilihat sebelumnya. Uji tunggal inilah yang mencegah algoritmanya berputar selamanya, dan inilah yang membuat "sentuhan pertama menang" menjadi benar.
- **Baris 13.** Catlah v kelabu: ia kini ditemukan dan bergabung dengan perbatasannya.
- **Baris 14.** Jaraknya satu lebih banyak daripada jarak u. Kita mencapai v dengan mengambil satu sisi tambahan dari u.
- **Baris 15.** Pendahulunya adalah u.
- **Baris 16.** Taruhlah v di *belakang* antrean. Ia akan ditangani sesudah segala sesuatu yang sudah menunggu — dan itu berarti segala sesuatu yang lebih dekat ke s.
- **Baris 17.** Keluarkan u dari antrean; kita sudah selesai dengannya.
- **Baris 18.** Catlah u hitam. Setiap sisi yang meninggalkan u kini sudah diperiksa.

> **PERINGATAN — dua keseleo yang perlu diwaspadai, dan apa yang keduanya ajarkan.** Dua baris pada **Gambar 10.15** mudah keliru ketika algoritmanya disalin dengan tangan. Baris 07 harus terbaca $\pi[s] \leftarrow \text{NIL}$, bukan $\pi[u] \leftarrow \text{NIL}$, dan baris 15 harus terbaca $\pi[v] \leftarrow u$, bukan $\pi[u] \leftarrow u$ — kita sedang mencatat bahwa v ditemukan dari u, jadi pendahulu v-lah yang berubah, bukan pendahulu u. Alasan kita berlama-lama padanya adalah bahwa keseleo yang kedua itulah kutu sungguhan yang paling sering muncul pada kode BFS tulisan tangan. Jika Anda menulis $\pi[u] \leftarrow u$, programnya tetap berhenti, tetap mengunjungi setiap simpul, dan tetap menghitung setiap jarak dengan benar — lalu mengembalikan omong kosong ketika Anda memintanya sebuah lintasan. Kutu yang merusak hanya bagian jawaban yang tidak langsung Anda lihat adalah jenis kutu yang paling mahal yang ada.

Disusun dalam notasi buku ini, algoritmanya adalah **Pseudokode 10.1**. Nomor barisnya sama dengan nomor baris **Gambar 10.15**, sehingga pembacaan baris demi baris di atas berlaku tanpa perubahan.

Pseudokode 10.1 Penelusuran melebar dari sumber s, dalam notasi buku ini, dengan nomor baris yang sama seperti **Gambar 10.15**.

```
BFS(G, s)
for each vertex u in G.V - {s}
    color[u] = WHITE
    d[u] = INFINITY
    pi[u] = NIL
color[s] = GRAY
d[s] = 0
pi[s] = NIL                  // s, bukan u: keseleo yang lazim
Q = {s}
while Q is not empty
    u = head[Q]
    for each v in Adj[u]
        if color[v] == WHITE
            color[v] = GRAY
            d[v] = d[u] + 1
            pi[v] = u        // v, bukan u: keseleo yang lazim
            ENQUEUE(Q, v)
    DEQUEUE(Q)
    color[u] = BLACK
```

10.7.4 Contoh yang dikerjakan, selangkah demi selangkah

Graf berikut adalah graf yang tepat untuk belajar. Simpulnya ditata dalam dua baris berisi empat:

$$V = \{r, s, t, u, v, w, x, y\}, \qquad E = \{(r,s),\ (r,v),\ (s,w),\ (t,u),\ (t,w),\ (t,x),\ (u,y),\ (w,x),\ (x,y)\}.$$

Delapan simpul, sembilan sisi. Sebagai pemeriksaan atas cara kita membaca gambarnya, marilah kita tegaskan lema jabat tangannya: $\deg(r) = 2$, $\deg(s) = 2$, $\deg(t) = 3$, $\deg(u) = 2$, $\deg(v) = 1$, $\deg(w) = 3$, $\deg(x) = 3$, $\deg(y) = 2$; totalnya $2 + 2 + 3 + 2 + 1 + 3 + 3 + 2 = 18 = 2 \times 9$. Benar.

Sumbernya adalah s. Satu hal lagi harus ditetapkan sebelum sebuah penelusuran dapat diulang: **urutan penyimpanan tiap daftar ketetanggaan**, sebab urutan itulah yang menentukan simpul mana di antara dua simpul yang sama dekatnya yang ditemukan lebih dulu. Membaca isi antrean dari **Gambar 10.16** sampai **Gambar 10.19** memberi tahu kita urutannya, dan urutan itu adalah:

$$\mathrm{Adj}[s] = [w, r],\ \mathrm{Adj}[r] = [s, v],\ \mathrm{Adj}[w] = [s, t, x],\ \mathrm{Adj}[t] = [u, w, x],\ \mathrm{Adj}[x] = [t, w, y],\ \mathrm{Adj}[v] = [r],\ \mathrm{Adj}[u] = [t, y],\ \mathrm{Adj}[y] = [u, x].$$

> **PERINGATAN:** *Jarak* yang dihitung sebuah BFS tidak bergantung pada urutan daftar ketetanggaannya — jarak adalah sifat grafnya. *Pohonnya*, larik π-nya, dan urutan antreannya bergantung. Jadi jika penelusuran Anda berbeda dari penelusuran seorang teman tentang simpul mana yang ditemukan lebih dulu, mungkin kalian berdua benar; jika kalian berbeda tentang sebuah jarak, setidaknya salah satu dari kalian salah.

Sekarang penelusurannya. Tiap langkah bernomor adalah satu putaran gelung `while`: ambil kepala antreannya, susuri daftar ketetanggaannya, lalu hitamkan. Antreannya ditulis dari kiri ke kanan dengan kepalanya di kiri, dan tiap simpul ditampilkan beserta jaraknya.

Inisialisasi. Setiap simpul putih dengan $d = \infty$, kecuali s: kelabu, $d[s] = 0$, $\pi[s] =$ NIL. Masukkan s ke antrean.

$$Q = (s{:}\,0)$$

Langkah 1. Keluarkan s**.** $\text{Adj}[s] = [w, r]$. w putih: cat kelabu, $d[w] = d[s] + 1 = 1$, $\pi[w] = s$, masukkan ke antrean. $Q = (w: 1)$. r putih: cat kelabu, $d[r] = 1$, $\pi[r] = s$, masukkan ke antrean. $Q = (w: 1,\ r: 1)$. Hitamkan s.

$$Q = (w: 1,\ r: 1)$$

Langkah 2. Keluarkan w**.** $\text{Adj}[w] = [s, t, x]$. s hitam — lewati. (Inilah algoritmanya menyadari bahwa ia pernah ke sini.) t putih: kelabu, $d[t] = d[w] + 1 = 2$, $\pi[t] = w$, masukkan ke antrean. x putih: kelabu, $d[x] = 2$, $\pi[x] = w$, masukkan ke antrean. Hitamkan w.

$$Q = (r: 1,\ t: 2,\ x: 2)$$

Langkah 3. Keluarkan r**.** $\text{Adj}[r] = [s, v]$. s hitam — lewati. v putih: kelabu, $d[v] = d[r] + 1 = 2$, $\pi[v] = r$, masukkan ke antrean. Hitamkan r.

$$Q = (t: 2,\ x: 2,\ v: 2)$$

Berhentilah di sini dan pandangi antreannya. Ia memuat tiga simpul, semuanya berjarak 2. Cincin berjarak 1 — w dan r — kini seluruhnya hitam, dan seluruh cincin berjarak 2 kelabu dan menunggu. Cincinnya nyata.

Langkah 4. Keluarkan t**.** $\text{Adj}[t] = [u, w, x]$. u putih: kelabu, $d[u] = d[t] + 1 = 3$, $\pi[u] = t$, masukkan ke antrean. w hitam — lewati. x **kelabu** — lewati. Perhatikan bedanya: x ditemukan pada langkah 2 dari w, dan ia masih duduk di antrean. Ada sisi (t, x), jadi kita bisa saja mencapai x dari t dan menuliskan $d[x] = d[t] + 1 = 3$. Kita tidak melakukannya, sebab x sudah punya jarak yang lebih kecil. Sentuhan pertama menang. Hitamkan t.

$$Q = (x: 2,\ v: 2,\ u: 3)$$

Langkah 5. Keluarkan x**.** $\text{Adj}[x] = [t, w, y]$. t hitam — lewati. w hitam — lewati. y putih: kelabu, $d[y] = d[x] + 1 = 3$, $\pi[y] = x$, masukkan ke antrean. Hitamkan x.

$$Q = (v: 2,\ u: 3,\ y: 3)$$

Langkah 6. Keluarkan v**.** $\text{Adj}[v] = [r]$. r hitam — lewati. Tidak ada yang ditemukan; v adalah jalan buntu. Hitamkan v.

$$Q = (u: 3,\ y: 3)$$

Langkah 7. Keluarkan u**.** $\text{Adj}[u] = [t, y]$. t hitam — lewati. y kelabu — lewati. Hitamkan u.

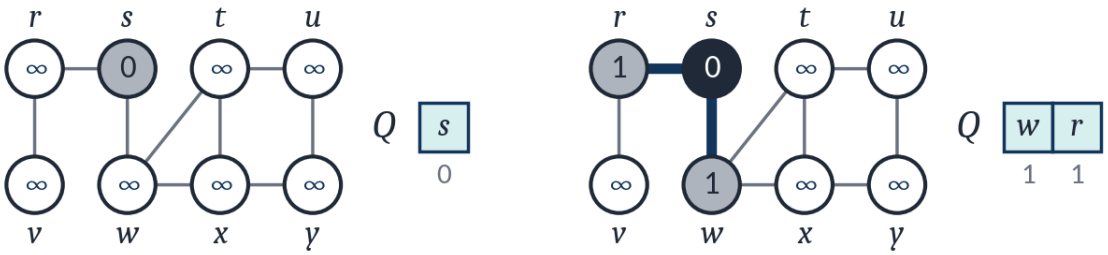
$$Q = (y: 3)$$

Langkah 8. Keluarkan y**.** $\text{Adj}[y] = [u, x]$. Keduanya hitam — lewati keduanya. Hitamkan y.

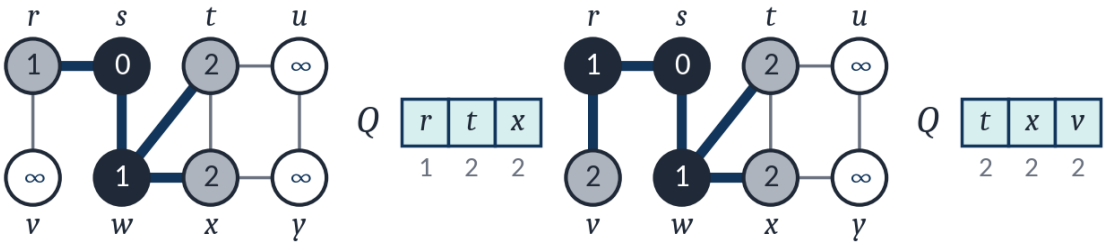
$$Q = (\ ) \quad \text{— kosong, jadi gelung while-nya berakhir.}$$

Gambar-gambar untuk penelusuran ini adalah **Gambar 10.16** untuk langkah 1 dan 2, **Gambar 10.17** untuk langkah 3 dan 4, **Gambar 10.18** untuk langkah 5 dan 6, serta **Gambar 10.19** untuk langkah 7 dan 8, yang berakhir pada **Gambar 10.20**. Dalam gambar-gambar itu, lingkaran putih adalah simpul putih, lingkaran kelabu adalah simpul kelabu, lingkaran

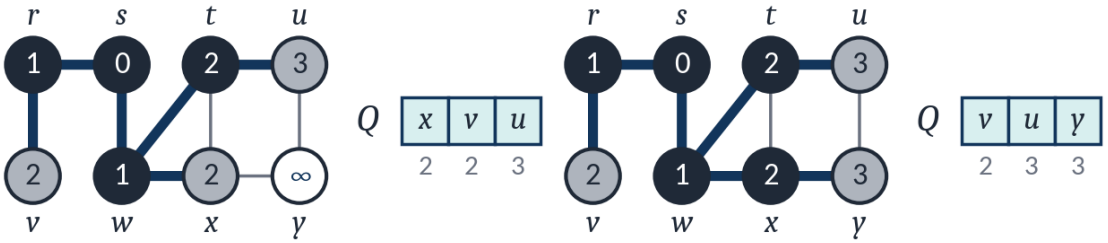
hitam penuh adalah simpul hitam, bilangan di dalam tiap lingkaran adalah d, dan sisi tebal adalah sisi pohon — sebuah sisi $(\pi[v], v)$.



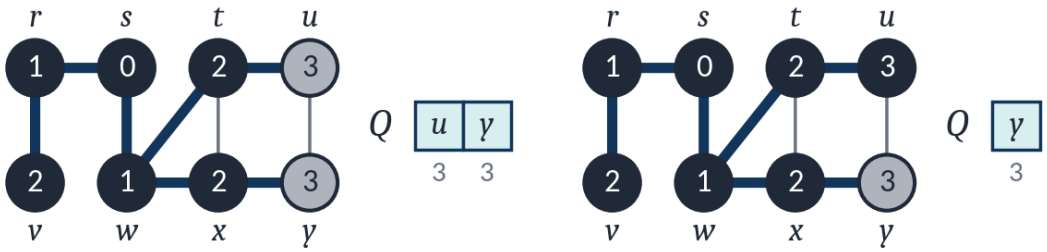
Gambar 10.16 Contoh BFS, dua keadaan yang pertama. **Kiri:** sesudah inisialisasi. Hanya s yang kelabu dan hanya s yang punya angka, 0; setiap simpul lain menampilkan ∞. Antreannya memuat s. **Kanan:** sesudah langkah 1. Kedua tetangga s kelabu dengan jarak 1, kedua sisi pohon s–w dan s–r digambar tebal, s hitam, dan antreannya memuat w lalu r — dalam urutan itu, yang menetapkan urutan Adj$[s]$ sebagai $[w, r]$.



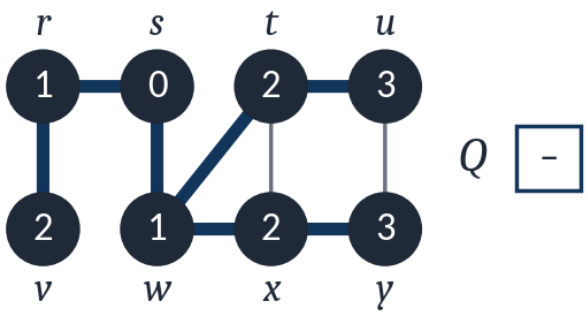
Gambar 10.17 Langkah 2 dan 3. **Kiri:** w telah dikeluarkan dari antrean dan dihitamkan, menemukan t dan x pada jarak 2; antreannya r, t, x. **Kanan:** r telah dikeluarkan dan dihitamkan, menemukan v pada jarak 2; antreannya t, x, v — tiga simpul yang semuanya berjarak 2, yaitu cincin kedua yang lengkap.



Gambar 10.18 Langkah 4 dan 5. **Kiri:** t dikeluarkan, menemukan u pada jarak 3; sisi (t, x) diperiksa lalu ditolak, sebab x sudah kelabu dengan $d[x] = 2$. **Kanan:** x dikeluarkan, menemukan y pada jarak 3; antreannya v, u, y.



Gambar 10.19 Langkah 6 dan 7. **Kiri:** v dikeluarkan; satu-satunya tetangganya, r, sudah hitam, jadi tidak ada yang ditemukan dan v sekadar berubah hitam. **Kanan:** u dikeluarkan; kedua tetangganya sudah terlihat. Antreannya tinggal y seorang.



Gambar 10.20 Hasilnya. Setiap simpul hitam, antreannya kosong, tiap simpul membawa jaraknya dari s, dan ketujuh sisi tebalnya adalah pohon melebarnya. Ada delapan simpul dan tujuh sisi pohon, persis sebagaimana dituntut hitungan **Gambar 10.10** atas sebuah pohon.

10.7.5 Membaca jawabannya

Tabel 10.7 adalah penelusuran yang sama dalam bentuk tabel, dan itulah bentuk yang harus Anda pakai setiap kali sebuah penelusuran diminta — Bab 12 mengerjakan soal persis sejenis ini: satu baris per langkah, dengan antreannya ditulis lengkap.

Tabel 10.7 BFS lengkap §10.7.4, satu baris per putaran gelung **while**. Setiap simpul dimasukkan ke antrean tepat sekali dan dikeluarkan tepat sekali, dan itulah sebabnya ada delapan langkah untuk delapan simpul. Perhatikan bahwa jarak di dalam antreannya tidak pernah menurun dari kiri ke kanan dan tidak pernah merentang lebih dari satu nilai — sifat yang dibuktikan di Lema 10.3.

Langkah	Dikeluarkan	Adj Yang diperiksa	Ditemukan, beserta d	Dilewati, dan sebabnya	Antrean sesudah langkah itu
—	—	—	$s: d = 0$	—	$(s: 0)$
1	s	w, r	$w: 1, r: 1$	—	$(w: 1,\ r: 1)$
2	w	s, t, x	$t: 2, x: 2$	s hitam	$(r: 1,\ t: 2,\ x: 2)$
3	r	s, v	$v: 2$	s hitam	$(t: 2,\ x: 2,\ v: 2)$
4	t	u, w, x	$u: 3$	w hitam, x kelabu	$(x: 2,\ v: 2,\ u: 3)$
5	x	t, w, y	$y: 3$	t, w hitam	$(v: 2,\ u: 3,\ y: 3)$
6	v	r	—	r hitam	$(u: 3,\ y: 3)$
7	u	t, y	—	t hitam, y kelabu	$(y: 3)$
8	y	u, x	—	keduanya hitam	$(\ )$

Dan **Tabel 10.8** adalah jawabannya: kedua larik yang memang hendak dihasilkan penelusuran itu.

Tabel 10.8 Keluaran BFS dari s: larik jarak d, larik pendahulu π, dan lintasan terpendek ke tiap simpul yang dibaca dari π dengan menyusuri pendahulunya kembali ke s lalu membaliknya. Panjang tiap lintasan sama dengan nilai d simpulnya, sebagaimana dituntut Teorema 10.4.

v	r	s	t	u	v	w	x	y
$d[v]$	1	0	2	3	2	1	2	3
$\pi[v]$	s	NIL	w	t	r	s	w	x
lintasan terpendek dari s	$s\,r$	s	$s\,w\,t$	$s\,w\,t\,u$	$s\,r\,v$	$s\,w$	$s\,w\,x$	$s\,w\,x\,y$

Bacalah salah satu lintasan itu dari π dengan tangan, untuk memastikan Anda bisa. Ambillah $v = y$. $\pi[y] = x$; $\pi[x] = w$; $\pi[w] = s$; $\pi[s] =$ NIL, jadi berhenti. Menuliskannya sambil berjalan memberikan y, x, w, s; membaliknya memberikan s, w, x, y. Panjangnya 3 sisi, dan $d[y] = 3$. Keduanya cocok, sebagaimana memang harus.

10.7.6 Programnya

Kode 10.2 adalah algoritma **Gambar 10.15** dalam C++, yang mencetak penelusurannya sendiri supaya Anda dapat mencocokkannya dengan **Tabel 10.7** baris demi baris.

Kode 10.2 Penelusuran melebar dalam C++, yang mencetak antreannya pada setiap langkah. Daftar ketetanggaannya dituliskan secara eksplisit alih-alih dibangun dengan `addEdge`, supaya *urutannya* persis sama dengan **Gambar 10.16** sampai **Gambar 10.19** dan penelusuran yang tercetak dapat dibandingkan dengan **Tabel 10.7** baris demi baris.

```
// Penelusuran melebar pada graf Bab 10, dengan antreannya dicetak
// pada setiap langkah, persis seperti yang ditunjukkan gambar Subbab 10.7.4.
//   V = {r,s,t,u,v,w,x,y}
//   E = {rs, rv, sw, tu, tw, tx, uy, wx, xy}
#include <iostream>
#include <vector>
#include <deque>
#include <string>
#include <algorithm>
#include <climits>
using namespace std;

enum Color { WHITE, GRAY, BLACK };
static const char* CNAME[] = {"white", "gray ", "black"};
static const int INF = INT_MAX;

struct Graph {
    int n;
    vector<vector<int>> adj;
    explicit Graph(int n_) : n(n_), adj(n_) {}
    void addEdge(int u, int v) { adj[u].push_back(v); adj[v].push_back(u); }
};

static const string NAME = "rstuvwxy";
static string qToString(const deque<int>& Q, const vector<int>& d) {
    if (Q.empty()) return "( empty )";
    string s = "(";
    for (size_t i = 0; i < Q.size(); ++i) {
        if (i) s += " ";
        s += NAME[Q[i]];
        s += ":";
        s += to_string(d[Q[i]]);
    }
    return s + ")";
}

// Mengembalikan d[] dan pi[]; mencetak antreannya sesudah tiap masuk dan keluar antrean.
void bfs(const Graph& G, int s, vector<int>& d, vector<int>& pi, bool trace) {
    vector<Color> color(G.n, WHITE);
    d.assign(G.n, INF);
    pi.assign(G.n, -1);
```

Kode 10.2 (Lanjutan) *Penelusuran melebar dalam C++, yang mencetak antreannya pada setiap langkah. Daftar ketetanggaannya dituliskan secara eksplisit alih-alih dibangun dengan addEdge, supaya urutannya persis sama dengan Gambar 10.16 sampai Gambar 10.19 dan penelusuran yang tercetak dapat dibandingkan dengan Tabel 10.7 baris demi baris.*

```
    color[s] = GRAY; d[s] = 0; pi[s] = -1;   // sumbernya: kelabu, jarak 0
    deque<int> Q; Q.push_back(s);
    int step = 0;
    if (trace) cout << "  init      enqueue " << NAME[s] << "   Q = " << qToString(Q,d) <<
"\n";

    while (!Q.empty()) {
        int u = Q.front();                      // u = head[Q]
        Q.pop_front();                          // Dequeue(Q)
        ++step;
        if (trace) cout << "  step " << step << "    dequeue " << NAME[u]
                        << "   Q = " << qToString(Q,d) << "\n";
        for (int v : G.adj[u]) {                // pindai Adj[u] sekali, dan hanya sekali
            if (color[v] == WHITE) {
                color[v] = GRAY;
                d[v]  = d[u] + 1;               // satu sisi lebih jauh daripada u
                pi[v] = u;
                Q.push_back(v);
                if (trace) cout << "            discover " << NAME[v]
                                << " via " << NAME[u] << ", d=" << d[v]
                                << "  Q = " << qToString(Q,d) << "\n";
            } else if (trace) {
                cout << "            skip " << NAME[v] << " (already "
                     << CNAME[color[v]] << ")\n";
            }
        }
        color[u] = BLACK;                       // u sudah selesai
        if (trace) cout << "            " << NAME[u] << " -> black\n";
    }
}

vector<int> pathTo(int v, const vector<int>& pi) {
    vector<int> p;
    for (int x = v; x != -1; x = pi[x]) p.push_back(x);
    reverse(p.begin(), p.end());
    return p;
}
```

Kode 10.2 (Lanjutan) *Penelusuran melebar dalam C++, yang mencetak antreannya pada setiap langkah. Daftar ketetanggaannya dituliskan secara eksplisit alih-alih dibangun dengan addEdge, supaya urutannya persis sama dengan Gambar 10.16 sampai Gambar 10.19 dan penelusuran yang tercetak dapat dibandingkan dengan Tabel 10.7 baris demi baris.*

```
int main() {
    Graph G(8);
    // r=0 s=1 t=2 u=3 v=4 w=5 x=6 y=7 ; urutan ketetanggaan yang dipakai gambarnya
    G.adj[1] = {5,0};        // s: w lalu r  -- urutan yang ditunjukkan antrean pada gambar
    G.adj[0] = {1,4};        // r: s, v
    G.adj[5] = {1,2,6};      // w: s, t, x
    G.adj[2] = {3,5,6};      // t: u, w, x
    G.adj[6] = {2,5,7};      // x: t, w, y
    G.adj[4] = {0};          // v: r
    G.adj[3] = {2,7};        // u: t, y
    G.adj[7] = {3,6};        // y: u, x

    vector<int> d, pi;
    cout << "BFS from s\n";
    bfs(G, 1, d, pi, true);

    cout << "\nresult\n  vertex :";
    for (int i = 0; i < 8; ++i) cout << "  " << NAME[i];
    cout << "\n  d      :";
    for (int i = 0; i < 8; ++i) cout << "  " << d[i];
    cout << "\n  pi     :";
    for (int i = 0; i < 8; ++i) cout << "  " << (pi[i] < 0 ? '-' : NAME[pi[i]]);
    cout << "\n";

    cout << "\nbreadth-first tree edges (pi[v], v)\n  ";
    for (int v = 0; v < 8; ++v)
        if (pi[v] >= 0) cout << "(" << NAME[pi[v]] << "," << NAME[v] << ") ";
    cout << "\n";

    cout << "\nshortest paths read out of pi\n";
    for (int v = 0; v < 8; ++v) {
        cout << "  s to " << NAME[v] << " (" << d[v] << (d[v]==1 ? " edge" : " edges") <<
"): ";
        for (int x : pathTo(v, pi)) cout << NAME[x];
        cout << "\n";
    }
    return 0;
}
```

Keluarannya, selengkapnya:

```
BFS from s
  init      enqueue s   Q = (s:0)
  step 1    dequeue s   Q = ( empty )
            discover w via s, d=1  Q = (w:1)
            discover r via s, d=1  Q = (w:1 r:1)
            s -> black
  step 2    dequeue w   Q = (r:1)
            skip s (already black)
            discover t via w, d=2  Q = (r:1 t:2)
            discover x via w, d=2  Q = (r:1 t:2 x:2)
            w -> black
  step 3    dequeue r   Q = (t:2 x:2)
            skip s (already black)
            discover v via r, d=2  Q = (t:2 x:2 v:2)
            r -> black
  step 4    dequeue t   Q = (x:2 v:2)
            discover u via t, d=3  Q = (x:2 v:2 u:3)
            skip w (already black)
            skip x (already gray )
            t -> black
  step 5    dequeue x   Q = (v:2 u:3)
            skip t (already black)
            skip w (already black)
            discover y via x, d=3  Q = (v:2 u:3 y:3)
            x -> black
  step 6    dequeue v   Q = (u:3 y:3)
            skip r (already black)
            v -> black
  step 7    dequeue u   Q = (y:3)
            skip t (already black)
            skip y (already gray )
            u -> black
  step 8    dequeue y   Q = ( empty )
            skip u (already black)
            skip x (already black)
            y -> black

result
  vertex :  r  s  t  u  v  w  x  y
  d      :  1  0  2  3  2  1  2  3
  pi     :  s  -  w  t  r  s  w  x

breadth-first tree edges (pi[v], v)
  (s,r) (w,t) (t,u) (r,v) (s,w) (w,x) (x,y)

shortest paths read out of pi
  s to r (1 edge): sr
  s to s (0 edges): s
  s to t (2 edges): swt
  s to u (3 edges): swtu
  s to v (2 edges): srv
  s to w (1 edge): sw
  s to x (2 edges): swx
  s to y (3 edges): swxy
```

Bandingkanlah keluaran itu dengan **Tabel 10.7** dan dengan **Gambar 10.16** sampai **Gambar 10.20**. Antrean sesudah tiap langkah sama persis pada ketiganya: (w, r), lalu (r, t, x), lalu (t, x, v), lalu (x, v, u), lalu (v, u, y), lalu (u, y), lalu (y), lalu kosong. Jaraknya sama persis. Ketujuh sisi pohonnya adalah tujuh sisi tebal pada **Gambar 10.20**. Programnya, tabelnya, dan gambarnya semuanya sepakat, dan kesepakatan itulah gunanya mengerjakan ketiganya.

10.7.7 Mengapa ongkosnya $O(V + E)$

Penurunannya terdiri atas lima baris. Marilah kita ambil satu per satu dan menaruh alasannya di bawah masing-masing.

1. Inisialisasinya memakan $O(V)$. Baris 01–04 menyentuh tiap simpul sekali dan mengerjakan tiga penugasan, jadi $\Theta(V)$.

2. Sebuah simpul dimasukkan ke antrean hanya jika ia putih, dan ia berhenti menjadi putih pada saat yang sama. Baris 12 menguji keputihannya, dan baris 13 langsung mengecatnya kelabu. Tidak ada apa pun yang pernah mengecat sebuah simpul menjadi putih lagi. Karena itu **tiap simpul dimasukkan ke antrean paling banyak sekali**, dan karena setiap pengeluaran membuang satu simpul, tiap simpul dikeluarkan dari antrean paling banyak sekali pula.

3. Masuk dan keluar antrean masing-masing berongkos $O(1)$, jadi biaya total semua operasi antreannya adalah $O(V)$. Ada paling banyak $|V|$ dari masing-masingnya, dengan biaya tetap per operasi. (Dengan `std::deque` atau larik biasa yang dipakai secara melingkar, ini sungguh-sungguh tetap; dengan `std::list` ia juga tetap. Memakai `std::vector::erase(begin())` sebagai pengeluaran dari antrean akan membuatnya $\Theta(V)$ per operasi dan seluruh penelusurannya $\Theta(V^2)$ — salah satu jebakan di §10.11.)

4. Daftar ketetanggaan sebuah simpul dipindai ketika simpul itu dikeluarkan dari antrean, dan hanya ketika itu. Gelung dalamnya, baris 11, duduk di dalam badan gelung `while`, yang berjalan sekali per pengeluaran. Karena tiap simpul dikeluarkan paling banyak sekali, $\mathrm{Adj}[u]$ dipindai paling banyak sekali untuk tiap u.

5. Jumlah panjang semua daftar ketetanggaannya adalah $\Theta(E)$, jadi waktu $O(E)$ dihabiskan untuk memindainya. Inilah Akibat 2 lema jabat tangan Persamaan (10.1), §10.4.3: $\sum_v \deg(v) = 2|E|$ pada graf tak berarah, dan $\sum_v \deg^+(v) = |E|$ pada graf berarah. Setiap entri yang diperiksa berongkos $O(1)$ — satu uji warna dan, paling banyak, empat penugasan serta satu dorongan.

Menjumlahkan potongan-potongannya:

$$T(n) = \underbrace{O(V)}_{\text{inisialisasi}} + \underbrace{O(V)}_{\text{operasi antrean}} + \underbrace{O(E)}_{\text{memindai daftarnya}} = O(V + E). \tag{10.4}$$

Ringkasan Persamaan (10.4) tepat: *waktu jalan totalnya $O(V + E)$ — linear terhadap ukuran representasi daftar ketetanggaan G*. Anak kalimat terakhir itulah yang perlu diingat. Masukannya **memang** sepanjang $\Theta(V + E)$ kata ketika disimpan sebagai daftar ketetanggaan, jadi algoritma yang berjalan dalam waktu $O(V + E)$ mengerjakan sejumlah kerja tetap per kata masukannya. Anda tidak dapat lebih baik daripada itu, sebab Anda tidak dapat membaca masukannya lebih cepat daripada itu. Penelusuran melebar bersifat *optimal*, bukan sekadar cepat.

> **PERINGATAN:** $O(V + E)$ adalah klaim tentang representasi **daftar ketetanggaan**. Jalankan algoritma yang persis sama atas **matriks** ketetanggaan dan baris 11 menjadi "untuk v dari 1 sampai $|V|$", yang berongkos $\Theta(V)$ per simpul yang dikeluarkan, ada atau tidak ada sisinya. Totalnya menjadi $\Theta(V^2)$. §10.9.8 mengukur persis hal ini, dan pada 8 000 simpul dengan 32 000 sisi, versi matriksnya memakan kira-kira 136 kali lebih lama.

Mengapa $O(V + E)$ dan bukan $O(E)$? Sebab kedua sukunya saling bebas. Sebuah graf boleh punya banyak simpul dan sangat sedikit sisi — sepuluh ribu simpul tanpa satu pun sisi adalah graf yang sah — dan inisialisasinya tetap harus menyentuh setiap simpul, jadi suku V-nya tidak dapat dibuang. Sebaliknya sebuah graf boleh punya sedikit simpul dan banyak sisi — K_{1000} punya seribu simpul dan setengah juta sisi — dan kemudian suku E-nya mendominasi. Tak satu suku

pun menelan suku yang lain, jadi keduanya dituliskan. Ketika grafnya terhubung kita tahu $|E| \geq |V| - 1$, dan kemudian $O(V + E) = O(E)$; banyak makalah mengandaikan keterhubungan dan menulis $O(E)$ karena alasan itu.

10.7.8 Pohon melebar

Larik π bukan sekadar setumpuk penunjuk. Ia adalah sebuah pohon, yang disebut **subgraf pendahulu**.

> **DEFINISI.** Sesudah menjalankan BFS dari s, **subgraf pendahulu** dari G adalah $G_\pi = (V_\pi, E_\pi)$ dengan

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}, \qquad E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}. \tag{10.5}$$

> G_π disebut **pohon melebar**, dan sisi-sisinya disebut **sisi pohon**.

Bacalah kedua pembentuk himpunan pada Persamaan (10.5) dengan suara keras. V_π adalah "simpul yang punya pendahulu, ditambah sumbernya" — dan karena sebuah simpul memperoleh pendahulu persis ketika ia ditemukan, V_π persis merupakan himpunan simpul yang dicapai penelusurannya. E_π adalah "untuk tiap simpul semacam itu selain sumbernya, satu sisi yang menemukannya". Jadi $|E_\pi| = |V_\pi| - 1$: setiap simpul di dalam pohonnya kecuali akarnya menyumbang tepat satu sisi.

Pohon melebar punya tiga sifat, dan kita kini dapat melihat masing-masingnya pada contoh yang kita kerjakan.

- **V_π terdiri atas simpul-simpul yang terjangkau dari s.** Pada contoh kita kedelapan simpulnya tercapai, jadi $V_\pi = V$.
- **$|E_\pi| = |V_\pi| - 1$, dan G_π terhubung, jadi G_π adalah pohon.** Contoh kita menghasilkan tujuh sisi pohon untuk delapan simpul — bacalah dari keluaran programnya: (s, r), (s, w), (w, t), (w, x), (r, v), (t, u), (x, y). Tujuh adalah delapan kurang satu. Ia terhubung sebab setiap simpul di dalamnya punya lintasan ke s menyusuri pendahulunya. Menurut §10.4.11, terhubung ditambah $|E| = |V| - 1$ menjadikannya pohon.
- **Untuk setiap $v \in V_\pi$ ada tepat satu lintasan sederhana dari s ke v di dalam G_π, dan lintasan itu juga merupakan lintasan terpendek dari s ke v di dalam G.** Ketunggalannya adalah sifat pohon. Bahwa ia *terpendek* adalah teorema bagian berikutnya.

Pohon melebar adalah **pohon merentang** dari komponen yang memuat s — sebuah subgraf merentang (§10.4.10) yang kebetulan berupa pohon. Inilah yang pertama dari beberapa algoritma dalam buku ini yang menghasilkan pohon merentang; Bab 11 menghasilkan yang *termurah*.

Bandingkanlah sisi pohonnya dengan kesembilan sisi grafnya untuk melihat apa yang dibuang BFS. Sisi G adalah (r, s), (r, v), (s, w), (t, u), (t, w), (t, x), (u, y), (w, x), (x, y). Tujuh menjadi sisi pohon. Dua yang tersisa adalah (t, x) — ditolak pada langkah 4, sebab x sudah kelabu — dan (u, y) — ditolak pada langkah 7, karena sebab yang sama. Secara umum, $|E| - (|V_\pi| - 1)$ sisi ditolak, di sini $9 - 7 = 2$. Setiap sisi yang ditolak menghubungkan dua simpul yang jaraknya berbeda paling banyak satu; itulah Akibat 10.5 di bawah, dan itulah fakta yang menjadi sandaran pengujian kebipartitan.

10.7.9 BFS menghitung jarak terpendek: pembuktiannya

Kita sampai pada alasan mengapa BFS penting. Dinyatakan sebagai daftar sifat, klaimnya adalah:

Diberikan graf $G = (V, E)$, BFS menemukan semua simpul yang terjangkau dari simpul sumber s. Ia menghitung jarak terpendek ke semua simpul yang terjangkau. Ia menghitung sebuah pohon melebar yang memuat semua simpul terjangkau itu. Untuk sembarang simpul v yang terjangkau dari s, lintasan di dalam pohon melebar dari s ke v bersesuaian dengan lintasan terpendek di dalam G.

Keempat kalimat itu adalah sebuah teorema, dan kita akan membuktikannya. Ingatlah kembali dari §10.4.5 bahwa $\delta(s, v)$ menyatakan jarak lintasan terpendek yang sesungguhnya — banyaknya sisi terkecil pada lintasan mana pun dari s ke v, atau ∞ jika tidak ada. Klaimnya adalah bahwa ketika BFS selesai, $d[v] = \delta(s, v)$ untuk setiap v. Pembuktiannya terdiri atas empat potongan: sebuah fakta tentang grafnya, satu paruh yang bebas dari batas atas, sebuah fakta tentang antreannya, lalu teoremanya.

Lema 10.1 (sebuah fakta tentang grafnya, tanpa algoritma apa pun). *Misalkan $G = (V, E)$ sebuah graf dan $s \in V$. Untuk setiap sisi $(u, v) \in E$,*

$$\delta(s, v) \leq \delta(s, u) + 1. \tag{10.6}$$

Bukti. Dua kasus. Jika u terjangkau dari s, ambillah sebuah lintasan terpendek dari s ke u — lintasan itu punya $\delta(s, u)$ sisi — lalu tempelkan sisi (u, v). Hasilnya adalah lintasan dari s ke v dengan $\delta(s, u) + 1$ sisi. Lintasan terpendek ke v tidak lebih panjang daripada lintasan tertentu itu, jadi $\delta(s, v) \leq \delta(s, u) + 1$. Jika u tidak terjangkau dari s maka $\delta(s, u) = \infty$ dan ketaksamaannya berlaku secara sepele, sebab tidak ada yang melampaui ∞. ■

Dengan kata lain: *mengambil satu langkah lagi tidak mungkin membuat Anda lebih dari satu langkah lebih jauh dari rumah.* Kentara, dan tak tergantikan.

Lema 10.2 (BFS tidak pernah menaksir terlalu rendah). *Pada akhir BFS dari s, berlaku $d[v] \geq \delta(s, v)$ untuk setiap $v \in V$.*

Bukti. Dengan induksi atas banyaknya simpul yang sudah dimasukkan ke antrean sejauh ini — artinya, kita menunjukkan bahwa pernyataan "$d[v] \geq \delta(s, v)$ untuk semua v" benar sesudah inisialisasinya dan dipertahankan oleh setiap pemasukan ke antrean.

Basis. Sesudah inisialisasi, $d[s] = 0 = \delta(s, s)$, dan $d[v] = \infty \geq \delta(s, v)$ untuk setiap v lainnya. Benar.

Langkah. Perhatikan saat sebuah simpul putih v ditemukan ketika $\mathrm{Adj}[u]$ sedang dipindai, sehingga baris 14 menetapkan $d[v] \leftarrow d[u] + 1$. Hipotesis induksi memberikan $d[u] \geq \delta(s, u)$. Karena itu

$$d[v] = d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v),$$

dengan ketaksamaan terakhirnya adalah Lema 10.1, Persamaan (10.6), yang diterapkan pada sisi (u, v). Dan tidak ada baris lain dalam algoritmanya yang pernah mengubah nilai d mana pun: begitu ditetapkan, $d[v]$ tidak pernah disentuh lagi, sebab v tidak lagi putih dan baris 12 menjaga satu-satunya penugasannya. Jadi sifatnya dipertahankan. ■

Jadi jawaban BFS tidak pernah terlalu kecil. Separuh teoremanya. Separuh yang lain memerlukan sebuah fakta tentang *antreannya*, dan inilah inti persoalannya.

Lema 10.3 (antreannya memuat paling banyak dua jarak berbeda, terurut). *Misalkan selama eksekusi BFS antreannya memuat simpul $\langle v_1, v_2, \ldots, v_r \rangle$, dengan v_1 di kepalanya. Maka*

$$d[v_1] \leq d[v_2] \leq \cdots \leq d[v_r] \leq d[v_1] + 1. \tag{10.7}$$

Bukti. Dengan induksi atas banyaknya operasi antrean yang telah dilakukan.

Basis. Segera sesudah inisialisasi, antreannya adalah $\langle s \rangle$: satu elemen, sehingga kedua rantai ketaksamaannya terbaca $d[s] \leq d[s] \leq d[s] + 1$. Benar.

Langkah, untuk sebuah pengeluaran. Misalkan sifatnya berlaku untuk $\langle v_1, \ldots, v_r \rangle$ dan kita membuang kepalanya v_1, sehingga tersisa $\langle v_2, \ldots, v_r \rangle$ (jika $r = 1$ antreannya menjadi kosong dan tidak ada yang perlu dibuktikan). Kita harus memeriksa kepala barunya, v_2. Kita memerlukan $d[v_r] \leq d[v_2] + 1$. Kita tahu $d[v_r] \leq d[v_1] + 1$ dan $d[v_1] \leq d[v_2]$, jadi $d[v_r] \leq d[v_2] + 1$. Ketaksamaan internalnya $d[v_2] \leq \cdots \leq d[v_r]$ terwaris tanpa berubah. Benar.

Langkah, untuk sebuah pemasukan. Misalkan kita sedang memindai $\mathrm{Adj}[u]$ untuk simpul u yang paling baru dikeluarkan, dan kita menemukan sebuah v yang putih, menetapkan $d[v] = d[u] + 1$ lalu menambahkan v ke antreannya, sehingga antreannya menjadi $\langle v_1, \ldots, v_r, v \rangle$. Dua hal harus diperiksa.

Pertama, $d[v_r] \leq d[v]$. Pada saat u dikeluarkan, ia adalah kepala antreannya, jadi menurut hipotesis induksi pada saat itu setiap simpul yang kala itu ada di antreannya punya nilai d sekurang-kurangnya $d[u]$ dan paling banyak $d[u] + 1$. Setiap simpul yang dimasukkan sejak itu diberi nilai $d[u] + 1$. Jadi setiap simpul yang kini ada di antreannya punya nilai d di dalam $\{d[u], d[u] + 1\}$; khususnya $d[v_r] \leq d[u] + 1 = d[v]$. Bagus.

Kedua, $d[v] \leq d[v_1] + 1$. Dari pengamatan yang sama, $d[v_1] \geq d[u]$, jadi $d[v_1] + 1 \geq d[u] + 1 = d[v]$. Bagus. ■

Ketaksamaan lema itu, Persamaan (10.7), mengatakan sesuatu yang sangat konkret, dan Anda dapat melihatnya pada setiap baris **Tabel 10.7**: **antreannya selalu terurut menurut jarak, dan ia selalu memuat paling banyak dua jarak berbeda, k dan $k + 1$.** Lihatlah langkah 3: antreannya $(t{:}\,2, x{:}\,2, v{:}\,2)$ — satu jarak. Langkah 4: $(x{:}\,2, v{:}\,2, u{:}\,3)$ — dua jarak berdampingan, terurut. Tidak pernah $(u{:}\,3, v{:}\,2)$, dan tidak pernah 2,4 tanpa 3. Itulah disiplin yang membuat cincinnya nyata.

Akibat. *Simpul dikeluarkan dari antrean dalam urutan d yang tidak menurun, dan tiap simpul dimasukkan paling banyak sekali, jadi jika v dimasukkan sebelum w maka $d[v] \leq d[w]$.*

Teorema 10.4 (kebenaran penelusuran melebar). *Misalkan BFS dijalankan pada graf $G = (V, E)$ dari sumber s. Maka BFS menemukan setiap simpul $v \in V$ yang terjangkau dari s, ia tidak menemukan satu pun simpul yang tak terjangkau, dan pada terminasinya $d[v] = \delta(s, v)$ untuk setiap $v \in V$. Lebih jauh, untuk setiap $v \neq s$ yang terjangkau, salah satu lintasan terpendek dari s ke v adalah lintasan dari s ke $\pi[v]$ yang disusul sisi $(\pi[v], v)$.*

Bukti. Andaikan, demi kontradiksi, ada simpul yang menerima nilai d yang berbeda dari jaraknya yang sesungguhnya. Menurut Lema 10.2 kita tahu $d[v] \geq \delta(s, v)$ selalu, jadi nilai yang salah pasti **terlalu besar**. Di antara semua simpul dengan $d[v] \neq \delta(s, v)$, misalkan v adalah salah satu yang $\delta(s, v)$-nya terkecil; jika beberapa seri, pilih mana saja. Maka

$$d[v] > \delta(s, v). \tag{10.8}$$

Perhatikan pertama bahwa $v \neq s$, sebab $d[s] = 0 = \delta(s, s)$. Perhatikan kedua bahwa v terjangkau dari s: seandainya tidak, $\delta(s, v) = \infty$, dan karena $d[v] \geq \delta(s, v)$ kita akan punya $d[v] = \infty$ pula sehingga tidak ada selisih. Jadi $\delta(s, v)$ adalah bilangan berhingga yang sekurang-kurangnya 1.

Karena v terjangkau dan $v \neq s$, ada sebuah lintasan terpendek dari s ke v; misalkan u adalah simpul tepat sebelum v pada lintasan itu. Maka $(u, v) \in E$ dan

$$\delta(s, v) = \delta(s, u) + 1.$$

Karena $\delta(s, u) = \delta(s, v) - 1 < \delta(s, v)$, dan v dipilih dengan jarak salah *terkecil*, u tidak mungkin salah. Jadi $d[u] = \delta(s, u)$. Mensubstitusikannya ke Persamaan (10.8):

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1. \tag{10.9}$$

Sekarang perhatikan saat algoritmanya mengeluarkan u dari antrean — dan itu pasti terjadi, sebab $d[u]$ berhingga, jadi u ditemukan, jadi u dimasukkan ke antrean, dan segala yang dimasukkan akhirnya dikeluarkan. Pada saat itu algoritmanya memindai $\text{Adj}[u]$ dan memeriksa sisi (u, v). Ada tiga kemungkinan warna v pada saat itu, dan kita akan menyingkirkan ketiganya.

Kasus 1: v putih. Maka baris 14 dieksekusi dan menetapkan $d[v] \leftarrow d[u] + 1$, bertentangan dengan Persamaan (10.9), yang menyatakan $d[v] > d[u] + 1$.

Kasus 2: v hitam. Simpul hitam sudah dikeluarkan dari antrean. Menurut akibat Lema 10.3, simpul dikeluarkan dalam urutan d yang tidak menurun, jadi v yang dikeluarkan sebelum u memberikan $d[v] \leq d[u]$, yang bertentangan dengan Persamaan (10.9) bahkan lebih keras lagi.

Kasus 3: v kelabu. Maka v ditemukan ketika daftar ketetanggaan suatu simpul w sedang dipindai, dan w dikeluarkan sebelum u (atau w adalah u itu sendiri, tetapi kemudian Kasus 1 berlaku pada saat yang lebih awal itu). Karena w dikeluarkan tidak lebih lambat daripada u, akibatnya memberikan $d[w] \leq d[u]$. Dan nilai v ditetapkan dari w: $d[v] = d[w] + 1 \leq d[u] + 1$. Sekali lagi ini bertentangan dengan Persamaan (10.9).

Ketiga kasus itu mustahil, jadi v semacam itu tidak ada, dan $d[v] = \delta(s, v)$ untuk setiap simpul.

Yang tersisa adalah mencatat kedua klaim tentang simpul *mana* yang ditemukan. Jika v terjangkau maka $\delta(s, v)$ berhingga, jadi $d[v]$ berhingga, jadi $d[v]$ ditetapkan oleh baris 14, dan itu hanya terjadi pada penemuan — sehingga v ditemukan. Jika v tak terjangkau maka $\delta(s, v) = \infty$, jadi $d[v] = \infty$, jadi baris 14 tidak pernah menyala untuk v dan ia tidak pernah ditemukan.

Terakhir, klaim lintasannya. Untuk $v \neq s$ yang terjangkau kita punya $d[v] = d[\pi[v]] + 1$ menurut konstruksinya (baris 14 menetapkannya demikian), dan baru saja kita tunjukkan bahwa d sama dengan δ, jadi $\delta(s, v) = \delta(s, \pi[v]) + 1$. Jadi lintasan terpendek ke $\pi[v]$ yang disusul sisi $(\pi[v], v)$ adalah lintasan ke v berpanjang $\delta(s, v)$ — sebuah lintasan terpendek. Dengan induksi atas $d[v]$, menyusuri π sampai kembali ke s menghasilkan lintasan terpendek dari s ke v; dan itu persis merupakan lintasan pohon tunggal di dalam G_π, sebab G_π adalah pohon yang sisinya adalah pasangan $(\pi[v], v)$. ■

> **WAWASAN KUNCI, dengan bahasa sehari-hari.** Mengapa sebuah antrean menemukan lintasan terpendek? Sebab antrean melayani perbatasannya dalam urutan ia ditemukan, dan urutan penemuan *itulah* urutan jarak. Karena itu algoritmanya menyelesaikan setiap simpul berjarak k sebelum menyentuh simpul mana pun berjarak $k + 1$, sehingga ketika ia pertama kali menyentuh sebuah simpul, ia sedang berdiri pada pendekatan terdekat yang mungkin. Tumpukan akan merusak hal ini — ia akan menghunjam ke jarak 17 sebelum menyelesaikan jarak 1 — dan itulah persis sebabnya penelusuran mendalam **tidak** menghitung lintasan terpendek.

Akibat 10.5 (setiap sisi menghubungkan aras yang berbeda paling banyak satu). *Untuk setiap sisi* (u, v) *pada graf tak berarah,* $|d[u] - d[v]| \leq 1$ *sesudah BFS dari sembarang sumber yang mencapai keduanya.*

Bukti. Menurut Teorema 10.4, $d = \delta$. Menurut Lema 10.1, Persamaan (10.6), yang diterapkan pada (u, v), $\delta(s, v) \leq \delta(s, u) + 1$; diterapkan pada (v, u), $\delta(s, u) \leq \delta(s, v) + 1$. Bersama-sama keduanya menyatakan bahwa kedua nilainya berbeda paling banyak satu. ■

Akibat 10.5 menggolongkan setiap sisi graf tak berarah sesudah BFS menjadi tepat dua jenis: **sisi pohon**, yang menghubungkan aras k ke aras $k + 1$ dan dipakai untuk menemukan simpul di aras $k + 1$; atau **sisi silang**, yang menghubungkan dua simpul pada aras yang sama, atau menghubungkan aras k dan $k + 1$ tanpa menjadi sisi yang menemukannya. Yang *mustahil* adalah sisi yang menghubungkan aras k ke aras $k + 2$ — seandainya sisi semacam itu ada, simpul yang jauh itu pasti sudah ditemukan satu cincin lebih awal. Inilah fakta yang dipakai §10.9.3: sisi searas persis merupakan siklus ganjil, dan karena itu persis merupakan penghalang kebipartitan.

10.8 Penelusuran Mendalam

10.8.1 Labirin, benang, dan sekaleng cat

Kiasan klasik untuk penelusuran mendalam [2] memperluas kiasan BFS dengan tepat satu benda, dan benda tambahan itulah seluruh bedanya:

Penelusuran mendalam (DFS) pada graf tak berarah G bagaikan mengembara di dalam labirin dengan seutas benang **dan sekaleng cat**. Kita mulai di simpul s, mengikatkan ujung benang kita ke titik itu dan mengecat s "dikunjungi (ditemukan)". Berikutnya kita menandai s sebagai simpul kita saat ini yang kita sebut u. Sekarang kita menyusuri sebuah sisi sembarang (u, v). Jika sisi (u, v) menuntun kita ke simpul v yang sudah dikunjungi, kita kembali ke u. Jika simpul v belum dikunjungi, kita ulurkan benang kita, berpindah ke v, mengecat v "dikunjungi", menetapkan v sebagai simpul kita saat ini, dan mengulangi langkah-langkah sebelumnya.

Akhirnya kita akan sampai pada titik ketika semua sisi yang bersisian dengan u menuntun ke simpul yang sudah dikunjungi. Kita lalu berbalik mundur dengan menggulung benang kita kembali ke simpul v yang telah dikunjungi sebelumnya. Kemudian v menjadi simpul kita saat ini dan kita mengulangi langkah-langkah sebelumnya. Lalu, jika semua sisi yang bersisian dengan v menuntun ke simpul yang sudah dikunjungi, kita berbalik mundur seperti sebelumnya. Kita terus berbalik mundur menyusuri lintasan yang telah kita tempuh, menemukan dan menjelajahi sisi-sisi yang belum terjelajah, dan mengulangi prosedurnya.

Pada penelusuran melebar Anda tidak pernah berpindah: Anda berdiri di s dan membiarkan benangnya terulur melingkar. Pada penelusuran mendalam **Anda berjalan**. Anda memilih sebuah lorong dan mengikutinya sejauh lorong itu menuju, lalu lorong berikutnya dari sana, dan seterusnya, sampai Anda menabrak sebuah ruangan yang semua pintu keluarnya menuntun ke tempat yang sudah Anda cat. Barulah kemudian Anda mundur — satu ruangan setiap kali, sambil mencoba tiap pintu keluar yang belum terjelajah saat melewatinya.

Benangnya masih di sana, dan kini ia mengerjakan tugas kedua. Pada BFS benangnya mengukur jarak. Pada DFS benangnya **adalah rute kembali**: ia merekam urutan persis ruangan yang telah Anda lewati, dan mundur berarti mengikutinya secara terbalik. Urutan itu, disimpan terurut dengan ruangan terbaru di depan, adalah sebuah **tumpukan** — dan pada bentuk rekursif

algoritmanya ia adalah tumpukan panggilan programnya sendiri, dan itulah sebabnya bentuk rekursifnya begitu pendek.

Catnya adalah warna §10.6, dan DFS memakai ketiga rona itu padahal BFS boleh dibilang hanya memerlukan dua. Tepatnya:

- *Sebuah simpul berwarna **putih** jika ia belum ditemukan.*
- *Sebuah simpul berwarna **kelabu** jika ia sudah ditemukan tetapi belum semua sisinya ditemukan.* Setara dengan itu: Anda sedang berdiri di ruangan itu, atau Anda berada di suatu tempat lebih dalam di lorong yang bermula di sana. Simpul kelabu persis merupakan ruangan-ruangan yang dilewati benang Anda.
- *Sebuah simpul berwarna **hitam** sesudah semua simpul yang bertetangga dengannya ditemukan (daftar ketetanggaannya diperiksa sepenuhnya).* Anda telah selesai dengan ruangan itu dan telah menarik benang Anda melewatinya.

> **WAWASAN KUNCI:** Simpul kelabu pada sebuah DFS membentuk **lintasan**, bukan cincin: persis lintasan dari akar pohon yang sedang dikerjakan ke simpul tempat Anda berdiri. Dalam satu frasa: *simpul kelabu membentuk rantai linear, yang bersesuaian dengan tumpukan simpul yang belum dijelajahi secara tuntas.* Pengamatan tunggal inilah asal-usul setiap teorema DFS dalam bab ini. "Apakah v kelabu?" berarti "apakah v leluhur dari tempat saya berdiri?", dan itulah sebabnya simpul kelabu di depan Anda berarti ada siklus.

10.8.2 Algoritmanya

Tidak seperti BFS, DFS secara alami ditulis sebagai dua prosedur, sebab ia mengerjakan dua hal: ia *memulai* penelusuran, dan ia *menjalankan* penelusuran. **Gambar 10.21** memberikan keduanya.

```
DFS (G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
        then DFS-VISIT (u)

DFS-VISIT (u)
color[u] ← GRAY                    ▷ Simpul putih u ditemukan.
d[u] ← time                        ▷ Catat waktu penemuan.
time ← time + 1                    ▷ Majukan jam global.
for each v ∈ Adj[u]                ▷ Jelajahi semua sisi (u, v).
    do if color[v] = WHITE
        then DFS-VISIT (v)
color[u] ← BLACK                   ▷ Hitamkan u; ia selesai.
f[u] ← time                        ▷ Catat waktu selesai.
time ← time + 1                    ▷ Majukan jam global.
```

Gambar 10.21 Penelusuran mendalam. Prosedur luar DFS(G) memutihkan setiap simpul, menjalankan jamnya, lalu memanggil DFS-Visit pada setiap simpul yang masih putih; tiap panggilan semacam itu mengakarkan satu pohon baru dalam hutan mendalamnya. Prosedur dalam DFS-Visit(u) adalah penelusurannya sendiri: cat u kelabu, cap waktu penemuannya, rekursilah ke setiap tetangga yang putih, lalu cat u hitam dan cap waktu selesainya.

Strukturnya adalah sebagai berikut:

- *Inisialisasi — warnai semua simpul putih.*
- *Kunjungi tiap dan setiap simpul putih memakai DFS-Visit.*
- *Tiap panggilan ke DFS-Visit(u) mengakarkan sebuah pohon baru dalam hutan mendalamnya di simpul u.*

Butir kedua itu adalah beda struktural yang besar dari BFS. **Penelusuran melebar mengambil sebuah sumber dan menjelajahi satu komponen. Penelusuran mendalam tidak mengambil sumber dan menjelajahi seluruh grafnya.** Gelung luar DFS menjamin bahwa jika grafnya tak terhubung, atau berarah sedemikian rupa sehingga sebagian simpulnya tak terjangkau dari akar pertamanya, simpul-simpul itu tetap ditelusuri — dari akar yang baru. Karena itu hasilnya bukan pohon melainkan **hutan**, dengan satu pohon per akar.

> **PERINGATAN — jamnya meleset satu antara prosedur dan contoh yang dikerjakan.** Pada **Gambar 10.21** jamnya dimulai pada $time \leftarrow 0$ dan tiap cap diambil *sebelum* penambahannya ($d[u] \leftarrow time$, lalu $time \leftarrow time + 1$), sehingga simpul pertama yang ditemukan akan menerima $d = 0$. Contoh yang dikerjakan, **Gambar 10.22** dan seterusnya, bermula dari 1. Kedua kelaziman itu sama-sama benar sempurna dan tak satu pun mengubah teorema mana pun — setiap pernyataan dalam bab ini menyangkut *urutan* pencap waktunya, tidak pernah nilai mutlaknya. Agar cocok dengan gambarnya, dan dengan CLRS [1], buku ini menambah lebih dulu: $time \leftarrow time + 1$, lalu $d[u] \leftarrow time$. Setiap kali Anda melaporkan

pencap waktu, katakanlah kelaziman mana yang Anda pakai; jawaban berbasis 1 dan jawaban berbasis 0 sama-sama benar.

Dalam notasi buku ini, dengan jam yang ditambah lebih dulu, kedua prosedurnya adalah **Pseudokode 10.2**:

Pseudokode 10.2 Penelusuran mendalam: prosedur luarnya memulai pohon baru pada setiap simpul yang masih putih, dan DFS-Visit menjalankan penelusurannya sambil mencap waktu penemuan dan waktu selesai.

```
DFS(G)
 1  for each vertex u in G.V
 2      color[u] = WHITE
 3      pi[u] = NIL
 4  time = 0
 5  for each vertex u in G.V
 6      if color[u] == WHITE
 7          DFS-VISIT(u)

DFS-VISIT(u)
 1  color[u] = GRAY                // simpul putih u baru saja ditemukan
 2  time = time + 1
 3  d[u] = time                    // cap waktu penemuannya
 4  for each v in Adj[u]           // jelajahi setiap sisi (u, v)
 5      if color[v] == WHITE
 6          pi[v] = u
 7          DFS-VISIT(v)
 8  color[u] = BLACK               // u sudah selesai
 9  time = time + 1
10  f[u] = time                    // cap waktu selesainya
```

Perhatikan betapa jauh lebih kecil ini daripada BFS. Tidak ada antrean, sebab rekursinya *adalah* tumpukannya; tidak ada larik jarak, sebab DFS tidak menghitung jarak. Yang ia hitung sebagai gantinya adalah dua pencap waktu per simpul, dan segala sesuatu dari §10.8.7 ke depan diperas dari kedua pencap itu.

10.8.3 Kedua pencap waktu

DEFINISI. DFS memelihara satu jam global, yaitu pencacah yang hanya pernah bertambah. Tiap simpul u menerima dua cap darinya: **waktu penemuan** $d[u]$, yaitu bacaan jamnya pada saat u berubah kelabu, dan **waktu selesai** $f[u]$, yaitu bacaannya pada saat u berubah hitam. Untuk setiap simpul, $d[u] < f[u]$.

Invariannya — *untuk setiap simpul u, ketaksamaan $d[u] < f[u]$ harus berlaku* — punya alasan yang langsung: jamnya maju di antara kedua cap itu, setidaknya sebanyak satu detak yang dihabiskan untuk menyelesaikan u, dan tidak pernah berjalan mundur.

Dua fakta numeris mengikutinya, dan keduanya dipakai terus-menerus.

Jamnya berakhir tepat pada $2|V|$. Setiap simpul ditemukan tepat sekali dan diselesaikan tepat sekali, dan tiap peristiwa memajukan jamnya sebanyak satu. Jadi ada $2|V|$ detak, dan pencap waktunya persis merupakan bilangan bulat 1 sampai $2|V|$, masing-masing dipakai sekali. Pada contoh enam simpul kita, capnya akan menjadi 1 sampai 12.

Sebuah simpul putih sebelum $d[u]$, kelabu di antara $d[u]$ dan $f[u]$, dan hitam sesudah $f[u]$. Pernyataan ulang ini menjadikan pencap waktunya sebagai *selang*: simpul u "menduduki" selang tertutup $[d[u], f[u]]$ pada jamnya, dan ia kelabu persis sepanjang selang itu. Struktur selang-selang itulah teorema tanda kurung §10.8.8.

10.8.4 Contoh yang dikerjakan, selangkah demi selangkah

Contohnya adalah graf **berarah** — DFS adalah tempat kearahan mulai penting — dengan enam simpul:

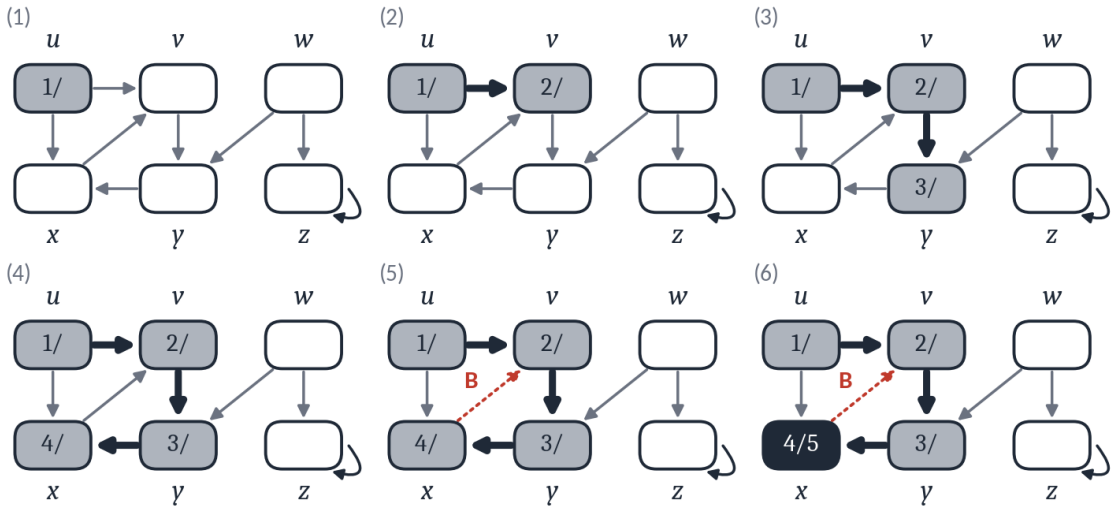
$$V = \{u, v, w, x, y, z\}, \qquad E = \{u \to v,\ u \to x,\ v \to y,\ w \to y,\ w \to z,\ x \to v,\ y \to x,\ z \to z\}.$$

Delapan busur, termasuk satu gelung-diri $z \to z$. Derajat masuk dan keluarnya adalah derajat pada **Tabel 10.1**. Gelung luarnya memperhatikan simpul-simpulnya dalam urutan u, v, w, x, y, z, dan tiap daftar ketetanggaannya berada dalam urutan yang ditulis di atas.

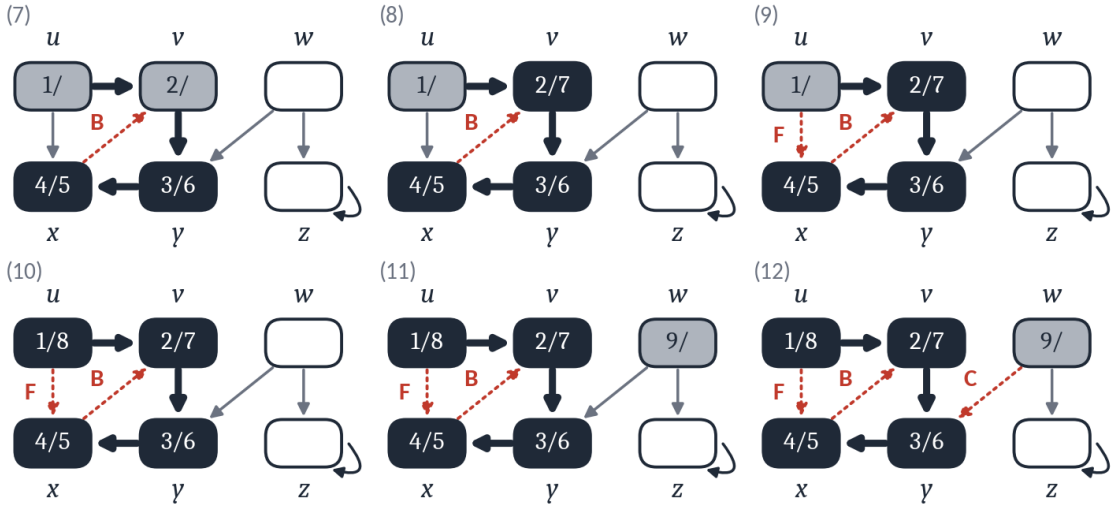
Tiap langkah bernomor di bawah ini adalah satu detak jam atau satu pemeriksaan sisi. Pada aras rincian inilah DFS menjadi kentara; apa pun yang lebih kasar dan ia tetap misterius.

Langkah 1. Gelung luarnya sampai di u, yang putih. Panggil DFS-Visit(u). Cat u kelabu; $d[u] = 1$. **Langkah 2.** Pindai Adj$[u] = [v, x]$. Mula-mula v: putih. Jadi (u, v) adalah **sisi pohon**; tetapkan $\pi[v] = u$ dan rekursilah. Cat v kelabu; $d[v] = 2$. **Langkah 3.** Pindai Adj$[v] = [y]$. y putih. (v, y) adalah **sisi pohon**; $\pi[y] = v$; rekursilah. Cat y kelabu; $d[y] = 3$. **Langkah 4.** Pindai Adj$[y] = [x]$. x putih. (y, x) adalah **sisi pohon**; $\pi[x] = y$; rekursilah. Cat x kelabu; $d[x] = 4$. **Langkah 5.** Pindai Adj$[x] = [v]$. v **kelabu**. Jadi v adalah leluhur x — memang $v \to y \to x$ adalah rantai simpul kelabu saat ini — dan (x, v) adalah **sisi balik**. Jangan rekursi. Inilah algoritmanya mendeteksi siklus berarah $v \to y \to x \to v$. **Langkah 6.** Adj$[x]$ habis. Cat x hitam; $f[x] = 5$. Kembali ke y. **Langkah 7.** Adj$[y]$ habis. Cat y hitam; $f[y] = 6$. Kembali ke v. **Langkah 8.** Adj$[v]$ habis. Cat v hitam; $f[v] = 7$. Kembali ke u. **Langkah 9.** Kembali di Adj$[u]$, tetangga berikutnya adalah x. Ia **hitam**, dan $d[u] = 1 < 4 = d[x]$, jadi x adalah *keturunan* u dan (u, x) adalah **sisi maju**. Jangan rekursi. **Langkah 10.** Adj$[u]$ habis. Cat u hitam; $f[u] = 8$. Kembali ke gelung luarnya. Pohon pertama hutannya lengkap: ia memuat u, v, y, x. **Langkah 11.** Gelung luarnya mencoba v — hitam, lewati. Lalu w — putih. Panggil DFS-Visit(w), mengakarkan **pohon kedua**. Cat w kelabu; $d[w] = 9$. **Langkah 12.** Pindai Adj$[w] = [y, z]$. Mula-mula y: **hitam**, dan $d[w] = 9 > 3 = d[y]$, jadi y bukan leluhur maupun keturunan w — ia berada di pohon yang sama sekali berbeda. (w, y) adalah **sisi silang**. **Langkah 13.** Berikutnya z: putih. (w, z) adalah **sisi pohon**; $\pi[z] = w$; rekursilah. Cat z kelabu; $d[z] = 10$. **Langkah 14.** Pindai Adj$[z] = [z]$. z **kelabu** — ia dirinya sendiri, dan ia sedang kelabu. Jadi (z, z) adalah **sisi balik**, dan gelung-dirinya dilaporkan dengan benar sebagai siklus berarah berpanjang 1. Tidak diperlukan kasus khusus apa pun. **Langkah 15.** Adj$[z]$ habis. Cat z hitam; $f[z] = 11$. **Langkah 16.** Adj$[w]$ habis. Cat w hitam; $f[w] = 12$. Gelung luarnya mendapati x, y, z semuanya hitam, lalu berhenti. Jamnya berakhir pada $12 = 2 \times 6$, sebagaimana diramalkan.

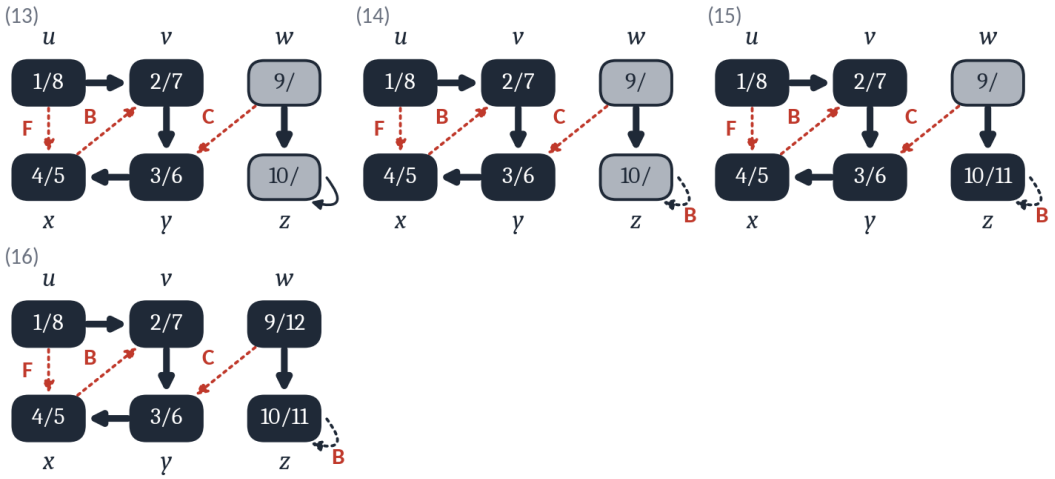
Gambar-gambar untuk keenam belas langkah ini adalah **Gambar 10.22**, **Gambar 10.23**, dan **Gambar 10.24** — enam panel, enam panel, dan empat panel, satu per langkah, dalam urutan baca.



Gambar 10.22 Langkah 1 sampai 6 contoh DFS, dalam urutan baca: u berubah kelabu dengan $d = 1$; v ditemukan lewat sisi pohon dengan $d = 2$; y dengan $d = 3$; x dengan $d = 4$; sisi (x, v) diperiksa, didapati menunjuk ke simpul kelabu, dan diberi label **B** untuk sisi balik, digambar sebagai panah putus-putus; dan akhirnya x berubah hitam dengan $f = 5$, ditampilkan sebagai $4/5$.



Gambar 10.23 Langkah 7 sampai 12. Rekursinya terurai, menghitamkan y pada $3/6$ dan v pada $2/7$; kembali di daftar ketetanggaan u, sisi (u, x) diperiksa dan diberi label **F** untuk sisi maju; u selesai pada $1/8$, melengkapkan pohon pertamanya; gelung luarnya lalu mengakarkan pohon kedua di w dengan $d = 9$; dan sisi (w, y) diberi label **C** untuk sisi silang, sebab y milik pohon pertama.



Gambar 10.24 Langkah 13 sampai 16. z ditemukan dari w lewat sisi pohon dengan $d = 10$; gelung-diri (z, z) diperiksa dan diberi label **B**, sebab pada saat itu z kelabu; z selesai pada $10/11$; dan w selesai pada $9/12$. Setiap simpul kini hitam dan jamnya telah mencapai $12 = 2|V|$.

Tabel 10.9 adalah jawabannya, dan **Tabel 10.10** adalah penggolongan sisinya. Keduanya pantas disalin dengan tangan sekali.

Tabel 10.9 Pencap waktu dan pendahulu sesudah DFS atas graf §10.8.4. Kedua akarnya adalah u dan w, dua simpul yang tidak punya pendahulu. Kedua belas capnya adalah bilangan bulat 1 sampai 12, masing-masing dipakai sekali.

v	u	v	w	x	y	z
$d[v]$	1	2	9	4	3	10
$f[v]$	8	7	12	5	6	11
$\pi[v]$	NIL	u	NIL	y	v	w

Tabel 10.10 Setiap busur contohnya, digolongkan, beserta uji warna yang membuat keputusannya. Kedelapan busurnya muncul: tiga sisi pohon di pohon pertama, satu di pohon kedua, dua sisi balik, satu maju, satu silang.

Busur	Golongan	Uji yang menentukannya
$u \to v$	pohon	v berwarna WHITE
$v \to y$	pohon	y berwarna WHITE
$y \to x$	pohon	x berwarna WHITE
$x \to v$	balik	v berwarna GRAY — seorang leluhur
$u \to x$	maju	x berwarna BLACK dan $d[u] < d[x]$ — seorang keturunan
$w \to y$	silang	y berwarna BLACK dan $d[w] > d[y]$ — bukan keduanya
$w \to z$	pohon	z berwarna WHITE
$z \to z$	balik	z berwarna GRAY — dirinya sendiri, dan kelabu

10.8.5 Hutan mendalam

Subgraf pendahulu untuk DFS didefinisikan sedikit berbeda dari subgraf pendahulu BFS:

DEFINISI. Sesudah sebuah DFS, subgraf pendahulunya adalah $G_\pi = (V, E_\pi)$ dengan

$$E_\pi = \{(\pi[v], v) \in E : v \in V \text{ dan } \pi[v] \neq \text{NIL}\}.$$

(10.10)

G_π adalah **hutan mendalam**; keping-keping terhubungnya adalah **pohon mendalam**, dan sisi-sisinya adalah **sisi pohon**.

Beda antara Persamaan (10.10) dan Persamaan (10.5) milik BFS adalah bahwa himpunan simpulnya di sini adalah *seluruh* V, bukan hanya bagian yang terjangkau. Karena gelung luar DFS mengunjungi setiap simpul, tidak ada simpul yang tertinggal; simpul yang tak terjangkau dari apa pun sekadar menjadi akar pohonnya sendiri yang berisi satu simpul. Jadi hutan mendalam selalu merupakan hutan **merentang** dari G (§10.4.10).

Pada contoh kita, $E_\pi = \{(u,v),(v,y),(y,x),(w,z)\}$ — empat sisi pohon, enam simpul, dua pohon. Hitungannya adalah $|V| - (\text{banyaknya pohon}) = 6 - 2 = 4$, dan itulah aturan umumnya: hutan dengan k pohon pada n simpul punya $n - k$ sisi.

Kedua pohonnya adalah

$$\underbrace{u \to v \to y \to x}_{\text{pohon berakar di } u}, \qquad \underbrace{w \to z}_{\text{pohon berakar di } w}.$$

10.8.6 Tanpa rekursi

Rekursi itu elegan tetapi ia memakai tumpukan panggilan programnya, yang pada kebanyakan sistem berukuran beberapa megabita — cukup untuk beberapa puluh ribu panggilan bersarang, tidak lebih. Penelusuran mendalam pada graf yang kebetulan memuat lintasan panjang akan meluapkannya. Graf lintasan sejuta simpul akan merontokkan DFS rekursif mana pun. Jadi bentuk iteratifnya pantas diketahui, dan menuliskannya bersifat mendidik, sebab ia menyingkapkan satu hal yang disembunyikan rekursinya.

Hal yang disembunyikan rekursinya adalah ini: ketika Anda kembali ke sebuah simpul sesudah panggilan rekursif, Anda harus melanjutkan daftar ketetanggaannya **dari tempat Anda berhenti**, bukan dari awal. Tumpukan berisi simpul saja tidak cukup; Anda memerlukan tumpukan berisi *posisi*. **Kode 10.3** menyimpan, untuk tiap simpul kelabu, baik simpulnya maupun sebuah indeks ke dalam daftar ketetanggaannya.

Kode 10.3 Penelusuran mendalam dengan tumpukan eksplisit, yang mencetak tumpukannya pada setiap langkah. Struktur `Frame` membawa posisi lanjutan `next` — tanpanya, kembali ke sebuah simpul akan memulai ulang daftar ketetanggaannya dan penelusurannya akan berputar-putar.

```
// Penelusuran mendalam tanpa rekursi: tumpukan eksplisit, tumpukannya dicetak
// pada setiap langkah. Graf dan pencap waktu yang sama dengan versi rekursifnya.
//   V = {u,v,w,x,y,z}, E = {u->v, u->x, v->y, w->y, w->z, x->v, y->x, z->z}
#include <iostream>
#include <vector>
#include <string>
using namespace std;

enum Color { WHITE, GRAY, BLACK };
static const string NAME = "uvwxyz";

struct Frame { int u; size_t next; };      // simpulnya, dan sejauh mana kita sampai di
Adj[u]

static string stackToString(const vector<Frame>& S) {
    if (S.empty()) return "[ ]";
    string s = "[";
    for (size_t i = 0; i < S.size(); ++i) {
        if (i) s += " ";
        s += NAME[S[i].u];
    }
    return s + "]";
}

int main() {
    const int n = 6;
    vector<vector<int>> adj(n);
    adj[0] = {1,3};    // u -> v, x
    adj[1] = {4};      // v -> y
    adj[2] = {4,5};    // w -> y, z
    adj[3] = {1};      // x -> v
    adj[4] = {3};      // y -> x
    adj[5] = {5};      // z -> z

    vector<Color> color(n, WHITE);
    vector<int> d(n,0), f(n,0), pi(n,-1);
    int time = 0, step = 0;

    cout << "DFS, iterative (explicit stack)\n";
    for (int root = 0; root < n; ++root) {
        if (color[root] != WHITE) continue;
        vector<Frame> S;
        color[root] = GRAY; d[root] = ++time;
        S.push_back({root, 0});
        cout << "  step " << ++step << "  push " << NAME[root]
             << " (root), d=" << d[root] << "   stack = " << stackToString(S) << "\n";
```

Kode 10.3 (Lanjutan) *Penelusuran mendalam dengan tumpukan eksplisit, yang mencetak tumpukannya pada setiap langkah. Struktur Frame membawa posisi lanjutan next — tanpanya, kembali ke sebuah simpul akan memulai ulang daftar ketetanggaannya dan penelusurannya akan berputar-putar.*

```
        while (!S.empty()) {
            Frame& top = S.back();
            if (top.next < adj[top.u].size()) {
                int v = adj[top.u][top.next++];        // ambil sisi berikutnya yang belum
terjelajah
                const char* kind = color[v] == WHITE ? "tree"
                                 : color[v] == GRAY  ? "back"
                                 : d[top.u] < d[v]    ? "forward" : "cross";
                cout << "  step " << ++step << "  edge (" << NAME[top.u] << ","
                     << NAME[v] << ") " << kind;
                if (color[v] == WHITE) {
                    color[v] = GRAY; d[v] = ++time; pi[v] = top.u;
                    S.push_back({v, 0});
                    cout << ": push " << NAME[v] << ", d=" << d[v];
                }
                cout << "   stack = " << stackToString(S) << "\n";
            } else {
                int u = top.u;
                S.pop_back();                          // Adj[u] habis: u selesai
                color[u] = BLACK; f[u] = ++time;
                cout << "  step " << ++step << "  pop  " << NAME[u]
                     << ", f=" << f[u] << "        stack = " << stackToString(S) << "\n";
            }
        }
    }

    cout << "\ntimestamps\n  vertex :";
    for (int i = 0; i < n; ++i) cout << "   " << NAME[i];
    cout << "\n  d/f    :";
    for (int i = 0; i < n; ++i) cout << " " << d[i] << "/" << f[i];
    cout << "\n\ntree edges\n  ";
    for (int v = 0; v < n; ++v)
        if (pi[v] >= 0) cout << "(" << NAME[pi[v]] << "," << NAME[v] << ") ";
    cout << "\n";
    return 0;
}
```

Keluarannya adalah penelusuran §10.8.4 dengan tumpukannya dibuat kasatmata:

```
DFS, iterative (explicit stack)
  step 1  push u (root), d=1   stack = [u]
  step 2  edge (u,v) tree: push v, d=2   stack = [u v]
  step 3  edge (v,y) tree: push y, d=3   stack = [u v y]
  step 4  edge (y,x) tree: push x, d=4   stack = [u v y x]
  step 5  edge (x,v) back   stack = [u v y x]
  step 6  pop  x, f=5        stack = [u v y]
  step 7  pop  y, f=6        stack = [u v]
  step 8  pop  v, f=7        stack = [u]
  step 9  edge (u,x) forward   stack = [u]
  step 10  pop  u, f=8        stack = [ ]
  step 11  push w (root), d=9   stack = [w]
  step 12  edge (w,y) cross   stack = [w]
  step 13  edge (w,z) tree: push z, d=10   stack = [w z]
  step 14  edge (z,z) back   stack = [w z]
  step 15  pop  z, f=11        stack = [w]
  step 16  pop  w, f=12        stack = [ ]

timestamps
  vertex :   u   v   w   x   y   z
  d/f    : 1/8 2/7 9/12 4/5 3/6 10/11

tree edges
  (u,v) (y,x) (v,y) (w,z)
```

Keenam belas langkah itu adalah keenam belas langkah §10.8.4 dan keenam belas panel **Gambar 10.22** sampai **Gambar 10.24**. Pencap waktunya sama persis dengan **Tabel 10.9**. Dan sekarang bandingkanlah lajur tumpukannya dengan lajur antrean **Tabel 10.7**: pada langkah 4 tumpukan DFS-nya adalah $[u\ v\ y\ x]$ — sedalam empat simpul, sebuah *rantai*, persis "rantai linear simpul kelabu" dari §10.8.1. Antrean BFS tidak pernah tampak seperti itu.

Dan inilah, demi kelengkapan, bentuk rekursifnya, yang lebih pendek dan itulah yang sebaiknya Anda tulis kecuali grafnya dalam.

Kode 10.4 Penelusuran mendalam, rekursif, pada contoh §10.8.4. Ia mencetak penelusurannya, pencap waktunya, penandakurungan §10.8.8, hutannya, dan penggolongan setiap busurnya.

```
// Penelusuran mendalam, bentuk rekursif, pada graf berarah contoh DFS
// Subbab 10.8.4:  V = {u,v,w,x,y,z}
//   E = {u->v, u->x, v->y, w->y, w->z, x->v, y->x, z->z}
// Mencetak waktu penemuan/selesai, penandakurungannya, hutan
// mendalamnya, dan penggolongan setiap sisinya.
#include <iostream>
#include <vector>
#include <string>
using namespace std;

enum Color { WHITE, GRAY, BLACK };
static const string NAME = "uvwxyz";

struct Digraph {
    int n;
    vector<vector<int>> adj;
    explicit Digraph(int n_) : n(n_), adj(n_) {}
    void addArc(int u, int v) { adj[u].push_back(v); }
};

struct DFS {
    const Digraph& G;
    vector<Color> color;
    vector<int> d, f, pi;
    int time = 0;
    string parens;                      // struktur tanda kurungnya
    vector<string> classified;          // satu baris per sisi
    vector<int> finishOrder;            // simpul dalam urutan selesainya

    explicit DFS(const Digraph& g)
        : G(g), color(g.n, WHITE), d(g.n, 0), f(g.n, 0), pi(g.n, -1) {}

    void run() {
        for (int u = 0; u < G.n; ++u)
            if (color[u] == WHITE) visit(u);
    }

    void visit(int u) {
        color[u] = GRAY;
        d[u] = ++time;                                  // waktu penemuan
        parens += "(" + string(1, NAME[u]) + " ";
        cout << "  d[" << NAME[u] << "] = " << d[u] << "   " << NAME[u]
             << " -> gray\n";
```

Kode 10.4 (Lanjutan) *Penelusuran mendalam, rekursif, pada contoh §10.8.4. Ia mencetak penelusurannya, pencap waktunya, penandakurungan §10.8.8, hutannya, dan penggolongan setiap busurnya.*

```
        for (int v : G.adj[u]) {
            string kind;
            if (color[v] == WHITE)      kind = "tree";
            else if (color[v] == GRAY)  kind = "back";     // v adalah leluhur
            else if (d[u] < d[v])        kind = "forward";  // v adalah keturunan
            else                         kind = "cross";
            classified.push_back("(" + string(1,NAME[u]) + "," + string(1,NAME[v])
                                 + ")  " + kind);
            cout << "    edge (" << NAME[u] << "," << NAME[v] << ") is a "
                 << kind << " edge\n";
            if (color[v] == WHITE) { pi[v] = u; visit(v); }
        }

        color[u] = BLACK;
        f[u] = ++time;                                  // waktu selesai
        finishOrder.push_back(u);
        parens += string(1, NAME[u]) + ") ";
        cout << "  f[" << NAME[u] << "] = " << f[u] << "   " << NAME[u]
             << " -> black\n";
    }
};

int main() {
    Digraph G(6);                        // u=0 v=1 w=2 x=3 y=4 z=5
    G.addArc(0,1); G.addArc(0,3);        // u->v, u->x
    G.addArc(1,4);                       // v->y
    G.addArc(2,4); G.addArc(2,5);        // w->y, w->z
    G.addArc(3,1);                       // x->v
    G.addArc(4,3);                       // y->x
    G.addArc(5,5);                       // z->z  (sebuah gelung-diri)

    DFS s(G);
    cout << "DFS, recursive\n";
    s.run();

    cout << "\ntimestamps\n  vertex :";
    for (int i = 0; i < 6; ++i) cout << "   " << NAME[i];
    cout << "\n  d/f    :";
    for (int i = 0; i < 6; ++i) cout << " " << s.d[i] << "/" << s.f[i];
    cout << "\n";

    cout << "\nparenthesization\n  " << s.parens << "\n";
```

Kode 10.4 (Lanjutan) *Penelusuran mendalam, rekursif, pada contoh §10.8.4. Ia mencetak penelusurannya, pencap waktunya, penandakurungan §10.8.8, hutannya, dan penggolongan setiap busurnya.*

```
    cout << "\ndepth-first forest (tree edges)\n  ";
    for (int v = 0; v < 6; ++v)
        if (s.pi[v] >= 0) cout << "(" << NAME[s.pi[v]] << "," << NAME[v] << ") ";
    cout << "\n  roots: ";
    for (int v = 0; v < 6; ++v) if (s.pi[v] < 0) cout << NAME[v] << " ";
    cout << "\n";

    cout << "\nedge classification\n";
    for (const string& line : s.classified) cout << "  " << line << "\n";

    cout << "\nvertices in order of finishing\n  ";
    for (int v : s.finishOrder) cout << NAME[v] << " ";
    cout << "\n";
    return 0;
}
```

Keluarannya:

```
DFS, recursive
  d[u] = 1   u -> gray
    edge (u,v) is a tree edge
  d[v] = 2   v -> gray
    edge (v,y) is a tree edge
  d[y] = 3   y -> gray
    edge (y,x) is a tree edge
  d[x] = 4   x -> gray
    edge (x,v) is a back edge
  f[x] = 5   x -> black
  f[y] = 6   y -> black
  f[v] = 7   v -> black
    edge (u,x) is a forward edge
  f[u] = 8   u -> black
  d[w] = 9   w -> gray
    edge (w,y) is a cross edge
    edge (w,z) is a tree edge
  d[z] = 10   z -> gray
    edge (z,z) is a back edge
  f[z] = 11   z -> black
  f[w] = 12   w -> black

timestamps
  vertex :   u   v   w   x   y   z
  d/f    : 1/8 2/7 9/12 4/5 3/6 10/11

parenthesization
  (u (v (y (x x) y) v) u) (w (z z) w)

depth-first forest (tree edges)
  (u,v) (y,x) (v,y) (w,z)
  roots: u w

edge classification
  (u,v)  tree
  (v,y)  tree
  (y,x)  tree
  (x,v)  back
  (u,x)  forward
  (w,y)  cross
  (w,z)  tree
  (z,z)  back

vertices in order of finishing
  x y v u z w
```

Yang rekursif dan yang iteratif sepakat pada setiap pencap waktu, pada setiap sisi pohon, dan pada golongan setiap busur — sebagaimana memang harus, sebab keduanya hanya berbeda pada tempat tumpukannya berada.

10.8.7 Mengapa ongkosnya $\Theta(V + E)$

Penurunannya, dalam empat langkah:

- *Gelung-gelung di dalam DFS masing-masing memakan waktu $\Theta(V)$, tidak termasuk waktu untuk mengeksekusi DFS-Visit.* Ada dua gelung semacam itu: gelung pemutihan dan gelung yang mencoba setiap simpul sebagai akar. Masing-masing berjalan $|V|$ kali dengan kerja tetap, dengan mengabaikan panggilan rekursifnya.
- *DFS-Visit dipanggil sekali untuk setiap simpul.* Mengapa tepat sekali? Sebab ia hanya dipanggil pada simpul putih, dan **tindakan pertamanya** adalah mengecat simpul itu kelabu. Tidak ada simpul yang pernah dicat putih kembali. Jadi jumlah total panggilannya paling banyak $|V|$; dan karena gelung luarnya mencoba setiap simpul, jumlahnya tepat $|V|$.

- *Untuk tiap DFS-Visit ada gelung yang mengiterasi seluruh* Adj$[v]$ *— biaya total DFS-Visit adalah* $\Theta(E)$*.* Daftar ketetanggaan tiap simpul disusuri tepat sekali, di dalam satu panggilannya. Jadi jumlah total entri daftar yang diperiksa adalah $\sum_v |\text{Adj}[v]|$, yaitu $|E|$ untuk graf berarah dan $2|E|$ untuk graf tak berarah — lema jabat tangan lagi.
- *Waktu jalan DFS adalah* $\Theta(V + E)$*.*

Jadi

$$T(n) = \underbrace{\Theta(V)}_{\text{dua gelung luarnya}} + \underbrace{\Theta(V)}_{|V|\text{ panggilan, overhead tetap tiap panggilan}} + \underbrace{\Theta(E)}_{\text{menyusuri semua}} \tag{10.11}$$
$$\Theta(V + E).$$

Perhatikan bahwa Persamaan (10.11) adalah Θ, bukan O, sedangkan Persamaan (10.4) hanya O: DFS selalu menyentuh setiap simpul dan setiap sisi, jadi batasnya ketat dari kedua arah. BFS dari sumber tunggal hanya $O(V + E)$, sebab jika grafnya tak terhubung BFS mungkin selesai lebih awal setelah hanya menyentuh sebagiannya.

Teknik dalam kedua penurunan itu pantas diberi nama, sebab ia berulang di sepanjang buku ini.

> **WAWASAN KUNCI — analisis agregat.** Kita tidak membatasi biaya satu iterasi lalu mengalikannya dengan banyaknya iterasi; itu akan memberikan $O(V) \times O(V) = O(V^2)$, sebuah batas yang benar tetapi tak berguna. Sebagai gantinya kita membatasi **totalnya** atas semua iterasi, dengan mengamati bahwa gelung-gelung dalamnya bersama-sama memeriksa tiap entri daftar ketetanggaan tepat sekali. Membebankan kerjanya pada *benda yang diproses* alih-alih pada *langkah yang memprosesnya* adalah seluruh triknya, dan Bab 14 menjadikannya metode formal dengan nama *amortized analysis* (analisis amortisasi).

10.8.8 Teorema tanda kurung

Sekarang strukturnya. Struktur ini paling baik diperkenalkan dengan sebuah gambaran yang lebih berharga daripada penjelasan apa pun: tulislah "(" disusul u ketika u ditemukan, dan u disusul ")" ketika u selesai, dalam urutan jam.

Waktu penemuan dan waktu selesai punya struktur tanda kurung. Nyatakan penemuan u dengan tanda kurung buka "(u"; nyatakan selesainya u dengan tanda kurung tutup "u)". Riwayat penemuan dan penyelesaiannya membentuk sebuah ungkapan yang terbentuk baik (tanda kurungnya bersarang secara benar).

Riwayat contoh kita, dibaca langsung dari keluaran **Kode 10.4**, adalah

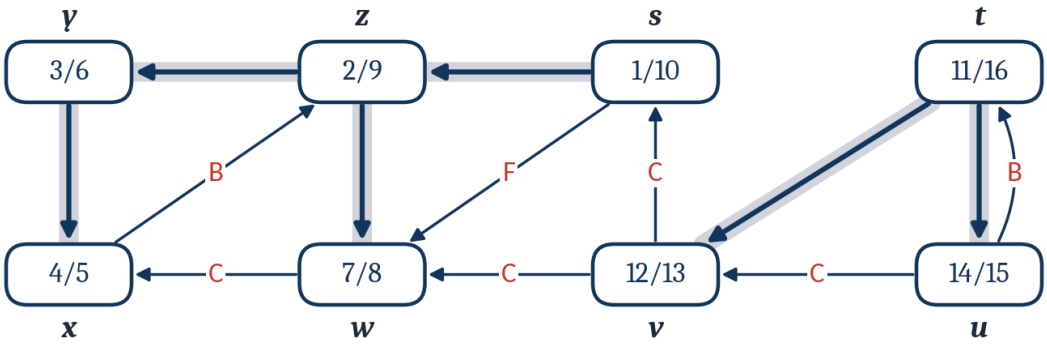
$$(u\ (v\ (y\ (x\ \ x)\ y)\ v)\ u) \quad (w\ (z\ \ z)\ w)$$

dan Anda dapat memeriksa dengan mata bahwa setiap tanda kurung menutup di tempat yang benar. Tidak pernah muncul sesuatu seperti $(u\ (v\ u)\ v)$.

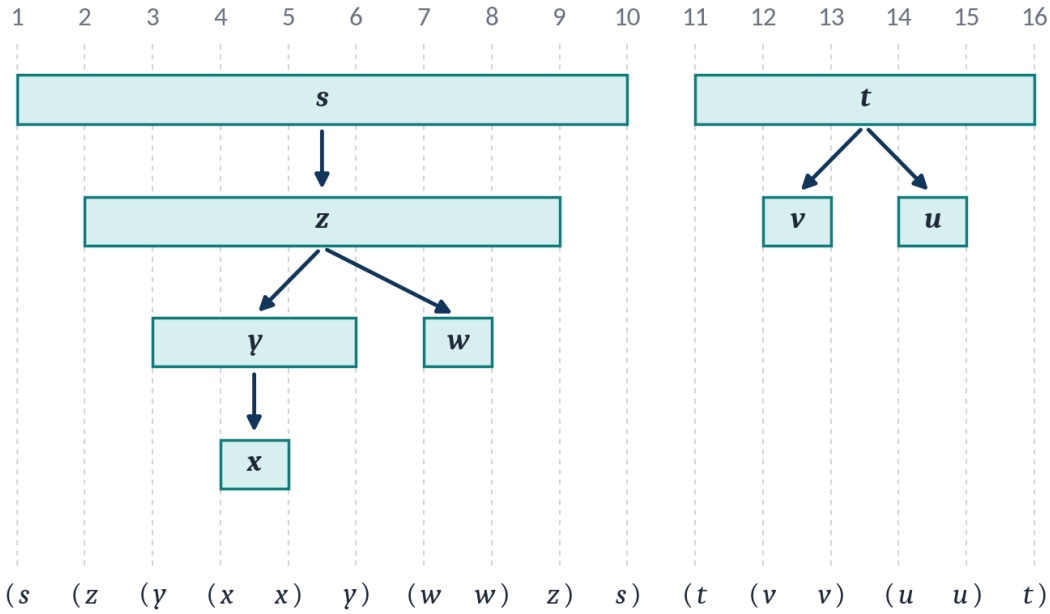
Untuk contoh kedua, kita memakai graf yang lebih besar, dan graf ini pantas dikerjakan sebab ia punya dua pohon dan keempat golongan sisinya. Busur-busurnya adalah

$$s \to z,\ s \to w,\ z \to y,\ z \to w,\ y \to x,\ x \to z,\ w \to x,\ v \to w,\ v \to s,\ t \to v,\ t \to u,\ u \to t,\ u \to v.$$

Gambar 10.25 adalah gambar graf itu yang berpencap waktu dan **Gambar 10.26** adalah struktur tanda kurung di bawahnya.



Gambar 10.25 Contoh DFS yang lebih besar, berpencap waktu. Tiap kotak simpul memuat d/f: s adalah $1/10$, z adalah $2/9$, y adalah $3/6$, x adalah $4/5$, w adalah $7/8$, t adalah $11/16$, v adalah $12/13$, u adalah $14/15$. Panah kelabu tebal adalah sisi pohon; panah tipisnya diberi label **B** (balik), **F** (maju), dan **C** (silang). Gambar tunggal ini membawa pencap waktunya, hutannya, dan penggolongan sisinya yang lengkap sekaligus.



Gambar 10.26 Struktur tanda kurung **Gambar 10.25**. Di atas, hutan mendalamnya digambar ulang dengan sisi pohon menunjuk ke bawah. Di bawah, jamnya dari 1 sampai 16 dengan satu batang mendatar per simpul, merentang $[d, f]$, dan penandakurungannya ditulis di sepanjang bagian bawah: $(s\ (z\ (y\ (x\ x)\ y)\ (w\ w)\ z)\ s)\ (t\ (v\ v)\ (u\ u)\ t)$. Setiap pasang batangnya entah bersarang atau saling lepas; tidak ada yang bertindihan sebagian.

Kode 10.5 menjalankan contoh kedua ini, untuk menegaskan bahwa cara kita membaca gambarnya sama dengan gambarnya.

Kode 10.5 Contoh DFS kedua dijalankan selengkapnya, untuk memeriksa pencap waktu dan penandakurungan **Gambar 10.25** dan **Gambar 10.26**.

```
// DFS pada contoh berarah kedua, yang struktur tanda kurungnya
// digambar di Subbab 10.8.8:
//   V = {s,t,u,v,w,x,y,z}
//   E = {s->z, s->w, z->y, z->w, y->x, x->z, w->x, v->w, v->s, t->v, t->u,
//        u->t, u->v}
// Simpulnya dikunjungi dalam urutan s, t, u, v, w, x, y, z pada gelung utamanya.
#include <iostream>
#include <vector>
#include <string>
using namespace std;

enum Color { WHITE, GRAY, BLACK };
static const string NAME = "stuvwxyz";     // s=0 t=1 u=2 v=3 w=4 x=5 y=6 z=7

struct DFS {
    vector<vector<int>> adj;
    vector<Color> color;
    vector<int> d, f, pi;
    int time = 0;
    string parens;
    vector<pair<string,string>> edges;

    explicit DFS(vector<vector<int>> a)
        : adj(move(a)), color(adj.size(), WHITE),
          d(adj.size(),0), f(adj.size(),0), pi(adj.size(),-1) {}

    void run() { for (size_t u = 0; u < adj.size(); ++u) if (color[u]==WHITE)
visit((int)u); }

    void visit(int u) {
        color[u] = GRAY; d[u] = ++time;
        parens += "(" + string(1,NAME[u]) + " ";
        for (int v : adj[u]) {
            string kind = color[v]==WHITE ? "tree"
                        : color[v]==GRAY  ? "back"
                        : d[u] < d[v]      ? "forward" : "cross";
            edges.push_back({string(1,NAME[u]) + "->" + string(1,NAME[v]), kind});
            if (color[v]==WHITE) { pi[v]=u; visit(v); }
        }
        color[u] = BLACK; f[u] = ++time;
        parens += string(1,NAME[u]) + ") ";
    }
};
```

Kode 10.5 (Lanjutan) *Contoh DFS kedua dijalankan selengkapnya, untuk memeriksa pencap waktu dan penandakurungan Gambar 10.25 dan Gambar 10.26.*

```
int main() {
    vector<vector<int>> adj(8);
    adj[0] = {7,4};      // s -> z, w
    adj[7] = {6,4};      // z -> y, w
    adj[6] = {5};        // y -> x
    adj[5] = {7};        // x -> z
    adj[4] = {5};        // w -> x
    adj[3] = {4,0};      // v -> w, s
    adj[1] = {3,2};      // t -> v, u
    adj[2] = {1,3};      // u -> t, v

    DFS s(adj);
    s.run();

    cout << "timestamps\n  vertex :";
    for (int i = 0; i < 8; ++i) cout << "     " << NAME[i];
    cout << "\n  d/f    :";
    for (int i = 0; i < 8; ++i) cout << " " << s.d[i] << "/" << s.f[i]
                                    << (s.f[i] < 10 ? "  " : "");
    cout << "\n\nparenthesization\n  " << s.parens << "\n";
    cout << "\nedge classification\n";
    for (auto& e : s.edges) cout << "  " << e.first << "  " << e.second << "\n";
    return 0;
}
```

Keluarannya:

```
timestamps
  vertex :     s     t     u     v     w     x     y     z
  d/f    : 1/10 11/16 14/15 12/13 7/8   4/5   3/6   2/9

parenthesization
  (s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

edge classification
  s->z  tree
  z->y  tree
  y->x  tree
  x->z  back
  z->w  tree
  w->x  cross
  s->w  forward
  t->v  tree
  v->w  cross
  v->s  cross
  t->u  tree
  u->t  back
  u->v  cross
```

Setiap pencap waktu cocok dengan **Gambar 10.25**, dan penandakurungannya sama karakter demi karakter dengan yang tertulis di sepanjang bagian bawah **Gambar 10.26**. Sekarang teoremanya.

> **TEOREMA TANDA KURUNG (Teorema 10.6).** *Pada penelusuran mendalam mana pun atas graf mana pun, untuk sembarang dua simpul u dan v, tepat satu di antara ketiga hal berikut berlaku:*

1. *selang $[d[u], f[u]]$ dan $[d[v], f[v]]$* ***saling lepas****, dan tidak ada di antara u dan v yang merupakan keturunan yang lain di dalam hutan mendalamnya;*
2. $[d[u], f[u]]$ ***termuat seluruhnya di dalam*** $[d[v], f[v]]$*, dan u adalah keturunan v;*
3. $[d[v], f[v]]$ ***termuat seluruhnya di dalam*** $[d[u], f[u]]$*, dan v adalah keturunan u.*

Khususnya, selang-selang itu tidak pernah ***bertindihan sebagian****.*

Bukti. Andaikan tanpa mengurangi keumumannya bahwa $d[u] < d[v]$ — artinya, u ditemukan lebih dulu. (Jika $d[v] < d[u]$, tukarkan namanya; keduanya tidak mungkin sama, sebab jamnya tidak pernah mengulang sebuah nilai.) Ada dua kasus.

Kasus A: $d[v] < f[u]$. Jadi v ditemukan ketika u masih kelabu. Sebuah simpul kelabu persis selama panggilan DFS-Visit-nya sendiri berada di tumpukan, jadi DFS-Visit(v) bermula selama DFS-Visit(u), yang berarti DFS-Visit(v) bersarang di dalam DFS-Visit(u). Dan panggilan yang bersarang harus kembali sebelum pemanggilnya kembali — ini sifat tumpukan panggilannya sendiri, bukan sifat graf. Jadi v selesai sebelum u selesai: $f[v] < f[u]$. Karena itu

$$d[u] < d[v] < f[v] < f[u], \tag{10.12}$$

dan itu persis kasus 3: selang v duduk tegas di dalam selang u, dan v ditemukan dari rantai simpul kelabu yang bermula di u, jadi v adalah keturunan u.

Kasus B: $f[u] < d[v]$. Maka u selesai sebelum v ditemukan, jadi

$$d[u] < f[u] < d[v] < f[v],$$

dan kedua selangnya saling lepas. Tidak ada yang dapat menjadi keturunan yang lain: agar v menjadi keturunan u, DFS-Visit(v) harus bersarang di dalam DFS-Visit(u), dan itu menuntut $d[v] < f[u]$ — salah di sini; dan agar u menjadi keturunan v kita memerlukan $d[v] < d[u]$, juga salah. Inilah kasus 1.

Hanya itulah kedua kasusnya, sebab $d[v] \neq f[u]$ (seluruh $2|V|$ capnya berbeda-beda) sehingga entah $d[v] < f[u]$ atau $f[u] < d[v]$. Tindihan *sebagian* akan menuntut sesuatu seperti $d[u] < d[v] < f[u] < f[v]$: tetapi kita baru saja menunjukkan bahwa $d[v] < f[u]$ memaksa $f[v] < f[u]$, jadi susunan itu tidak mungkin muncul. ■

Gerak batin di balik pembuktiannya pantas disimpan: *sembarang dua selang entah saling lepas atau terkurung; selang yang bertindihan akan berarti menyelesaikan seorang leluhur sebelum menyelesaikan seorang keturunan, atau memulai seorang keturunan tanpa memulai leluhurnya.* Keduanya mustahil pada sebuah tumpukan panggilan.

Akibat 10.7 (penyarangan selang para keturunan). *Simpul v adalah keturunan sejati simpul u di dalam hutan mendalam G jika dan hanya jika $d[u] < d[v] < f[v] < f[u]$, yaitu penyarangan* Persamaan (10.12)*.*

Inilah bentuk praktis teoremanya, dan ia mengubah pertanyaan tentang *bentuk pohon* menjadi pembandingan *empat bilangan bulat*. Ujilah pada **Tabel 10.9**: apakah x keturunan u? $d[u] = 1 < d[x] = 4 < f[x] = 5 < f[u] = 8$. Ya. Apakah y keturunan w? $d[w] = 9$ dan $d[y] = 3$, jadi $d[w] > d[y]$ dan rantainya gagal seketika. Tidak — dan memang keduanya berada di pohon yang berbeda.

10.8.9 Teorema lintasan putih

Satu lagi hasil struktural. Inilah alasan mengapa DFS menemukan segala yang seharusnya ia temukan, dan inilah mesin teorema DAG di §10.9.5.

TEOREMA LINTASAN PUTIH (Teorema 10.8). *Di dalam hutan mendalam G, simpul v adalah keturunan simpul u jika dan hanya jika, pada saat $d[u]$ ketika u ditemukan, ada lintasan dari u*

> *ke v yang **seluruhnya terdiri atas simpul putih** (selain u sendiri, yang berubah kelabu pada saat itu).*

Sketsa bukti, arah maju. Jika v adalah keturunan u, maka lintasan pohon dari u turun ke v ada. Setiap simpul di atasnya ditemukan sesudah u — menurut Akibat 10.7, $d[u] < d[w]$ untuk tiap w semacam itu — jadi pada waktu $d[u]$ setiap satu di antaranya masih putih. Lintasan pohon itu adalah sebuah lintasan putih.

Bukti, arah balik. Andaikan pada waktu $d[u]$ ada lintasan putih $u \rightsquigarrow v$ tetapi v tidak menjadi keturunan u. Di antara simpul pada lintasan itu yang gagal menjadi keturunan u, misalkan w adalah yang terdekat ke u, dan misalkan t adalah simpul tepat sebelum w pada lintasan itu, sehingga t *memang* menjadi keturunan u (atau $t = u$). Menurut Akibat 10.7, $f[t] \leq f[u]$.

Sekarang w putih pada waktu $d[u]$, jadi $d[u] < d[w]$. Dan w pasti ditemukan sebelum t selesai, sebab sisi (t, w) diperiksa selama DFS-Visit(t): entah w masih putih saat itu, dan dalam hal itu t menjadi induknya sehingga w *adalah* keturunan u — kontradiksi — atau w sudah ditemukan, dan dalam hal itu $d[w] < f[t]$. Bagaimanapun juga $d[u] < d[w] < f[t] \leq f[u]$, jadi $d[u] < d[w] < f[u]$, dan menurut teorema tanda kurung selang w bersarang di dalam selang u, sehingga w menjadi keturunan u. Kontradiksi. ■

Teoremanya terbaca lebih enak sebagai semboyan: **penelusuran mendalam dari u mencapai segala sesuatu yang terjangkau dari u melalui wilayah yang belum terjelajah.** Ia tidak dapat dihalangi oleh simpul yang belum ia lihat, hanya oleh simpul yang sudah ia lihat.

10.8.10 Menggolongkan sisinya

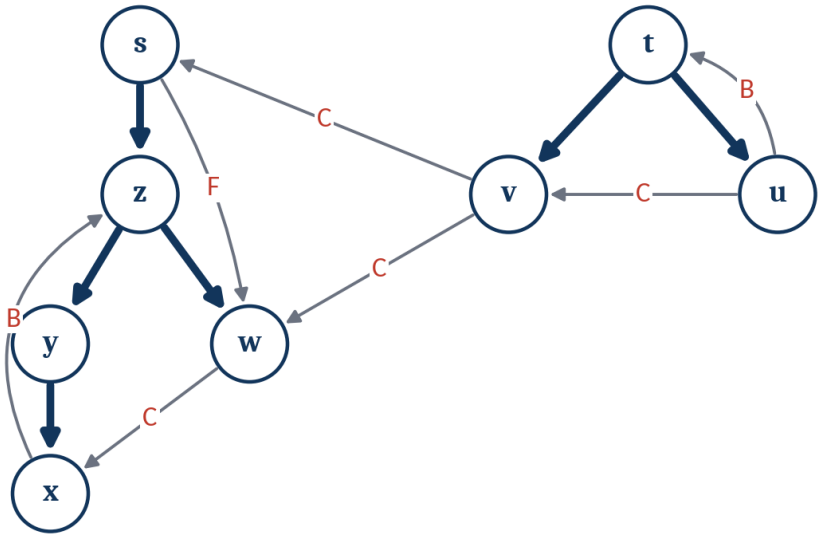
Kita sudah memakai keempat golongan sisi secara tak resmi; kini keempatnya memperoleh definisi, dengan uji warna terlampir pada masing-masingnya.

> **DEFINISI.** Tetapkan sebuah penelusuran mendalam atas G, yang menghasilkan hutan G_π. Setiap sisi (u, v) dari G jatuh ke tepat satu dari empat golongan, yang ditentukan oleh warna v pada saat sisi (u, v) diperiksa dari simpul kelabu u:

- **sisi pohon** — v berwarna WHITE — adalah sisi G_π; ia menemukan simpul baru;
- **sisi balik** — v berwarna GRAY — berjalan dari u ke seorang leluhur u di dalam G_π (gelung-diri terhitung sisi balik);
- **sisi maju** — v berwarna BLACK dan $d[u] < d[v]$ — adalah sisi bukan pohon dari u ke seorang keturunan sejati u;
- **sisi silang** — v berwarna BLACK dan $d[u] > d[v]$ — adalah segala sisanya: di antara dua subpohon pada pohon yang sama, atau di antara dua pohon yang berbeda.

Ringkasnya, untuk dua yang pertama: *sisi pohon (kelabu ke putih) — menjumpai simpul baru; sisi balik (kelabu ke kelabu) — dari keturunan ke leluhur*, dan untuk dua yang berikutnya *sisi maju (kelabu ke hitam) — dari leluhur ke keturunan; sisi silang (kelabu ke hitam) — sisanya, antarpohon atau antarsubpohon*. Perhatikan bahwa sisi maju dan sisi silang dibedakan oleh uji warna yang sama, jadi pencap waktunya harus dilihat untuk membedakan keduanya: sisi maju menunjuk *turun* ke wilayah yang sudah selesai di bawah Anda, sisi silang menunjuk *ke samping* ke wilayah yang sudah selesai di sebelah Anda.

Gambar 10.27 adalah gambar terbaik tentang hal ini, sebab ia menggambar ulang seluruh grafnya sehingga sisi pohonnya menggantung ke bawah; begitu gambarnya dibuat demikian, kata *balik*, *maju*, dan *silang* memerikan persis apa yang dilihat mata.



Gambar 10.27 Graf **Gambar 10.25** digambar ulang sebagai hutan mendalamnya, dengan sisi pohon tebal dan menunjuk ke bawah. Kini penggolongannya menjadi kasatmata: $x \to z$ dan $u \to t$ melengkung kembali **ke atas** menuju seorang leluhur (**B**); $s \to w$ menunjuk **ke bawah** melewati satu aras antara menuju seorang keturunan (**F**); $w \to x$, $v \to w$, $v \to s$, dan $u \to v$ berjalan **ke samping** di antara cabang atau di antara kedua pohonnya (**C**).

Dua catatan praktis.

Golongan mana yang penting? Terus terang saja: *sisi pohon dan sisi balik itu penting; kebanyakan algoritma tidak membedakan sisi maju dan sisi silang.* Sisi pohon memberi Anda hutannya, dan karena itu strukturnya. Sisi balik memberi Anda siklusnya, dan karena itu ketakbersiklusan, keterhubungan kuat, titik artikulasi, dan jembatan. Sisi maju dan sisi silang biasanya diabaikan.

Golongan mana yang bahkan mungkin muncul? Ini bergantung pada apakah grafnya berarah, dan jawabannya adalah teorema yang sungguh-sungguh berguna.

TEOREMA 10.9. *Pada penelusuran mendalam atas graf **tak berarah** G, setiap sisi adalah sisi pohon atau sisi balik. Sisi maju dan sisi silang tidak pernah muncul.*

Bukti. Misalkan (u, v) sebuah sisi, dan andaikan tanpa mengurangi keumumannya $d[u] < d[v]$; jadi u ditemukan lebih dulu. Sebuah sisi digolongkan pada saat pertama kali ia diperiksa, dari ujung mana pun hal itu terjadi, jadi kita bertanya ujung mana yang lebih dulu.

Langkah 1: v adalah keturunan u. Karena grafnya tak berarah, $v \in \text{Adj}[u]$, sehingga DFS-Visit(u) memeriksa sisi itu sebelum u selesai. Pada saat itu v pasti sudah ditemukan — jika ia masih putih, ia ditemukan tepat pada saat itu. Jadi $d[u] < d[v] < f[u]$, dan menurut teorema tanda kurung (Kasus A dalam pembuktiannya) v adalah keturunan u, dengan $f[v] < f[u]$.

Langkah 2: jika sisinya pertama kali diperiksa dari pihak u, ia adalah sisi pohon. Pada saat itu DFS-Visit(u) adalah panggilan yang sedang berjalan, sehingga u adalah simpul kelabu yang terdalam. Seandainya v kelabu, ia akan menjadi keturunan u yang kelabu dan lebih dalam daripada u — mustahil. Seandainya v hitam, DFS-Visit(v) sudah memindai seluruh Adj[v], yang memuat u, sehingga sisinya sudah diperiksa dari pihak v — bertentangan dengan "pertama kali". Jadi v putih, dan (u, v) adalah sisi pohon.

Langkah 3: jika sisinya pertama kali diperiksa dari pihak v, ia adalah sisi balik. Pemeriksaan itu terjadi selama DFS-Visit(v), dan menurut Langkah 1 panggilan itu bersarang di dalam DFS-Visit(u). Jadi u masih kelabu, dan sisi menuju simpul kelabu adalah sisi balik.

> Bagaimanapun juga sisinya adalah sisi pohon atau sisi balik; sisi maju maupun sisi silang menuntut ujung yang hitam pada pemeriksaan pertama, dan kita baru saja menunjukkan bahwa hal itu tidak mungkin terjadi. ■

Akibat praktisnya adalah uji siklus satu baris untuk graf tak berarah di §10.9.2: karena satu-satunya sisi bukan pohon adalah sisi balik, *sembarang* sisi bukan pohon berarti ada siklus.

Tabel 10.11 Keempat golongan sisi: graf mana yang dapat memuatnya, uji warna yang mengenalinya, pola pencap waktu yang dihasilkannya, dan untuk apa masing-masingnya berguna. Pada graf tak berarah hanya dua lajur pertamanya yang mungkin, dan itulah Teorema 10.9.

	Pohon	Balik	Maju	Silang
Graf tak berarah	ya	ya	tak pernah	tak pernah
Graf berarah	ya	ya	ya	ya
Warna v saat (u, v) diperiksa	putih	kelabu	hitam	hitam
Uji pencap waktu	—	$d[v] < d[u] < f[u] < f[v]$	$d[u] < d[v] < f[v] < f[u]$	$d[v] < f[v] < d[u] < f[u]$
Apa yang ia beri tahu Anda	hutannya	ada siklus	tidak banyak	tidak banyak

10.9 Enam Hal yang Dapat Dikerjakan Kedua Penelusuran

Kita kini punya dua penelusuran berwaktu linear dan setumpuk teorema tentangnya. Bagian ini membelanjakan modal itu. Enam persoalan, tak satu pun menyebut penelusuran dalam pernyataannya, semuanya diselesaikan dalam $O(V + E)$ oleh sebuah penelusuran dengan kira-kira lima baris tambahan.

10.9.1 Komponen terhubung

Persoalan. Diberikan sebuah graf tak berarah, berapa komponen terhubung yang ia punya, dan setiap simpul berada di komponen yang mana?

Gagasan. Menurut §10.4.8, penelusuran yang dimulai di dalam sebuah komponen tidak pernah dapat meninggalkannya, sebab tidak ada sisi yang menyeberang antarkomponen. Jadi: mulailah sebuah penelusuran di simpul 0; segala yang ia capai adalah komponen 0. Carilah simpul bernomor terkecil yang *tidak* ia capai; mulailah penelusuran di sana; segala yang *itu* capai adalah komponen 1. Ulangi sampai tidak ada simpul yang tersisa tanpa label.

Kedua penelusuran sama-sama bisa dipakai. BFS dipakai di bawah sebab label komponennya menggantikan larik `color` seluruhnya: simpul tanpa label adalah putih, dan simpul berlabel sudah pernah dilihat.

Kebenaran. Dua klaim. *Segala yang berlabel c sungguh-sungguh berada di satu komponen:* setiap simpul yang dicapai penelusuran dari s punya lintasan dari s, jadi semuanya saling terjangkau melalui s. *Segala yang berada di komponen s sungguh-sungguh memperoleh label c:* menurut Teorema 10.4, BFS menemukan setiap simpul yang terjangkau dari s, dan pada graf tak berarah "terjangkau dari s" persis berarti "berada di komponen s".

Biaya. Gelung luarnya berjalan $|V|$ kali, dan penelusurannya bersama-sama memasukkan tiap simpul ke antrean paling banyak sekali dan memindai tiap daftar ketetanggaan paling banyak sekali — sebab simpul yang sudah berlabel dilewati. Jadi totalnya $O(V + E)$, bukan $O(V \cdot (V +$

E)). Ini analisis agregat lagi: kerjanya dibebankan pada simpul dan sisinya, bukan pada iterasi gelung luarnya.

Kode 10.6 memberi label pada graf **Gambar 10.9**, yang punya tiga komponen.

Kode 10.6 Komponen terhubung lewat penelusuran melebar berulang, pada graf yang berbentuk seperti **Gambar 10.9**. Begitu `comp[]` terbangun, "apakah u dan v terhubung?" menjadi satu pembandingan bilangan bulat.

```
// Komponen terhubung sebuah graf tak berarah, lewat BFS berulang.
// Grafnya berbentuk seperti gambar tiga komponen di Subbab 10.4.8:
// sebuah segitiga, sebuah bintang, dan sebuah keping berisi lima simpul.
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

// comp[v] = indeks komponen yang memuat v; mengembalikan banyaknya komponen
int components(const vector<vector<int>>& adj, vector<int>& comp) {
    int n = (int)adj.size();
    comp.assign(n, -1);
    int c = 0;
    for (int s = 0; s < n; ++s) {
        if (comp[s] != -1) continue;         // sudah ditempatkan oleh penelusuran
sebelumnya
        queue<int> Q; Q.push(s); comp[s] = c;
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            for (int v : adj[u])
                if (comp[v] == -1) { comp[v] = c; Q.push(v); }
        }
        ++c;                                  // s menghabiskan satu komponen utuh
    }
    return c;
}

int main() {
    const int n = 12;
    vector<vector<int>> adj(n);
    auto add = [&](int u, int v) { adj[u].push_back(v); adj[v].push_back(u); };
    add(0,1); add(1,2); add(2,0);                                    // segitiga
    add(3,4); add(3,5); add(3,6);                                    // bintang
    add(7,8); add(7,9); add(7,10); add(8,9); add(9,10); add(9,11); add(10,11);

    vector<int> comp;
    int k = components(adj, comp);
    cout << "|V| = " << n << ", components = " << k << "\n\n";

    vector<vector<int>> members(k);
    for (int v = 0; v < n; ++v) members[comp[v]].push_back(v);
    for (int c = 0; c < k; ++c) {
        cout << "  component " << c << " (" << members[c].size() << " vertices): ";
        for (int v : members[c]) cout << v << " ";
        cout << "\n";
    }
```

Kode 10.6 (Lanjutan) *Komponen terhubung lewat penelusuran melebar berulang, pada graf yang berbentuk seperti Gambar 10.9. Begitu comp[] terbangun, "apakah u dan v terhubung?" menjadi satu pembandingan bilangan bulat.*

```
    cout << "\nsame-component queries, answered in O(1) from comp[]\n";
    const int q[4][2] = {{0,2},{0,4},{4,6},{8,11}};
    for (auto& p : q)
        cout << "  " << p[0] << " and " << p[1] << ": "
             << (comp[p[0]] == comp[p[1]] ? "connected" : "not connected") << "\n";
    return 0;
}
```

Keluarannya:

```
|V| = 12, components = 3

  component 0 (3 vertices): 0 1 2
  component 1 (4 vertices): 3 4 5 6
  component 2 (5 vertices): 7 8 9 10 11

same-component queries, answered in O(1) from comp[]
  0 and 2: connected
  0 and 4: not connected
  4 and 6: connected
  8 and 11: connected
```

Tiga komponen berukuran 3, 4, dan 5 — segitiga, bintang, dan keping berisi lima simpul pada **Gambar 10.9**. Perhatikan apa yang Anda beli dengan blok terakhirnya: sesudah satu lintasan $O(V + E)$, setiap kueri keterhubungan di kemudian hari berongkos satu pembandingan. Itulah pola baku sebuah algoritma prapemrosesan, dan itulah sebabnya label komponen layak dihitung sekalipun Anda hanya mengharapkan beberapa kueri.

Dan keterhubungannya gratis. Sebuah graf terhubung tepat ketika cacahannya 1. Jadi `components(adj, comp) == 1` adalah uji keterhubungannya, dan ongkosnya satu lintasan linear.

10.9.2 Deteksi siklus

Persoalan. Apakah graf ini memuat siklus?

Di sini kasus tak berarah dan kasus berarah sungguh-sungguh berbeda, dan bedanya bersifat mendidik.

Tak berarah. Menurut Teorema 10.9, DFS atas graf tak berarah hanya menghasilkan sisi pohon dan sisi balik. Sisi balik berarti ada siklus: sisi (u, v) ditambah lintasan pohon dari v turun ke u menutup sebuah siklus. Jadi grafnya punya siklus jika dan hanya jika DFS-nya menemukan sisi balik — artinya, jika dan hanya jika ia pernah memeriksa sisi yang menuju simpul bukan putih.

Tetapi ada jebakannya, dan inilah perangkap pemula yang klasik. Ketika penelusurannya berada di x, sesudah tiba dari y, sisi (x, y) berada di Adj$[x]$ dan y berwarna kelabu. Itu bukan siklus; itu sisi yang kita lalui saat masuk, dilihat dari ujung yang lain. Jadi ujinya harus mengabaikan sisi yang kembali ke induknya — **tepat sekali**. Mengabaikannya sekali dan tidak lebih itulah yang membuat programnya juga mendeteksi siklus *multigraf*: dua sisi sejajar di antara x dan y sungguh-sungguh merupakan siklus berpanjang 2, sehingga perjumpaan kedua dengan induknya memang harus dilaporkan. **Kode 10.7** melakukannya dengan satu baris: `if (v == p) { p = -2; continue; }` — sesudah melewati induknya sekali, `p` ditetapkan pada nomor simpul yang mustahil sehingga tidak ada tetangga berikutnya yang dapat cocok dengannya.

Berarah. Teorema 10.9 tidak berlaku, jadi keempat golongannya muncul, dan hanya sisi *balik* yang menandakan siklus. Sisi maju dan sisi silang tidak: $u \to x$ pada contoh kita adalah sisi maju

dan u, x tidak terletak pada siklus bersama. Karena itu ujinya bukan "bukan putih" melainkan tepat "**kelabu**". Dan kelabu berarti, menurut §10.8.1, *seorang leluhur dari tempat saya berdiri* — sehingga busur yang menuju simpul kelabu menutup sebuah siklus berarah melalui rantai simpul kelabunya, dan tidak ada yang lain yang melakukannya.

> **PERINGATAN:** Kutu tunggal yang paling sering muncul pada deteksi siklus berarah adalah menguji `visited[v]` alih-alih `gray[v]`. Larik `bool visited[]` tidak dapat membedakan kelabu dari hitam, jadi ia melaporkan sisi silang dan sisi maju sebagai siklus. Graf $u \to v$, $u \to w$, $v \to w$ sama sekali tidak punya siklus, tetapi uji berbasis `visited` menandai $v \to w$ dan melaporkan adanya siklus. Inilah persis keadaan yang menjadi alasan keberadaan skema tiga warna.

Biaya. Satu DFS: $O(V + E)$, dan ia dapat berhenti lebih awal begitu ia menemukan sebuah sisi balik.

Jalan pintas untuk kasus tak berarah. Jika grafnya terhubung, ia adalah pohon tepat ketika $|E| = |V| - 1$ (§10.4.11), jadi mencacah sisinya menjawab pertanyaannya tanpa penelusuran apa pun. Lebih umum lagi, graf tak berarah dengan k komponen adalah hutan tepat ketika $|E| = |V| - k$, dan jika $|E| > |V| - k$ ia punya siklus. Ini berongkos satu lintasan untuk mencacah komponennya dan pantas diketahui; tetapi ia hanya memberi tahu Anda *apakah*, bukan *di mana*, dan ia tidak mengatakan apa-apa pada kasus berarah.

Kode 10.7 Deteksi siklus untuk graf tak berarah maupun berarah, diuji pada tujuh graf termasuk contoh-contoh bab ini sendiri. Uji tak berarahnya menerima sembarang tetangga bukan putih selain induknya; uji berarahnya bersikeras pada warna **kelabu**.

```
// Deteksi siklus, kedua ragamnya.
//   tak berarah: sisi balik yang bukan sisi yang kita lalui saat tiba
//   berarah:    sisi yang menuju simpul GRAY
#include <iostream>
#include <vector>
#include <string>
using namespace std;

enum Color { WHITE, GRAY, BLACK };

// ---------- tak berarah ----------
struct UndirectedCycle {
    vector<vector<int>> adj;
    vector<Color> color;
    vector<int> parent;
    int a = -1, b = -1;                      // kedua ujung sisi yang bermasalah

    explicit UndirectedCycle(vector<vector<int>> g)
        : adj(move(g)), color(adj.size(), WHITE), parent(adj.size(), -1) {}

    bool hasCycle() {
        for (size_t s = 0; s < adj.size(); ++s)
            if (color[s] == WHITE && visit((int)s, -1)) return true;
        return false;
    }
    bool visit(int u, int p) {
        color[u] = GRAY; parent[u] = p;
        for (int v : adj[u]) {
            if (v == p) { p = -2; continue; }   // lewati sisi yang kita lalui saat masuk,
sekali saja
            if (color[v] != WHITE) { a = u; b = v; return true; }
            if (visit(v, u)) return true;
        }
        color[u] = BLACK;
        return false;
    }
};

// ---------- berarah ----------
struct DirectedCycle {
    vector<vector<int>> adj;
    vector<Color> color;
    int a = -1, b = -1;

    explicit DirectedCycle(vector<vector<int>> g)
        : adj(move(g)), color(adj.size(), WHITE) {}
```

Kode 10.7 (Lanjutan) *Deteksi siklus untuk graf tak berarah maupun berarah, diuji pada tujuh graf termasuk contoh-contoh bab ini sendiri. Uji tak berarahnya menerima sembarang tetangga bukan putih selain induknya; uji berarahnya bersikeras pada warna kelabu.*

```
    bool hasCycle() {
        for (size_t s = 0; s < adj.size(); ++s)
            if (color[s] == WHITE && visit((int)s)) return true;
        return false;
    }
    bool visit(int u) {
        color[u] = GRAY;
        for (int v : adj[u]) {
            if (color[v] == GRAY) { a = u; b = v; return true; }  // sisi balik
            if (color[v] == WHITE && visit(v)) return true;
        }
        color[u] = BLACK;
        return false;
    }
};

static vector<vector<int>> undirectedFrom(int n, const vector<pair<int,int>>& E) {
    vector<vector<int>> adj(n);
    for (auto e : E) { adj[e.first].push_back(e.second); adj[e.second].push_back(e.first);
}
    return adj;
}
static vector<vector<int>> directedFrom(int n, const vector<pair<int,int>>& E) {
    vector<vector<int>> adj(n);
    for (auto e : E) adj[e.first].push_back(e.second);
    return adj;
}

int main() {
    cout << "undirected graphs\n";
    {   // graf bab ini: V={a..e}, E={ab,ac,ad,be,cd,ce,de}
        UndirectedCycle t(undirectedFrom(5, {{0,1},{0,2},{0,3},{1,4},{2,3},{2,4},{3,4}}));
        cout << "  a-e graph of the chapter : " << (t.hasCycle() ? "has a cycle" :
"acyclic");
        if (t.a >= 0) cout << "  (back edge " << char('a'+t.a) << "-" << char('a'+t.b) <<
")";
        cout << "\n";
    }
    {   // pohon dengan 5 simpul
        UndirectedCycle t(undirectedFrom(5, {{0,1},{0,2},{1,3},{1,4}}));
        cout << "  tree on 5 vertices      : " << (t.hasCycle() ? "has a cycle" :
"acyclic") << "\n";
    }
    {   // hutan: dua pohon, |E| = |V| - 2
        UndirectedCycle t(undirectedFrom(6, {{0,1},{1,2},{3,4},{4,5}}));
        cout << "  forest of two paths     : " << (t.hasCycle() ? "has a cycle" :
"acyclic") << "\n";
    }
    {   // satu segitiga ditambah satu simpul terasing
        UndirectedCycle t(undirectedFrom(4, {{0,1},{1,2},{2,0}}));
        cout << "  triangle + isolate      : " << (t.hasCycle() ? "has a cycle" :
"acyclic") << "\n";
```

Kode 10.7 (Lanjutan) *Deteksi siklus untuk graf tak berarah maupun berarah, diuji pada tujuh graf termasuk contoh-contoh bab ini sendiri. Uji tak berarahnya menerima sembarang tetangga bukan putih selain induknya; uji berarahnya bersikeras pada warna kelabu.*

```
    }

    cout << "\ndirected graphs\n";
    {   // contoh DFS bab ini: u->v,u->x,v->y,w->y,w->z,x->v,y->x,z->z
        DirectedCycle t(directedFrom(6,
{{0,1},{0,3},{1,4},{2,4},{2,5},{3,1},{4,3},{5,5}}));
        cout << "  DFS example of the chapter: " << (t.hasCycle() ? "has a directed cycle"
: "is a DAG");
        if (t.a >= 0) cout << "  (back edge " << "uvwxyz"[t.a] << "->" << "uvwxyz"[t.b] <<
")";
        cout << "\n";
    }
    {   // DAG berpakaian, dengan penomoran simpul yang dipakai nanti di bab ini
        DirectedCycle t(directedFrom(9,
{{0,1},{0,3},{1,2},{3,2},{5,6},{5,7},{6,7},{6,3},{8,7}}));
        cout << "  dressing DAG              : " << (t.hasCycle() ? "has a directed cycle"
: "is a DAG") << "\n";
    }
    {   // DAG yang sama dengan satu busur tambahan jacket -> shirt, yang menutup sebuah
siklus
        DirectedCycle t(directedFrom(9,
{{0,1},{0,3},{1,2},{3,2},{5,6},{5,7},{6,7},{6,3},{8,7},{2,0}}));
        cout << "  dressing DAG + jacket->shirt: " << (t.hasCycle() ? "has a directed
cycle" : "is a DAG") << "\n";
    }
    return 0;
}
```

Keluarannya:

```
undirected graphs
  a-e graph of the chapter : has a cycle  (back edge c-a)
  tree on 5 vertices       : acyclic
  forest of two paths      : acyclic
  triangle + isolate       : has a cycle

directed graphs
  DFS example of the chapter: has a directed cycle  (back edge x->v)
  dressing DAG              : is a DAG
  dressing DAG + jacket->shirt: has a directed cycle
```

Setiap barisnya dapat diperiksa dengan tangan. Graf bab ini punya segitiga $a\,c\,d$ pada **Gambar 10.7**, dan sisi balik yang dilaporkan, c–a, adalah salah satu dari ketiga sisinya. Sisi balik yang dilaporkan untuk contoh DFS-nya, $x \to v$, persis merupakan sisi yang diberi label **B** pada **Gambar 10.22**. Pohon berisi lima simpul dan hutan berisi dua lintasan itu tak bersiklus, sebagaimana **Gambar 10.10** membuat kita mengharapkannya. Dan menambahkan satu busur ke sebuah DAG merusak ketakbersiklusannya — topik §10.9.5.

10.9.3 Menguji kebipartitan

Persoalan. Dapatkah simpul-simpulnya dibelah menjadi dua kubu sehingga setiap sisi menyeberang di antara keduanya (§10.4.13)?

Gagasan. Jalankan BFS dan warnailah tiap simpul menurut keparitasan jaraknya: kubu 0 jika $d[v]$ genap, kubu 1 jika ganjil. Menurut Akibat 10.5, setiap sisi menghubungkan aras yang berbeda paling banyak satu — sehingga sisi *pohon* selalu menyeberang, sebab aras yang berurutan punya paritas yang berlawanan. Satu-satunya kegagalan yang mungkin adalah sisi yang menghubungkan dua simpul pada aras yang **sama**, yang akan menghubungkan dua simpul berparitas sama. Jadi:

kerjakan BFS-nya, dan setiap kali sebuah sisi menuntun ke simpul yang sudah berwarna, periksalah bahwa warnanya berbeda. Jika suatu saat tidak berbeda, grafnya tidak bipartit; jika tidak pernah begitu, penempatan yang kita bangun adalah bipartisi yang sah, dan itu membuktikan kebipartitannya.

Mengapa kegagalannya sungguh-sungguh kegagalan. Andaikan sisi (u, v) menghubungkan dua simpul dengan $d[u] = d[v] = k$. Susurilah lintasan pohon dari u naik ke akarnya s, dan dari v naik ke s; misalkan z adalah simpul pertama yang keduanya bagi bersama. Kedua penggal lintasannya, $z \rightsquigarrow u$ dan $z \rightsquigarrow v$, punya panjang yang sama, katakanlah ℓ. Maka $z \rightsquigarrow u$, sisi (u, v), dan $v \rightsquigarrow z$ yang dibalik membentuk jalan tertutup berpanjang $2\ell + 1$ — **ganjil**. Ia memuat sebuah siklus ganjil. Dan menurut §10.4.13, graf yang punya siklus ganjil tidak bipartit. Jadi jawaban "tidak" dari algoritmanya berdasar, bukan sekadar kegagalan menemukan pewarnaan.

Graf tak terhubung. Tiap komponen diwarnai secara terpisah, mulai dari sumbernya sendiri. Itu benar, sebab tidak ada sisi yang menyeberang antarkomponen, jadi tidak ada kendala yang mengaitkan keduanya. Inilah sebabnya gelung atas semua simpul diperlukan, bukan hanya satu BFS.

Biaya. Satu BFS per komponen: $O(V + E)$ seluruhnya.

Kode 10.8 Pengujian kebipartitan lewat pewarnaan dua warna selama sebuah BFS. Ketika berhasil ia mencetak kedua kubunya, dan itulah sertifikatnya; ketika gagal ia menyebut sisi yang tidak dapat diwarnai, dan sisi itu menunjuk pada siklus ganjil.

```
// Pengujian kebipartitan lewat pewarnaan dua warna dengan BFS.
// Sebuah graf bipartit tepat ketika ia dapat diwarnai dengan 2 warna sehingga tidak ada
sisi
// yang menghubungkan dua simpul berwarna sama; BFS memberikan warna d[v] mod 2.
#include <iostream>
#include <vector>
#include <queue>
#include <string>
using namespace std;

// Mengembalikan true dan mengisi side[] dengan 0/1 jika bipartit; selain itu mengembalikan
false
// dan meninggalkan di bad[] kedua ujung sisi yang tidak dapat diwarnai.
bool bipartite(const vector<vector<int>>& adj, vector<int>& side, pair<int,int>& bad) {
    int n = (int)adj.size();
    side.assign(n, -1);
    for (int s = 0; s < n; ++s) {
        if (side[s] != -1) continue;          // komponen ini sudah selesai
        side[s] = 0;
        queue<int> Q; Q.push(s);
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            for (int v : adj[u]) {
                if (side[v] == -1) { side[v] = 1 - side[u]; Q.push(v); }
                else if (side[v] == side[u]) { bad = {u, v}; return false; }
            }
        }
    }
    return true;
}

static vector<vector<int>> build(int n, const vector<pair<int,int>>& E) {
    vector<vector<int>> adj(n);
    for (auto e : E) { adj[e.first].push_back(e.second); adj[e.second].push_back(e.first);
}
    return adj;
}

static void report(const string& what, int n, const vector<pair<int,int>>& E) {
    vector<int> side; pair<int,int> bad{-1,-1};
    bool ok = bipartite(build(n, E), side, bad);
    cout << "  " << what << ": ";
    if (!ok) {
        cout << "NOT bipartite (edge " << bad.first << "-" << bad.second
             << " joins two vertices of the same side)\n";
        return;
    }
    cout << "bipartite; side 0 = {";
    bool first = true;
```

Kode 10.8 (Lanjutan) *Pengujian kebipartitan lewat pewarnaan dua warna selama sebuah BFS. Ketika berhasil ia mencetak kedua kubunya, dan itulah sertifikatnya; ketika gagal ia menyebut sisi yang tidak dapat diwarnai, dan sisi itu menunjuk pada siklus ganjil.*

```
    for (int v = 0; v < n; ++v) if (side[v]==0) { cout << (first?"":",") << v; first=false;
}
    cout << "}, side 1 = {";
    first = true;
    for (int v = 0; v < n; ++v) if (side[v]==1) { cout << (first?"":",") << v; first=false;
}
    cout << "}\n";
}

int main() {
    cout << "bipartiteness tests\n";
    report("even cycle C6      ", 6, {{0,1},{1,2},{2,3},{3,4},{4,5},{5,0}});
    report("odd cycle C5       ", 5, {{0,1},{1,2},{2,3},{3,4},{4,0}});
    report("path P4            ", 4, {{0,1},{1,2},{2,3}});
    report("complete K_{2,3}   ", 5, {{0,2},{0,3},{0,4},{1,2},{1,3},{1,4}});
    report("K4 (complete, n=4) ", 4, {{0,1},{0,2},{0,3},{1,2},{1,3},{2,3}});
    report("chapter graph a..e ", 5, {{0,1},{0,2},{0,3},{1,4},{2,3},{2,4},{3,4}});
    report("two disjoint edges ", 4, {{0,1},{2,3}});
    return 0;
}
```

Keluarannya:

```
bipartiteness tests
  even cycle C6      : bipartite; side 0 = {0,2,4}, side 1 = {1,3,5}
  odd cycle C5       : NOT bipartite (edge 2-3 joins two vertices of the same side)
  path P4            : bipartite; side 0 = {0,2}, side 1 = {1,3}
  complete K_{2,3}   : bipartite; side 0 = {0,1}, side 1 = {2,3,4}
  K4 (complete, n=4) : NOT bipartite (edge 1-2 joins two vertices of the same side)
  chapter graph a..e : NOT bipartite (edge 2-3 joins two vertices of the same side)
  two disjoint edges : bipartite; side 0 = {0,2}, side 1 = {1,3}
```

Tujuh jawaban, semuanya cocok dengan analisis tangan §10.4.13. Siklus genap terbelah menurut paritas; siklus ganjil tidak bisa; $K_{2,3}$ terbelah menjadi dua kubu yang memang dimaksudkan; K_4 tidak bisa, sebab tiga simpul mana pun di antara keempatnya membentuk segitiga; dan graf bab ini sendiri gagal pada sisi 2–3, yaitu c–d — satu sisi segitiga $a\ c\ d$, persis seperti yang diramalkan.

10.9.4 Lintasan terpendek pada graf tak berbobot

Ini tidak begitu merupakan sebuah penerapan melainkan pernyataan tentang apa itu BFS, tetapi ia pantas diberi judul sendiri sebab inilah algoritma graf yang paling banyak dipakai di dunia.

Persoalan. Diberikan graf tak berarah atau berarah tanpa bobot sisi dan sebuah sumber s, carilah untuk setiap simpul v banyaknya sisi paling sedikit pada lintasan mana pun dari s ke v, dan salah satu lintasan semacam itu.

Penyelesaian. BFS dari s. Teorema 10.4 menyatakan $d[v] = \delta(s, v)$; §10.7.8 menyatakan bahwa larik π menyandikan lintasan terpendek ke setiap simpul yang terjangkau. Itulah seluruh jawabannya, sudah dihitung dan sudah dicetak oleh **Kode 10.2**.

Tiga penghalusan yang pantas diketahui.

Satu sasaran. Jika Anda hanya menginginkan jarak ke satu sasaran t, berhentilah begitu t dikeluarkan dari antrean. Jawabannya sudah final, sebab simpul dikeluarkan dalam urutan d yang tidak menurun (akibat Lema 10.3), jadi tidak ada yang ditemukan belakangan yang dapat memperbaikinya. Ini bisa menjadi penghematan yang sangat besar: jaringan jalan mungkin punya jutaan simpul dan sasarannya mungkin hanya berjarak tiga lompatan.

Banyak sumber. Jika Anda menginginkan, untuk setiap simpul, jarak ke anggota *terdekat* dari sebuah himpunan sumber S, masukkan seluruh S ke antrean pada awalnya, masing-masing dengan $d = 0$. Segala sesuatu dalam pembuktian Teorema 10.4 tetap berjalan tanpa perubahan dengan $\delta(S, v) = \min_{s \in S} \delta(s, v)$ menggantikan $\delta(s, v)$; alasannya adalah bahwa antrean awalnya tetap memenuhi Lema 10.3, sebab semua isinya punya d yang sama. Satu BFS, $O(V + E)$ — bukan $|S|$ penelusuran terpisah.

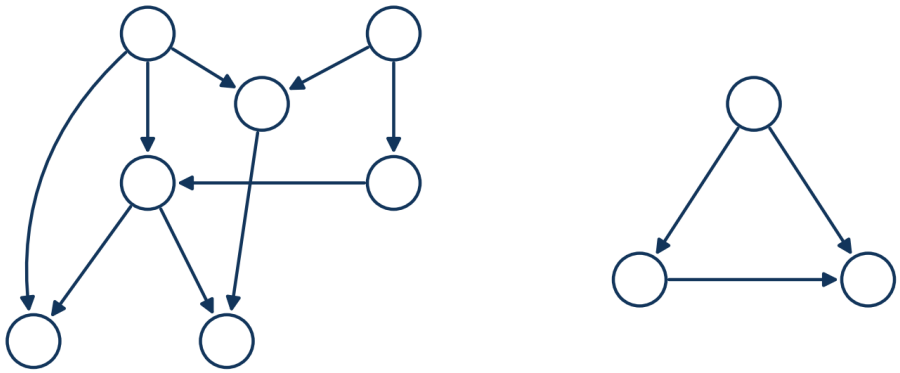
Graf berarah. BFS benar pada graf berarah kata demi kata; tidak ada apa pun di §10.7.9 yang memakai kesetangkupan. Satu-satunya bedanya adalah Adj$[u]$ kini berarti kepala dari busur yang meninggalkan u, sehingga jaraknya dihitung sepanjang busur menurut arahnya yang benar, dan $\delta(s, v)$ tidak harus sama dengan $\delta(v, s)$.

Apa yang tidak dapat dikerjakan BFS. Mengulangi peringatan §10.4.14 karena inilah kesalahpahaman yang paling mahal dalam bab ini: jika sisinya membawa panjang, BFS salah. Pakailah algoritma Dijkstra (Bab 13), yang merupakan BFS dengan antreannya diganti antrean berprioritas. Ada satu perkecualian yang berguna: jika setiap bobotnya sama dengan konstanta c yang sama, maka lintasan dengan sisi paling sedikit juga merupakan lintasan dengan bobot terkecil, dan bobotnya adalah $c \cdot d[v]$. Dan ada trik murah kedua untuk bobot di dalam $\{0,1\}$: pakailah antrean berujung ganda, dengan mendorong tetangga berbobot 0 ke depan dan tetangga berbobot 1 ke belakang. Itulah *0-1 BFS*, ia tetap $O(V + E)$, dan ia pantas diingat untuk soal lomba.

10.9.5 Graf berarah tak bersiklus

> **DEFINISI. Graf berarah tak bersiklus** (*directed acyclic graph*, DAG) adalah graf berarah tanpa siklus berarah.

Gambar 10.28 menggambarkan dua DAG, dan kalimat yang perlu diingat tentang untuk apa DAG itu adalah ini: *sering dipakai untuk menunjukkan prasyarat di antara peristiwa, yaitu peristiwa a harus terjadi sebelum b. Contohnya adalah eksekusi kode paralel.*



Gambar 10.28 Dua graf berarah tak bersiklus. Pada masing-masingnya, ikutilah panahnya dari mana saja dan Anda tidak akan pernah dapat kembali ke tempat Anda memulai: tidak ada siklus berarah. DAG adalah model alami bagi sehimpunan prasyarat — "ini sebelum itu" — dan himpunan prasyarat yang tak bersiklus persis merupakan himpunan yang dapat dipenuhi.

Hampir setiap keadaan "ini harus mendahului itu" adalah DAG, atau seharusnya begitu: tugas-tugas sebuah sistem *build*, mata kuliah sebuah program studi beserta prasyaratnya, medan-medan sebuah lembar kerja beserta rumusnya, tahapan sebuah proses manufaktur, riwayat versi sebuah repositori sumber, lapisan-lapisan sebuah jaringan saraf. Ketika graf semacam itu ternyata *tidak* tak bersiklus, spesifikasinya rusak, dan mengatakannya adalah sebuah jasa.

Pencirian yang membuat semua ini dapat dihitung adalah **teorema DAG**, dan pembuktiannya adalah pemakaian teorema lintasan putih yang elok.

TEOREMA DAG (Teorema 10.10). *Graf berarah G tak bersiklus jika dan hanya jika penelusuran mendalam atas G tidak menghasilkan sisi balik.*

Bukti. Ada dua arah, dan kita buktikan keduanya.

(⇒, dalam bentuk kontraposisi: sisi balik mengakibatkan siklus.) Andaikan DFS-nya menghasilkan sisi balik (u, v). Menurut definisi di §10.8.10, v adalah leluhur u di dalam hutan mendalamnya. Jadi ada lintasan sisi pohon dari v turun ke u, yang khususnya merupakan lintasan berarah dari v ke u di dalam G. Busur (u, v) menutupnya menjadi siklus berarah. Karena itu G tidak tak bersiklus. (Jika $u = v$ maka "lintasannya" kosong dan busurnya adalah gelung-diri, yaitu siklus berarah berpanjang 1.)

(⇐, dalam bentuk kontraposisi: siklus mengakibatkan sisi balik.) Andaikan G memuat siklus berarah c. Di antara simpul-simpul c, misalkan v adalah **yang pertama ditemukan** oleh DFS-nya, dan misalkan u adalah simpul tepat sebelum v pada siklusnya, sehingga (u, v) adalah busur c.

Pada saat $d[v]$ ketika v ditemukan, setiap simpul c yang lain masih putih, sebab v adalah yang pertama di antaranya yang ditemukan. Jadi penggal siklus yang berjalan dari v maju mengelilinginya sampai ke u adalah lintasan simpul putih dari v ke u. Menurut teorema lintasan putih (Teorema 10.8), u karena itu menjadi **keturunan** v di dalam hutan mendalamnya.

Sekarang perhatikan busur (u, v), yang diperiksa selama DFS-Visit(u). Karena u adalah keturunan v, Akibat 10.7 memberikan $d[v] < d[u] < f[u] < f[v]$ — sehingga pada saat (u, v) diperiksa, jamnya menunjukkan suatu waktu di dalam $[d[u], f[u]] \subset [d[v], f[v]]$, yang berarti v masih **kelabu**. Busur yang menuju simpul kelabu adalah sisi balik. ■

Ringkasan padat atas paruh keduanya pantas dihafalkan: *begitu v ditemukan, seluruh siklus dari v ke u berwarna putih; kita harus mengunjungi semua simpul yang terjangkau pada lintasan putih ini sebelum kembali dari DFS-Visit(v), yaitu simpul u menjadi keturunan v; jadi (u, v) adalah sisi balik.*

Teorema 10.10 persis merupakan apa yang diimplementasikan uji berarah **Kode 10.7**, dan ia juga menjelaskan peringatan di §10.9.2: teoremanya mengatakan sisi *balik*, dan hanya uji kelabu yang menemukan sisi balik.

10.9.6 Pengurutan topologis

Persoalan. Diberikan sebuah DAG, daftarkan semua simpulnya dalam sebuah urutan sedemikian sehingga untuk setiap busur (u, v), u datang sebelum v.

DEFINISI. Pengurutan topologis sebuah DAG $G = (V, E)$ adalah pengurutan linear atas V sedemikian sehingga untuk setiap busur $(u, v) \in E$, u muncul sebelum v dalam urutan itu.

Bayangkan simpul-simpulnya ditata pada sebuah garis mendatar; pengurutan topologis adalah susunan yang di dalamnya **setiap panah menunjuk ke kanan**. **Gambar 10.30** persis merupakan gambaran itu.

Mengapa ia harus DAG: seandainya ada siklus berarah $v_1 \to v_2 \to \cdots \to v_k \to v_1$, maka v_1 harus datang sebelum v_2, yang datang sebelum v_3, ..., yang datang sebelum v_1 — sehingga v_1 datang sebelum dirinya sendiri. Mustahil. Sebaliknya, kita akan segera membuktikan bahwa setiap DAG *punya* pengurutan topologis, jadi "adalah DAG" dan "punya pengurutan topologis" adalah syarat yang sama.

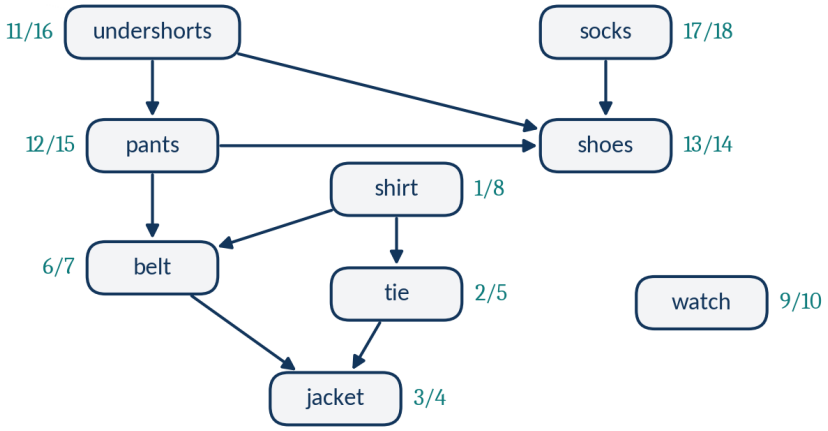
Algoritmanya adalah **Pseudokode 10.3**, dan panjangnya tiga baris:

Pseudokode 10.3 Pengurutan topologis dengan penelusuran mendalam: simpul-simpul dalam urutan waktu selesai yang menurun.

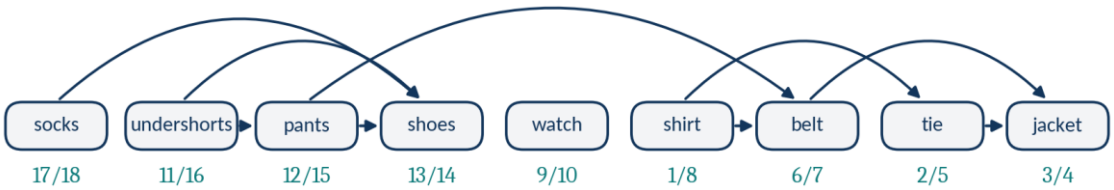
```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times f[v] for each vertex v
2  as each vertex is finished, insert it onto the FRONT of a linked list
3  return the linked list of vertices
```

Jadi urutan topologisnya adalah simpul-simpulnya **dalam urutan waktu selesai yang menurun** — dan langkah 2 mencapainya tanpa pengurutan apa pun, sebab menyisipkan tiap simpul di depan begitu ia selesai secara otomatis membalik urutan selesainya.

Contohnya. Kita memakai persoalan berpakaian yang klasik: busana yang dikenakan seseorang pada pagi hari, dengan panah dari x ke y yang berarti *seseorang harus selesai dengan x sebelum ia dapat mengerjakan y*. Gerak batinnya dalam satu baris: *sebuah tugas hanya dapat dijadwalkan ketika semua subtugasnya sudah dijadwalkan.* **Gambar 10.29** adalah DAG-nya dengan pencap waktu DFS di atasnya, dan **Gambar 10.30** adalah urutan yang dihasilkannya.



Gambar 10.29 DAG berpakaian, diberi pencap waktu oleh sebuah DFS. Sembilan busana; panah dari x ke y berarti x harus dikenakan sebelum y. Capnya adalah shirt 1/8, tie 2/5, jacket 3/4, belt 6/7, watch 9/10, undershorts 11/16, pants 12/15, shoes 13/14, socks 17/18 — delapan belas cap untuk sembilan simpul, sebagaimana dituntut $2|V|$. Si watch sama sekali tidak punya panah: ia boleh dikenakan kapan saja.



Gambar 10.30 Urutan topologisnya: kesembilan simpul yang sama ditata dari kiri ke kanan dalam urutan waktu selesai yang menurun — socks 18, undershorts 16, pants 15, shoes 14, watch 10, shirt 8, belt 7, tie 5, jacket 4 — dan kini **setiap panah menunjuk ke kanan**. Itulah yang disebut pengurutan topologis.

Kebenaran. Pembuktiannya, yang akan kita paparkan selengkapnya, bersandar pada satu klaim tunggal.

> **TEOREMA 10.11.** *Jika G adalah DAG, maka untuk setiap busur $(u, v) \in E$ berlaku $f[v] < f[u]$. Sebagai akibatnya, mendaftarkan simpul-simpulnya dalam urutan f yang menurun adalah pengurutan topologis.*

Bukti. Perhatikan saat busur (u, v) diperiksa. Pada saat itu u kelabu, sebab kita berada di dalam DFS-Visit(u). Ada tepat tiga kemungkinan bagi warna v:

- v **kelabu.** Maka (u, v) adalah sisi balik, dan menurut Teorema 10.10 G punya siklus berarah — bertentangan dengan andaian bahwa G adalah DAG. Jadi kasus ini tidak mungkin terjadi.

- v **putih.** Maka DFS-Visit(v) dipanggil seketika, dari dalam DFS-Visit(u). Jadi v menjadi keturunan u, dan menurut teorema tanda kurung selang v bersarang di dalam selang u: $f[v] < f[u]$.
- v **hitam.** Maka v sudah selesai, dan u belum — ia masih kelabu. Jadi $f[v]$ sudah tercap dan $f[u]$ belum: $f[v] < f[u]$.

Pada kedua kasus yang mungkin, $f[v] < f[u]$. Jadi untuk setiap busur, kepalanya selesai sebelum ekornya. Sekarang daftarkanlah simpulnya dalam urutan f yang menurun. Untuk sembarang busur (u, v), $f[u] > f[v]$ menempatkan u lebih awal dalam daftarnya daripada v. Itulah persis definisi pengurutan topologis. ■

Biaya. Pembukuannya singkat: *penelusuran mendalam memakan waktu $O(V + E)$ (menurut* Persamaan (10.11)*); menyisipkan masing-masing dari $|V|$ simpulnya ke depan senarai berantainya: $O(1)$ per penyisipan. Jadi waktu jalan totalnya adalah $O(V + E)$.* Perhatikan bahwa tidak ada *pengurutan* yang dilakukan, meskipun namanya demikian — menyisipkan di depan begitu tiap simpul selesai memberikan urutan f menurun secara cuma-cuma. Seandainya kita mengumpulkan pasangan $(v, f[v])$ lalu mengurutkannya, ongkosnya akan menjadi $O(V \log V + E)$, yang lebih buruk tanpa manfaat apa pun.

Tidak tunggal. Sebuah DAG biasanya punya banyak pengurutan topologis. Pada **Gambar 10.30**, si watch duduk di antara shoes dan shirt, tetapi ia bisa ditaruh di mana saja: ia tidak punya panah. Urutan mana yang Anda peroleh bergantung pada urutan gelung DFS luarnya mencoba simpul dan pada urutan tiap daftar ketetanggaannya. Mana pun di antaranya adalah jawaban yang benar; jika sebuah program harus menghasilkan satu urutan tertentu — katakanlah yang terkecil secara leksikografis — diperlukan algoritma yang berbeda (lihat Soal 10.14).

Kode 10.9 menjalankan contoh berpakaian itu lalu *memeriksa* keluarannya sendiri, dengan mengecek bahwa setiap busur memang menunjuk ke kanan.

Kode 10.9 Pengurutan topologis DAG berpakaian lewat penelusuran mendalam, disusul lintasan pemeriksaan yang mengecek bahwa setiap busur menunjuk maju dalam urutan yang dihasilkan.

```
// Pengurutan topologis lewat DFS: begitu tiap simpul selesai, dorong ia ke depan
// sebuah senarai. Grafnya adalah contoh berpakaian Subbab 10.9.6.
#include <iostream>
#include <vector>
#include <list>
#include <string>
#include <iomanip>
using namespace std;

enum Color { WHITE, GRAY, BLACK };

struct TopoSort {
    vector<vector<int>> adj;
    vector<string> name;
    vector<Color> color;
    vector<int> d, f;
    int time = 0;
    list<int> order;                      // dibangun dengan mendorong ke depan
    bool cyclic = false;

    TopoSort(vector<vector<int>> g, vector<string> nm)
        : adj(move(g)), name(move(nm)), color(adj.size(), WHITE),
          d(adj.size(),0), f(adj.size(),0) {}

    void run() { for (size_t u = 0; u < adj.size(); ++u) if (color[u]==WHITE)
visit((int)u); }

    void visit(int u) {
        color[u] = GRAY; d[u] = ++time;
        for (int v : adj[u]) {
            if (color[v] == GRAY) cyclic = true;      // sisi balik: bukan DAG
            if (color[v] == WHITE) visit(v);
        }
        color[u] = BLACK; f[u] = ++time;
        order.push_front(u);                           // O(1) per simpul
    }
};
```

Kode 10.9 (Lanjutan) *Pengurutan topologis DAG berpakaian lewat penelusuran mendalam, disusul lintasan pemeriksaan yang mengecek bahwa setiap busur menunjuk maju dalam urutan yang dihasilkan.*

```
int main() {
    // Gelung utamanya memindai simpul dalam urutan indeks ini, dan itulah yang menetapkan
    // pencap waktu di bawah.
    vector<string> name = {"shirt","tie","jacket","belt","watch",
                           "undershorts","pants","shoes","socks"};
    vector<vector<int>> adj(9);
    adj[0] = {1,3};        // shirt       -> tie, belt
    adj[1] = {2};          // tie         -> jacket
    adj[3] = {2};          // belt        -> jacket
    adj[5] = {6,7};        // undershorts -> pants, shoes
    adj[6] = {7,3};        // pants       -> shoes, belt
    adj[8] = {7};          // socks       -> shoes
    // watch (4) dan jacket (2) dan shoes (7) tidak punya busur keluar

    TopoSort t(adj, name);
    t.run();

    cout << "discovery and finishing times\n";
    for (size_t v = 0; v < adj.size(); ++v)
        cout << "  " << setw(12) << left << t.name[v] << " " << t.d[v] << "/" << t.f[v] <<
"\n";

    cout << "\nthe graph " << (t.cyclic ? "has a directed cycle; no topological order
exists"
                                         : "is a DAG") << "\n";

    cout << "\ntopological order (decreasing finishing time)\n  ";
    for (int v : t.order) cout << t.name[v] << " ";
    cout << "\n";

    cout << "\nverification: every arc goes left to right\n";
    vector<int> pos(adj.size());
    int i = 0;
    for (int v : t.order) pos[v] = i++;
    bool allOK = true;
    for (size_t u = 0; u < adj.size(); ++u)
        for (int v : adj[u]) {
            bool ok = pos[u] < pos[v];
            allOK = allOK && ok;
            cout << "  " << setw(12) << left << t.name[u] << " -> "
                 << setw(7) << left << t.name[v]
                 << "  position " << pos[u] << " -> " << pos[v]
                 << (ok ? "   ok" : "   VIOLATION") << "\n";
        }
    cout << (allOK ? "\nall arcs respected.\n" : "\nthe order is wrong.\n");
    return 0;
}
```

Keluarannya:

```
discovery and finishing times
  shirt        1/8
  tie          2/5
  jacket       3/4
  belt         6/7
  watch        9/10
  undershorts  11/16
  pants        12/15
  shoes        13/14
  socks        17/18

the graph is a DAG

topological order (decreasing finishing time)
  socks undershorts pants shoes watch shirt belt tie jacket

verification: every arc goes left to right
  shirt        -> tie      position 5 -> 7   ok
  shirt        -> belt     position 5 -> 6   ok
  tie          -> jacket   position 7 -> 8   ok
  belt         -> jacket   position 6 -> 8   ok
  undershorts  -> pants    position 1 -> 2   ok
  undershorts  -> shoes    position 1 -> 3   ok
  pants        -> shoes    position 2 -> 3   ok
  pants        -> belt     position 2 -> 6   ok
  socks        -> shoes    position 0 -> 3   ok

all arcs respected.
```

Setiap pencap waktunya persis sama dengan yang ada pada **Gambar 10.29**, dan urutan yang tercetak persis sama dengan yang ada pada **Gambar 10.30**: *socks, undershorts, pants, shoes, watch, shirt, belt, tie, jacket*. Kesembilan busurnya menunjuk ke kanan. Programnya sepakat dengan gambarnya baris demi baris.

Menatap ke depan. Bab 11 kembali ke pengurutan topologis dan mengerjakan dua hal dengannya yang tidak dapat dikerjakan bab ini. Pertama, ia memberikan algoritma yang *lain* — membuang simpul berderajat masuk nol berulang-ulang, dikenal sebagai algoritma Kahn — yang juga $O(V + E)$ dan punya kelebihan melaporkan simpul *mana* yang terlibat dalam siklus ketika grafnya bukan DAG. Kedua, ia memakai urutan topologisnya sebagai perancah untuk lintasan terpendek pada DAG, yang dapat dihitung dalam waktu linear dan dengan bobot negatif — sesuatu yang tidak dapat dikerjakan Dijkstra.

10.9.7 Jadi: melebar atau mendalam?

Tabel 10.12 Melebar melawan mendalam, pada sebelas sumbu. Keduanya berongkos sama dan berbeda pada satu baris kode; segala sesuatu lain di dalam tabel ini mengikuti dari satu baris itu.

	Penelusuran melebar	Penelusuran mendalam
Wadah	antrean (FIFO)	tumpukan (LIFO), biasanya tumpukan panggilannya
Mengambil sumber?	ya, satu per panggilan	tidak — ia menelusuri seluruh grafnya
Menghasilkan	sebuah pohon, dari satu komponen	sebuah hutan, merentang seluruh V
Merekam	$d[v]$ dan $\pi[v]$	$d[u]$, $f[u]$, dan $\pi[u]$
Biaya	$O(V+E)$	$\Theta(V+E)$
Ruang tambahan	$O(V)$ untuk antreannya	$O(V)$ untuk tumpukannya — atau tumpukan panggilannya
Menemukan lintasan terpendek?	**ya**, ketika sisinya tak berbobot	tidak
Urutan kunjungan	menurut jarak yang menaik dari sumbernya	sedalam mungkin, lalu mundur
Sisi bukan pohon	hanya antara aras k dan $k, k+1$	balik, maju, dan silang
Wajar untuk	lompatan paling sedikit, benda terdekat, *flood fill*, kerja aras demi aras, kebipartitan	siklus, pengurutan topologis, komponen terhubung kuat, titik artikulasi, labirin, runut balik
Risiko dalam praktik	antrean besar pada graf yang lebar	luapan tumpukan pada graf yang dalam

Aturan praktisnya pendek. **Jika pertanyaannya menyebut jarak, lompatan, terdekat, atau aras, pakailah BFS. Jika pertanyaannya tentang struktur — urutan, siklus, kebergantungan, keterjangkauan tanpa ukuran — pakailah DFS.** Dan jika bukan keduanya, mana pun boleh; pakailah yang dapat Anda tulis dengan benar pada percobaan pertama.

10.9.8 Diukur

Klaim sekuat $O(V+E)$ layak dicocokkan dengan jam, bukan hanya dengan pembuktian. **Kode 10.10** membangun graf terhubung acak dengan $|E| = 4|V|$, menjalankan BFS atas daftar ketetanggaan dan BFS yang sama atas matriks ketetanggaan, lalu melaporkan waktu per satuan $(V+E)$ dan per satuan V^2. Jika teorinya benar, yang pertama seharusnya tetap kira-kira tetap seiring membesarnya grafnya, dan begitu pula yang kedua.

Kode 10.10 Mengukur waktu BFS atas daftar ketetanggaan melawan BFS atas matriks ketetanggaan pada graf yang sama. Sebuah pohon merentang dibangun lebih dulu, sehingga grafnya pasti terhubung dan kedua penelusurannya harus mencapai seluruh $|V|$ simpul — dan programnya memeriksa hal itu sebelum melaporkan waktu apa pun.

```
// Mengukur klaimnya. BFS atas daftar ketetanggaan seharusnya berongkos Theta(V+E);
// BFS yang sama atas matriks ketetanggaan harus membayar Theta(V) per simpul yang keluar
antrean
// ada atau tidak ada sisinya, jadi Theta(V^2).
// Graf ujinya adalah graf terhubung acak dengan |E| = 4|V|.
#include <iostream>
#include <vector>
#include <queue>
#include <random>
#include <chrono>
#include <iomanip>
using namespace std;
using namespace std::chrono;

static long long bfsList(const vector<vector<int>>& adj, int s) {
    int n = (int)adj.size();
    vector<int> d(n, -1);
    queue<int> Q; Q.push(s); d[s] = 0;
    long long visited = 0;
    while (!Q.empty()) {
        int u = Q.front(); Q.pop(); ++visited;
        for (int v : adj[u]) if (d[v] < 0) { d[v] = d[u]+1; Q.push(v); }
    }
    return visited;
}

static long long bfsMatrix(const vector<vector<char>>& M, int s) {
    int n = (int)M.size();
    vector<int> d(n, -1);
    queue<int> Q; Q.push(s); d[s] = 0;
    long long visited = 0;
    while (!Q.empty()) {
        int u = Q.front(); Q.pop(); ++visited;
        for (int v = 0; v < n; ++v)                 // setiap v, ada sisi atau tidak
            if (M[u][v] && d[v] < 0) { d[v] = d[u]+1; Q.push(v); }
    }
    return visited;
}
```

Kode 10.10 (Lanjutan) *Mengukur waktu BFS atas daftar ketetanggaan melawan BFS atas matriks ketetanggaan pada graf yang sama. Sebuah pohon merentang dibangun lebih dulu, sehingga grafnya pasti terhubung dan kedua penelusurannya harus mencapai seluruh |V| simpul — dan programnya memeriksa hal itu sebelum melaporkan waktu apa pun.*

```
int main() {
    cout << "     |V|      |E|   list (ms)  matrix (ms)   list/(V+E) ns   matrix/V^2 ns\n";
    mt19937 rng(20260919);
    for (int n : {1000, 2000, 4000, 8000}) {
        int m = 4 * n;
        vector<vector<int>> adj(n);
        vector<vector<char>> M(n, vector<char>(n, 0));
        auto add = [&](int a, int b) {
            if (a == b || M[a][b]) return false;
            M[a][b] = M[b][a] = 1; adj[a].push_back(b); adj[b].push_back(a); return true;
        };
        for (int v = 1; v < n; ++v) add(v, (int)(rng() % v));   // pohon merentang lebih
dulu
        int placed = n - 1;
        while (placed < m) if (add((int)(rng()%n), (int)(rng()%n))) ++placed;

        auto t0 = steady_clock::now();
        long long a = bfsList(adj, 0);
        auto t1 = steady_clock::now();
        long long b = bfsMatrix(M, 0);
        auto t2 = steady_clock::now();
        if (a != n || b != n) { cout << "graph not connected\n"; return 1; }

        double msL = duration<double, milli>(t1 - t0).count();
        double msM = duration<double, milli>(t2 - t1).count();
        double perVE = msL * 1e6 / (n + m);
        double perV2 = msM * 1e6 / ((double)n * n);
        cout << setw(8) << n << setw(9) << m
             << setw(12) << fixed << setprecision(3) << msL
             << setw(13) << msM
             << setw(17) << setprecision(2) << perVE
             << setw(16) << perV2 << "\n";
    }
    return 0;
}
```

Keluarannya pada mesin yang dipakai menyiapkan bab ini:

```
     |V|      |E|   list (ms)  matrix (ms)   list/(V+E) ns   matrix/V^2 ns
    1000     4000       0.037        0.778             7.38            0.78
    2000     8000       0.072        3.105             7.25            0.78
    4000    16000       0.139       12.263             6.94            0.77
    8000    32000       0.353       47.902             8.83            0.75
```

Tabel 10.13 Pengukuran **Kode 10.10**, dengan nisbah pertumbuhannya ditambahkan. Waktu mutlaknya bergantung pada mesinnya dan berbeda beberapa puluh persen dari satu jalan ke jalan berikutnya, jadi *nisbahnya*-lah yang memikul klaimnya: melipatduakan grafnya kira-kira **melipatduakan** waktu daftar ketetanggaannya dan hampir persis **melipatempatkan** waktu matriksnya — tanda tangan $\Theta(V + E)$ dan $\Theta(V^2)$ berturut-turut. Kedua lajur terakhirnya, waktu dibagi biaya yang diramalkan, tetap di dalam faktor kecil sepanjang rentang ukuran delapan kali lipat, dan itulah arti "linear" dan "kuadratik" dalam praktik.

$\|V\|$	$\|E\|$	Daftar, ms	Matriks, ms	Waktu daftar ÷ sebelumnya	Waktu matriks ÷ sebelumnya	Ns per $(V+E)$	Ns per V^2
1 000	4 000	0.037	0.778	—	—	7.38	0.78
2 000	8 000	0.072	3.105	1.95	3.99	7.25	0.78
4 000	16 000	0.139	12.263	1.93	3.95	6.94	0.77
8 000	32 000	0.353	47.902	2.54	3.91	8.83	0.75

Bacalah tabelnya dengan cermat, sebab ia mengatakan dua hal yang berbeda.

Versi daftarnya linear. Waktunya kira-kira berlipat dua ketika grafnya berlipat dua — nisbahnya 1.95, 1.93, 2.54 — dan waktunya per satuan $(V + E)$ bertahan antara 6.9 dan 8.9 nanodetik sepanjang kenaikan ukuran delapan kali lipat. Ia tidak bertahan *persis* tetap, dan tidak jujur kalau kita mengaku demikian: baris terakhirnya melayang naik sebab pada 8 000 simpul daftar ketetanggaannya tidak lagi muat di tembolok cepat prosesornya dan sebagian pencarian tetangganya harus menjangkau memori yang lebih lambat. Analisis asimtotik dengan sengaja mengabaikan efek semacam itu, dan beginilah rupa "dengan sengaja mengabaikan" pada mesin yang sungguhan — layangan kira-kira seperempat, melawan faktor delapan pada ukuran masukannya.

Versi matriksnya kuadratik. Waktunya 0.778, 3.105, 12.263, 47.902 — masing-masing hampir persis empat kali waktu sebelumnya, dengan nisbah 3.99, 3.95, dan 3.91, persis sebagaimana dituntut V^2 ketika V berlipat dua. Dan waktunya per V^2 datar sampai dalam empat persen, antara 0.75 dan 0.78 nanodetik.

Senjangnya melebar tanpa batas. Pada 1 000 simpul, versi matriksnya kira-kira 21 kali lebih lambat. Pada 8 000 ia kira-kira 136 kali lebih lambat. Mengapa? Versi matriksnya berongkos kira-kira 0.77 ns per sel, dengan V^2 sel seluruhnya; versi daftarnya berongkos kira-kira 7.5 ns per entri, dengan $V + E = 5V$ entri seluruhnya. Nisbahnya karena itu kira-kira $0.77\,V^2/(7.5 \times 5V) \approx V/49$ — ia tumbuh sebanding dengan V. Rumus itu memberi kira-kira 160 pada $V = 8\,000$, dekat dengan 136 yang terukur, dan pada sejuta simpul ia meramalkan penelusuran matriks yang kira-kira dua puluh ribu kali lebih lambat — seandainya matriksnya dapat dialokasikan sama sekali, dan menurut **Tabel 10.4** ia tidak dapat. Inilah isi konkret peringatan di §10.7.7.

10.10 Profitina dalam Praktik

Profitina adalah **platform visibilitas AI (GEO)**: ia mengukur seberapa siap sebuah merek atau situs web untuk *dikutip oleh mesin penjawab AI*, dari sinyal yang sungguhan — seberapa mudah kontennya diekstraksi, apakah data terstruktur hadir dan terbentuk baik, otoritas teknis apa yang dibawa situsnya — lalu membantu pemiliknya membetulkan apa yang ia temukan. Ia hidup di produksi pada `app.profitina.com`. *Generative Engine Optimization* adalah bagi mesin penjawab seperti optimasi mesin pencari dahulu bagi daftar tautan.

Di mana bab ini muncul pada produk itu? Pada hal yang paling pertama dikerjakan pemindainya.

Sebuah situs web adalah graf berarah, dan pemindaiannya adalah penelusuran graf. Simpulnya adalah halaman — lebih tepatnya, URL kanonis. Sisinya adalah pranala, dan sisi itu berarah: tautan dari halaman muka ke halaman harga tidak memberi Anda tautan kembali. Untuk menilai sebuah situs, pemindainya harus mengunjungi halaman-halaman yang terjangkau dari sebuah URL awal, mengunjungi masing-masingnya tepat sekali, lalu berhenti. Itulah penelusuran melebar, dengan ketiga warna §10.6 dan tidak satu gagasan pun lebih:

- **putih** — URL yang kita lihat disebut tetapi belum diambil: ia duduk di antrean perbatasannya;
- **kelabu** — sudah diambil, dan tautannya kini sedang diekstraksi;
- **hitam** — sudah diambil, tautannya diekstraksi, sinyalnya dinilai.

Lupakanlah warnanya dan perayapnya akan mengunjungi ulang halaman selamanya, sebab situs web yang sungguhan penuh dengan siklus: setiap halaman menaut ke halaman muka, yang menaut kembali. Itu bukan kutu hipotetis; itu kutu pertama yang dipunyai setiap perayap.

Mengapa melebar, bukan mendalam. Tiga alasan, dan ketiganya adalah alasan mengapa bab ini menghabiskan waktu begitu lama pada **Tabel 10.12**.

Pertama, **halaman yang penting berada dekat halaman muka.** Halaman penting sebuah situs — muka, produk, harga, tentang kami, kontak — biasanya berjarak 0, 1, atau 2 dari akarnya. BFS mengunjunginya lebih dulu, menurut Teorema 10.4, sehingga pemindaian yang dipotong sesudah anggaran sekian halaman tetap sudah melihat halaman-halaman yang membawa bobot terbesar. Perayapan mendalam bisa saja menghabiskan seluruh anggarannya di satu cabang dalam arsip blog dari tahun 2019.

Kedua, **"kedalaman klik" adalah sinyal GEO yang sungguhan, dan BFS menghitungnya secara cuma-cuma.** $d[v]$ yang dihasilkan BFS persis merupakan banyaknya klik dari halaman muka ke halaman v — jarak lompatan-paling-sedikit §10.9.4. Halaman yang terkubur enam klik dalamnya lebih sulit dicapai perayap, manusia maupun mesin, dan itu pantas disampaikan kepada pemilik situsnya. Lariknya sudah ada di sana; tidak diperlukan kerja tambahan.

Ketiga, **BFS itu sopan.** Perbatasan melebar secara alami menyebarkan permintaan ke seluruh situsnya alih-alih menggempur satu direktori, dan ia bersela rapi dengan pembatas laju per host: Anda ambil kepala antreannya, tunggu selang waktu Anda, lalu ambil berikutnya.

Di mana algoritma lain bab ini muncul.

Komponen terhubung (§10.9.1) mengenali **gerombolan yatim** — kelompok halaman yang saling menaut tetapi tidak ditaut oleh apa pun yang terjangkau dari halaman muka. Halaman-halaman itu tak terlihat oleh perayap yang masuk lewat akarnya, yang berarti tak terlihat oleh mesin penjawab, dan itulah temuan yang pantas dilaporkan.

Deteksi siklus (§10.9.2) tidak dijalankan pada graf halamannya, tempat siklus itu lumrah dan tak berbahaya, melainkan pada **graf kebergantungan sinyal**: sebagian sinyal dihitung dari sinyal lain, dan pipeline yang mengevaluasinya adalah sebuah DAG. Jika suatu perubahan konfigurasi pernah membuat graf itu bersiklus, pipeline-nya akan berputar atau menemui jalan buntu. Pemeriksaan siklus pada saat mula, yang berongkos $O(V + E)$ pada graf berisi beberapa lusin simpul, mengubah gantungnya produksi menjadi sebuah pesan galat.

Pengurutan topologis (§10.9.6) lalu memberikan **urutan evaluasi sinyalnya**, sehingga tidak ada yang dihitung sebelum masukannya. Ini perhitungan yang sama dengan **Kode 10.9**, pada DAG yang berbeda.

Pilihan representasi §10.5 diputuskan **Tabel 10.6** tanpa ragu: situs dengan 50 000 halaman dan 400 000 tautan jelas-jelas jarang, jadi ia disimpan sebagai daftar ketetanggaan. Matriks ketetanggaan akan memerlukan 2.5×10^9 entri untuk graf yang daftar ketetanggaannya memerlukan kurang dari sejuta, dan pada ongkos per entri yang terukur di **Tabel 10.13** perayapannya juga akan beberapa ratus kali lebih lambat — kira-kira $0.77 \times 2.5 \times 10^9$ ns melawan $7.5 \times 4.5 \times 10^5$ ns, faktor sekitar 570.

Dua kendala dari luar algoritmanya pantas disebut, sebab pada produk yang sungguhan keduanya sama pentingnya dengan asimtotiknya. **UU PDP 27/2022** mengatur data pribadi, sehingga graf halamannya disimpan sebagai URL dan struktur tautan dan tidak lebih — tidak ada isi halaman yang mungkin memuat data pribadi yang disimpan melebihi apa yang diperlukan penilaiannya. Dan produknya **memberi saran; penggunalah yang memutuskan**: temuan kedalaman klik disajikan sebagai fakta terukur beserta penjelasannya, tidak pernah sebagai jaminan tentang bagaimana suatu mesin penjawab akan berperilaku.

10.10.1 Studi kasus yang dikerjakan: Profitina Laundry

Studi kasus yang dapat dijalankan dalam buku ini selalu **Profitina Laundry**, sebuah usaha kecil untuk pengajaran, bukan produk yang sungguhan. Ia adalah binatu dengan satu gerai di Keputih, dekat kampus ITS di Surabaya, seorang kurir bermotor, dan cita-cita menjanjikan penjemputan di hari yang sama.

Dua pertanyaan tiba pada pagi yang sama, dan keduanya pertanyaan graf.

Pertanyaan 1. "Kelurahan mana saja yang dapat kita janjikan penjemputan dua lompatan?" Kurirnya dapat berkendara dari satu kelurahan ke kelurahan yang bertetangga; gerainya ada di Keputih. Simpulnya adalah kelurahan yang dilayani; ada sisi antara dua kelurahan jika kurirnya dapat berkendara langsung di antara keduanya. Setiap perjalanan berongkos waktu kira-kira sama, jadi ini graf **tak berbobot** dan pertanyaannya adalah lompatan paling sedikit: BFS dari Keputih, lalu bacalah $d[v]$. (Seandainya perjalanannya berdurasi berbeda-beda, kita akan berada di Bab 13, dan ini akan menjadi algoritma yang salah — peringatan §10.4.14 berlaku untuk binatu sama seperti untuk apa pun.)

BFS itu menjawab pertanyaan kedua secara cuma-cuma. Kelurahan mana pun dengan $d[v] = \infty$ **sama sekali tidak terjangkau** — ia berada di komponen terhubung yang berbeda, dan tidak ada rute kurir yang ada. Itu bukan persoalan penjadwalan; itu persoalan gerai kedua, dan itu pantas diketahui sebelum sebuah janji dicetak di spanduk.

Pertanyaan 2. "Dalam urutan apa sekantong cucian harus dikerjakan?" Penerimaan, penimbangan, pemilahan, perawatan awal, pencucian, pengeringan, penyetrikaan, pelipatan, pengemasan, penagihan, pemberitahuan. Sebagian di antaranya harus mendahului yang lain — Anda tidak dapat mengeringkan sebelum mencuci — dan sebagian saling bebas: tagihannya dapat dibuat kapan saja sesudah penimbangan. Simpulnya adalah stasiun kerja; busur dari x ke y berarti x harus selesai sebelum y boleh mulai. Itu sebuah DAG, dan urutan kerja yang sah adalah pengurutan topologis. Dan jika seseorang menuliskan sehimpunan aturan yang *bukan* DAG, programnya harus mengatakannya dan menyebut siklusnya, alih-alih menghasilkan urutan yang tampak masuk akal tetapi tidak dapat diikuti.

Kode 10.11 menjawab keduanya.

Kode 10.11 Studi kasus Profitina Laundry: penelusuran melebar pada jaringan penjemputan antarkelurahan, dan pengurutan topologis pada DAG proses pencucian, beserta apa yang terjadi ketika spesifikasi prosesnya dibuat bersiklus.

```
// Profitina Laundry: dua persoalan graf dari satu hari kerja.
//   Bagian 1  jaringan penjemputan sebuah gerai, sebagai graf tak berarah
//             berisi kelurahan; BFS menjawab "berapa lompatan jauhnya, dan lewat
//             rute mana", dan komponennya memberi tahu zona mana yang tak dicapai kurir.
//   Bagian 2  graf proses pencucian, sebagai DAG; pengurutan topologis adalah
//             urutan kerja yang sah, dan sisi balik adalah kutu spesifikasi.
#include <iostream>
#include <vector>
#include <string>
#include <queue>
#include <list>
#include <algorithm>
#include <iomanip>
#include <climits>
using namespace std;

// ------------------------------------------------------ Bagian 1: BFS
struct Zones {
    vector<string> name;
    vector<vector<int>> adj;
    int edges = 0;

    int add(const string& s) { name.push_back(s); adj.emplace_back(); return
(int)name.size()-1; }
    void road(int a, int b) { adj[a].push_back(b); adj[b].push_back(a); ++edges; }
    int  n() const { return (int)name.size(); }
};

struct BfsResult { vector<int> d, pi; long long edgesScanned = 0; };

BfsResult bfs(const Zones& Z, int s) {
    BfsResult R;
    R.d.assign(Z.n(), INT_MAX);
    R.pi.assign(Z.n(), -1);
    R.d[s] = 0;
    queue<int> Q; Q.push(s);
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        for (int v : Z.adj[u]) {
            ++R.edgesScanned;                        // tiap slot berarah sekali
            if (R.d[v] == INT_MAX) { R.d[v] = R.d[u] + 1; R.pi[v] = u; Q.push(v); }
        }
    }
    return R;
}
```

Kode 10.11 (Lanjutan) *Studi kasus Profitina Laundry: penelusuran melebar pada jaringan penjemputan antarkelurahan, dan pengurutan topologis pada DAG proses pencucian, beserta apa yang terjadi ketika spesifikasi prosesnya dibuat bersiklus.*

```
// ------------------------------------------------------ Bagian 2: DFS
enum Color { WHITE, GRAY, BLACK };

struct Process {
    vector<string> name;
    vector<vector<int>> adj;
    int add(const string& s) { name.push_back(s); adj.emplace_back(); return
(int)name.size()-1; }
    void before(int a, int b) { adj[a].push_back(b); }   // a harus selesai sebelum b
    int  n() const { return (int)name.size(); }
};

struct Topo {
    const Process& P;
    vector<Color> color;
    list<int> order;
    bool cyclic = false;
    string cycle;                                   // siklusnya, dieja lengkap
    vector<int> grayStack;                          // rantai simpul kelabu
    explicit Topo(const Process& p) : P(p), color(p.n(), WHITE) {}
    void run() { for (int u = 0; u < P.n(); ++u) if (color[u]==WHITE) visit(u); }
    void visit(int u) {
        color[u] = GRAY;
        grayStack.push_back(u);
        for (int v : P.adj[u]) {
            if (color[v] == GRAY && !cyclic) {      // sisi balik menutup sebuah siklus
                cyclic = true;
                size_t i = 0;
                while (grayStack[i] != v) ++i;       // v ada di suatu tempat di bawah u
                for (; i < grayStack.size(); ++i) cycle += P.name[grayStack[i]] + " -> ";
                cycle += P.name[v];
            }
            if (color[v] == WHITE) visit(v);
        }
        color[u] = BLACK;
        grayStack.pop_back();
        order.push_front(u);
    }
};
```

Kode 10.11 (Lanjutan) *Studi kasus Profitina Laundry: penelusuran melebar pada jaringan penjemputan antarkelurahan, dan pengurutan topologis pada DAG proses pencucian, beserta apa yang terjadi ketika spesifikasi prosesnya dibuat bersiklus.*

```
int main() {
    cout << "================ Part 1: the pickup network ================\n";
    Zones Z;
    int keputih   = Z.add("Keputih");
    int gebang    = Z.add("Gebang Putih");
    int klampis   = Z.add("Klampis Ngasem");
    int semolo    = Z.add("Semolowaru");
    int sukolilo  = Z.add("Sukolilo");
    int medokan   = Z.add("Medokan Semampir");
    int manyar    = Z.add("Manyar Sabrangan");
    int nginden   = Z.add("Nginden Jangkungan");
    int mulyorejo = Z.add("Mulyorejo");
    int kalijudan = Z.add("Kalijudan");
    Z.add("Dukuh Setro");   // di seberang sungai: tidak ada jalan yang mencapainya, jadi
ia tak perlu nama di sini

    Z.road(keputih, gebang);
    Z.road(keputih, sukolilo);
    Z.road(gebang, klampis);
    Z.road(gebang, mulyorejo);
    Z.road(klampis, semolo);
    Z.road(klampis, manyar);
    Z.road(semolo, nginden);
    Z.road(semolo, medokan);
    Z.road(sukolilo, medokan);
    Z.road(manyar, nginden);
    Z.road(mulyorejo, kalijudan);
    // Dukuh Setro sengaja dibiarkan tanpa jalan

    cout << "|V| = " << Z.n() << " zones, |E| = " << Z.edges << " roads\n";
    BfsResult R = bfs(Z, keputih);
    cout << "\nBFS from the outlet in " << Z.name[keputih] << "\n";
    cout << "  " << setw(20) << left << "zone" << setw(6) << "hops" << "route\n";
    for (int v = 0; v < Z.n(); ++v) {
        cout << "  " << setw(20) << left << Z.name[v];
        if (R.d[v] == INT_MAX) { cout << setw(6) << "-" << "unreachable by road\n";
continue; }
        cout << setw(6) << R.d[v];
        vector<int> p;
        for (int x = v; x != -1; x = R.pi[x]) p.push_back(x);
        reverse(p.begin(), p.end());
        for (size_t i = 0; i < p.size(); ++i) cout << (i ? " > " : "") << Z.name[p[i]];
        cout << "\n";
    }
    cout << "\n  edge slots scanned = " << R.edgesScanned
         << ", and 2|E| = " << 2*Z.edges << "\n";
```

Kode 10.11 (Lanjutan) *Studi kasus Profitina Laundry: penelusuran melebar pada jaringan penjemputan antarkelurahan, dan pengurutan topologis pada DAG proses pencucian, beserta apa yang terjadi ketika spesifikasi prosesnya dibuat bersiklus.*

```
    int far = 0;
    for (int v = 0; v < Z.n(); ++v)
        if (R.d[v] != INT_MAX && R.d[v] > R.d[far]) far = v;
    cout << "  furthest reachable zone: " << Z.name[far] << " at " << R.d[far] << "
hops\n";
    cout << "  a 2-hop promise covers:";
    for (int v = 0; v < Z.n(); ++v)
        if (R.d[v] != INT_MAX && R.d[v] <= 2) cout << " " << Z.name[v] << ",";
    cout << " and nothing else\n";

    cout << "\n================ Part 2: the wash process ================\n";
    Process P;
    int intake  = P.add("intake");
    int weigh   = P.add("weigh");
    int sort_   = P.add("sort");
    int pretreat= P.add("pretreat");
    int wash    = P.add("wash");
    int dry     = P.add("dry");
    int iron    = P.add("iron");
    int fold    = P.add("fold");
    int pack    = P.add("pack");
    int invoice = P.add("invoice");
    int notify  = P.add("notify");

    P.before(intake, weigh);
    P.before(weigh, sort_);
    P.before(weigh, invoice);
    P.before(sort_, pretreat);
    P.before(sort_, wash);
    P.before(pretreat, wash);
    P.before(wash, dry);
    P.before(dry, iron);
    P.before(dry, fold);
    P.before(iron, fold);
    P.before(fold, pack);
    P.before(invoice, pack);
    P.before(pack, notify);

    Topo T(P);
    T.run();
    cout << "is a DAG: " << (T.cyclic ? "no" : "yes") << "\n";
    cout << "one legal order of work:\n  ";
    for (int v : T.order) cout << P.name[v] << " ";
    cout << "\n";
```

Kode 10.11 (Lanjutan) *Studi kasus Profitina Laundry: penelusuran melebar pada jaringan penjemputan antarkelurahan, dan pengurutan topologis pada DAG proses pencucian, beserta apa yang terjadi ketika spesifikasi prosesnya dibuat bersiklus.*

```
    // Sekarang rusakkan spesifikasinya: bersikeras bahwa tagihannya baru diterbitkan
    // sesudah pelanggannya diberi tahu.
    Process Bad = P;
    Bad.before(notify, invoice);
    Topo T2(Bad);
    T2.run();
    cout << "\nafter adding 'notify -> invoice':\n";
    cout << "  is a DAG: " << (T2.cyclic ? "no" : "yes") << "\n";
    if (T2.cyclic) cout << "  the cycle is: " << T2.cycle
                        << "\n  no order of work can satisfy this\n";
    return 0;
}
```

Keluarannya:

```
================ Part 1: the pickup network ================
|V| = 11 zones, |E| = 11 roads

BFS from the outlet in Keputih
  zone                hops  route
  Keputih             0     Keputih
  Gebang Putih        1     Keputih > Gebang Putih
  Klampis Ngasem      2     Keputih > Gebang Putih > Klampis Ngasem
  Semolowaru          3     Keputih > Gebang Putih > Klampis Ngasem > Semolowaru
  Sukolilo            1     Keputih > Sukolilo
  Medokan Semampir    2     Keputih > Sukolilo > Medokan Semampir
  Manyar Sabrangan    3     Keputih > Gebang Putih > Klampis Ngasem > Manyar Sabrangan
  Nginden Jangkungan  4     Keputih > Gebang Putih > Klampis Ngasem > Semolowaru > Nginden
Jangkungan
  Mulyorejo           2     Keputih > Gebang Putih > Mulyorejo
  Kalijudan           3     Keputih > Gebang Putih > Mulyorejo > Kalijudan
  Dukuh Setro         -     unreachable by road

  edge slots scanned = 22, and 2|E| = 22
  furthest reachable zone: Nginden Jangkungan at 4 hops
  a 2-hop promise covers: Keputih, Gebang Putih, Klampis Ngasem, Sukolilo, Medokan
Semampir, Mulyorejo, and nothing else

================ Part 2: the wash process ================
is a DAG: yes
one legal order of work:
  intake weigh invoice sort pretreat wash dry iron fold pack notify

after adding 'notify -> invoice':
  is a DAG: no
  the cycle is: pack -> notify -> invoice -> pack
  no order of work can satisfy this
```

Ada lima hal dalam keluaran itu yang pantas ditunjuk.

Jawaban atas pertanyaan 1 adalah keputusan bisnis yang dibuat dapat dihitung. Enam dari sebelas zonanya berada dalam dua lompatan; janji dua lompatan dapat dicetak dengan jujur untuk keenam zona itu dan tidak lebih. Nginden Jangkungan berjarak empat lompatan dan sebaiknya dihargai berbeda, atau dilayani dari gerai kedua.

Dukuh Setro dilaporkan tak terjangkau, bukan jauh. Nilai d-nya tetap ∞, dan itulah "belum diketahui terjangkau" dari §10.7.2 yang sedang mengerjakan tugasnya. Melaporkannya sebagai bilangan besar akan menjadi kebohongan; melaporkannya sebagai tak terjangkau adalah sebuah temuan. Inilah alasan praktis menginisialisasi jarak dengan ∞ alih-alih dengan -1 atau 0.

Cacahan slot sisinya tepat $2|E|$. Programnya mencacah 22 pemeriksaan untuk 11 sisi tak berarah. Itulah Akibat 2 lema jabat tangan (§10.4.3) yang diukur alih-alih dibuktikan, dan itulah isi konkret suku $O(E)$ di §10.7.7: penelusurannya melihat tiap sisi dari tiap ujungnya, sekali, dan tidak pernah lagi.

Rutenya keluar dari π, bukan dari penelusuran kedua. Setiap rute yang tercetak direkonstruksi dengan menyusuri pendahulunya kembali ke gerainya lalu membaliknya — operasi §10.7.5. Satu BFS menghasilkan sebelas rute.

Siklusnya disebut namanya, bukan sekadar dideteksi. Ketika aturan *pemberitahuan sebelum penagihan* ditambahkan, programnya melaporkan `pack -> notify -> invoice -> pack`. Itulah rantai simpul kelabu pada saat sisi baliknya ditemukan, dan itulah persis alasan tumpukan kelabunya disimpan: pesan galat yang berbunyi "ada siklus" mengirim orang berburu; pesan galat

yang menyebut ketiga langkah yang terlibat mengakhiri percakapannya. Teorema 10.10 adalah yang menjamin bahwa menemukan sisi balik sama dengan menemukan siklus.

10.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. Lupa menandai simpul sebagai sudah dilihat. Programnya berjalan selamanya, atau sampai memorinya habis. Inilah kesalahannya, dan setiap kesalahan lain adalah penghalusan dari kesalahan ini. Obatnya adalah disiplin: baris yang menemukan sebuah simpul dan baris yang menandainya harus ditulis *pada saat yang sama*, sebelum Anda menulis apa pun yang lain di dalam gelungnya.

2. Menandai saat keluar antrean alih-alih saat masuk antrean. Pada BFS, catlah v kelabu pada saat Anda mendorongnya (baris 13 pada **Gambar 10.15**), bukan ketika Anda kemudian mengambilnya. Jika Anda menandai saat pengambilan, simpul dengan tiga tetangga tak bertanda yang menunjuk kepadanya akan masuk ke antrean tiga kali; programnya tetap berhenti dan tetap menghitung jarak dengan benar, tetapi antreannya dapat membesar sampai $\Theta(E)$ alih-alih $\Theta(V)$ dan waktu jalannya memburuk. Pada graf padat inilah beda antara program yang jalan dan program yang kehabisan memori.

3. Memakai BFS pada graf berbobot. Diuraikan panjang lebar di §10.4.14 dan §10.9.4, dan diulang di sini karena inilah kekeliruan yang paling mahal dalam bab ini: BFS meminimumkan banyaknya sisi, bukan bobot totalnya. Gejalanya adalah jawaban yang *masuk akal tetapi salah*, dan itu jauh lebih buruk daripada kerontokan program. Bab 13 adalah perbaikannya.

4. Memakai `visited` padahal Anda memerlukan tiga warna. Untuk deteksi siklus berarah, ujinya adalah "apakah v **kelabu**", bukan "sudahkah saya melihat v" (§10.9.2). Penanda dua keadaan melaporkan sisi maju dan sisi silang sebagai siklus. Graf $u \to v$, $u \to w$, $v \to w$ tak bersiklus dan uji berbasis `visited` mengatakan sebaliknya.

5. Lupa induknya pada deteksi siklus tak berarah. Sebaliknya, pada graf *tak berarah* setiap sisi pohon tampak seperti sisi balik dari sudut pandang anaknya, sebab induknya kelabu dan berada di daftar ketetanggaan anaknya. Anda harus melewati sisi yang Anda lalui saat masuk — dan melewatinya tepat sekali, supaya sisi sejajar yang sungguhan tetap tertangkap (**Kode 10.7**).

6. Mengeluarkan dari antrean dengan `vector::erase(begin())`. Itu operasi $\Theta(V)$, dan melakukannya sekali per simpul membuat seluruh penelusurannya $\Theta(V^2)$. Pakailah `std::queue`, `std::deque`, atau larik dengan indeks kepala. Peringatan yang sama berlaku untuk `list::size()` pada implementasi lama dan untuk "buang dari depan" mana pun yang menggeser elemen.

7. Berekursi pada graf yang dalam. DFS rekursif pada lintasan sejuta simpul meluapkan tumpukan panggilan dan programnya mati tanpa pesan yang berguna. Jika grafnya mungkin dalam, pakailah bentuk iteratif **Kode 10.3** — dan ingatlah bahwa ia memerlukan indeks lanjutannya, bukan sekadar tumpukan simpul.

8. Melanjutkan daftar ketetanggaan dari awal. Inilah kutu khusus yang menjerat orang yang mengubah DFS rekursif menjadi iteratif: ketika kembali ke sebuah simpul, mereka memindai ulang Adj[u] dari indeks 0. Penelusurannya lalu berputar-putar atau memakan waktu kuadratik. `Frame { int u; size_t next; }` pada **Kode 10.3** ada justru untuk mencegah hal ini.

9. Membangun daftar ketetanggaan dengan hanya satu arah, pada graf tak berarah. `adj[u].push_back(v)` tanpa `adj[v].push_back(u)`. Penelusurannya lalu berhasil dari satu ujung

dan gagal diam-diam dari ujung yang lain. Tulislah satu penolong `addEdge` tunggal yang mengerjakan keduanya, dan jangan pernah mendorong langsung.

10. Mengalokasikan matriks $V \times V$ untuk graf jarang. $|V| = 10^6$ memerlukan 10^{12} entri. Alokasinya gagal, atau lebih buruk lagi, berhasil pada mesin dengan memori yang dijanjikan berlebih lalu tergagap-gagap. **Tabel 10.4** adalah tabel yang perlu dilihat sebelum menulis `vector<vector<char>> M(n, vector<char>(n))`.

11. Mengandaikan satu penelusuran mencakup seluruh grafnya. BFS dari s hanya mencapai komponen s. Jika tugasnya adalah "setiap simpul", Anda memerlukan gelung luar atas semua simpul yang sudah tertanam di dalam DFS dan tidak ada di dalam BFS (**Kode 10.6**).

12. Membaca $d[v] = \infty$ sebagai "sangat jauh". Artinya *tak terjangkau*. Mencetaknya, membandingkannya, atau menambahinya satu semuanya kutu; dalam C++, `INT_MAX + 1` adalah luapan bertanda dan karena itu perilaku tak terdefinisi. Ujilah penandanya secara eksplisit, seperti yang dilakukan **Kode 10.11**.

13. Mengharapkan jawaban yang tunggal. Jarak yang dihitung sebuah BFS bersifat tunggal; pohonnya, larik π-nya, urutan antreannya, pencap waktu DFS-nya, dan urutan topologisnya tidak — semuanya bergantung pada urutan daftar ketetanggaannya (§10.7.4, §10.9.6). Dua program yang sama-sama benar dapat mencetak pohon yang berbeda. Bandingkanlah hanya apa yang memang tertentukan.

14. Mengacaukan d pada BFS dengan d pada DFS. Pada BFS, $d[v]$ adalah sebuah *jarak*. Pada DFS, $d[u]$ adalah sebuah *waktu penemuan*. Keduanya besaran yang tak berhubungan yang berbagi satu huruf sebab rujukan baku [1] memakainya untuk keduanya. Ketika Anda menulis kedua algoritma dalam satu program, ganti nama salah satunya.

15. Mengklaim $O(V + E)$ sambil memakai matriks. Batas itu adalah pernyataan tentang daftar ketetanggaan (§10.7.7). **Tabel 10.13** adalah biaya kode yang sama atas sebuah matriks.

10.12 Ringkasan Bab

Sebuah **graf** $G = (V, E)$ adalah himpunan simpul dan himpunan sisi, dengan tiap sisi berupa sepasang simpul; pasangannya terurut pada graf berarah dan tak terurut pada graf tak berarah (**Gambar 10.1**). Gambarnya bukanlah grafnya.

Kosakatanya terhimpun dalam satu tabel, **Tabel 10.3**. Satu hasil di dalamnya yang dipakai di mana-mana adalah **lema jabat tangan** Persamaan (10.1), $\sum_v \deg(v) = 2|E|$, dibuktikan di §10.4.3 dengan mencacah ujung-sisi dengan dua cara; bentuk pemrogramnya adalah bahwa semua daftar ketetanggaannya bersama-sama memuat $2|E|$ entri, dan fakta itulah sumber E pada setiap waktu jalan dalam bab ini.

Ada **tiga representasi**. *Daftar sisi* hanya terbaik untuk mendaftarkan semua sisinya. *Daftar ketetanggaan* memakai ruang $\Theta(V + E)$ dan menjawab "daftarkan tetangga v" secara optimal, dan itulah satu-satunya pertanyaan yang diajukan sebuah penelusuran; ia adalah pilihan bawaan. *Matriks ketetanggaan* memakai ruang $\Theta(V^2)$ dan menjawab "apakah (u, v) sebuah sisi?" dalam waktu tetap; pakailah untuk graf padat dan algoritma bergaya matriks. **Tabel 10.5** memuat setiap ongkosnya dan **Tabel 10.6** keputusannya.

Penelusuran melebar mengelola perbatasannya sebagai **antrean**. Ia menghitung, untuk sumber s, jarak $d[v] = \delta(s, v)$ ke setiap simpul dan pendahulu $\pi[v]$ yang darinya sebuah lintasan terpendek dapat dibaca. Ia berjalan dalam $O(V + E)$, yang linear terhadap ukuran masukannya.

Teorema 10.4 membuktikan kebenarannya, bersandar pada Lema 10.1 (satu langkah tidak dapat membawa Anda dua langkah lebih jauh), Lema 10.2 (BFS tidak pernah menaksir terlalu rendah), dan Lema 10.3 (antreannya memuat paling banyak dua jarak berdampingan, terurut). Keluarannya adalah **pohon melebar**, yaitu pohon merentang dari komponen s yang setiap lintasan akar-ke-simpulnya merupakan lintasan terpendek.

Penelusuran mendalam mengelola perbatasannya sebagai **tumpukan** — biasanya tumpukan panggilannya. Ia tidak mengambil sumber dan menelusuri seluruh grafnya, menghasilkan **hutan mendalam** dan dua pencap waktu per simpul, yaitu waktu penemuan $d[u]$ dan waktu selesai $f[u]$. Ia berjalan dalam $\Theta(V + E)$. Teorema strukturalnya adalah **teorema tanda kurung** (selang sembarang dua simpul bersarang atau saling lepas, tidak pernah bertindihan sebagian) dan **teorema lintasan putih** (v menjadi keturunan u tepat ketika sebuah lintasan simpul putih membentang dari u ke v pada saat u ditemukan). Setiap sisi adalah sisi **pohon**, **balik**, **maju**, atau **silang**, ditentukan oleh warna kepalanya ketika ia diperiksa; pada graf tak berarah hanya sisi pohon dan sisi balik yang muncul (Teorema 10.9).

Enam penerapan, semuanya $O(V + E)$: *komponen terhubung* lewat penelusuran berulang; *deteksi siklus*, lewat sisi balik bukan induk pada graf tak berarah dan lewat busur yang menuju simpul kelabu pada graf berarah; *kebipartitan* lewat pewarnaan dua warna selama sebuah BFS, yang berhasil kecuali ada sisi yang menghubungkan dua simpul searas; *lintasan terpendek* pada graf tak berbobot, yang tak lain adalah BFS; **teorema DAG**, bahwa graf berarah tak bersiklus tepat ketika sebuah DFS tidak menemukan sisi balik; dan **pengurutan topologis**, yang mendaftarkan simpul sebuah DAG dalam urutan waktu selesai yang menurun dan tidak memerlukan pengurutan apa pun untuk melakukannya.

Gagasan tunggal yang paling dapat dipindahkan dari bab ini adalah gagasan di awal §10.6: **BFS dan DFS berbeda dalam satu baris — wadahnya — dan segala sesuatu lain tentang keduanya mengikuti dari situ.**

10.12.1 Menatap ke depan

Bab 11 menaruh bobot pada sisinya dan meminta pohon merentang termurah. Kedua algoritma yang ia berikan — Prim dan Kruskal — bersifat *greedy* dalam arti Bab 9, dan keduanya adalah penelusuran graf dalam arti bab ini: Prim adalah penelusuran melebar dengan antreannya diganti antrean berprioritas, dan Kruskal adalah sapuan atas daftar sisi §10.5.2. Bab 11 juga kembali ke pengurutan topologis, memberikan algoritma derajat masuk milik Kahn sebagai alternatif bagi §10.9.6, dan memakai urutan topologisnya untuk menyelesaikan lintasan terpendek pada DAG dalam waktu linear.

Bab 13 mengganti "sisi paling sedikit" dengan "bobot total terkecil" dan memberikan algoritma Dijkstra, yang persis merupakan BFS bab ini dengan `std::queue` diganti `std::priority_queue` — lalu Bellman-Ford, untuk ketika sebagian bobotnya negatif dan bahkan itu pun tidak cukup. Bab 15 meminta jarak antara *setiap* pasangan simpul, dan di sana matriks ketetanggaan §10.5.4 akhirnya menjadi representasi yang tepat, sebab algoritmanya adalah aritmetika matriks.

Segala sesuatu dalam keempat bab itu mengandaikan bahwa Anda dapat menjelajahi sebuah graf dalam waktu linear dan tahu mengapa ia linear. Itulah gunanya bab ini.

10.13 Soal Latihan

10.1 Untuk graf G pada **Gambar 10.1**: (a) tuliskan $\mathrm{Adj}[v]$ untuk kelima simpulnya; (b) berikan $\deg(v)$ untuk masing-masingnya; (c) periksalah lema jabat tangan secara numeris; (d) ada berapa pasangan tak terurut simpul berbeda yang *tidak* bertetangga, dan pasangan mana saja?

10.2 Buktikan bahwa pada graf tak berarah mana pun, banyaknya simpul berderajat ganjil selalu genap. Lalu putuskan, dengan alasan, apakah ada graf sederhana tak berarah dengan barisan derajat (3,3,3,3,3), dan apakah ada yang berbarisan derajat (4,4,4,4,4).

10.3 Pada graf **Gambar 10.1**, daftarkan **setiap** lintasan sederhana dari b ke d, dalam urutan panjang yang menaik. Nyatakan $\delta(b, d)$ dan katakan bagaimana Anda tahu bahwa itulah minimumnya.

10.4 Sebuah graf berarah punya $V = \{0,1,2,3\}$ dan daftar sisi $[(0,1), (0,2), (1,2), (2,0), (2,3), (3,3)]$. (a) Gambarlah. (b) Berikan daftar ketetanggaannya. (c) Berikan matriks ketetanggaannya. (d) Berikan $\deg^{+}(v)$ dan $\deg^{-}(v)$ untuk tiap v lalu periksalah kedua jumlahnya. (e) Cacahlah kata mesin yang diperlukan masing-masing dari ketiga representasinya, seperti pada **Tabel 10.4**.

10.5 Untuk tiap keadaan, pilihlah sebuah representasi dari **Tabel 10.6** dan berikan alasannya dalam satu kalimat. (a) Jejaring sosial dengan 3×10^8 pengguna, masing-masing punya kira-kira 200 teman, yang di atasnya Anda harus menjalankan BFS untuk mencari derajat keterpisahan. (b) Tabel jarak 40×40 antarkota, yang di atasnya Anda akan menjalankan algoritma lintasan terpendek semua pasangan. (c) Berkas berisi 10^7 sisi yang harus Anda urutkan menurut bobot lalu Anda proses berurutan. (d) Graf berisi 5 000 simpul yang gelung terdalam algoritmanya menanyakan "apakah (u, v) sebuah sisi?" kira-kira 10^9 kali.

10.6 Jalankan BFS pada graf §10.7.4 dengan sumber w alih-alih s, memakai urutan daftar ketetanggaan yang sama. Hasilkan sebuah tabel dalam format **Tabel 10.7**, lalu berikan larik d dan π akhirnya serta sisi pohonnya.

10.7 Tunjukkan bahwa untuk graf §10.7.4 dengan sumber s, **pohon** melebarnya sama saja bagaimanapun daftar ketetanggaannya diurutkan — meskipun urutan antreannya tidak. Lalu berikan sebuah graf berisi empat simpul yang pohonnya sungguh-sungguh bergantung pada urutannya, dan perlihatkan dua pohon yang berbeda.

10.8 Akibat 10.5 menyatakan bahwa pada graf tak berarah setiap sisi menghubungkan dua aras yang berbeda paling banyak satu. Paruh mana dari pernyataan itu yang bertahan pada graf **berarah**, dan paruh mana yang runtuh? Berikan sebuah graf berarah berisi tiga simpul yang di dalamnya sebuah busur menghubungkan aras 2 ke aras 0.

10.9 Jalankan DFS pada graf berarah §10.8.4 tetapi dengan gelung luarnya memperhatikan simpul dalam urutan z, y, x, w, v, u. Berikan pencap waktunya, akarnya, sisi pohonnya, penggolongan kedelapan busurnya, dan urutan selesainya. Bandingkan banyaknya pohonnya dengan **Tabel 10.9**.

10.10 Hanya dengan memakai pencap waktu **Gambar 10.25** — s 1/10, z 2/9, y 3/6, x 4/5, w 7/8, t 11/16, v 12/13, u 14/15 — putuskan, menurut Akibat 10.7 dan tanpa melihat gambarnya: (a) apakah x keturunan s? (b) apakah w keturunan y? (c) apakah v keturunan u, atau u keturunan v? (d) apakah z keturunan s? (e) apakah u keturunan t?

10.11 Masih hanya dengan memakai pencap waktu itu, golongkanlah masing-masing busur graf yang sama berikut ini: $s \to w, x \to z, w \to x, u \to t, v \to s$. Nyatakan aturan yang Anda pakai untuk masing-masingnya.

10.12 Berikan sebuah graf berarah berisi dua simpul, dan dua urutan bagi gelung luar DFS, sedemikian sehingga urutan yang satu menghasilkan hutan berisi satu pohon dan yang lain menghasilkan hutan berisi dua pohon. Apa yang dikatakan hal ini tentang klaim "DFS menghasilkan *sang* hutan mendalam dari G"?

10.13 Putuskan mana di antara berikut ini yang bipartit, dan untuk masing-masing yang tidak, perlihatkan sebuah siklus ganjil: (a) siklus C_7; (b) graf bipartit lengkap $K_{3,3}$; (c) sebuah pohon berisi 9 simpul; (d) graf **Gambar 10.1**; (e) graf **Gambar 10.9**.

10.14 Pengurutan topologis §10.9.6 menghasilkan *sebuah* urutan yang sah. Rancanglah sebuah algoritma yang menghasilkan urutan sah yang **terkecil secara leksikografis** (dengan memperlakukan simpulnya bernomor $0 \ldots |V| - 1$), nyatakan waktu jalannya, dan jelaskan dalam satu kalimat mengapa metode berbasis DFS §10.9.6 tidak dapat begitu saja disesuaikan.

10.15 Ubahlah BFS sehingga, selain $d[v]$, ia menghitung $cnt[v]$, yaitu **banyaknya lintasan terpendek yang berbeda** dari s ke v. Berikan perubahannya dalam tiga baris, berikan alasannya, lalu terapkan dengan tangan pada graf **Gambar 10.1** dengan sumber a.

10.16 **Diameter** sebuah pohon adalah panjang lintasan sederhana terpanjang di dalamnya. Buktikan bahwa prosedur berikut menghitungnya: jalankan BFS dari sembarang simpul, misalkan a adalah simpul terjauh yang ditemukan; jalankan BFS dari a, misalkan b adalah simpul terjauh dari a; jawabannya adalah $d[b]$. Lalu terapkan pada pohon dengan sisi 0–1, 1–2, 2–3, 3–4, 2–5, 5–6, 1–7, 7–8, 8–9.

10.17 Tulislah, dengan kata-kata lalu sebagai prosedur lima baris, sebuah uji $O(V + E)$ untuk "apakah graf tak berarah ini sebuah pohon?" yang hanya memakai satu BFS dan satu cacahan. Jelaskan mengapa kedua syarat yang Anda uji itu perlu.

10.18 Profitina Laundry membuka jalan baru antara Semolowaru dan Sukolilo. Hitunglah ulang BFS **Kode 10.11** dengan tangan, berikan cacahan lompatan baru setiap zonanya, dan katakan zona mana yang masuk ke janji dua lompatan yang sebelumnya tidak masuk. Apakah Dukuh Setro menjadi terjangkau?

10.14 Penyelesaian Soal Latihan

10.1 (a) Bacalah dari gambarnya:

$$\text{Adj}[a] = \{b, c, d\},\ \text{Adj}[b] = \{a, e\},\ \text{Adj}[c] = \{a, d, e\},\ \text{Adj}[d] = \{a, c, e\},\ \text{Adj}[e] = \{b, c, d\}.$$

(b) $\deg(a) = 3$, $\deg(b) = 2$, $\deg(c) = 3$, $\deg(d) = 3$, $\deg(e) = 3$ — kelima label pada **Gambar 10.4**.

(c) $3 + 2 + 3 + 3 + 3 = 14$ dan $2|E| = 2 \times 7 = 14$. Keduanya cocok.

(d) Ada $\binom{5}{2} = 10$ pasangan tak terurut dan 7 sisi, jadi 3 pasangan tidak bertetangga: $\{a, e\}$, $\{b, c\}$, $\{b, d\}$. Cocokkan masing-masingnya dengan E dan Anda tidak akan menemukannya.

10.2 *Bukti.* Belahlah V menjadi V_{genap} dan V_{ganjil} menurut paritas derajatnya. Maka

$$\sum_{v \in V_{\text{genap}}} \deg(v) + \sum_{v \in V_{\text{ganjil}}} \deg(v) = 2|E|,$$

yang genap. Jumlah pertamanya adalah jumlah bilangan genap, karena itu genap. Dengan mengurangkan, jumlah keduanya genap. Tetapi jumlah bilangan ganjil bernilai genap tepat ketika banyaknya suku genap. Jadi $|V_{\text{ganjil}}|$ genap. ■

(3,3,3,3,3): lima simpul berderajat ganjil, dan lima itu ganjil — mustahil menurut hasil yang baru saja dibuktikan. (Setara dengan itu, derajatnya berjumlah 15, yang tidak mungkin sama dengan $2|E|$.)

(4,4,4,4,4): derajatnya berjumlah $20 = 2|E|$, jadi $|E| = 10$; dan $10 = \binom{5}{2}$, sehingga grafnya haruslah K_5, yang di dalamnya setiap simpul memang berderajat 4. Jadi ya, dan grafnya tunggal: K_5.

10.3 Ada enam lintasan sederhana, dan **Tabel 10.14** mendaftarkan semuanya:

Tabel 10.14 Keenam lintasan sederhana dari b ke d pada graf **Gambar 10.1**, dalam urutan panjang yang menaik. Yang terpendek berpanjang 2; tidak ada yang berpanjang 1 sebab $(b, d) \notin E$.

Lintasan	Panjang	Sisi yang dipakai
$b\,a\,d$	2	$(a, b), (a, d)$
$b\,e\,d$	2	$(b, e), (d, e)$
$b\,a\,c\,d$	3	$(a, b), (a, c), (c, d)$
$b\,e\,c\,d$	3	$(b, e), (c, e), (c, d)$
$b\,a\,c\,e\,d$	4	$(a, b), (a, c), (c, e), (d, e)$
$b\,e\,c\,a\,d$	4	$(b, e), (c, e), (a, c), (a, d)$

$\delta(b, d) = 2$. Ia tidak mungkin 0, sebab $b \neq d$; ia tidak mungkin 1, sebab (b, d) adalah salah satu dari ketiga bukan-sisi yang ditemukan pada Soal 10.1(d); dan panjang 2 tercapai dua kali. Menurut argumen pemangkasan §10.4.5, lintasan terpendek selalu sederhana, jadi pencacahan ini sudah tuntas.

10.4 (a) Empat simpul; busur $0 \to 1$, $0 \to 2$, $1 \to 2$, $2 \to 0$, $2 \to 3$, dan sebuah gelung-diri di 3. (b) Adj[0] = [1,2], Adj[1] = [2], Adj[2] = [0,3], Adj[3] = [3]. (c) Dengan baris dan kolom dalam urutan 0,1,2,3:

$$M = \begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 1 \end{array}$$

Ia tidak setangkup, sebab grafnya berarah; dan $M[3][3] = 1$ merekam gelung-dirinya. (d) Derajat keluar 2,1,2,1 berjumlah 6; derajat masuk 1,1,2,2 berjumlah 6; dan $|E| = 6$. Kedua jumlahnya sama dengan $|E|$, sebagaimana dituntut §10.4.4. Simpul 0 berderajat masuk 1, jadi grafnya tidak punya sumber; simpul 3 berderajat keluar 1 (gelungnya sendiri), jadi tidak punya muara pula. (e) Daftar sisi: $2 \times 6 = 12$ kata. Daftar ketetanggaan: 4 kepala $+6$ entri (satu per busur, sebab berarah) $=10$ kata. Matriks ketetanggaan: $4^2 = 16$ kata. Matriksnya sudah menjadi yang terbesar bahkan pada empat simpul.

10.5 (a) **Daftar ketetanggaan.** $|E| \approx 3 \times 10^{10}$ melawan 9×10^{16} entri milik matriks, yang tidak dapat disimpan; dan BFS hanya menanyakan P3, yang dijawab daftarnya secara optimal. (b) **Matriks ketetanggaan.** $40^2 = 1\,600$ entri itu bukan apa-apa, grafnya padat dan berbobot, dan algoritma semua pasangan seperti Floyd-Warshall (Bab 15) *adalah* aritmetika matriks. (c) **Daftar**

sisi. Satu-satunya operasinya adalah P4, daftarkan semua sisi — lalu sebuah pengurutan. Inilah pola Kruskal (Bab 11). (d) **Matriks ketetanggaan.** $5\,000^2 = 2.5 \times 10^7$ entri kira-kira 25 MB pada satu bita per entri, yang terjangkau, dan ia mengubah 10^9 kueri P1 dari $O(\deg u)$ masing-masing menjadi $\Theta(1)$ masing-masing.

10.6 BFS dari w, dengan $\mathrm{Adj}[w] = [s, t, x]$, berjalan seperti yang ditunjukkan **Tabel 10.15**:

Tabel 10.15 Penelusuran melebar dari w pada graf §10.7.4. Bandingkan dengan **Tabel 10.7**: delapan langkah yang sama, tetapi pohon yang berbeda dan jarak yang berbeda, sebab sumbernya berpindah.

Langkah	Dikeluarkan	Ditemukan, beserta d	Antrean sesudahnya
—	—	$w: 0$	$(w: 0)$
1	w	$s: 1,\ t: 1,\ x: 1$	$(s: 1,\ t: 1,\ x: 1)$
2	s	$r: 2$	$(t: 1,\ x: 1,\ r: 2)$
3	t	$u: 2$	$(x: 1,\ r: 2,\ u: 2)$
4	x	$y: 2$	$(r: 2,\ u: 2,\ y: 2)$
5	r	$v: 3$	$(u: 2,\ y: 2,\ v: 3)$
6	u	—	$(y: 2,\ v: 3)$
7	y	—	$(v: 3)$
8	v	—	$(\,)$

Larik akhirnya: $d[w] = 0$; $d[s] = d[t] = d[x] = 1$; $d[r] = d[u] = d[y] = 2$; $d[v] = 3$. Pendahulunya: $\pi[s] = \pi[t] = \pi[x] = w$, $\pi[r] = s$, $\pi[u] = t$, $\pi[y] = x$, $\pi[v] = r$, $\pi[w] =$ NIL. Sisi pohonnya: $(w, s), (w, t), (w, x), (s, r), (t, u), (x, y), (r, v)$ — tujuh lagi, untuk delapan simpul. Perhatikan bahwa v kini berjarak 3 alih-alih 2: jarak adalah sifat *pasangannya*, bukan sifat grafnya semata.

10.7 *Mengapa pohonnya terpaksa di sini.* Simpul v dengan $d[v] = k$ hanya dapat ditemukan dari tetangga yang berjarak $k - 1$ (Teorema 10.4 ditambah Akibat 10.5), jadi pohonnya tunggal tepat ketika setiap simpul bukan sumber punya tepat **satu** tetangga yang selangkah lebih dekat. Periksalah ketujuhnya, sebagaimana dilakukan **Tabel 10.16**:

Tabel 10.16 Mengapa pohon melebar §10.7.4 tidak bergantung pada urutan ketetanggaannya: setiap simpul punya tepat satu tetangga yang selangkah lebih dekat ke s, sehingga ia tidak punya pilihan induk.

v	$d[v]$	Tetangga, beserta d-nya	Pada aras $d[v]-1$
r	1	$s: 0, v: 2$	hanya s
w	1	$s: 0, t: 2, x: 2$	hanya s
t	2	$u: 3, w: 1, x: 2$	hanya w
x	2	$t: 2, w: 1, y: 3$	hanya w
v	2	$r: 1$	hanya r
u	3	$t: 2, y: 3$	hanya t
y	3	$u: 3, x: 2$	hanya x

Urutan antreannya tetap berubah — dengan $\mathrm{Adj}[s] = [r, w]$ antrean sesudah langkah 1 adalah (r, w) alih-alih (w, r) — tetapi tidak ada simpul yang pernah punya pilihan induk, jadi π-nya sama saja.

Sebuah graf yang pohonnya tidak terpaksa. Ambillah siklus-4, C_4, pada $\{0,1,2,3\}$ dengan sisi 0–1, 1–2, 2–3, 3–0, dan sumber 0. Maka $d = (0,1,2,1)$ dan simpul 2 punya **dua** tetangga di aras 1, yaitu 1 dan 3. Dengan $\mathrm{Adj}[0] = [1,3]$ penelusurannya mencapai 2 dari 1 dan sisi pohonnya adalah

$(0,1), (0,3), (1,2)$; dengan $\text{Adj}[0] = [3,1]$ ia mencapai 2 dari 3 dan sisi pohonnya adalah $(0,1), (0,3), (3,2)$. Dua pohon yang berbeda, jarak yang sama.

10.8 Paruh yang bertahan adalah $\delta(s,v) \leq \delta(s,u) + 1$ untuk setiap busur (u,v) — Lema 10.1, yang pembuktiannya tidak pernah memakai kesetangkupan: tempelkan busurnya pada lintasan terpendek ke u. Yang runtuh adalah ketaksamaan sebaliknya $\delta(s,u) \leq \delta(s,v) + 1$, sebab Anda tidak dapat menyusuri busur secara mundur. Jadi pada graf berarah sebuah busur boleh jatuh dari aras mana pun ke aras mana pun yang lebih rendah, tetapi tidak pernah boleh naik lebih dari satu.

Contoh: $V = \{s,p,q\}$ dengan busur $s \to p$, $p \to q$, $q \to s$. BFS dari s memberikan $d[s] = 0$, $d[p] = 1$, $d[q] = 2$. Busur $q \to s$ menghubungkan aras 2 ke aras 0 — jatuh dua. Keberadaannya konsisten, sebab Lema 10.1 yang diterapkan pada (q,s) hanya menuntut $\delta(s,s) \leq \delta(s,q) + 1$, yaitu $0 \leq 3$.

10.9 Dengan gelung luarnya mencoba z, y, x, w, v, u, pencap waktunya keluar seperti pada **Tabel 10.17**:

Tabel 10.17 Pencap waktu ketika gelung luar DFS memindai z, y, x, w, v, u. Bandingkan **Tabel 10.9**, yang memakai urutan u, v, w, x, y, z: graf yang sama, algoritma yang sama, hutan yang sama sekali berbeda.

v	u	v	w	x	y	z
d/f	11/12	5/6	9/10	4/7	3/8	1/2

Akarnya: z, y, w, u — **empat** pohon, padahal urutan aslinya memberikan dua. Sisi pohonnya: hanya (y,x) dan (x,v); hutannya punya $6 - 4 = 2$ sisi, sebagaimana dituntut aturan §10.8.5.

Penggolongan kedelapan busurnya, dalam urutan pemeriksaannya: $z \to z$ balik; $y \to x$ pohon; $x \to v$ pohon; $v \to y$ balik; $w \to y$ silang; $w \to z$ silang; $u \to v$ silang; $u \to x$ silang.

Urutan selesainya: z, v, x, y, w, u.

Dua hal pantas diperhatikan. Busur $w \to z$ tadinya sisi *pohon* dan kini menjadi sisi *silang*, sebab z sudah ditelusuri lebih dulu sebagai akar. Dan sisi baliknya berganti identitas — $x \to v$ tadinya sisi balik dan kini menjadi sisi pohon, sedangkan $v \to y$ kini menjadi sisi baliknya — tetapi sisi balik tetap ada, sebagaimana dijamin Teorema 10.10 memang harus, sebab grafnya memang memuat siklus.

10.10 Menurut Akibat 10.7, v adalah keturunan sejati u tepat ketika $d[u] < d[v] < f[v] < f[u]$. (a) x keturunan s: $1 < 4 < 5 < 10$. **Ya.** (b) w keturunan y: memerlukan $3 < 7 < 8 < 6$, dan $8 < 6$ itu salah. Selang $[3,6]$ dan $[7,8]$ saling lepas, jadi menurut teorema tanda kurung tidak ada yang merupakan keturunan yang lain. **Tidak.** (c) $[14,15]$ dan $[12,13]$ saling lepas, jadi **tidak ada** yang merupakan keturunan yang lain; keduanya bersaudara, sama-sama anak t, yang selangnya $[11,16]$ memuat keduanya. (d) z keturunan s: $1 < 2 < 9 < 10$. **Ya.** (e) u keturunan t: $11 < 14 < 15 < 16$. **Ya.**

10.11 Aturannya, dari **Tabel 10.11**: busur (u,v) adalah sisi *balik* ketika $d[v] < d[u] < f[u] < f[v]$ (v adalah leluhur); sisi *maju* ketika $d[u] < d[v] < f[v] < f[u]$ dan ia bukan sisi pohonnya sendiri; sisi *silang* ketika $f[v] < d[u]$ (seluruh selang v sudah selesai sebelum selang u dimulai).

- $s \to w$: $d[s] = 1 < d[w] = 7 < f[w] = 8 < f[s] = 10$. Bersarang, w keturunan s, dan induk w adalah z bukan s — jadi **maju**.
- $x \to z$: $d[z] = 2 < d[x] = 4 < f[x] = 5 < f[z] = 9$. z adalah leluhur x — **balik**.
- $w \to x$: $f[x] = 5 < d[w] = 7$. Selang x menutup sebelum selang w membuka — **silang**.
- $u \to t$: $d[t] = 11 < d[u] = 14 < f[u] = 15 < f[t] = 16$. t adalah leluhur u — **balik**.

- $v \to s$: $f[s] = 10 < d[v] = 12$. Saling lepas, s selesai lebih dulu — **silang**. (Dan memang s dan v berada di pohon yang berbeda.)

Kelimanya sepakat dengan label pada **Gambar 10.25** dan dengan penggambaran ulang pada **Gambar 10.27**.

10.12 Ambillah $V = \{0,1\}$ dengan satu busur $0 \to 1$. Urutan luar 0,1: DFS-Visit(0) menemukan 1 lewat sisi pohon, jadi hutannya berisi **satu** pohon dengan sisi (0,1). Urutan luar 1,0: DFS-Visit(1) mendapati Adj[1] kosong lalu selesai; kemudian DFS-Visit(0) memeriksa $0 \to 1$, mendapati 1 hitam dengan $d[0] = 3 > d[1] = 1$, dan menyebutnya sisi silang. Hutannya berisi **dua** pohon bersimpul tunggal.

Pelajarannya adalah bahwa tidak ada yang namanya *sang* hutan mendalam dari G; yang ada hanyalah hutan mendalam dari G *di bawah sebuah urutan simpul tertentu dan urutan ketetanggaan tertentu*. Setiap teorema di §10.8 dinyatakan tentang "sebuah penelusuran mendalam", bukan "sang" penelusuran, justru karena alasan ini. Yang *memang* invarian adalah keberadaan sisi balik setiap kali G punya siklus (Teorema 10.10) — itu berlaku untuk setiap urutan, dan itulah sebabnya deteksi siklus adalah jawaban yang andal sementara hutannya tidak.

10.13 (a) C_7 adalah siklus ganjil, jadi **tidak bipartit**; siklus ganjilnya adalah seluruh grafnya, berpanjang 7. (b) $K_{3,3}$ **bipartit** menurut konstruksinya: kedua kubunya adalah kedua tripelnya, dan setiap sisi menyeberang. (c) Setiap pohon **bipartit**. Ia tak bersiklus, jadi ia sama sekali tidak punya siklus, khususnya tidak punya yang ganjil; secara konkret, warnailah menurut paritas kedalaman dari sembarang akar. (d) Graf **Gambar 10.1 tidak bipartit**. Siklus ganjilnya adalah segitiga $a\ c\ d\ a$ pada **Gambar 10.7**. (**Kode 10.8** melaporkan kegagalannya pada sisi c–d.) (e) Graf **Gambar 10.9 tidak bipartit**, sebab komponen pertamanya adalah segitiga. Perhatikan bahwa kebipartitan harus berlaku pada *setiap* komponen: satu komponen yang buruk merusak seluruh grafnya.

10.14 Pakailah **algoritma Kahn** [3] dengan antrean berprioritas. Hitunglah setiap derajat masuknya dalam $O(V + E)$. Masukkan setiap simpul berderajat masuk 0 ke dalam *min-heap*. Berulang-ulang, ambillah simpul terkecil yang tersedia, tambahkan ia ke keluarannya, lalu kurangi derajat masuk masing-masing tetangga keluarnya, sambil mendorong masuk simpul mana pun yang mencapai 0. Jika keluarannya berisi kurang dari $|V|$ simpul, grafnya bukan DAG.

Biaya. Tiap simpul didorong dan diambil sekali, masing-masing $O(\log V)$, dan tiap busur direlaksasi sekali dengan $O(1)$ ditambah paling banyak satu dorongan: $O((V + E)\log V)$.

Kebenaran pilihan rakusnya. Pada tiap langkah, simpul berderajat masuk 0 di antara yang belum dikeluarkan persis merupakan simpul yang boleh datang berikutnya secara sah. Memilih yang terkecil di antaranya itu aman: urutan sah mana pun yang menempatkan simpul y yang lebih besar di posisi berikutnya dapat ditulis ulang dengan memindahkan simpul terkecil x yang tersedia ke depan sisa barisannya — tidak ada apa pun sebelum x di sisa itu yang dituntut mendahuluinya, sebab derajat masuk x yang tersisa adalah 0 — dan hasilnya lebih kecil secara leksikografis. Jadi tidak ada urutan yang lebih kecil secara leksikografis.

Mengapa DFS tidak dapat begitu saja disesuaikan. Metode DFS §10.9.6 menghasilkan urutannya **secara terbalik**, menurut waktu selesai, jadi pilihan rakus yang dibuat di depan DFS-nya adalah pilihan tentang *ujung* keluarannya. Memilih simpul terkecil untuk selesai lebih dulu tidak membuat daftar yang selesai itu terkecil ketika dibalik. Tidak ada aturan lokal di ujung DFS-nya yang bersesuaian dengan "terkecil dulu" di ujung keluarannya.

10.15 Tiga baris. Simpanlah sebuah larik cnt dengan $cnt[s] = 1$ dan $cnt[v] = 0$ selain itu, lalu di dalam gelung atas $v \in \text{Adj}[u]$ gantilah badannya dengan kedua baris **Pseudokode 10.4**:

Pseudokode 10.4 Gelung dalam BFS yang diperluas untuk mencacah lintasan terpendek: $cnt[v]$ menghimpun cacah milik semua tetangga yang satu aras lebih dekat ke sumber.

```
for each v in Adj[u]
    if d[v] == UNSET
        d[v] = d[u] + 1;  cnt[v] = cnt[u];  ENQUEUE(Q, v)   // lintasan terpendek pertama
ditemukan
    else if d[v] == d[u] + 1
        cnt[v] = cnt[v] + cnt[u]                            // satu keluarga lintasan
terpendek lagi
```

Alasannya. Setiap lintasan terpendek dari s ke v berpanjang $d[v]$ sehingga ia tiba di v dari suatu tetangga u dengan $d[u] = d[v] - 1$; sebaliknya, mengawali lintasan terpendek mana pun ke u semacam itu dengan sisi (u, v) memberikan lintasan terpendek ke v. Jadi $cnt[v] = \sum\{cnt[u] : u \in \text{Adj}[v],\ d[u] = d[v] - 1\}$. BFS-nya menyinggahi setiap u semacam itu sebelum v dikeluarkan dari antrean — sebab simpul dikeluarkan dalam urutan d yang tidak menurun — dan masing-masingnya menyumbang bagiannya tepat sekali, entah ketika v pertama kali ditemukan atau di dalam `else if`-nya. Tiap sisi tetap diperiksa sekali, jadi ongkosnya tetap $O(V + E)$.

Dengan tangan, pada **Gambar 10.1** *dengan sumber* a. $d[a] = 0$, $cnt[a] = 1$. Dari a kita menemukan b, c, d, masing-masing berjarak 1 dan masing-masing dengan $cnt = 1$. Lalu keluarkan b: tetangganya e putih, jadi $d[e] = 2$ dan $cnt[e] = cnt[b] = 1$. Keluarkan c: e kini punya $d[e] = 2 = d[c] + 1$, jadi $cnt[e] += cnt[c]$, sehingga menjadi 2. Keluarkan d: begitu pula $cnt[e] += cnt[d]$, sehingga menjadi 3. Jawaban akhirnya:

$$cnt[a] = 1, \quad cnt[b] = cnt[c] = cnt[d] = 1, \quad cnt[e] = 3.$$

Dan memang ketiga lintasan terpendek dari a ke e adalah $a\ b\ e$, $a\ c\ e$, dan $a\ d\ e$.

10.16 *Bukti.* Misalkan pohonnya T dan misalkan $P = p \cdots q$ adalah lintasan terpanjang, berpanjang D. Jalankan BFS dari sembarang simpul r dan misalkan a adalah simpul terjauh dari r. Kita klaim bahwa a adalah ujung *suatu* lintasan terpanjang. Dua fakta tentang pohon mengerjakan seluruhnya: di antara dua simpul mana pun ada tepat satu lintasan sederhana, sehingga δ diukur sepanjang lintasan itu; dan lintasan yang meninggalkan P di sebuah simpul tidak dapat kembali ke P, sebab hal itu akan menutup sebuah siklus.

Misalkan w adalah simpul P yang terdekat ke r. Setiap lintasan dari r ke sebuah simpul P pertama kali mencapai P di w. Ada dua kasus, menurut apakah lintasan dari r ke a menyentuh P.

Kasus 1: lintasan dari r ke a menyentuh P. Lintasan itu mencapai P di w, berjalan di sepanjang P untuk sementara, lalu meninggalkan P untuk selamanya di suatu simpul z (mungkin $z = w$, atau $z = a$). Karena setangkup, kita boleh menamai ujung-ujung P sehingga z terletak di antara w dan q. Maka $\delta(r, a) = \delta(r, z) + \delta(z, a)$ dan, karena lintasan dari r ke q juga melewati z, $\delta(r, q) = \delta(r, z) + \delta(z, q)$. Karena a adalah simpul terjauh dari r, $\delta(r, a) \geq \delta(r, q)$, dan mengurangkan $\delta(r, z)$ memberi $\delta(z, a) \geq \delta(z, q)$. Sekarang berjalanlah dari p di sepanjang P ke z lalu keluar ke a: inilah lintasan sederhana berpanjang $\delta(p, z) + \delta(z, a) \geq \delta(p, z) + \delta(z, q) = D$. Ia tidak mungkin lebih panjang daripada D, jadi panjangnya tepat D, dan a adalah ujung sebuah lintasan terpanjang.

Kasus 2: lintasan dari r ke a tidak menyentuh P. Misalkan y adalah simpul terakhir yang dimiliki bersama oleh lintasan dari r ke a dan lintasan dari r ke w; maka $y \neq w$, sehingga $\delta(y, w) \geq 1$.

Seperti tadi, $\delta(r,a) \geq \delta(r,q)$ menjadi $\delta(y,a) \geq \delta(y,w) + \delta(w,q)$. Tetapi kemudian lintasan sederhana $p \cdots w \cdots y \cdots a$ berpanjang $\delta(p,w) + \delta(w,y) + \delta(y,a) \geq \delta(p,w) + 2\,\delta(w,y) + \delta(w,q) = D + 2\,\delta(w,y) > D$, yang mustahil sebab D adalah yang terpanjang. Jadi Kasus 2 tidak pernah terjadi.

Karena itu BFS kedua, dari a, menemukan sebuah simpul b pada jarak tepat D, dan D adalah diameternya. Dua kali jalan BFS, $O(V+E) = O(V)$ pada sebuah pohon. ■

Diterapkan. Pohonnya punya sisi 0–1, 1–2, 2–3, 3–4, 2–5, 5–6, 1–7, 7–8, 8–9: sepuluh simpul, sembilan sisi, jadi ia sebuah pohon. BFS dari 0 memberikan jarak 0,1,2,3,4,3,4,2,3,4, jadi simpul terjauhnya adalah $a = 4$, pada jarak 4. BFS dari 4 memberikan 4,3,2,1,0,3,4,4,5,6, jadi yang terjauh adalah $b = 9$ pada jarak 6. **Diameternya adalah** 6, diwujudkan lintasan 4 3 2 1 7 8 9. Perhatikan bahwa jawaban BFS pertamanya, 4, *bukan* diameternya — jalan keduanya sangat penting.

10.17 *Dengan kata-kata.* Graf tak berarah adalah pohon tepat ketika ia **terhubung** dan punya tepat $|V| - 1$ **sisi** (§10.4.11: dua mana pun di antara terhubung, tak bersiklus, dan $|E| = |V| - 1$ mengakibatkan yang ketiga). Jadi: cacahlah sisinya; jalankan satu BFS dari simpul 0; periksalah bahwa ia mencapai seluruh $|V|$ simpulnya. **Pseudokode 10.5** adalah prosedurnya.

Pseudokode 10.5 Menguji apakah sebuah graf tak berarah dengan $n \geq 1$ simpul adalah pohon: satu pencacahan sisi dan satu penelusuran melebar.

```
IS-TREE(adj, n)
1  if COUNT-EDGES(adj) != n - 1            // banyaknya sisi salah
2      return FALSE
3  d = BFS(adj, 0)                         // satu penelusuran, O(V+E)
4  for v = 0 to n - 1
5      if d[v] == UNSET                    // tidak terhubung
6          return FALSE
7  return TRUE
```

Mengapa kedua syaratnya perlu. Tidak ada yang mengakibatkan yang lain. Segitiga ditambah satu simpul terasing punya $|V| = 4$ dan $|E| = 3 = |V| - 1$, tetapi ia **tidak terhubung** — jadi cacahan sisinya saja tidak cukup. Segitiga saja punya $|V| = 3$, terhubung, tetapi punya $|E| = 3 > |V| - 1$ dan memuat siklus — jadi keterhubungannya saja tidak cukup. Bersama-sama keduanya memaksakan ketakbersiklusan: graf terhubung yang punya siklus punya sekurang-kurangnya $|V|$ sisi, sebab menghapus satu sisi siklusnya membuatnya tetap terhubung dan graf terhubung mana pun memerlukan sekurang-kurangnya $|V| - 1$ sisi.

Satu kasus khusus yang perlu dijaga: untuk $n = 0$ uji cacahan sisinya terbaca $0 \neq -1$ dan mengembalikan false, dan BFS dari simpul 0 akan berada di luar jangkauan. Tanganilah graf kosongnya sebelum apa pun yang lain, atau tuntutlah $n \geq 1$.

10.18 Jalan baru Semolowaru–Sukolilo menjadikan Sukolilo, yang sudah berjarak 1, sebagai tetangga Semolowaru; **Tabel 10.18** memberikan keadaan sebelum dan sesudahnya.

Tabel 10.18 Dampak satu jalan baru terhadap cacahan lompatan Profitina Laundry. Satu zona bergabung ke janji dua lompatan dan satu zona meninggalkan ekor empat lompatan; zona yang sama sekali tidak punya jalan tidak terpengaruh, sebab menambahkan sisi di tempat lain tidak dapat menciptakan lintasan ke komponen yang berbeda.

Zona	Lompatan sebelumnya	Lompatan sesudahnya	Perubahan
Keputih	0	0	—
Gebang Putih	1	1	—
Sukolilo	1	1	—
Klampis Ngasem	2	2	—
Medokan Semampir	2	2	—
Mulyorejo	2	2	—
Semolowaru	3	2	**membaik**, kini lewat Sukolilo
Manyar Sabrangan	3	3	—
Kalijudan	3	3	—
Nginden Jangkungan	4	3	membaik, kini lewat Semolowaru
Dukuh Setro	∞	∞	tetap tak terjangkau

Semolowaru adalah satu-satunya zona yang masuk ke janji dua lompatan, yang menjadikannya dari enam zona menjadi tujuh. Nginden Jangkungan membaik dari empat lompatan menjadi tiga tetapi tetap tidak masuk janjinya.

Dukuh Setro tidak menjadi terjangkau. Ia sama sekali tidak punya sisi yang bersisian, jadi ia adalah komponen terhubung tersendiri (§10.4.8), dan tidak ada sisi yang ditambahkan di antara dua zona *lain* yang dapat menciptakan lintasan ke dalamnya. Menurut fakta 3 di §10.4.8, penelusuran yang dimulai di komponen Keputih tidak pernah dapat meninggalkannya. Hanya jalan yang **menyentuh Dukuh Setro** yang akan mengubah hal ini — dan itulah persis temuan yang diperlukan binatunya: persoalannya bukan penjadwalan, melainkan bahwa zona itu berada di komponen yang berbeda.

Dari mana gagasan-gagasan ini berasal. Skema tiga warna, pencap waktunya, teorema tanda kurung dan teorema lintasan putih, penggolongan sisinya, serta pengurutan topologis berbasis DFS semuanya disajikan di sini dalam bentuk yang diberikan Cormen, Leiserson, Rivest, dan Stein [1]; kedua contoh penelusuran yang dikerjakan itu adalah milik mereka. Gambaran labirin-dengan-benang untuk BFS, perluasan kaleng-cat untuk DFS, dan gambar daftar sisi serta daftar ketetanggaan di §10.5 berasal dari Goodrich dan Tamassia [2]. Penelusuran mendalam sebagai teknik umum untuk algoritma graf berwaktu linear berasal dari Tarjan [4], dan algoritma derajat masuk untuk pengurutan topologis yang disebut di Soal 10.14 adalah milik Kahn [3]. Makalah Dijkstra [5] adalah asal-usul algoritma lintasan terpendek berbobot yang akan disajikan Bab 13 sebagai BFS dengan antrean berprioritas; Kleinberg dan Tardos [6] memberikan paparan yang luar biasa jernih tentang BFS dan DFS sebagai dua disiplin bagi perbatasan yang sama; Sedgewick dan Wayne [7] adalah sumber terbaik untuk rincian implementasi dan untuk banyak penerapan kecil kedua penelusuran itu; Halim, Halim, dan Effendy [8] membahas varian pemrograman kompetitifnya termasuk 0-1 BFS yang disebut di §10.9.4; dan Knuth [9] memberikan sejarah pengurutan topologis dan teknik senarai berantai yang dipakai di §10.9.6.

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] M. T. Goodrich, R. Tamassia, and D. M. Mount, *Data Structures and Algorithms in C++*, 2nd ed. Hoboken, NJ, USA: Wiley, 2011.

[3] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962.

[4] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, Jun. 1972.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[6] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[7] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[8] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, 4th ed. Raleigh, NC, USA: Lulu Press, 2020.

[9] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

Bab 11
Pohon Merentang Minimum, DAG, dan Pengurutan Topologis

Di akhir bab ini Anda akan mampu memandang sebuah graf berarah dan memutuskan, dalam waktu linear, apakah ia memuat sebuah siklus — sekaligus membuktikan bahwa pengujian Anda itu benar. Anda akan mampu mengambil sehimpunan tugas yang saling mendahului dan menghasilkan sebuah urutan yang sah untuk mengerjakannya, dengan salah satu dari dua cara baku, serta menjelaskan mengapa sebuah graf biasanya punya banyak urutan semacam itu, bukan satu. Anda akan mampu menyatakan apa itu pohon merentang, membuktikan bahwa setiap pohon merentang sebuah graf dengan $|V|$ simpul memiliki tepat $|V| - 1$ sisi, dan menerangkan mengapa pohon merentang yang *termurah* adalah pertanyaan yang layak diajukan. Anda akan mampu menyatakan dan membuktikan sifat potongan — satu-satunya teorema yang menjadi tumpuan setiap algoritma pohon merentang minimum yang pernah ada — lalu memakainya untuk membuktikan dua di antaranya benar: Prim–Jarník, yang menumbuhkan satu pohon dari sebuah akar, dan Kruskal, yang melebur sebuah hutan dari sisi termurah ke atas. Anda akan mampu menelusuri keduanya dengan tangan pada graf contoh bab ini, dengan mencetak antrean prioritas atau partisi himpunan saling lepasnya pada setiap langkah. Anda akan mampu mengimplementasikan hutan himpunan saling lepas dengan gabung menurut peringkat dan pemampatan lintasan, serta menjelaskan mengapa biaya tiap operasinya begitu dekat dengan konstan sehingga dalam praktik boleh Anda anggap konstan. Dan ketika seseorang bertanya algoritma mana dari keduanya yang sebaiknya dipakai, Anda akan mampu menjawabnya dengan alasan, bukan dengan selera.

11.1 Dari Mana Kita Melanjutkan

Rujukan baku untuk bahan ini adalah Bab 21 Cormen, Leiserson, Rivest, dan Stein [1], dan sejumlah gambar dalam bab ini mengikuti gambar mereka dengan dekat. Bab ini berdiri di atas dua bab terdahulu, dan ada baiknya disebutkan dengan tepat apa yang diambilnya dari masing-masing.

Dari **Bab 10** ia mengambil grafnya sendiri. Sebuah *graf* $G = (V, E)$ adalah sebuah pasangan: sehimpunan V berisi **simpul** (*vertices*), dan sehimpunan $E \subseteq V \times V$ berisi **sisi** (*edges*), yang tiap sisinya merupakan sepasang simpul. Jika pasangannya terurut — jika (u, v) dan (v, u) berarti dua hal yang berbeda — grafnya disebut **berarah**; jika tidak, ia **tak berarah**. Kita menuliskan $|V|$ untuk banyaknya simpul dan $|E|$ untuk banyaknya sisi, dan, seperti di Bab 10, kita melepas garis tegaknya di dalam notasi asimtotik: $O(V + E)$ selalu berarti $O(|V| + |E|)$. Kita menyatakan sebuah graf melalui **senarai ketetanggaan**-nya: $Adj[u]$ adalah senarai simpul yang ditunjuk oleh u (kasus berarah) atau yang bersambung dengannya (kasus tak berarah). Bab 10 memberi kita dua cara menelusuri graf, pencarian melebar-dahulu dan pencarian mendalam-dahulu, masing-masing dalam waktu $O(V + E)$, dan dari pencarian mendalam-dahulu ia memberi kita empat catatan pembukuan yang dipakai terus-menerus oleh bab ini.

> **DEFINISI (diulang dari Bab 10).** Sebuah pencarian mendalam-dahulu memelihara satu **jam** yang mulai dari 0 dan berdetak maju satu setiap kali ada sesuatu yang terjadi. Untuk tiap

simpul u ia mencatat sebuah **waktu penemuan** $d[u]$, yaitu nilai jam pada saat u pertama kali dicapai, dan sebuah **waktu penyelesaian** $f[u]$, yaitu nilai jam pada saat pencariannya selesai memeriksa setiap sisi yang keluar dari u dan hendak mundur darinya. Selalu berlaku $d[u] < f[u]$. Tiap simpul berwarna **putih** sebelum waktu $d[u]$ (belum ditemukan), **kelabu** antara $d[u]$ dan $f[u]$ (sudah ditemukan, tetapi belum selesai diurus), dan **hitam** sesudah $f[u]$ (selesai). **Pendahulu** $\pi[u]$ adalah simpul tempat u ditemukan, atau NIL bila u memulai sebuah pohon baru dalam hutan mendalam-dahulunya.

Bab 10 juga menggolongkan sisi-sisi sebuah graf berarah menurut apa yang ditemukan pencariannya di ujung seberang masing-masing:

Tabel 11.1 Keempat jenis sisi yang dapat dilalui sebuah pencarian mendalam-dahulu pada graf berarah, digolongkan menurut warna ujung seberangnya pada saat sisi itu diperiksa. Seluruh isi §11.2 berputar pada baris kedua.

Nama	Warna v saat (u,v) dijelajahi	Artinya
sisi pohon	putih	v ditemukan lewat sisi ini; $\pi[v] = u$
sisi balik	kelabu	v adalah leluhur u di dalam hutan mendalam-dahulunya
sisi maju	hitam, dan $d[u] < d[v]$	v adalah keturunan u, yang dicapai lebih dulu lewat jalan lain
sisi silang	hitam, dan $d[v] < d[u]$	selebihnya: antarpohon, atau antarsubpohon

Dari **Bab 9** bab ini mengambil *disiplin* perancangan *greedy*. Sebuah **algoritma *greedy*** (rakus) membangun penyelesaiannya sepotong demi sepotong, dan setiap kali ia menghadapi pilihan, ia mengambil potongan yang tampak paling baik pada saat itu, tanpa pernah meninjau ulang. Itu terdengar sebagai cara yang gegabah untuk menyelesaikan persoalan optimasi, dan Bab 9 menerangkan dengan tepat apa yang harus dibuktikan lebih dahulu sebelum cara itu diizinkan.

DEFINISI (diulang dari Bab 9). Sebuah persoalan memiliki **sifat pilihan *greedy*** bila selalu ada penyelesaian optimal global yang memuat pilihan terbaik secara lokal itu. Ia memiliki **substruktur optimal** bila, sesudah pilihan itu diambil, penyelesaian optimal bagi seluruh persoalannya memuat penyelesaian optimal bagi persoalan lebih kecil yang tersisa.

11.1.1 Yang masih kurang

Bab 10 menelusuri graf tetapi tidak pernah *mengukurnya*. Setiap sisi bernilai sama: satu langkah. Itu tepat benar ketika pertanyaannya "bisakah saya sampai ke sana?" atau "berapa sisi paling sedikit jaraknya?", dan tepat salah ketika pertanyaannya "berapa ongkosnya?". Sebuah jalan bisa panjang atau pendek; sebuah kabel bisa murah atau mahal; sebuah rit kurir menghabiskan lebih banyak bensin pada satu ruas daripada ruas lain. Begitu kita menempelkan sebuah bilangan pada tiap sisi, kita punya **graf berbobot**, dan sekeluarga pertanyaan baru pun terbuka. Bab ini menjawab salah satunya — *berapa himpunan sisi termurah yang menjaga semuanya tetap terhubung?* — secara tuntas, dari asas yang paling dasar, dengan dua algoritma beserta bukti masing-masing. Bab 13 akan menjawab yang kedua, *berapa rute termurah dari sini ke sana?*, dan Bab 15 yang ketiga, *berapa rute termurah antara setiap pasang tempat?*

Sebelum semua itu, ada satu bagian Bab 10 yang belum selesai dan perlu dibereskan. Pencarian mendalam-dahulu meninggalkan kita memegang sehimpunan waktu penyelesaian tanpa alasan yang jelas untuk memedulikannya. Bab ini dibuka dengan menunjukkan bahwa waktu penyelesaian itu, sendirian saja, menjawab sebuah pertanyaan yang sama sekali tidak kelihatan seperti pencarian.

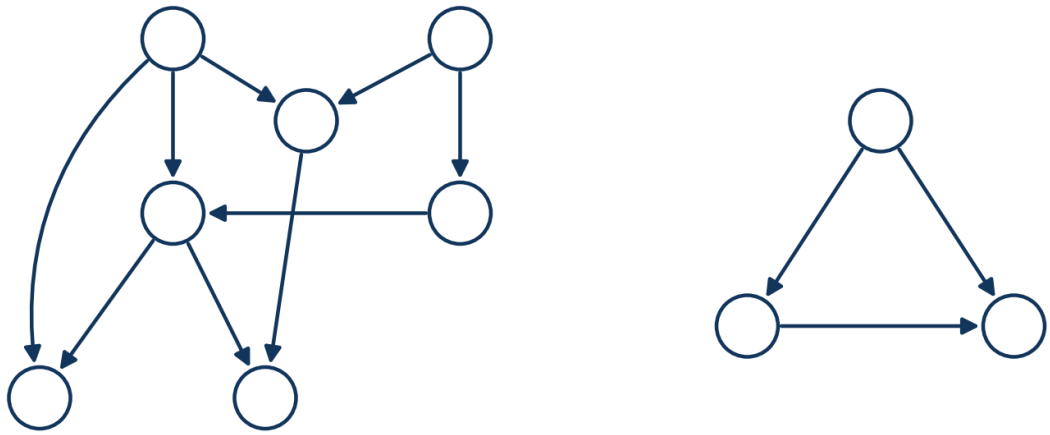
11.2 Graf Berarah Asiklik

11.2.1 Siklus, dan apa artinya tidak punya siklus

Mulailah dari kata *siklus*. Di dalam sebuah graf berarah, sebuah **lintasan** dari v_0 ke v_k adalah barisan simpul $v_0, v_1, \dots, v_k$ sedemikian sehingga tiap pasangan yang berurutan merupakan sebuah sisi: $(v_{i-1}, v_i) \in E$ untuk $i = 1,2, \dots, k$. Bilangan k adalah **panjang** lintasan itu, dihitung dalam sisi, bukan simpul. Sebuah lintasan disebut **siklus** bila panjangnya sekurang-kurangnya satu dan ia kembali ke tempatnya berangkat, yaitu $v_0 = v_k$.

> **DEFINISI.** Sebuah **graf berarah asiklik** (*directed acyclic graph*), yang di mana-mana disingkat **DAG**, adalah graf berarah yang tidak memuat siklus. Setara dengan itu: tidak ada simpul yang dari sana Anda dapat berangkat menyusuri anak panahnya lalu kembali lagi kepadanya.

Gambar 11.1 memperlihatkan dua graf, dan keduanya DAG. Ikuti anak panah graf kiri dari simpul mana pun dan anak panahnya akan habis; Anda tidak pernah kembali. Graf tiga simpul di kanan lebih mendidik, sebab ia *tampak* seperti siklus: ketiga sisinya membentuk segitiga. Tetapi bacalah arahnya. Simpul atas menunjuk ke kedua simpul lainnya, dan simpul kiri bawah menunjuk ke simpul kanan bawah. Berangkatlah dari mana saja, ikuti anak panahnya, dan Anda tidak akan pernah kembali — simpul kanan bawah sama sekali tidak punya jalan keluar. Segitiga sisi baru merupakan siklus berarah bila anak panahnya berputar mengelilinginya ke satu arah. Balikkan satu sisi saja — jadikan simpul kanan bawah menunjuk kembali ke simpul atas — maka anak panahnya berjalan dari atas ke kiri bawah, ke kanan bawah, lalu ke atas lagi: sebuah siklus berarah sepanjang tiga, dan grafnya bukan DAG lagi. Satu siklus semacam itu saja sudah cukup untuk mendiskualifikasi sebuah graf, sebesar dan setertib apa pun sisanya.



Gambar 11.1 Dua graf berarah asiklik. **Kiri:** ikuti anak panahnya dari mana saja dan anak panahnya selalu habis. **Kanan:** tiga sisi yang membentuk segitiga tetapi tetap asiklik, sebab anak panahnya tidak berputar mengelilinginya ke satu arah; membalik sisi dari simpul atas ke simpul kanan bawah akan menutup sebuah siklus berarah.

> **DENGAN BAHASA SEHARI-HARI.** Sebuah DAG adalah sehimpunan hal yang harus terjadi dalam urutan tertentu, tanpa bisa dibantah. “Anda harus memasang pondasi sebelum dinding, dan dinding sebelum atap” adalah sebuah DAG. “Anda harus bertanya kepada manajer sebelum bertanya kepada kasir, dan kepada kasir sebelum kepada manajer” bukan DAG — itu perintah yang mustahil dilaksanakan, dan alasan kemustahilannya persis karena ia memuat sebuah siklus.

11.2.2 Empat DAG sehari-hari

Definisinya kering. Alasan DAG itu penting adalah bahwa bentuk graf yang satu ini muncul di mana pun ada satu hal yang harus mendahului hal lain. Dalam satu baris: sebuah DAG sering dipakai untuk menunjukkan pendahuluan antarperistiwa — peristiwa a harus terjadi sebelum peristiwa b. Empat contoh, tidak satu pun yang dibuat-buat.

Kebergantungan pembangunan program. Sebuah program dibangun dari berkas sumber. `main.o` memerlukan `main.cpp` beserta setiap *header* yang disertakannya; `app` memerlukan `main.o` dan `graph.o`; `graph.o` memerlukan `graph.cpp` dan `graph.h`. Gambarlah anak panah dari tiap berkas ke tiap berkas yang bergantung kepadanya dan Anda memperoleh sebuah DAG. `make` dan setiap sistem pembangun sesudahnya mengerjakan dua hal dengan DAG itu: ia memeriksa bahwa graf itu memang asiklik (siklus di dalam graf pembangunan adalah kutu — berkas A tidak mungkin memerlukan B sementara B memerlukan A), dan ia mencari sebuah urutan untuk mengompilasi. Ketika Anda melihat `make` melaporkan *circular dependency dropped*, Anda sedang menyaksikan sebuah algoritma deteksi siklus berbicara.

Prasyarat mata kuliah. Di ITS, *Perancangan dan Analisis Algoritma* mensyaratkan *Struktur Data*, yang mensyaratkan *Dasar Pemrograman*. Gambarlah anak panah dari tiap mata kuliah ke mata kuliah yang dibukanya. Bila grafnya asiklik, seorang mahasiswa dapat lulus; setiap rencana studi yang sah adalah sebuah urutan pada DAG itu. Bila seseorang tanpa sengaja menjadikan mata kuliah X prasyarat Y dan Y prasyarat X, kurikulumnya tidak terpenuhi, dan siklusnya adalah buktinya.

Penjadwalan tugas. Sebuah order cucian harus ditimbang sebelum dihargai, dipilah sebelum dicuci, dicuci sebelum dikeringkan, dikeringkan sebelum disetrika. Beberapa pasangan tidak terurut — menimbang dan memilah boleh terjadi dalam urutan mana pun, atau serentak di dua meja. Sebuah DAG mencatat persis kendala yang ada dan tidak lebih, dan itulah sebabnya ia model yang tepat: ia tidak mengarang urutan di tempat yang tidak memerlukannya.

Eksekusi kode paralel. Bila instruksi a menuliskan sebuah nilai yang dibaca instruksi b, maka a harus berjalan lebih dulu; bila dua instruksi tidak menyentuh apa pun yang sama, keduanya boleh berjalan serentak di inti yang berbeda. DAG kebergantungan sebuah blok kode memberi tahu kompilator dengan tepat penataan ulang mana yang sah, dan seberapa besar keparalelan yang tersedia.

Pada keempatnya, dua pertanyaan langsung muncul, dan keduanya adalah pertanyaan bagian ini dan bagian berikutnya: **apakah ia asiklik?** dan, kalau ya, **berikan saya sebuah urutan yang sah.**

11.2.3 Teorema DAG

Inilah teorema yang membuat pertanyaan pertama menjadi murah untuk dijawab. Bab 10 sudah menjumpainya sebagai Teorema 10.10; di sini kita menyatakannya kembali dan membuktikannya lagi, dalam bentuk yang disesuaikan dengan keperluan berikutnya.

> **TEOREMA 11.1 (teorema DAG).** Sebuah graf berarah G bersifat asiklik **jika dan hanya jika** sebuah pencarian mendalam-dahulu atas G tidak menghasilkan sisi balik.

Perhatikan apa yang kita peroleh dari situ. Sebuah sisi balik, menurut **Tabel 11.1**, adalah sisi (u, v) yang diperiksa oleh pencarian mendalam-dahulu pada saat v berwarna kelabu. Mendeteksi warna kelabu hanyalah satu pencarian di dalam larik. Jadi teoremanya berkata: jalankan satu pencarian mendalam-dahulu, dan begitu Anda memeriksa sebuah sisi yang ujung seberangnya

kelabu, berhentilah — grafnya punya siklus. Bila Anda sampai ke ujung tanpa hal itu pernah terjadi, grafnya asiklik. Karena pencarian mendalam-dahulu berongkos $O(V + E)$ dan kita hanya menambahkan kerja sebanyak konstanta untuk tiap sisi, **deteksi siklus berongkos** $O(V + E)$. Itu semurah sekadar membaca grafnya, yang merupakan hal terbaik yang bisa diharapkan algoritma mana pun.

Frasa *jika dan hanya jika* berarti teoremanya punya dua paruh, dan keduanya perlu dibuktikan:

1. **Bila ada sisi balik, maka G punya siklus.** (Sisi balik adalah bukti *adanya* siklus.)
2. **Bila G punya siklus, maka ada sisi balik.** (Sebuah siklus tidak dapat bersembunyi dari pencariannya.)

Paruh pertama mudah. Paruh kedualah yang memikul beban, sebab ia harus menyingkirkan kemungkinan bahwa ada siklus yang tertata begitu licik sehingga pencariannya berjalan melewatinya tanpa menyadarinya. Tidak satu pun paruh boleh dilewati: setengah bukti akan meninggalkan kita dengan sebuah pengujian yang sahih tetapi mungkin buta.

11.2.4 Membuktikan teorema DAG

Bagian 1: sisi balik berakibat siklus.

Misalkan pencariannya memeriksa sebuah sisi (u, v) dan menemukan v berwarna kelabu, sehingga (u, v) adalah sisi balik. Menurut definisi kelabu, v sudah ditemukan dan belum selesai. Simpul-simpul kelabu pada sembarang saat selama pencarian mendalam-dahulu persis sama dengan simpul-simpul yang berada di tumpukan rekursi saat itu — yaitu rantai pemanggilan dari akar pohon mendalam-dahulu yang sedang dikerjakan turun sampai ke simpul yang sedang digarap sekarang. (Bab 10 menyebutnya "simpul-simpul kelabu membentuk sebuah rantai linear".) Karena u adalah simpul yang sedang digarap, dan v berwarna kelabu, maka v duduk di suatu tempat pada rantai itu, sejajar dengan u atau di atasnya. Artinya: v **adalah leluhur** u **di dalam hutan mendalam-dahulunya**, atau $v = u$ sendiri.

Hubungan keleluhuran di dalam hutan mendalam-dahulu dibangun dari sisi-sisi pohon, dan setiap sisi pohon adalah sisi G yang sungguhan. Jadi ada sebuah lintasan di dalam G dari v turun ke u:

$$v \rightsquigarrow u.$$

Sekarang tambahkan sisi (u, v) di ujung lintasan itu. Kita tiba kembali di v. Itu adalah jalan tertutup dengan panjang sekurang-kurangnya satu, yang berarti sebuah siklus. (Bila $v = u$, sisi (u, u) itu sendiri adalah siklus sepanjang satu — sebuah gelung diri.) Maka G tidak asiklik. ■

Bagian 2: siklus berakibat sisi balik.

Inilah arah yang memerlukan kehati-hatian, dan buktinya berputar pada satu gagasan, yang oleh Bab 10 disebut **sifat lintasan putih**: pada saat sebuah simpul v ditemukan, setiap simpul yang dapat dicapai dari v melalui lintasan yang seluruhnya berwarna putih akan menjadi keturunan v di dalam pohon mendalam-dahulu sebelum v selesai.

Misalkan G memuat sebuah siklus c. Pilih v sebagai **simpul pertama dari** c **yang ditemukan oleh pencarian mendalam-dahulunya**. Pasti ada yang pertama, sebab pencariannya pada akhirnya menemukan setiap simpul dan jamnya mengurutkan penemuan-penemuannya. Pilih u sebagai pendahulu v *pada siklus itu*, sehingga $(u, v) \in E$ dan u terletak pada c. Kita akan menunjukkan bahwa (u, v) adalah sisi balik.

Tinjaulah saat v ditemukan, yaitu waktu jam $d[v]$. Karena v adalah simpul pertama dari c yang ditemukan, **setiap simpul lain pada c masih putih pada saat itu** — termasuk u. Sekarang berjalanlah mengelilingi siklusnya dari v menurut arah anak panahnya:

$$v \to v_1 \to v_2 \to \cdots \to u.$$

Itu adalah sebuah lintasan dari v ke u, dan pada waktu $d[v]$ setiap simpul padanya sesudah v berwarna putih. Ia sebuah lintasan putih. Menurut sifat lintasan putih, pencariannya tidak dapat mundur dari v sebelum ia mengunjungi setiap simpul yang dapat dicapai dari v lewat lintasan putih; jadi u menjadi keturunan v di dalam pohon mendalam-dahulunya, dan u ditemukan serta diselesaikan benar-benar di dalam selang $(d[v], f[v])$.

Sekarang: pada suatu saat di dalam selang itu pencariannya berdiri di u dan memeriksa sisi-sisi u, salah satunya (u, v). Apa warna v ketika itu? Jamnya berada pada suatu waktu t dengan $d[v] < t < f[v]$, dan menurut aturan warna di §11.1 sebuah simpul berwarna kelabu persis antara waktu penemuan dan waktu penyelesaiannya. Jadi v kelabu. Sebuah sisi yang diperiksa dari simpul kelabu menuju simpul kelabu adalah, menurut **Tabel 11.1**, sisi balik. Maka (u, v) adalah sisi balik, dan pencariannya memang menemukan satu. ■

> **DENGAN BAHASA SEHARI-HARI.** Bagian 2 berkata bahwa sebuah siklus tidak dapat bersembunyi. Pilihlah simpul siklus yang pertama kali Anda tubruk. Pada saat itu seluruh sisa siklusnya belum terjelajahi, jadi ia jalan putih yang lapang yang menuntun kembali ke pintu rumah Anda sendiri. Pencarian mendalam-dahulu selalu menyusuri jalan putih yang lapang sampai ke ujungnya sebelum berbalik — jadi ia akan berjalan mengelilingi siklusnya sampai tuntas lalu mendapati dirinya mengetuk pintu kamar yang sedang ia tempati. Ketukan itulah sisi baliknya.

11.2.5 Mengapa pilihan "simpul pertama" itu penting

Mahasiswa sering mencoba Bagian 2 dengan memilih sembarang simpul pada siklusnya, dan buktinya patah. Inilah sebabnya kata *pertama* memikul beban.

Ambil siklus $x \to y \to z \to x$ dan andaikan pencariannya kebetulan menemukan y lebih dahulu, lalu menyelesaikan y dan z seluruhnya, dan baru kemudian — dari suatu simpul yang tidak berkaitan — menemukan x. Seandainya kita memilih x sebagai simpul awal dan menengok lintasan $x \to y$, kita akan mendapati y sudah **hitam**, bukan putih. Sisi (x, y) dengan demikian adalah sisi maju atau sisi silang, bukan sisi balik, dan argumennya runtuh.

Memilih simpul v yang *pertama ditemukan* pada siklus itu menyingkirkan kemungkinan tersebut sejak dari konstruksinya: tidak ada simpul lain pada siklus itu yang dapat berwarna selain putih pada waktu $d[v]$, sebab "pertama ditemukan" berarti tepat bahwa tidak satu pun dari mereka telah ditemukan. Pada contoh di atas, simpul yang pertama ditemukan adalah y; pendahulu y pada siklus itu adalah x; dan sisi (x, y) — yang diperiksa belakangan, dari keturunannya x, sementara y masih kelabu — itulah sisi baliknya. Buktinya menemukan sisi balik di mana pun ia berada, tetapi hanya bila Anda membiarkannya memilih sisi mana yang hendak ditengok.

> **JEBAKAN.** Teorema 11.1 berbicara tentang graf *berarah*, dan penggolongan sisi empat arah yang dipakainya adalah penggolongan untuk graf berarah. Pada graf **tak berarah** teorema itu, sebagaimana dinyatakan, salah dengan cara yang membosankan: setiap sisi $\{u, v\}$ muncul baik di $Adj[u]$ maupun di $Adj[v]$, sehingga pencariannya akan memeriksa (v, u) segera sesudah sampai di v dari u lalu mendapati u berwarna kelabu. Itu bukan siklus; itu sisi yang sama dilihat dua kali. Deteksi siklus tak berarah memerlukan aturan tambahan "abaikan sisi tempat

Anda datang", dan begitu aturan itu ditambahkan, sisi balik kembali berarti siklus. Soal 11.3 meminta Anda mengerjakan ini dengan benar.

11.3 Pengurutan Topologis

11.3.1 Apa yang kita inginkan

Kini kita sampai pada pertanyaan kedua: diberi sebuah DAG berisi pendahuluan, hasilkan sebuah urutan yang membuat tugas-tugasnya dapat dikerjakan secara sah. Definisinya:

DEFINISI. Sebuah **pengurutan topologis** (*topological sort*) dari graf berarah asiklik $G = (V, E)$ adalah pengurutan linear atas seluruh simpulnya sedemikian sehingga untuk setiap sisi $(u, v) \in E$, simpul u muncul **sebelum** simpul v di dalam urutan itu.

Bongkarlah setiap katanya. *Pengurutan linear* berarti sebuah senarai yang memuat tiap simpul tepat sekali — bukan pohon, bukan sehimpunan kelompok, melainkan satu barisan tunggal. *Seluruh* simpulnya berarti simpul terpencil pun ikut masuk ke senarainya, walaupun tidak ada apa pun yang mengekangnya. Dan syaratnya adalah syarat atas setiap sisi: bila Anda membentangkan simpul-simpulnya dari kiri ke kanan menurut urutan yang dihasilkan, **setiap anak panah harus menunjuk ke kanan.** Tidak boleh satu pun anak panah menunjuk balik. Itulah seluruh spesifikasinya, dan gambar itu layak Anda bawa di kepala, sebab begitu pula cara Anda memeriksa sebuah jawaban.

DENGAN BAHASA SEHARI-HARI. Pengurutan topologis adalah daftar kerja yang sungguh-sungguh dapat Anda kerjakan dari atas ke bawah tanpa pernah terhalang. Setiap kali Anda sampai ke sebuah butir, segala yang menjadi sandarannya sudah tercoret.

Yang pertama harus ditegakkan adalah bahwa urutan semacam itu memang ada. Ia ada, dan alasannya adalah Teorema 11.1 yang dibalik: graf yang punya siklus tidak punya urutan topologis (bila u mendahului v dan v mendahului u, yang mana yang dituntut siklusnya?), dan — sebagaimana akan dibuktikan secara konstruktif oleh kedua algoritma di bawah — graf yang tidak punya siklus selalu punya sekurang-kurangnya satu.

11.3.2 Mengapa urutan topologis (hampir) tidak pernah tunggal

Pemula mengharapkan satu jawaban. Biasanya jawabannya sangat banyak. Alasannya sederhana: definisinya hanya mengekang pasangan yang dihubungkan oleh sisi. Dua simpul yang tidak punya lintasan di antaranya ke arah mana pun boleh muncul di keluarannya dalam urutan mana pun, dan kedua keluaran itu benar.

Ambil contoh terkecil: $V = \{a, b\}$, $E = \varnothing$. Baik a, b maupun b, a adalah urutan topologis. Ambil $V = \{a, b, c\}$ dengan satu sisi (a, b). Maka c boleh di mana saja: c, a, b dan a, c, b dan a, b, c semuanya sah, dan itu baru tiga di antaranya.

Berapa banyak seluruhnya? Untuk DAG berupa satu rantai tunggal $v_1 \to v_2 \to \cdots \to v_n$, tepat satu: setiap pasangan terkekang. Untuk DAG tanpa satu sisi pun, tepat $n!$: tidak ada yang terkekang. Graf yang nyata berada di antaranya, dan cacahnya adalah banyaknya **perluasan linear** (*linear extensions*) dari urutan parsial yang didefinisikan DAG itu — sebuah besaran yang sungguh-sungguh sukar dihitung (ia #P-lengkap), yang dengan sendirinya merupakan isyarat bahwa biasanya jumlahnya banyak.

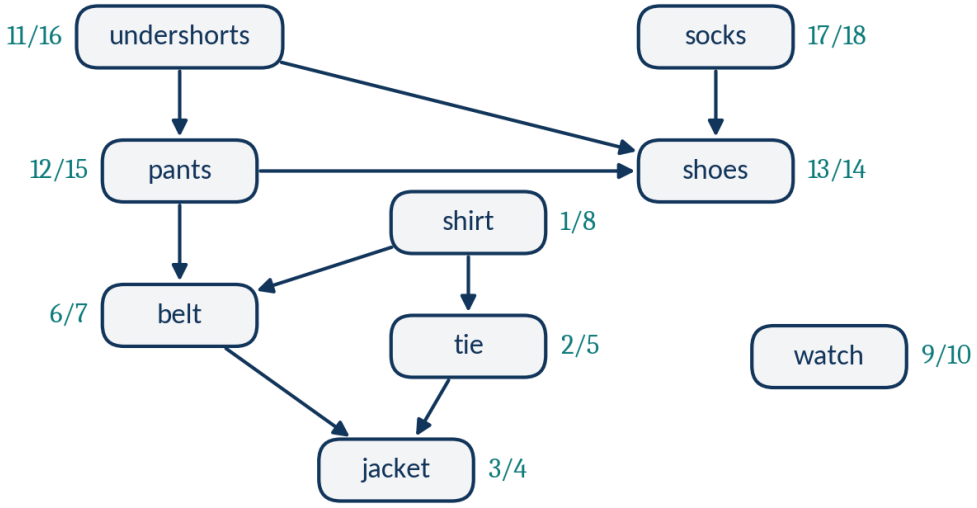
JEBAKAN. Karena jawabannya tidak tunggal, Anda tidak dapat memeriksa sebuah pengurutan topologis dengan membandingkannya terhadap satu jawaban tersimpan, dan Anda tidak dapat menguji sebuah implementasi dengan membandingkan keluarannya terhadap sebuah

untai tersimpan. Anda harus memeriksa *sifatnya*: bentangkan keluarannya, nomori posisinya, lalu pastikan $\text{pos}[u] < \text{pos}[v]$ untuk setiap sisi (u, v). Kedua program di bagian ini berakhir dengan mengerjakan persis itu, dan program Anda pun sebaiknya begitu.

Kita akan melihat ketidaktunggalan itu secara cuma-cuma: kedua algoritma di bawah, dijalankan pada graf yang sama, menghasilkan dua urutan yang sama sekali berbeda, dan keduanya benar.

11.3.3 Contoh berjalan: berpakaian

Gambar 11.2 adalah contoh berjalan bab ini, dan ia DAG kecil terbaik di dalam kepustakaan sebab semua orang sudah menyelesaikannya dengan tangan pagi tadi. Sembilan helai pakaian. Anak panah dari x ke y berarti *Anda harus selesai dengan x sebelum dapat mengerjakan y*.



Gambar 11.2 DAG pendahuluan untuk berpakaian, dengan cap waktu pencarian mendalam-dahulu $d[v]/f[v]$ tertulis di samping tiap simpul. Sembilan simpul, sembilan sisi. `watch` sama sekali tidak punya sisi: tidak ada yang mengekang kapan ia dikenakan.

Bacalah kesembilan sisinya dari gambar itu:

Tabel 11.2 Kesembilan kendala pendahuluan pada **Gambar 11.2**, dibaca dari anak panahnya. Perhatikan apa yang *tidak* ada di sini: tidak ada yang mengaitkan `socks` dengan `shirt`, dan sama sekali tidak ada yang mengaitkan `watch` dengan apa pun. Kebebasan itulah sebabnya jawabannya tidak tunggal.

Sisi	Dengan kata-kata
undershorts → pants	celana dalam dikenakan sebelum celana panjang
undershorts → shoes	dan sebelum sepatu
pants → shoes	celana panjang sebelum sepatu
pants → belt	celana panjang sebelum ikat pinggang
shirt → belt	kemeja sebelum ikat pinggang (sebab ia dimasukkan)
shirt → tie	kemeja sebelum dasi
belt → jacket	ikat pinggang sebelum jas
tie → jacket	dasi sebelum jas
socks → shoes	kaus kaki sebelum sepatu

Apakah ia DAG? Pasti, sebab kalau tidak, berpakaian akan menjadi mustahil. Sebentar lagi kita akan memastikannya secara mekanis: programnya melaporkan `back edge found: no`.

11.3.4 Algoritma 1: pencarian mendalam-dahulu, dan waktu penyelesaiannya

Algoritmanya, **Pseudokode 11.1**, panjangnya tiga baris, dan seluruh kecerdikannya ada di baris 2.

Pseudokode 11.1 Pengurutan topologis dengan pencarian mendalam-dahulu, yang mencatat tiap simpul pada saat simpul itu selesai.

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times f[v] for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

Perhatikan apa yang *tidak* dikerjakannya: ia tidak mengurutkan apa pun, dalam arti membandingkan pasangan. Ia menjalankan satu pencarian mendalam-dahulu lalu mendorong tiap simpul ke depan sebuah senarai pada saat pencariannya selesai dengan simpul itu. Karena mendorong ke depan senarai berkait berongkos $O(1)$, senarainya berakhir memuat simpul-simpulnya dalam **urutan waktu penyelesaian yang menurun**, dan — inilah klaimnya — urutan itu adalah urutan topologis.

Mengapa waktu penyelesaian yang menurun merupakan urutan yang benar? Inilah gambarannya sebelum pembuktiannya. Sebuah simpul selesai hanya ketika segala sesuatu yang dapat dicapai darinya sudah selesai. Jadi hal-hal yang bergantung pada u semuanya selesai *sebelum* u selesai. Bila kemudian kita mendaftar simpul-simpulnya dari yang paling akhir selesai ke yang paling awal, u keluar mendahului segala sesuatu yang bergantung kepadanya. Dan itulah yang diminta definisinya.

11.3.5 Contohnya ditelusuri, satu cap waktu demi satu cap waktu

Jalankan pencariannya dengan simpul-simpul pada gelung luarnya ditinjau dalam urutan `shirt, tie, jacket, belt, watch, undershorts, pants, shoes, socks`, dan tiap senarai ketetanggaannya dalam urutan yang diberikan **Tabel 11.2**. Inilah setiap detak jamnya.

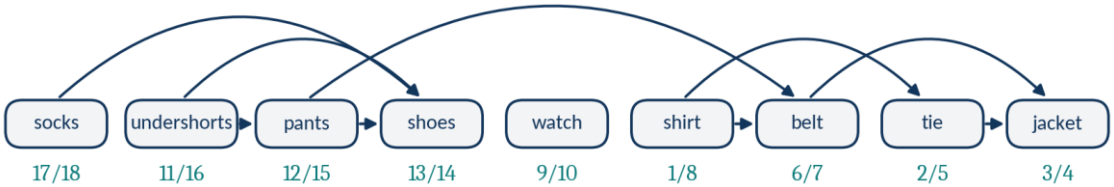
Tabel 11.3 Setiap detak jam mendalam-dahulu pada **Gambar 11.2**. Jamnya maju tepat satu pada tiap penemuan dan tiap penyelesaian, sehingga ia berakhir di $2|V| = 18$. Bandingkan pasangan d/f-nya dengan angka yang tertulis di samping simpul-simpulnya pada **Gambar 11.2**: keduanya cocok persis.

Jam	Peristiwa	Keadaan
1	temukan `shirt`	d[shirt] $= 1$; `shirt` kelabu
2	dari `shirt`, `tie` putih: temukan ia	d[tie] $= 2$
3	dari `tie`, `jacket` putih: temukan ia	d[jacket] $= 3$
4	`jacket` tidak punya sisi keluar: selesaikan ia	f[jacket] $= 4$; senarai = `jacket`
5	kembali di `tie`, tidak ada sisi lagi: selesaikan ia	f[tie] $= 5$; senarai = `tie, jacket`
6	kembali di `shirt`, `belt` putih: temukan ia	d[belt] $= 6$
7	dari `belt`, `jacket` **hitam** — tidak ada yang perlu dikerjakan; selesaikan `belt`	f[belt] $= 7$; senarai = `belt, tie, jacket`
8	kembali di `shirt`, tidak ada sisi lagi: selesaikan ia	f[shirt] $= 8$
9	gelung luar: `watch` putih, temukan ia	d[watch] $= 9$
10	`watch` tidak punya sisi keluar: selesaikan ia	f[watch] $= 10$
11	gelung luar: `undershorts` putih, temukan ia	d[undershorts] $= 11$
12	dari `undershorts`, `pants` putih: temukan ia	d[pants] $= 12$
13	dari `pants`, `shoes` putih: temukan ia	d[shoes] $= 13$
14	`shoes` tidak punya sisi keluar: selesaikan ia	f[shoes] $= 14$
15	kembali di `pants`, `belt` hitam: selesaikan `pants`	f[pants] $= 15$
16	kembali di `undershorts`, `shoes` hitam: selesaikan ia	f[undershorts] $= 16$
17	gelung luar: `socks` putih, temukan ia	d[socks] $= 17$
18	`shoes` hitam; selesaikan `socks`	f[socks] $= 18$

Sekarang bacalah waktu penyelesaiannya, dari yang terbesar lebih dulu:

$$\underbrace{18}_{\text{socks}}, \underbrace{16}_{\text{undershorts}}, \underbrace{15}_{\text{pants}}, \underbrace{14}_{\text{shoes}}, \underbrace{10}_{\text{watch}}, \underbrace{8}_{\text{shirt}}, \underbrace{7}_{\text{belt}}, \underbrace{5}_{\text{tie}}, \underbrace{4}_{\text{jacket}}.$$

Itulah jawabannya, dan **Gambar 11.3** menggambarkannya: kesembilan simpul yang sama dibentangkan dalam satu garis, dengan kesembilan anak panahnya kini semua melengkung *maju*, dari kiri ke kanan, dan tidak satu pun melengkung balik.



Gambar 11.3 Hasilnya: kesembilan simpul **Gambar 11.2** dibentangkan dalam urutan waktu penyelesaian yang menurun. Setiap anak panah kini menunjuk ke kanan, dan itu persis definisi pengurutan topologis. Pasangan d/f dicetak di bawah tiap simpul supaya urutannya dapat diperiksa terhadap **Tabel 11.3**.

Periksalah dengan tangan terhadap **Tabel 11.2**. `undershorts` (posisi 2) sebelum `pants` (3) dan sebelum `shoes` (4): ya. `pants` sebelum `shoes` dan sebelum `belt` (7): ya. `shirt` (6) sebelum `belt` (7) dan `tie` (8): ya. `belt` (7) sebelum `jacket` (9): ya. `tie` (8) sebelum `jacket` (9): ya. `socks` (1) sebelum `shoes` (4): ya. Kesembilan-sembilannya.

Perhatikan pula betapa ganjil jawaban itu bila dibaca sebagai saran. Ia menyuruh Anda mengenakan kaus kaki paling dahulu dan sepatu di urutan keempat — sebelum kemeja. Itu *sah*: tidak ada apa pun di **Tabel 11.2** yang mengatakan kemeja harus mendahului sepatu. Ia hanya bukan yang akan Anda lakukan. Pengurutan topologis menghormati kendala yang Anda berikan dan tidak lebih, dan itu sebuah keunggulan; bila Anda ingin sepatu dikenakan belakangan, tambahkanlah sisi yang mengatakan demikian.

11.3.6 Mengapa ia benar

Buktinya adalah argumen tiga kasus. Inilah selengkapnya.

> **TEOREMA 11.2.** Misalkan G sebuah DAG. Maka untuk setiap sisi $(u, v) \in E$ berlaku $f[v] < f[u]$. Akibatnya, mendaftar simpul-simpulnya dalam urutan f yang menurun merupakan pengurutan topologis atas G.

Bukti. Tinjaulah sembarang sisi (u, v), dan tinjau saat pencarian mendalam-dahulunya memeriksa sisi itu. Pada saat itu pencariannya sedang berdiri di u, sehingga u berwarna kelabu. Ada tepat tiga kemungkinan warna v, dan kita ambil satu per satu.

Kasus 1: v kelabu. Maka (u, v) adalah sisi balik, dan menurut Bagian 1 Teorema 11.1 grafnya memuat siklus. Tetapi G adalah DAG. Kontradiksi — jadi kasus ini tidak mungkin terjadi. (Hanya di sinilah hipotesis keasiklikan dipakai, dan ia tak tergantikan: pada graf bersiklus teoremanya salah, sebagaimana mestinya.)

Kasus 2: v putih. Maka v ditemukan lewat sisi ini juga, sehingga v menjadi anak u di dalam pohon mendalam-dahulunya. Pencariannya harus menyelesaikan v sebelum ia dapat mundur dari u, sebab pemanggilan rekursif atas v bersarang di dalam pemanggilan atas u. Maka $f[v] < f[u]$.

Kasus 3: v hitam. Maka v sudah selesai, pada suatu waktu jam yang benar-benar lebih awal daripada sekarang. Dan u belum selesai — ia kelabu — sehingga $f[u]$ adalah suatu waktu yang benar-benar lebih kemudian daripada sekarang. Maka $f[v] <$ sekarang $< f[u]$, yang memberikan $f[v] < f[u]$.

Pada setiap kasus yang mungkin, $f[v] < f[u]$. Jadi di dalam senarai yang terurut menurut f yang menurun, simpul u muncul sebelum v. Karena (u, v) adalah sisi sembarang, ini berlaku untuk setiap sisi, dan itu persis definisi pengurutan topologis. ■

> **DENGAN BAHASA SEHARI-HARI.** Apa pun yang ditunjuk u entah selesai di dalam kunjungan u sendiri (kasus 2) atau sudah selesai sebelum u mulai memedulikannya (kasus 3). Bagaimanapun juga, ia selesai lebih dulu. Jadi "yang terakhir selesai" sama dengan "yang paling hulu", dan mengurutkan menurut yang terakhir selesai menempatkan hal-hal yang di hulu di depan.

11.3.7 Waktu jalannya $\Theta(V + E)$

Pembukuannya cukup dua baris, dan keduanya layak diuraikan.

- Pencarian mendalam-dahulunya berongkos $\Theta(V + E)$. Kedua gelung atas simpulnya berongkos $\Theta(V)$; `DFS-Visit` dipanggil tepat sekali untuk tiap simpul (ia hanya dipanggil pada simpul putih dan langsung mewarnai argumennya kelabu, sehingga tidak ada simpul yang dikunjungi dua kali); dan sepanjang seluruh pemanggilan itu gelung dalamnya berjalan atas $Adj[u]$ satu kali untuk tiap u, sedangkan $\sum_u |Adj[u]| = |E|$ pada graf berarah. Totalnya: $\Theta(V + E)$.
- Menyisipkan sebuah simpul di depan senarai berkait berongkos $O(1)$, dan kita mengerjakannya sekali tiap simpul, sehingga $O(V)$ seluruhnya.

Dijumlahkan: $\Theta(V + E) + O(V) = \Theta(V + E)$. Karena algoritma mana pun sekurang-kurangnya harus melihat masukannya, dan masukannya berukuran $\Theta(V + E)$ pada representasi senarai ketetanggaan, batas ini optimal sampai dengan faktor konstanta. Tidak ada lagi yang tersisa untuk diperbaiki.

> **JEBAKAN.** Penyisipannya harus di **depan**, dan harus terjadi pada waktu **penyelesaian**, bukan pada waktu penemuan. Menempelkan di belakang memberikan waktu penyelesaian yang *menaik*, yang merupakan kebalikan urutan topologisnya — setiap anak panah menunjuk ke arah yang salah. Dan mencatat pada waktu penemuan memberikan urutan yang kebetulan ditempuh pencariannya, yang sama sekali bukan urutan topologis: pada **Tabel 11.3** urutan penemuannya dimulai `shirt, tie, jacket`, sedangkan `jacket` punya dua pendahulu yang belum muncul.

11.3.8 Program selengkapnya

Kode 11.1 adalah seluruhnya, siap dikompilasi. Ia membawa warnanya, jamnya, d, f, dan π secara eksplisit, sehingga cap waktu yang dicetaknya dapat dibandingkan baris demi baris dengan **Tabel 11.3**; ia mengawasi sisi kelabu-ke-kelabu, sehingga ia juga menjawab pertanyaan siklus di §11.2; dan ia berakhir dengan memverifikasi sifat topologisnya, alih-alih memercayainya begitu saja.

Kode 11.1 Pengurutan topologis memakai waktu penyelesaian mendalam-dahulu, pada DAG berpakaian. Kompilasilah dengan `g++ -std=c++17 -O2 -o toposort toposort.cpp`.

```
// Pengurutan topologis memakai waktu penyelesaian DFS.
// Mereproduksi contoh "berpakaian" dari Subbab 11.3.3.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <list>
using namespace std;

enum Color { WHITE, GRAY, BLACK };

struct Graph {
    int n;                          // banyaknya simpul
    vector<string>       name;      // name[u] - nama simpul u yang dapat dicetak
    vector<vector<int>>  adj;       // adj[u]  - simpul-simpul yang ditunjuk u
    explicit Graph(const vector<string>& names)
        : n((int)names.size()), name(names), adj(names.size()) {}
    void addEdge(int u, int v) { adj[u].push_back(v); }
};

struct DfsState {
    vector<Color> color;
    vector<int>    d, f, pi;
    int            clock_ = 0;      // jam global yang naik secara monoton
    list<int>      order;           // senarai berkait TOPOLOGICAL-SORT, baris 2
    bool           hasBackEdge = false;
};

void dfsVisit(const Graph& G, DfsState& S, int u) {
    S.color[u] = GRAY;
    S.d[u] = ++S.clock_;                      // u ditemukan
    for (int v : G.adj[u]) {
        if (S.color[v] == GRAY) S.hasBackEdge = true;   // kelabu -> kelabu adalah sisi
balik
        if (S.color[v] == WHITE) { S.pi[v] = u; dfsVisit(G, S, v); }
    }
    S.color[u] = BLACK;
    S.f[u] = ++S.clock_;                      // u selesai
    S.order.push_front(u);                    // sisipkan di DEPAN senarainya
}
```

Kode 11.1 (Lanjutan) *Pengurutan topologis memakai waktu penyelesaian mendalam-dahulu, pada DAG berpakaian. Kompilasilah dengan g++ -std=c++17 -O2 -o toposort toposort.cpp.*

```
DfsState topologicalSort(const Graph& G) {
    DfsState S;
    S.color.assign(G.n, WHITE);
    S.d.assign(G.n, 0); S.f.assign(G.n, 0); S.pi.assign(G.n, -1);
    for (int u = 0; u < G.n; ++u)
        if (S.color[u] == WHITE) dfsVisit(G, S, u);
    return S;
}

int main() {
    // Urutan indeks menetapkan urutan gelung luarnya, dan dengan itu cap waktunya.
    vector<string> names = { "shirt", "tie", "jacket", "belt", "watch",
                             "undershorts", "pants", "shoes", "socks" };
    enum { SHIRT, TIE, JACKET, BELT, WATCH, UNDERSHORTS, PANTS, SHOES, SOCKS };
    Graph G(names);
    G.addEdge(SHIRT, TIE);        G.addEdge(SHIRT, BELT);
    G.addEdge(TIE, JACKET);
    G.addEdge(BELT, JACKET);
    G.addEdge(UNDERSHORTS, PANTS); G.addEdge(UNDERSHORTS, SHOES);
    G.addEdge(PANTS, SHOES);      G.addEdge(PANTS, BELT);
    G.addEdge(SOCKS, SHOES);
    // "watch" sama sekali tidak punya sisi.

    DfsState S = topologicalSort(G);

    cout << "timestamps (d/f) in the order the DFS assigned them\n";
    vector<int> byDiscovery(G.n);
    for (int u = 0; u < G.n; ++u) byDiscovery[u] = u;
    for (int i = 0; i < G.n; ++i)                        // seleksi sederhana menurut d[]
        for (int j = i + 1; j < G.n; ++j)
            if (S.d[byDiscovery[j]] < S.d[byDiscovery[i]]) swap(byDiscovery[i],
byDiscovery[j]);
    for (int u : byDiscovery)
        cout << "  " << left << setw(12) << G.name[u] << right
             << setw(3) << S.d[u] << "/" << S.f[u] << "\n";

    cout << "back edge found: " << (S.hasBackEdge ? "yes" : "no") << "\n";
    cout << "topological order: ";
    for (int u : S.order) cout << G.name[u] << " ";
    cout << "\n";
```

Kode 11.1 (Lanjutan) *Pengurutan topologis memakai waktu penyelesaian mendalam-dahulu, pada DAG berpakaian. Kompilasilah dengan g++ -std=c++17 -O2 -o toposort toposort.cpp.*

```
    cout << "check: every edge points forward\n";
    vector<int> pos(G.n); int k = 0;
    for (int u : S.order) pos[u] = k++;
    bool ok = true;
    for (int u = 0; u < G.n; ++u)
        for (int v : G.adj[u])
            if (pos[u] >= pos[v]) { ok = false;
                cout << "  VIOLATED: " << G.name[u] << " -> " << G.name[v] << "\n"; }
    cout << (ok ? "  all 9 edges satisfied" : "  order is not topological") << "\n";
    return 0;
}
```

Keluarannya:

```
timestamps (d/f) in the order the DFS assigned them
  shirt         1/8
  tie           2/5
  jacket        3/4
  belt          6/7
  watch         9/10
  undershorts  11/16
  pants        12/15
  shoes        13/14
  socks        17/18
back edge found: no
topological order: socks undershorts pants shoes watch shirt belt tie jacket
check: every edge points forward
  all 9 edges satisfied
```

Setiap cap waktunya cocok dengan angka yang tertulis di samping simpul itu pada **Gambar 11.2**, dan urutannya cocok persis dengan **Gambar 11.3**.

11.3.9 Algoritma 2: algoritma derajat masuk dari Kahn

Algoritma kedua lebih tua daripada yang pertama — A. B. Kahn menerbitkannya pada 1962 [2] — dan inilah yang ditemukan sendiri oleh kebanyakan orang bila Anda meminta mereka menyusun daftar kerja dari sebuah graf kebergantungan. Ia bekerja dari depan, bukan dari belakang.

> **DEFINISI. Derajat masuk** sebuah simpul v, ditulis $\text{indeg}(v)$, adalah banyaknya sisi yang menunjuk *kepadanya*: $\text{indeg}(v) = |\{u : (u, v) \in E\}|$. **Derajat keluarnya** adalah banyaknya yang menunjuk menjauh darinya.

Simpul berderajat masuk 0 tidak punya apa pun yang menunggunya — tidak ada prasyarat, tidak ada apa pun di hulunya. Jadi aman untuk dikerjakan **sekarang**. Pengamatan tunggal itulah seluruh algoritmanya, **Pseudokode 11.2**:

Pseudokode 11.2 Pengurutan topologis Kahn: berulang kali keluarkan simpul yang tidak punya pendahulu tersisa; kekurangan pada akhirnya membuktikan adanya siklus.

```
KAHN(G)
 1  compute indeg[v] for every vertex v
 2  R = { v : indeg[v] == 0 }              // R adalah himpunan "siap"
 3  L = empty list
 4  while R is not empty
 5      remove some vertex u from R
 6      append u to L
 7      for each v in Adj[u]
 8          indeg[v] = indeg[v] - 1        // u sudah selesai, jadi v menunggu satu lebih
sedikit
 9          if indeg[v] == 0
10              add v to R
11  if |L| < |V|
12      report "G has a cycle"
13  return L
```

Ada dua rincian yang layak disinggahi. Pertama, baris 5 berkata *suatu* simpul: simpul siap mana pun boleh, dan pada pilihan itulah seluruh ketidaktunggalannya bersemayam. Pakailah tumpukan dan Anda memperoleh satu urutan; antrean, urutan lain; *min-heap* atas indeks simpulnya, urutan lain lagi; dan setiap urutan itu benar. Kedua, baris 11–12 adalah uji siklus yang datang cuma-cuma. Bila grafnya punya siklus, tidak ada simpul pada siklus itu yang derajat masuknya dapat mencapai 0 — masing-masing menunggu pendahulunya mengelilingi gelung itu, selamanya — sehingga simpul-simpul itu, beserta apa pun di hilirnya, tidak pernah masuk ke R dan tidak pernah sampai ke L. Kekurangannya itulah diagnosisnya.

11.3.10 Kahn ditelusuri pada DAG yang sama

Mulailah dengan menghitung mata panah pada **Gambar 11.2**:

$$\text{indeg:shirt} = 0,\ \text{tie} = 1,\ \text{jacket} = 2,\ \text{belt} = 2,\ \text{watch} = 0,\ \text{undershorts} = 0,\ \text{pants} = 1,\ \text{shoes} = 3,\ \text{socks} = 0.$$

Jadi $R = \{\text{shirt,watch,undershorts,socks}\}$ — empat hal yang boleh Anda kerjakan lebih dahulu. (Bandingkan dengan algoritma DFS, yang tidak menawarkan pilihan semacam itu; di sini kebebasannya terlihat sejak baris pertama.) Dengan mengambil simpul siap berindeks terkecil setiap kalinya, jalannya menjadi seperti berikut.

Tabel 11.4 Algoritma Kahn pada **Gambar 11.2**, satu pengeluaran tiap baris. Sembilan simpul keluar dari sembilan langkah, jadi tidak ada siklus. Bandingkan hasilnya dengan **Gambar 11.3**: kedua urutan itu tidak berbagi satu posisi pun, dan keduanya benar.

Langkah	Keluarkan	Derajat masuk yang jatuh ke 0	Himpunan siap sesudahnya
1	`shirt`	`tie`	watch, undershorts, socks, tie
2	`tie`	—	watch, undershorts, socks
3	`watch`	—	undershorts, socks
4	`undershorts`	`pants`	socks, pants
5	`pants`	`belt`	socks, belt
6	`belt`	`jacket`	socks, jacket
7	`jacket`	—	socks
8	`socks`	`shoes`	shoes
9	`shoes`	—	(kosong)

Urutan yang dihasilkannya adalah

$$\text{shirt, tie, watch, undershorts, pants, belt, jacket, socks, shoes,}$$

yang sama sekali tidak mirip jawaban mendalam-dahulu pada **Gambar 11.3**. Periksalah ia terhadap **Tabel 11.2** juga: `undershorts` (4) sebelum `pants` (5) dan `shoes` (9) — ya; `pants` (5) sebelum `shoes` (9) dan `belt` (6) — ya; `shirt` (1) sebelum `belt` (6) dan `tie` (2) — ya; `belt` (6) dan `tie` (2) sebelum `jacket` (7) — ya; `socks` (8) sebelum `shoes` (9) — ya. Sembilan dari sembilan. **Kedua jawabannya benar, dan keduanya bukan jawaban yang sama.** Itulah §11.3.2 yang dijadikan nyata.

11.3.11 Mengapa Kahn benar

Ada dua hal yang harus dibuktikan: bahwa yang keluar merupakan urutan topologis, dan bahwa semuanya keluar (bila grafnya asiklik).

Keluarannya adalah urutan topologis. Peliharalah invarian ini: *sebuah simpul ditempelkan ke L hanya sesudah setiap pendahulunya ditempelkan.* Ia berlaku pada awalnya, ketika L kosong. Ia dipelihara oleh baris 6: simpul u yang dikeluarkan dari R memiliki $\text{indeg}[u] = 0$ pada saat itu, dan pencacah u bermula dari derajat masuknya yang sebenarnya lalu dikurangi tepat sekali untuk tiap pendahulu yang ditempelkan (baris 8). Jadi $\text{indeg}[u] = 0$ berarti *semua* pendahulu u sudah berada di L. Maka untuk setiap sisi (p, u), p mendahului u di dalam L — persis definisi pengurutan topologis. ■

Semuanya keluar, bila G asiklik. Andaikan tidak: gelungnya berhenti dengan R kosong dan beberapa simpul tersisa. Misalkan X himpunan simpul yang tersisa itu; X tidak kosong. Setiap $v \in X$ memiliki indeg$[v] > 0$ pada akhirnya, sehingga v punya sekurang-kurangnya satu pendahulu yang tidak berada di L — yaitu pendahulu di dalam X. Jadi dari sembarang simpul X kita dapat melangkah mundur ke simpul X yang lain, dan dari yang itu mundur lagi, dan seterusnya selamanya. Tetapi X berhingga, sehingga suatu simpul pasti berulang, dan simpul yang berulang pada jalan mundur adalah sebuah siklus berarah. Itu bertentangan dengan keasikliannya. Maka $X = \varnothing$ dan $|L| = |V|$. ■

Argumen yang sama, dibaca maju, adalah *pendeteksi* siklusnya: bila $|L| < |V|$, himpunan sisa X tidak kosong dan argumen di atas memproduksi sebuah siklus di dalamnya. Jadi uji pada baris 11 bukan heuristik; ia eksak.

11.3.12 Waktu jalan Kahn

- Baris 1 menghitung semua derajat masuk dengan menyapu tiap senarai ketetanggaan satu kali: $\Theta(V + E)$.
- Baris 2 memindai simpul-simpulnya satu kali: $\Theta(V)$.
- Badan gelung `while` berjalan sekali tiap simpul, sebab tiap simpul masuk ke R paling banyak sekali (ia masuk hanya ketika pencacahnya *mencapai* 0, dan pencacahnya tidak pernah naik). Tiap putarannya mengerjakan $O(1)$ kerja ditambah satu sapuan atas $Adj[u]$. Dijumlahkan atas semua u: $\Theta(V + E)$.
- Baris 11–12 berongkos $O(1)$.

Totalnya $\Theta(V + E)$ — sama dengan metode mendalam-dahulu, dan optimal pula. Bila R berupa tumpukan atau antrean biasa, setiap operasi atasnya $O(1)$ dan batasnya persis seperti yang dinyatakan; bila Anda memakai *heap* supaya keluarannya deterministik, sebagaimana dilakukan program di bawah, tiap operasinya berongkos $O(\lg V)$ dan totalnya menjadi $O(V \lg V + E)$, yang merupakan harga yang dibayar semata-mata demi kerapian.

11.3.13 Program selengkapnya, dengan deteksi siklus

Kode 11.2 adalah seluruh algoritmanya, dengan himpunan siapnya ditampung di dalam *min-heap* supaya urutan yang dicetak dapat diulang, dan dengan satu jalan kedua pada graf yang sengaja dirusak supaya uji siklusnya dapat terlihat bekerja.

Kode 11.2 Pengurutan topologis derajat masuk dari Kahn, berikut uji siklus yang cuma-cuma. Kompilasilah dengan `g++ -std=c++17 -O2 -o kahn kahn.cpp`.

```
// Pengurutan topologis memakai derajat masuk (algoritma Kahn), pada DAG yang sama,
// ditambah uji siklus pada graf yang sengaja dibuat tidak asiklik.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
using namespace std;

struct Graph {
    int n;
    vector<string>      name;
    vector<vector<int>> adj;
    explicit Graph(const vector<string>& names)
        : n((int)names.size()), name(names), adj(names.size()) {}
    void addEdge(int u, int v) { adj[u].push_back(v); }
};

// Mengembalikan simpul-simpul menurut urutan keluarnya. Jika urutan itu
// lebih pendek daripada G.n, grafnya punya siklus: simpul yang tertinggal persis
// yang derajat masuknya tak pernah mencapai 0, yaitu siklusnya dan apa pun di hilirnya.
// `trace` mencetak satu baris per pengeluaran, memperlihatkan apa yang dibebaskannya.
vector<int> kahn(const Graph& G, bool trace) {
    vector<int> indeg(G.n, 0);
    for (int u = 0; u < G.n; ++u)
        for (int v : G.adj[u]) ++indeg[v];

    // Min-heap atas indeksnya membuat keluarannya deterministik; wadah apa pun bisa
dipakai.
    priority_queue<int, vector<int>, greater<int>> ready;
    for (int u = 0; u < G.n; ++u) if (indeg[u] == 0) ready.push(u);

    if (trace) {
        cout << "  initial in-degrees:";
        for (int u = 0; u < G.n; ++u) cout << " " << G.name[u] << "=" << indeg[u];
        cout << "\n";
    }
```

Kode 11.2 (Lanjutan) *Pengurutan topologis derajat masuk dari Kahn, berikut uji siklus yang cuma-cuma. Kompilasilah dengan g++ -std=c++17 -O2 -o kahn kahn.cpp.*

```
    vector<int> order;
    int step = 0;
    while (!ready.empty()) {
        int u = ready.top(); ready.pop();
        order.push_back(u);
        vector<int> freed;
        for (int v : G.adj[u]) if (--indeg[v] == 0) { ready.push(v); freed.push_back(v); }
        if (trace) {
            cout << "  step " << setw(2) << ++step << ": emit " << left << setw(12)
                 << G.name[u] << right << " freed:";
            if (freed.empty()) cout << " -";
            for (int v : freed) cout << " " << G.name[v];
            cout << "\n";
        }
    }
    return order;
}

int main() {
    vector<string> names = { "shirt", "tie", "jacket", "belt", "watch",
                             "undershorts", "pants", "shoes", "socks" };
    enum { SHIRT, TIE, JACKET, BELT, WATCH, UNDERSHORTS, PANTS, SHOES, SOCKS };
    Graph G(names);
    G.addEdge(SHIRT, TIE);         G.addEdge(SHIRT, BELT);
    G.addEdge(TIE, JACKET);        G.addEdge(BELT, JACKET);
    G.addEdge(UNDERSHORTS, PANTS); G.addEdge(UNDERSHORTS, SHOES);
    G.addEdge(PANTS, SHOES);       G.addEdge(PANTS, BELT);
    G.addEdge(SOCKS, SHOES);

    cout << "Kahn on the dressing DAG\n";
    vector<int> ord = kahn(G, true);
    cout << "  order: ";
    for (int u : ord) cout << G.name[u] << " ";
    cout << "\n";

    vector<int> pos(G.n); for (int i = 0; i < (int)ord.size(); ++i) pos[ord[i]] = i;
    bool ok = true;
    for (int u = 0; u < G.n; ++u)
        for (int v : G.adj[u]) if (pos[u] >= pos[v]) ok = false;
    cout << "  valid topological order: " << (ok ? "yes" : "no") << "\n\n";
```

Kode 11.2 (Lanjutan) *Pengurutan topologis derajat masuk dari Kahn, berikut uji siklus yang cuma-cuma. Kompilasilah dengan g++ -std=c++17 -O2 -o kahn kahn.cpp.*

```
    // Sekarang rusakkan. Menambahkan jacket -> undershorts menutup siklus berarah
    // undershorts -> pants -> belt -> jacket -> undershorts.
    cout << "Kahn on the same graph plus the edge jacket -> undershorts\n";
    Graph H = G;
    H.addEdge(JACKET, UNDERSHORTS);
    vector<int> bad = kahn(H, false);
    cout << "  vertices emitted: " << bad.size() << " of " << H.n << " -- ";
    for (int u : bad) cout << H.name[u] << " ";
    cout << "\n  stuck (in-degree never fell to 0):";
    vector<bool> out(H.n, false);
    for (int u : bad) out[u] = true;
    for (int u = 0; u < H.n; ++u) if (!out[u]) cout << " " << H.name[u];
    cout << "\n  verdict: " << ((int)bad.size() < H.n
             ? "cycle detected, no topological order exists" : "acyclic") << "\n";
    return 0;
}
```

Keluarannya:

```
Kahn on the dressing DAG
  initial in-degrees: shirt=0 tie=1 jacket=2 belt=2 watch=0 undershorts=0 pants=1 shoes=3
socks=0
  step  1: emit shirt        freed: tie
  step  2: emit tie          freed: -
  step  3: emit watch        freed: -
  step  4: emit undershorts  freed: pants
  step  5: emit pants        freed: belt
  step  6: emit belt         freed: jacket
  step  7: emit jacket       freed: -
  step  8: emit socks        freed: shoes
  step  9: emit shoes        freed: -
  order: shirt tie watch undershorts pants belt jacket socks shoes
  valid topological order: yes

Kahn on the same graph plus the edge jacket -> undershorts
  vertices emitted: 4 of 9 -- shirt tie watch socks
  stuck (in-degree never fell to 0): jacket belt undershorts pants shoes
  verdict: cycle detected, no topological order exists
```

Derajat masuk pada baris pertamanya adalah yang kita hitung dengan tangan, kesembilan langkahnya adalah **Tabel 11.4**, dan jalan kedua memperlihatkan uji siklusnya bekerja: empat simpul lolos, dan lima yang tersangkut adalah siklus empat simpul `undershorts` → `pants` → `belt` → `jacket` → `undershorts` ditambah `shoes`, yang duduk di hilirnya dan tidak akan pernah dapat dibebaskan.

11.3.14 Yang mana dari keduanya yang dipakai

Biaya keduanya sama dan keduanya benar, sehingga pilihannya dijatuhkan atas dasar yang lain.

Tabel 11.5 Memilih di antara kedua algoritma pengurutan topologis. Keduanya $\Theta(V + E)$ dan keduanya benar; segala yang ada di tabel ini soal rekayasa, bukan soal kompleksitas.

Pertimbangan	Waktu penyelesaian DFS	Derajat masuk Kahn
Baris kode, bila DFS sudah tertulis	2 baris tambahan	~12
Rekursi	ya — graf yang dalam dapat melimpahkan tumpukan	tidak, murni iteratif
Deteksi siklus	ya, lewat tetangga kelabu; melaporkan *sisinya*	ya, lewat $L< V$; melaporkan *himpunan yang tersangkut*
Kendali atas urutan keluarannya	tidak ada: pencariannya yang memutuskan	penuh: pilih wadah himpunan siapnya
Mendeteksi siklus **lebih awal**	ya, pada sisi yang bermasalah	tidak, hanya sesudah gelungnya berakhir
Wajar untuk tatanan inkremental / aliran	tidak	ya: alirkan simpul baru ke R begitu ia siap
Meluas ke lintasan terpanjang / penjadwalan	canggung	langsung (lihat §11.13.3)

Dalam praktik: bila Anda sudah punya pencarian mendalam-dahulu dan grafnya dangkal, ambil versi dua barisnya. Bila grafnya besar dan mungkin dalam, bila Anda menginginkan keluarannya dalam urutan tertentu (menurut abjad, atau menurut prioritas), atau bila Anda *sekaligus* ingin menghitung waktu mulai paling awal sambil berjalan, ambil Kahn.

11.4 Pohon Merentang

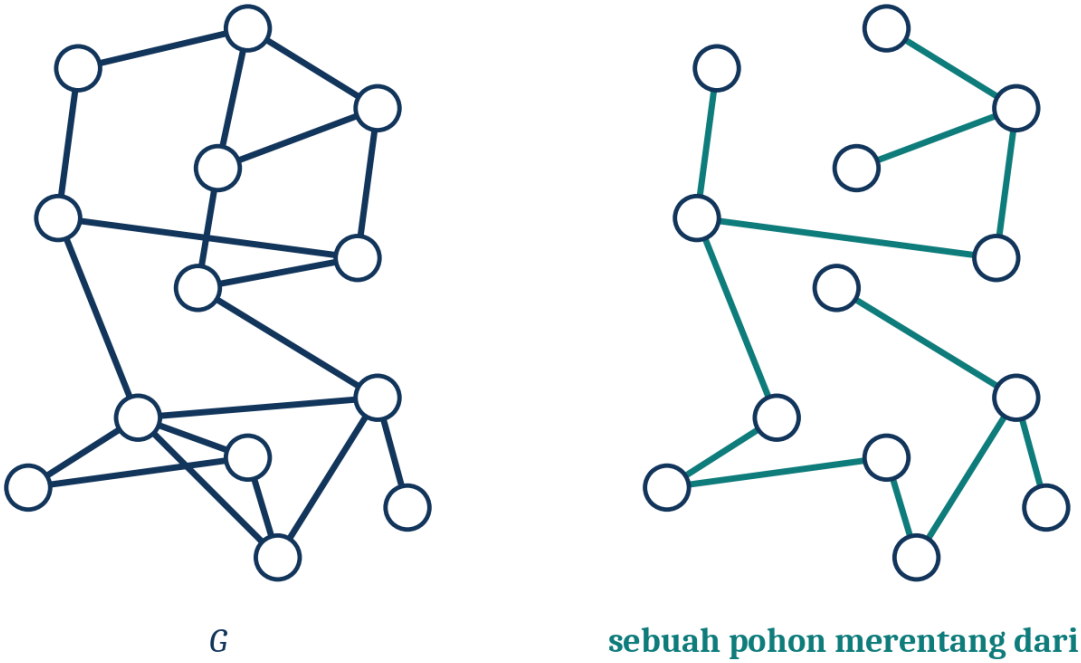
Kini kita meninggalkan graf berarah. Untuk sisa bab ini setiap graf bersifat **tak berarah**, **terhubung**, dan **berbobot**.

11.4.1 Definisi

DEFINISI. Sebuah **pohon** adalah graf tak berarah yang terhubung dan tidak punya siklus. Sebuah **pohon merentang** (*spanning tree*; sebagian sumber menulisnya *pohon merentang*) dari graf terhubung $G = (V, E)$ adalah upagraf T yang (i) merupakan pohon dan (ii) memuat **semua** simpul G.

Kedua syaratnya sama-sama bekerja. Lepaskan (i) dan Anda boleh menyimpan setiap sisi, yang memang terhubung tetapi punya siklus. Lepaskan (ii) dan satu sisi tunggal pun memenuhi syarat, yang memang pohon tetapi hampir tidak merentang apa-apa. Bersama-sama keduanya berkata: *simpanlah setiap tempat, buanglah setiap tautan yang berlebih, dan jagalah semuanya tetap dapat dicapai.*

Gambar 11.4 menggambarkan definisinya. Di kiri, sebuah graf G dengan tiga belas simpul dan jauh lebih banyak sisi daripada yang diperlukannya. Di kanan, satu pohon merentang G: tiga belas simpul yang sama, tetapi hanya dua belas sisi, tertata sedemikian rupa sehingga seluruhnya masih satu bagian dan ada tepat satu rute antara sembarang dua simpul.



Gambar 11.4 Sebuah pohon merentang. **Kiri:** graf G. **Kanan:** sebuah pohon merentang G — setiap simpul G hadir, grafnya masih terhubung, dan tidak ada siklus yang tersisa. Hitunglah sisi di sebelah kanan: dua belas, untuk tiga belas simpul.

DENGAN BAHASA SEHARI-HARI. Ambillah peta jalan sebuah provinsi lalu hapuslah jalan-jalannya satu per satu, selama Anda tidak pernah memutus sebuah kota dari sisanya. Berhentilah ketika menghapus satu jalan lagi akan membuat ada yang terkucil. Yang tersisa adalah pohon merentang: setiap kota masih dapat dicapai, dan tidak ada satu jalan pun yang berlebih.

11.4.2 Pohon merentang punya tepat $|V| - 1$ sisi

Fakta ini dipakai terus-menerus — setiap algoritma di dalam bab ini berhenti sesudah menambahkan tepat $|V| - 1$ sisi — sehingga ia layak dibuktikan, bukan sekadar dinyatakan.

TEOREMA 11.3. Misalkan T sebuah pohon dengan $n \geq 1$ simpul. Maka T punya tepat $n - 1$ sisi.

Bukti, dengan induksi atas n.

Kasus basis, $n = 1$. Satu simpul tunggal tanpa sisi bersifat terhubung dan asiklik, jadi ia sebuah pohon, dan ia punya $0 = 1 - 1$ sisi. Benar.

Langkah induksi. Andaikan teoremanya berlaku untuk semua pohon dengan simpul kurang dari n, dan misalkan T sebuah pohon dengan $n \geq 2$ simpul. Kita klaim lebih dahulu bahwa T punya simpul berderajat 1 — sebuah **daun**. Andaikan tidak: setiap simpul berderajat ≥ 2 (ia tidak mungkin berderajat 0, sebab T terhubung dan $n \geq 2$). Maka mulailah dari sembarang simpul lalu berjalanlah, setiap kalinya keluar lewat sisi selain sisi tempat Anda datang; derajat ≥ 2 menjamin sisi semacam itu selalu ada, sehingga jalan itu tidak pernah berhenti. Karena V berhingga, suatu simpul pasti dikunjungi dua kali, dan potongan jalan di antara kedua kunjungan itu adalah sebuah siklus. Tetapi T asiklik. Kontradiksi, jadi sebuah daun ℓ ada.

Hapuslah ℓ beserta satu sisi yang menempel padanya. Hasilnya T' tetap asiklik (menghapus sesuatu tidak mungkin menciptakan siklus) dan tetap terhubung: lintasan mana pun di dalam T antara dua simpul yang bertahan tidak mungkin melewati ℓ, sebab masuk ke ℓ lalu keluar lagi memerlukan dua sisi berbeda pada ℓ, sedangkan ℓ hanya punya satu. Jadi T' adalah pohon dengan $n - 1$ simpul, dan menurut hipotesis induksinya ia punya $n - 2$ sisi. Menambahkan kembali ℓ beserta sisinya memberikan T dengan $(n - 2) + 1 = n - 1$ sisi. ■

Dua akibat jatuh dari situ, dan keduanya akan dipakai:

- **Pohon merentang sebuah graf terhubung dengan $|V|$ simpul punya tepat $|V| - 1$ sisi.** Langsung, sebab pohon merentang adalah pohon atas seluruh $|V|$ simpulnya.
- **Menambahkan sembarang sisi G ke sebuah pohon merentang menciptakan tepat satu siklus.** Pohonnya sudah menghubungkan kedua ujung u dan v lewat sebuah lintasan tunggal; sisi baru $\{u, v\}$ menutup lintasan itu menjadi sebuah siklus, dan karena lintasannya tunggal, siklusnya pun tunggal. Inilah fakta yang akan menjadi sandaran bukti sifat potongan di §11.7.

JEBAKAN. "Tepat $n - 1$ sisi" tidak dengan sendirinya menjadikan sebuah graf pohon. Graf dengan 4 simpul yang terdiri atas sebuah segitiga ditambah satu simpul terpencil punya 3 sisi dan bukan pohon — ia punya siklus dan ia tak terhubung. Anda memerlukan *dua* dari ketiga sifatnya (terhubung, asiklik, $n - 1$ sisi) untuk memperoleh yang ketiga; satu saja tidak pernah cukup.

11.4.3 Ada berapa banyak pohon merentang?

Cukup banyak untuk membuat pencarian paksa menjadi sia-sia, dan hanya fakta itulah yang kita perlukan. Untuk graf **lengkap** dengan n simpul, rumus Cayley [3] memberikan cacahnya secara eksak: n^{n-2}. Untuk $n = 9$, yaitu ukuran graf contoh bab ini, itu berarti $9^7 = 4\,782\,969$ pohon merentang — untuk graf yang dapat Anda gambar di atas tisu. Untuk $n = 20$ jumlahnya $20^{18} \approx 2.6 \times 10^{23}$. Mencacah semuanya lalu memilih yang termurah bukanlah algoritma; itu cara untuk tidak pernah selesai.

11.5 Persoalan Pohon Merentang Minimum

11.5.1 Bobot

DEFINISI. Sebuah **fungsi bobot** pada graf $G = (V, E)$ adalah pemetaan $w: E \to \mathbb{R}$ yang memberikan sebuah bilangan real $w(u, v)$ kepada tiap sisi $\{u, v\}$. **Bobot sebuah upagraf** adalah jumlah bobot sisi-sisinya; khususnya, untuk pohon merentang T,

$$w(T) = \sum_{\{u,v\} \in T} w\,(u, v). \tag{11.1}$$

Jadi latarnya adalah graf tak berarah terhubung $G = (V, E)$ bersama sebuah fungsi bobot yang memberikan biaya, panjang, atau nilai lain kepada tiap sisinya, dan bobot sebuah pohon adalah jumlah Persamaan (11.1). Kata *bobot* sengaja dipilih netral; dalam sebuah penerapan ia bisa berarti meter kabel, rupiah biaya pembangunan, milidetik kelambatan, atau kilometer jalan.

> **DEFINISI.** Sebuah **pohon merentang minimum** (*minimum spanning tree*, MST) dari graf berbobot terhubung G adalah pohon merentang T dari G sedemikian sehingga, dengan bobot diukur menurut Persamaan (11.1), $w(T) \leq w(T')$ untuk setiap pohon merentang T' dari G. **Persoalan pohon merentang minimum** berbunyi: diberi G dan w, carilah satu.

Perhatikan kata sandangnya: *sebuah* pohon merentang minimum. Mungkin ada beberapa yang berbobot sama, dan §11.12 membahas persis kapan hal itu terjadi.

11.5.2 Sebuah persoalan nyata beserta ongkosnya

Inilah persoalannya dalam wujud yang biasanya ia datangi kita. Sebuah paguyuban sembilan desa di Jawa Timur menginginkan air perpipaan. Sebuah survei telah menetapkan, untuk tiap pasang desa yang cukup dekat untuk disambungkan, berapa ratus meter pipa yang diperlukan sambungan itu — medannya membuat sebagian loncatan yang tampak pendek menjadi mahal dan sebagian yang panjang justru murah. Setiap desa harus berakhir tersambung ke jaringannya; tidak penting *bagaimana* air sampai ke sebuah desa, yang penting ia sampai. Pipa memakan biaya per meter. **Sambungan mana yang sebaiknya dibangun?**

Tiga pengamatan mengubahnya menjadi persoalan MST persis.

1. *Jawabannya harus terhubung dan merentang kesembilan desanya* — itulah persyaratannya.
2. *Jawabannya tidak boleh memuat siklus.* Seandainya memuat, satu pipa pada siklus itu dapat dibuang dan setiap desa tetap tersambung — lewat sisa siklusnya — dengan biaya yang benar-benar lebih rendah. Jadi jawaban yang optimal tidak punya siklus. Karena itu ia sebuah pohon merentang.
3. *Di antara pohon-pohon merentang kita menghendaki yang termurah.* Dan itulah definisi MST.

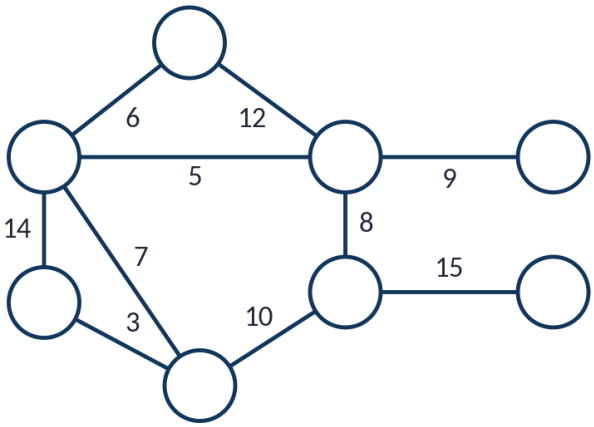
Ketiga baris yang sama berlaku, tanpa perubahan, untuk memasang serat optik antargedung kampus, untuk memilih penerbangan mana yang harus dipertahankan sebuah maskapai supaya jaringannya tetap terhubung, untuk mengawati papan rangkaian dengan solder paling sedikit, dan — di §11.13 — untuk merutekan mobil antar-jemput sebuah binatu.

> **JEBAKAN.** MST meminimumkan bobot **total** jaringannya. Ia *tidak* meminimumkan jarak antara sepasang simpul tertentu. Pada **Gambar 11.5** rute MST dari satu sudut graf ke sudut lainnya bisa jauh lebih panjang daripada sisi langsungnya, sebab sisi langsung itu terlalu mahal untuk dipertahankan. Bila yang Anda inginkan adalah *perjalanan* yang murah, yang Anda inginkan adalah lintasan terpendek, dan itu Bab 13, sedangkan pohon lintasan terpendek umumnya **bukan** MST. Mengacaukan keduanya adalah kekeliruan yang paling sering terjadi pada bahan ini.

11.5.3 Sebuah graf contoh

Gambar 11.5 adalah graf berbobot pertama kita: delapan simpul dan sepuluh sisi, dengan bobot 3, 5, 6, 7, 8, 9, 10, 12, 14, dan 15. Pandanglah ia lalu ajukan pertanyaannya. Sisi berbobot 15 mencuat ke sebuah sudut — tetapi ia satu-satunya sisi yang mencapai simpul di sudut itu,

sehingga ia tidak dapat dibuang, mahal atau tidak. Sisi berbobot 14, sebaliknya, adalah satu dari dua jalan menuju ujung bawahnya, dan jalan yang lain, berbobot 3, jauh lebih murah. MST adalah versi sistematis dari penalaran itu.



Gambar 11.5 Sebuah graf berbobot: graf tak berarah yang terhubung dengan sebuah bobot pada tiap sisinya. Sebagian sisi yang mahal tak terhindarkan (yang 15 adalah satu-satunya jalan menuju simpulnya); sebagian yang tampak murah justru berlebih. Membedakan keduanya adalah seluruh persoalannya.

Untuk penelusuran Prim dan Kruskal yang dikerjakan, kita memakai graf yang sedikit lebih besar, dengan sembilan simpul a sampai i dan empat belas sisi. Kita akan menjumpainya di §11.9.5 dan memakainya untuk segala sesuatu sesudah itu, supaya kedua algoritmanya dapat dibandingkan pada masukan yang identik.

11.5.4 Mengapa pencarian paksa sia-sia, dan apa yang kita perlukan sebagai gantinya

Menurut §11.4.3 pohon merentang dapat berjumlah eksponensial, sehingga pencacahan gugur. Kita juga tidak dapat sekadar mengambil $|V| - 1$ sisi termurah: pada **Gambar 11.5** ketiga sisi termurahnya adalah 3, 5, dan 6, dan tidak ada jaminan ketiganya bahkan membentuk sebuah hutan, apalagi hutan yang merentang. (Pada graf sembilan simpul §11.9.5, sisi (c, i) dan (f, g) berbobot 2, (c, f) berbobot 4, dan (g, i) berbobot 6 membentuk siklus $c - i - g - f - c$, dan keempatnya termasuk delapan sisi termurah; mengambil $|V| - 1$ sisi termurah secara *membuta* menghasilkan sebuah siklus.)

Yang kita perlukan adalah sebuah teorema yang mengizinkan kita berkomitmen pada satu sisi setiap kalinya, secara permanen, dengan jaminan bahwa kita belum menyingkirkan optimumnya. Ada dua teorema semacam itu, dan dua bagian berikutnya membuktikannya. §11.6 membuktikan bahwa persoalannya memiliki **substruktur optimal**, sehingga jawaban parsial dapat dilengkapi tanpa menoleh ke belakang. §11.7 membuktikan **sifat potongan**, yang memberi tahu kita sisi tunggal mana yang aman untuk dikomitmenkan. Bersama-sama keduanya melisensikan sebuah algoritma *greedy* — dan, menurut standar Bab 9, lisensi itulah yang dituntut kerakusan.

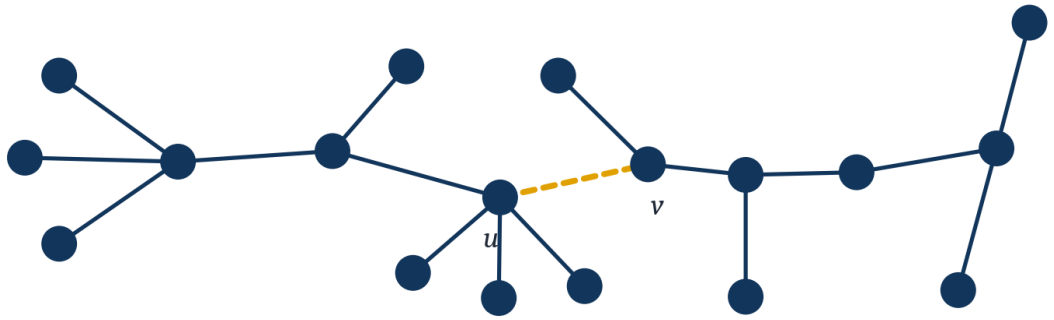
11.6 Substruktur Optimal MST

11.6.1 Klaimnya

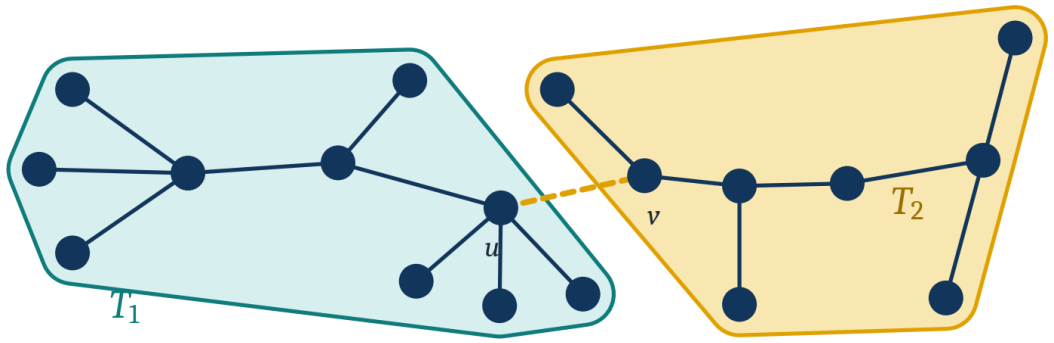
Klaimnya paling baik dinyatakan dengan sebuah gambar. Misalkan T sebuah pohon merentang minimum dari G, dan misalkan (u, v) sembarang sisi T. Hapuslah sisi itu. Karena T sebuah pohon, menghapus satu sisi memutusnya menjadi tepat dua bagian — sebutlah T_1 dan T_2. (Tepat dua:

menghapus sebuah sisi dari pohon tidak mungkin meninggalkannya tetap terhubung, sebab sisi itu adalah lintasan tunggal antara kedua ujungnya; dan ia tidak mungkin memecahnya menjadi tiga, sebab masing-masing dari kedua komponen yang tersisa masih terhubung di dalam dirinya.) Tulislah V_1 dan V_2 untuk himpunan simpulnya, sehingga $V_1 \cup V_2 = V$ dan $V_1 \cap V_2 = \varnothing$.

Gambar 11.6 menggambarkan pembelahan itu, dan **Gambar 11.7** adalah gambar yang sama dengan kedua paruhnya dilingkari dan dinamai.



Gambar 11.6 Sebuah pohon merentang minimum T, dengan salah satu sisinya (u, v) digambar sebagai garis putus-putus. Menghapus sisi itu membelah T menjadi dua bagian.



Gambar 11.7 Gambar 11.6 dengan anotasi: kedua bagian yang tersisa sesudah sisi putus-putus (u, v) dihapus dinamai T_1 (kiri) dan T_2 (kanan). Klaim Teorema 11.4 adalah bahwa masing-masingnya sendiri merupakan pohon merentang minimum bagi upagraf G yang diinduksi oleh himpunan simpulnya sendiri.

Karena sisi-sisi T persis terdiri atas sisi-sisi T_1, sisi-sisi T_2, dan (u, v) sendiri, bobotnya bertambah begitu saja:

$$w(T) = w(u, v) + w(T_1) + w(T_2). \tag{11.2}$$

Sekarang misalkan $G_1 = (V_1, E_1)$ adalah upagraf G yang **diinduksi** oleh V_1 — artinya E_1 adalah setiap sisi G yang kedua ujungnya berada di V_1 — dan definisikan $G_2 = (V_2, E_2)$ dengan cara yang sama.

> **TEOREMA 11.4 (substruktur optimal).** Dengan notasi di atas, T_1 adalah pohon merentang minimum bagi G_1, dan T_2 adalah pohon merentang minimum bagi G_2.

11.6.2 Buktinya

Bukti. Pertama, T_1 *memang* pohon merentang G_1: ia asiklik (upagraf sebuah pohon), ia terhubung (ditunjukkan di atas), dan himpunan simpulnya persis V_1. Jadi satu-satunya pertanyaan adalah apakah ia minimum. Karena simetri, cukup berargumen tentang T_1 saja.

Andaikan ia tidak minimum. Maka ada pohon merentang $T_1{}'$ bagi G_1 dengan

$$w(T_1{}') < w(T_1). \tag{11.3}$$

Bangunlah sebuah upagraf G yang baru dengan merekatkan kembali kedua bagiannya memakai paruh yang lebih murah:

$$T' = T_1{}' \ \cup \ \{(u, v)\} \ \cup \ T_2.$$

Apakah T' pohon merentang G? Himpunan simpulnya $V_1 \cup V_2 = V$, jadi ia merentang. Ia terhubung: $T_1{}'$ menghubungkan segala sesuatu di dalam V_1, T_2 menghubungkan segala sesuatu di dalam V_2, dan sisi (u, v) — yang punya satu ujung di masing-masing sisi, sebab ia sisi yang tadi kita hapus — menyambungkan keduanya. Dan ia asiklik: ia punya

$$(|V_1| - 1) + 1 + (|V_2| - 1) = |V| - 1$$

sisi menurut Teorema 11.3 yang diterapkan pada $T_1{}'$ dan T_2, sedangkan graf terhubung dengan $|V|$ simpul dan $|V| - 1$ sisi adalah sebuah pohon. Jadi T' adalah pohon merentang G.

Berapa bobotnya? Dengan penguraian yang sama seperti Persamaan (11.2), lalu dengan pengandaian Persamaan (11.3),

$$w(T') = w(u, v) + w(T_1{}') + w(T_2) \ < \ w(u, v) + w(T_1) + w(T_2) = w(T).$$

Jadi T' adalah pohon merentang G yang benar-benar lebih murah daripada T. Tetapi T adalah pohon merentang *minimum*. Kontradiksi. Maka $T_1{}'$ semacam itu tidak ada, dan T_1 adalah pohon merentang minimum bagi G_1. ■

> **DENGAN BAHASA SEHARI-HARI.** Inilah **argumen gunting-dan-tempel** dari Bab 9, dan ia layak disebut namanya sebab Anda akan memakainya berulang-ulang. Anda mengandaikan suatu *bagian* dari penyelesaian optimal tidak optimal pada dirinya sendiri; Anda menggunting bagian itu lalu menempelkan yang lebih baik; Anda memeriksa bahwa hasilnya masih penyelesaian yang sah; dan Anda mengamati bahwa hasilnya lebih murah daripada sesuatu yang seharusnya termurah. Kontradiksi itu membuktikan bahwa bagian tadi memang sudah optimal sejak semula.

11.6.3 Yang tidak diberikan oleh substruktur optimal sendirian

Teorema 11.4 perlu tetapi belum cukup. Ia mengatakan bahwa *begitu Anda mengetahui sebuah sisi MST*, persoalannya terbelah bersih menjadi dua persoalan lebih kecil yang saling bebas dan dapat diselesaikan tanpa pernah menoleh ke belakang. Yang justru tidak dikatakannya adalah **sisi mana yang harus diambil lebih dahulu**. Bacalah ia baik-baik: ia dimulai dengan "misalkan (u, v) sembarang sisi T" — ia mengandaikan kita sudah memegang sebuah MST.

Substruktur optimal adalah lisensi untuk bersikap *greedy*; ia bukan aturan *greedy*-nya. Untuk aturannya kita memerlukan teorema jenis lain: yang menyebut sebuah sisi, sebelum MST mana pun diketahui, lalu mengesahkan bahwa suatu MST memuatnya. Itulah sifat potongan, dan itulah teorema terpenting di dalam bab ini.

11.7 Sifat Potongan

Teorema ini sering disebut **teorema pilihan *greedy*** untuk pohon merentang minimum. Segala sesuatu yang menyusul — Prim, Kruskal, Borůvka, dan setiap algoritma MST yang pernah diterbitkan siapa pun — adalah strategi yang berbeda-beda untuk menerapkannya. Ia layak dinyatakan pelan-pelan.

11.7.1 Potongan, sisi yang melintas, dan menghormati sebuah himpunan

Tiga potong kosakata lebih dahulu.

> **DEFINISI.** Sebuah **potongan** (*cut*) dari $G = (V, E)$ adalah partisi simpul-simpulnya menjadi dua bagian yang tidak kosong: sebuah himpunan $S \subseteq V$ dan komplemennya $V - S$. Kita menuliskan potongan itu sebagai $(S, V - S)$.
>
> Sebuah sisi $\{u, v\}$ **melintasi** potongan itu bila tepat satu ujungnya berada di S: yaitu $u \in S$ dan $v \in V - S$, atau sebaliknya.
>
> Sebuah potongan **menghormati** sehimpunan sisi A bila tidak ada sisi A yang melintasinya — setiap sisi A kedua ujungnya berada di sisi yang sama.

> **DENGAN BAHASA SEHARI-HARI.** Gambarlah sebuah gelung tertutup di atas petanya. Desa-desa di dalamnya adalah S, yang di luar adalah $V - S$. Pipa-pipa yang menyeberangi garis pensil Anda adalah sisi yang melintas. Bila Anda menggambar garis itu sedemikian rupa sehingga ia tidak memotong satu pun pipa yang sudah Anda janjikan untuk dibangun, potongan itu *menghormati* janji-janji Anda.

Karena G terhubung dan kedua sisinya tidak kosong, sekurang-kurangnya satu sisi melintasi setiap potongan. Jadi frasa "sisi berbobot minimum yang melintasi potongan itu" selalu menyebut sesuatu. Bila beberapa sisi yang melintas seri sebagai yang minimum, ia menyebut salah satu di antaranya.

> **DEFINISI.** Sisi berbobot minimum yang melintasi sebuah potongan disebut **sisi ringan** (*light edge*) bagi potongan itu.

11.7.2 Teoremanya

Inilah teoremanya:

> **TEOREMA 11.5 (sifat potongan, alias sifat pilihan *greedy* untuk MST).** Misalkan $G = (V, E)$ graf tak berarah berbobot yang terhubung, misalkan $S \subseteq V$ dengan $S \neq \varnothing$ dan $S \neq V$, dan misalkan (u, v) sisi G berbobot minimum yang menghubungkan S dengan $V - S$. Maka (u, v) termasuk ke dalam **suatu** pohon merentang minimum T dari G.

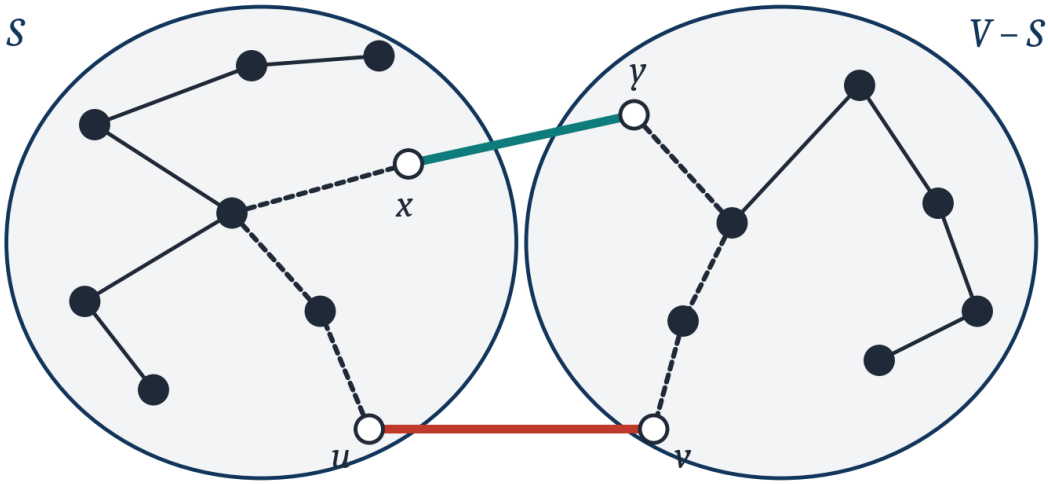
Setiap pembatasnya penting, jadi bacalah satu per satu.

- "**S adalah sembarang himpunan bagian sejati yang tidak kosong.**" Anda boleh menggambar gelungnya di mana saja. Teoremanya berlaku sekaligus untuk seluruh $2^{|V|} - 2$ potongan. Inilah yang membuatnya dapat dipakai: sebuah algoritma boleh memilih potongan yang cocok baginya, dan dua pilihan yang berbeda menghasilkan dua algoritma yang berbeda di bab ini.
- **"sisi berbobot minimum yang menghubungkan S dengan $V - S$."** Minimum *hanya di antara sisi-sisi yang melintas*. Boleh jadi ada sisi yang jauh lebih murah di tempat lain pada grafnya; sisi-sisi itu tidak relevan bagi potongan ini.
- **"termasuk ke dalam suatu pohon merentang minimum."** Bukan ke dalam *setiap* MST. Bila dua sisi yang melintas seri sebagai yang teringan, masing-masing termasuk ke dalam suatu MST,

dan keduanya boleh jadi termasuk ke dalam MST yang berbeda. Teoremanya menjanjikan keberadaan, bukan ketunggalan, dan §11.12 menerangkan perbedaannya.

11.7.3 Buktinya, di dalam sebuah gambar

Gambar 11.8 menggambarkan argumennya, dan buktinya lebih mudah diikuti dengan gambar itu di hadapan Anda daripada tanpanya. Kedua daerah yang diarsir adalah S (kiri) dan $V - S$ (kanan). Garis-garis hitam adalah sisi-sisi sebuah pohon merentang minimum T. Sisi merah di bawah adalah sisi ringan kita (u, v), yang — demi argumennya — *tidak* berada di dalam T. Garis putus-putus adalah lintasan yang disediakan T dari u ke v, dan (x, y), yang digambar hijau toska, adalah sisi pertama pada lintasan itu yang melintas dari S ke $V - S$.



Gambar 11.8 Argumen pertukaran yang membuktikan Teorema 11.5. S dan $V - S$ adalah kedua gumpalannya. Sisi merah (u, v) di bawah adalah sisi ringannya, yang diandaikan *tidak* berada di dalam pohonnya. Garis putus-putus adalah lintasan tunggal yang disediakan pohonnya dari u ke v; sisi toska (x, y) adalah sisi pertama lintasan itu yang melintasi potongannya. Menukar (x, y) dengan (u, v) adalah langkah yang diambil buktinya.

Bukti. Misalkan T sembarang pohon merentang minimum dari G. Bila $(u, v) \in T$ kita sudah selesai: (u, v) termasuk ke dalam sebuah MST, yaitu T. Jadi andaikan

$$(u, v) \notin T.$$

Kita akan mengonstruksi pohon merentang lain T' yang memuat (u, v) dan berbobot tidak lebih daripada T. Karena T minimum, T' dengan demikian juga pasti minimum, dan teoremanya mengikuti.

Langkah 1: temukan lintasannya. T adalah pohon merentang, jadi ia menghubungkan setiap pasang simpul, dan menurut §11.4.2 ia melakukannya lewat lintasan yang **tunggal**. Misalkan P lintasan tunggal di dalam T dari u ke v — garis putus-putus pada **Gambar 11.8**.

Langkah 2: temukan sisi yang melintasnya. Lintasan P bermula di u, yang berada di S, dan berakhir di v, yang berada di $V - S$. Berjalanlah menyusuri P dari u. Anda mulai di dalam S dan berakhir di luarnya, sehingga pada suatu langkah Anda pasti melangkahi batasnya. Misalkan (x, y) sisi **pertama** pada P yang melintasi potongannya, dengan $x \in S$ dan $y \in V - S$. Ia ada, menurut argumen berjalan tadi, dan ia sisi T sebab $P \subseteq T$.

Langkah 3: bandingkan bobotnya. Hanya di sinilah hipotesisnya dipakai. Sisi (x, y) melintasi potongan $(S, V - S)$. Sisi (u, v) adalah sisi **berbobot minimum** yang melintasi potongan itu. Maka

$$w(u, v) \leq w(x, y). \tag{11.4}$$

Langkah 4: lakukan pertukarannya. Definisikan

$$T' = (T \setminus \{(x, y)\}) \cup \{(u, v)\}.$$

Klaim: T' adalah pohon merentang. Menghapus (x, y) dari T membelahnya menjadi dua komponen; satu memuat x, satunya memuat y. Nah, lintasan P tadi berjalan dari u melalui x lalu y dan terus ke v, sehingga u berada di komponen yang sama dengan x (potongan P dari u ke x selamat dari penghapusan itu, sebab (x, y) satu-satunya sisi P yang kita hapus) dan v berada di komponen yang sama dengan y. Maka menambahkan sisi (u, v) menyambung kembali kedua komponennya. Hasilnya terhubung, merentang V, dan punya $|V| - 1$ sisi — satu dihapus, satu ditambahkan — sehingga menurut akibat Teorema 11.3 ia sebuah pohon. Jadi T' adalah pohon merentang G, dan menurut konstruksinya $(u, v) \in T'$.

Langkah 5: hitung harganya. Dengan Persamaan (11.4), gantilah $w(u, v)$ dengan $w(x, y)$ yang lebih besar (atau sama):

$$w(T') = w(T) - w(x, y) + w(u, v) \;\leq\; w(T) - w(x, y) + w(x, y) = w(T).$$

Jadi $w(T') \leq w(T)$. Tetapi T adalah pohon merentang minimum, sehingga tidak ada pohon merentang yang berbobot kurang daripada T; maka $w(T') = w(T)$, dan T' sendiri adalah pohon merentang minimum. Ia memuat (u, v). ■

Versi yang lebih singkat dari Langkah 5 berbunyi "ini memperbaiki T — kontradiksi, sebab T seharusnya MST". Ungkapan itu dituju kepada kasus tegasnya: bila $w(u, v) < w(x, y)$ maka $w(T') < w(T)$ secara tegas, yang langsung bertentangan dengan keminimuman T. Versi di atas menangani kasus seri juga, dan itulah sebabnya ia berakhir dengan "termasuk ke dalam *suatu* MST" alih-alih "termasuk ke dalam *setiap* MST". Kedua bacaan itu benar; yang berhati-hati lebih berguna, sebab graf yang nyata memang punya bobot yang seri.

11.7.4 Membacanya sekali lagi, pelan-pelan

Buktinya pendek tetapi ia sedang mengerjakan sesuatu yang halus, jadi inilah ia dengan kata-kata biasa.

Kita mencemaskan bahwa berkomitmen pada sisi lintas termurah mungkin sebuah kekeliruan — bahwa optimum yang sejati memakai sisi lintas yang lain, dan dengan menyambar sisi kita, kita sudah merusak segalanya. Buktinya berkata: tidak. Andaikan optimum T menghindari sisi kita (u, v). Maka T toh tetap harus pergi dari u ke v dengan suatu cara, dan rute apa pun yang ditempuhnya harus melintasi batas itu sekurang-kurangnya sekali. Temukan tempat ia melintas. Sisi lintas itu tidak mungkin lebih murah daripada sisi kita — sisi kitalah sisi lintas termurah yang ada. Jadi ambillah sisi kita sebagai ganti sisi mereka. Pohonnya tetap sebuah pohon, sebab kita memotongnya di satu tempat lalu menyambungnya persis di tempat kita memotong. Dan tagihannya tidak naik. Karena itu ada pohon optimal yang memuat sisi kita, dan menyambarnya tadi memang aman.

> **RINGKASAN SATU KALIMAT.** *Cara termurah untuk menyeberangi sebuah batas selalu menjadi bagian dari suatu jaringan termurah, sebab jaringan mana pun yang lain toh harus menyeberangi batas itu di suatu tempat, dan Anda selalu dapat menukar seberangan mereka dengan seberangan Anda tanpa merusaknya dan tanpa membayar lebih.*

11.7.5 Akibat yang sungguh-sungguh dipakai algoritmanya

Teorema 11.5 berbicara tentang papan tulis yang masih kosong. Kedua algoritma di bab ini sudah berkomitmen pada sejumlah sisi dan memerlukan teoremanya supaya tetap berlaku. Inilah versi yang mereka pakai.

> **AKIBAT 11.6.** Misalkan A himpunan bagian dari sisi-sisi suatu pohon merentang minimum dari G. Misalkan $(S, V - S)$ sembarang potongan yang **menghormati** A, dan misalkan (u, v) sisi ringan bagi potongan itu. Maka $A \cup \{(u, v)\}$ juga merupakan himpunan bagian dari sisi-sisi suatu pohon merentang minimum dari G.

Sketsa bukti. Misalkan $T \supseteq A$ sebuah MST. Bila $(u, v) \in T$, selesai. Bila tidak, jalankan pertukaran §11.7.3 untuk memperoleh $T' = (T \setminus \{(x, y)\}) \cup \{(u, v)\}$, yang merupakan MST yang memuat (u, v). Kita harus memeriksa bahwa $A \subseteq T'$, yaitu bahwa sisi yang kita buang tadi bukan salah satu sisi kita. Tetapi (x, y) melintasi potongannya, sedangkan potongannya menghormati A, sehingga tidak ada sisi A yang melintasinya — maka $(x, y) \notin A$. Karena itu $A \cup \{(u, v)\} \subseteq T'$. ■

Baris terakhir itu persis alasan kata *menghormati* didefinisikan. Ia adalah jaminan bahwa pertukarannya tidak pernah mengorbankan sisi yang sudah kita janjikan untuk dibangun.

11.7.6 Sebuah pemeriksaan yang dikerjakan

Ambillah graf sembilan simpul bab ini (digambar pada **Gambar 11.9** dan dipakai sejak §11.9 dan seterusnya) lalu ambil $S = \{a\}$. Sisi-sisi yang melintasi potongan ini adalah $\{a, b\}$ berbobot 4 dan $\{a, h\}$ berbobot 8. Sisi ringannya adalah $\{a, b\}$, berbobot 4. Teorema 11.5 menjanjikan bahwa suatu MST memuat $\{a, b\}$ — dan memang kedua MST graf itu memuatnya (§11.12.3 menemukan keduanya).

Sekarang ambil $S = \{g, h\}$. Sisi-sisi yang melintas adalah $\{a, h\}$=8, $\{b, h\}$=11, $\{h, i\}$=7, $\{g, i\}$=6, $\{f, g\}$=2. Sisi ringannya adalah $\{f, g\}$, berbobot 2 — dan ia berada di dalam kedua MST-nya. Terakhir ambil $S = \{d\}$: sisi yang melintas $\{c, d\}$=7, $\{d, e\}$=9, $\{d, f\}$=14; sisi ringannya $\{c, d\}$=7, sekali lagi berada di dalam kedua MST-nya. Tiga potongan yang berbeda, tiga komitmen yang aman, tanpa satu pun MST dihitung. Itulah kekuatan teoremanya: ia mengesahkan sisi satu per satu, dari keterangan yang sepenuhnya lokal.

11.8 Algoritma MST Generik

11.8.1 Sisi aman

Akibat 11.6 menyarankan sebuah nama bagi jenis sisi yang boleh kita komitmenkan.

> **DEFINISI.** Misalkan A sehimpunan sisi yang termuat di dalam suatu pohon merentang minimum dari G. Sebuah sisi $(u, v) \notin A$ disebut **aman bagi** A bila $A \cup \{(u, v)\}$ masih termuat di dalam suatu pohon merentang minimum dari G.

Dalam satu frasa, *safe edge* (sisi aman) adalah sisi yang tidak merusak sifat A, dan sifat yang dimaksud persis adalah *dapat diperluas menjadi sebuah MST*. Akibat 11.6 dengan demikian adalah sebuah mesin penghasil sisi aman: **carilah potongan yang menghormati A, lalu ambil sisi ringannya.**

11.8.2 Algoritma generiknya

Dari situ lahirlah **Pseudokode 11.3**, kerangka yang diisi oleh kedua algoritma bab ini:

Pseudokode 11.3 Algoritma pohon merentang minimum generik: tumbuhkan himpunan sisi A satu sisi aman setiap kalinya.

```
GENERIC-MST(G, w)
1  A = {}                                  // memuat sisi-sisi yang termasuk dalam sebuah
MST
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A union {(u, v)}
5  return A
```

Lima baris, dan hanya baris 3 yang merupakan kerja yang sesungguhnya. Baris 1, 4, dan 5 adalah pembukuan, sedangkan pengujian baris 2 sekadar "sudah adakah $|V| - 1$ sisi?", menurut Teorema 11.3.

11.8.3 Invarian gelungnya

> **INVARIAN.** *Sebelum tiap putaran gelung `while`, A merupakan himpunan bagian dari sisi-sisi suatu pohon merentang minimum dari G.*

Inisialisasi. Sebelum putaran pertama $A = \varnothing$, dan himpunan kosong adalah himpunan bagian dari sisi-sisi MST mana pun, yang sekurang-kurangnya satu di antaranya ada (graf berhingga yang terhubung punya berhingga banyak pohon merentang, sehingga ada satu yang berbobot minimum).

Pemeliharaan. Badannya menambahkan sebuah sisi yang aman bagi A, dan "aman" didefinisikan tepat sebagai syarat yang menjaga invariannya tetap benar.

Terminasi. Gelungnya berakhir ketika A membentuk sebuah pohon merentang. Pada saat itu A adalah pohon merentang *sekaligus* himpunan bagian dari sisi-sisi suatu MST T. Sebuah pohon merentang punya $|V| - 1$ sisi, dan demikian pula T; sehimpunan $|V| - 1$ sisi yang termuat di dalam sehimpunan $|V| - 1$ sisi adalah himpunan itu sendiri. Jadi $A = T$, dan A adalah pohon merentang minimum. ■

Dan gelungnya memang berakhir: tiap putaran menambahkan tepat satu sisi, sehingga sesudah $|V| - 1$ putaran A punya $|V| - 1$ sisi. Apakah ia juga *membentuk pohon merentang* pada saat itu? Ya — invariannya mengatakan A termuat di dalam sebuah pohon merentang dengan $|V| - 1$ sisi, sehingga pada ukuran itu ia sama dengan pohon tersebut. Dan baris 3 selalu dapat menemukan sisi aman selama $|A| < |V| - 1$: ambillah sembarang potongan yang memisahkan dua komponen A (potongan semacam itu ada, sebab A saat itu belum merentang), dan Akibat 11.6 menyediakan sisi ringannya.

11.8.4 Versi yang lebih khusus

Kerangka itu langsung diperhalus dengan menyulihkan Akibat 11.6 ke tempat kata "aman", yang memberi **Pseudokode 11.4**:

Pseudokode 11.4 Algoritma generik dengan kata "aman" diganti Akibat 11.6: pilih potongan yang menghormati A lalu ambil sisi ringannya.

```
MORE-SPECIFIC-MST(G, w)
1    A = {}                                // memuat sisi-sisi yang termasuk dalam sebuah
MST
2    while A does not form a spanning tree
3.1      make a cut (S, V-S) of G that respects A
3.2      take the min-weight edge (u, v) connecting S to V-S
4        A = A union {(u, v)}
5    return A
```

Kini ini adalah algoritma dengan tepat satu derajat kebebasan: **bagaimana Anda memilih potongannya?** Baris 3.1 adalah satu-satunya keputusan yang tersisa, dan ia seluruh perbedaan antara kedua algoritma yang masih tersisa di bab ini.

11.8.5 Dua cara memilih potongannya

Tabel 11.6 Kedua algoritma bab ini adalah algoritma yang sama dengan dua jawaban berbeda atas baris 3.1 `MoreSpecific-MST`. Selebihnya — bukti kebenarannya, cacah $|V| - 1$ sisinya, invarian gelungnya — dipakai bersama.

	Prim–Jarník	Kruskal
Rupa A	satu pohon tunggal, tumbuh dari sebuah akar r	sebuah hutan berisi beberapa pohon
Potongan yang dipilih	S = simpul-simpul yang sudah berada di pohonnya	S = simpul-simpul komponen yang memuat salah satu ujung sisi calon berikutnya
Apakah ia menghormati A?	ya: tiap sisi A kedua ujungnya di dalam pohonnya	ya: tiap sisi A terletak seluruhnya di dalam satu komponen
Cara sisi ringannya ditemukan	sebuah antrean prioritas atas simpul di luar S	urutkan semua sisi sekali, lalu uji satu per satu
Digerakkan oleh	simpul	sisi

Tabel 11.6 adalah tesis untuk sisa bab ini. Bacalah ia sekali lagi sesudah §11.10 dan ia akan berbicara lebih banyak.

11.9 Prim–Jarník

11.9.1 Gagasannya: satu awan yang tumbuh

Algoritma ini ditemukan oleh Vojtěch Jarník pada 1930 dan secara terpisah oleh Robert Prim pada 1957 [4], [5]; buku ini menyebutnya Prim–Jarník, dan itu adil bagi keduanya. (Otakar Borůvka sebenarnya sudah menyelesaikan persoalan ini pada 1926, untuk kelistrikan Moravia; Graham dan Hell [6] mengisahkan riwayat siapa menemukan apa dan kapan, yang ternyata ruwet.) Algoritma ini dapat dilukiskan dalam empat baris, dan baris-baris itu layak dipaparkan sebab di dalamnya terkandung seluruh rancangannya:

Algoritma berbasis simpul. Menumbuhkan satu pohon T, satu simpul setiap kalinya. Sebuah awan menyelimuti bagian T yang sudah dihitung. Labelilah simpul v di luar awan itu dengan $key[v]$ — bobot minimum sebuah sisi yang menghubungkan v ke sebuah simpul di dalam awannya, dengan $key[v] = \infty$ bila sisi semacam itu tidak ada.

Jadi A selalu berupa satu pohon tunggal. Potongannya selalu potongan yang itu-itu juga: S = awannya, $V - S$ = selebihnya. Potongan itu menghormati A dengan sendirinya, sebab tiap sisi A kedua ujungnya berada di dalam awan. Dan sisi ringan baginya adalah sisi termurah dari awannya ke luar — dan untuk itulah key ada.

11.9.2 Kedua lariknya: key dan π

> **DEFINISI.** Untuk tiap simpul v yang belum berada di dalam pohonnya, $key[v]$ adalah bobot sisi termurah yang menghubungkan v ke *sembarang* simpul yang sudah berada di dalam pohonnya, dan $\pi[v]$ adalah simpul pohon di ujung seberang sisi itu. Bila belum ada sisi yang menghubungkan v ke pohonnya, $key[v] = \infty$ dan $\pi[v] = \text{NIL}$.

Dua invarian membuat algoritmanya bekerja, dan keduanya mudah dilihat:

- **Minimum key atas semua simpul di luar pohonnya adalah bobot sisi ringan bagi potongan itu.** Sebab setiap sisi yang melintas adalah sisi dari suatu simpul luar v ke pohonnya, dan $key[v]$ adalah sisi semacam itu yang termurah di v; meminimumkan atas v berarti meminimumkan atas semua sisi yang melintas.
- **Pasangan $(\pi[v], v)$ bagi v yang meminimumkan itu *adalah* sisi ringan tersebut.** Menurut definisi π.

Jadi "keluarkan simpul yang kuncinya terkecil" dan "tambahkan sisi $(\pi[u], u)$" bersama-sama melaksanakan satu putaran `MoreSpecific-MST`. Itulah seluruh algoritmanya.

11.9.3 Algoritmanya

Pseudokode 11.5 memberikan prosedurnya.

Pseudokode 11.5 Algoritma Prim–Jarník: $key[v]$ adalah bobot sisi termurah dari v ke dalam pohon yang sedang tumbuh, dan $\pi[v]$ adalah ujung lainnya.

```
MST-PRIM(G, w, r)
01  Q = G.V                          // Q: simpul-simpul yang belum masuk T
02  for each u in Q
03      key[u] = INFINITY
04  key[r] = 0
05  pi[r] = NIL
06  while Q is not empty
07      u = EXTRACT-MIN(Q)           // u menjadi bagian dari T
08      for each v in Adj[u]
09          if v in Q and w(u, v) < key[v]
10              pi[v] = u
11              key[v] = w(u, v)
```

Baris 01–05 menyiapkan: semuanya berada di luar pohonnya, semuanya tak terjangkau (∞) kecuali akar r yang dipilih, yang diberi kunci 0 supaya ia yang pertama dikeluarkan. Baris 07 mengambil simpul luar termurah lalu memasukkannya. Baris 08–11 kemudian mengerjakan **satu-satunya** hal lain yang perlu dikerjakan: kini setelah u berada di dalam, sebagian simpul luar mungkin punya rute masuk ke pohonnya yang lebih murah daripada sebelumnya — yaitu lewat u. Baris 09–11 memperbarui simpul-simpul itu, dan hanya itu. Pengujian `v in Q` sangat penting: begitu v berada di dalam, kuncinya menjadi riwayat dan tidak boleh disentuh lagi.

> **JEBAKAN.** Syarat pada baris 09 adalah $w(u, v) < key[v]$, lebih kecil secara tegas. Bukan $\leq$, dan bukan pembandingan terhadap $key[u]$ atau terhadap suatu jumlah berjalan. Prim bukan Dijkstra: di sini tidak ada panjang lintasan yang terakumulasi. $key[v]$ adalah bobot **satu** sisi. Menuliskan `key[u] + w(u,v) < key[v]` mengubah Prim menjadi Dijkstra dan memberi Anda pohon lintasan terpendek, yang merupakan pohon yang berbeda. Soal 11.8 membuat perbedaan itu nyata pada graf tiga simpul.

Perhatikan pula bahwa algoritmanya tidak pernah membangun sehimpunan sisi secara eksplisit. MST-nya tercatat di dalam π: ketika gelungnya selesai, pohonnya adalah $\{(\pi[v], v) : v \in V, v \neq r\}$, yang menurut konstruksinya punya $|V| - 1$ sisi.

11.9.4 Antrean prioritas

Baris 07 memerlukan kunci terkecil, dan baris 10–11 mengubah kunci. Abstraksi yang menopang keduanya adalah antrean prioritas:

> **DEFINISI.** Sebuah **antrean prioritas** (*priority queue*) memelihara sehimpunan S berisi elemen, yang masing-masingnya punya nilai terkait yang disebut **kunci**, dan mendukung: • `BuildPQ(S)` — inisialisasi antreannya supaya memuat elemen-elemen S; • `ExtractMin(S)` — kembalikan lalu buang elemen S yang kuncinya terkecil; • `ModifyKey(S, x, newkey)` — ubah kunci x (di sini selalu *menurunkannya*, sehingga ia sering disebut `DecreaseKey`).

Sebuah **tumpukan biner** (*binary heap*) — struktur dari Bab 5 — mengimplementasikan ketiganya: `BuildPQ` dalam $O(n)$, serta `ExtractMin` dan `ModifyKey` masing-masing dalam $O(\lg n)$. Itulah pilihan bakunya, dan pilihan yang diambil program di bawah.

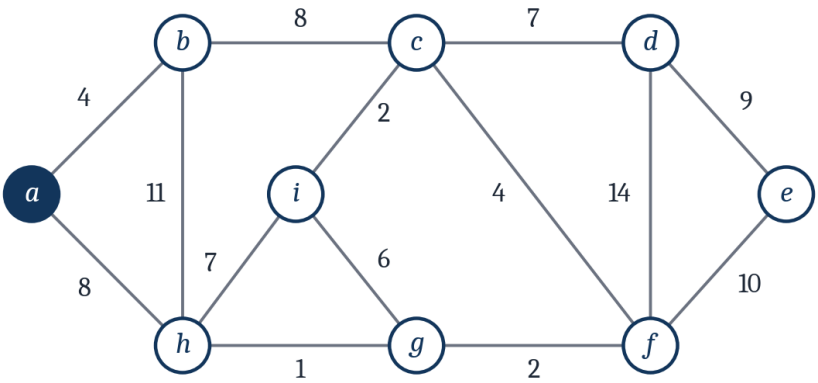
11.9.5 Contohnya ditelusuri, langkah demi langkah

Penelusurannya memakan sembilan panel, yang direproduksi di sini satu per langkah. Grafnya punya sembilan simpul $a \ldots i$ dan empat belas sisi; simpul hitam pejal sudah berada di dalam pohonnya, simpul berongga masih di dalam Q, dan sorotan kelabu menandai sisi-sisi yang sudah dikomitmenkan. Akarnya adalah a.

Kita akan membaca antrean prioritasnya pada tiap langkah sebagai senarai entri `simpul:kunci/π`, terurut menurut kuncinya.

Langkah 1. Q memuat kesembilan simpul dengan $key[a] = 0$ dan setiap kunci lainnya ∞. `ExtractMin` mengembalikan a. Tidak ada sisi yang ditambahkan — a adalah akarnya. Kini kedua tetangga a diperbarui: $key[b] \leftarrow 4, \pi[b] \leftarrow a$, dan $key[h] \leftarrow 8, \pi[h] \leftarrow a$.

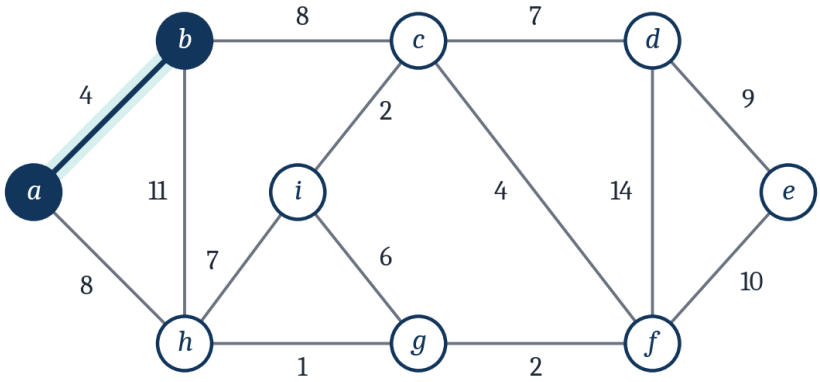
$$Q = \{\, b{:}\,4/a,\ \ h{:}\,8/a,\ \ c{:}\,\infty,\ \ d{:}\,\infty,\ \ e{:}\,\infty,\ \ f{:}\,\infty,\ \ g{:}\,\infty,\ \ i{:}\,\infty \,\}$$



Gambar 11.9 Langkah 1 penelusuran Prim: pohonnya baru berupa akar a. Ini sekaligus grafnya sendiri — sembilan simpul, empat belas sisi, bobot 1 sampai 14 — dan inilah graf yang dipakai untuk setiap penelusuran di sisa bab ini.

Langkah 2 (**Gambar 11.10**). Kunci terkecil di dalam Q adalah $b{:}\,4$. Keluarkan b, lalu komitmenkan sisi $(\pi[b], b) = (a, b)$ berbobot 4. Jumlah berjalan 4. Kini perbarui dari b: c punya $key[c] = \infty > 8 = w(b, c)$, sehingga $key[c] \leftarrow 8, \pi[c] \leftarrow b$; sedangkan h sudah punya $key[h] = 8$, sementara $w(b, h) = 11$ tidak lebih kecil, sehingga h dibiarkan.

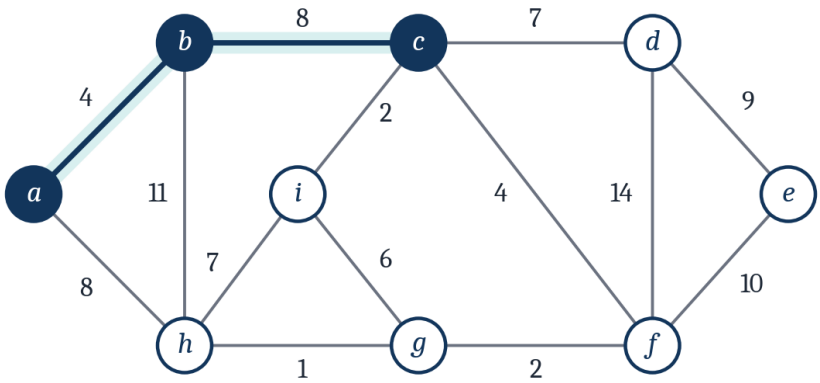
$$Q = \{\, c{:}\,8/b,\ \ h{:}\,8/a,\ \ d{:}\,\infty,\ \ e{:}\,\infty,\ \ f{:}\,\infty,\ \ g{:}\,\infty,\ \ i{:}\,\infty \,\}$$



Gambar 11.10 Langkah 2: sisi ringan bagi potongan $S = \{a\}$ adalah (a, b) berbobot 4, sehingga b bergabung ke pohonnya.

Langkah 3. Kini ada **seri**: c dan h sama-sama berkunci 8. Salah satunya boleh diambil — Teorema 11.5 mengesahkan keduanya — dan **Gambar 11.11** mengambil c. Keluarkan c, komitmenkan (b, c) berbobot 8, total 12. Memperbarui dari c: $key[i] \leftarrow 2/c$, $key[f] \leftarrow 4/c$, $key[d] \leftarrow 7/c$.

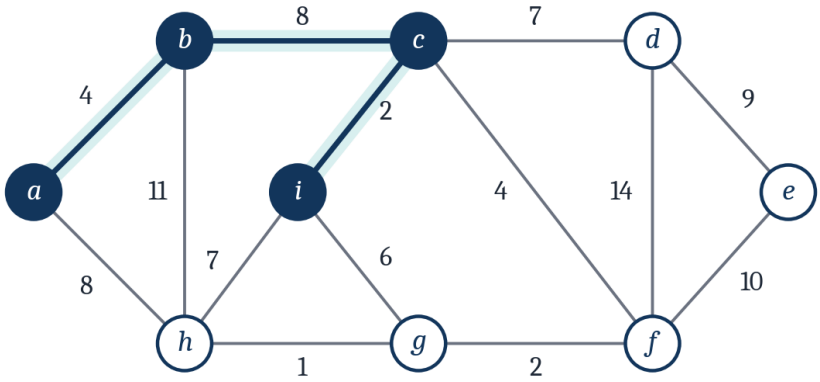
$$Q = \{\, i\colon 2/c,\ \ f\colon 4/c,\ \ d\colon 7/c,\ \ h\colon 8/a,\ \ e\colon \infty,\ \ g\colon \infty \,\}$$



Gambar 11.11 Langkah 3: dengan $S = \{a, b\}$ sisi-sisi yang melintas adalah $(b, c) = 8$, $(a, h) = 8$, dan $(b, h) = 11$. Dua di antaranya seri sebagai yang teringan; penelusurannya mengambil (b, c).

Langkah 4 (Gambar 11.12). Keluarkan i: 2, komitmenkan (c, i) berbobot 2, total 14. Dari i: $key[g] \leftarrow 6/i$ dan $key[h]$ membaik dari $8/a$ menjadi $7/i$, sebab $w(i, h) = 7 < 8$. Amati pembaruan terakhir itu baik-baik — di situlah algoritmanya mengubah pikirannya tentang *bagaimana* mencapai h, tanpa pernah mengubah pikirannya tentang sisi yang sudah dikomitmenkan.

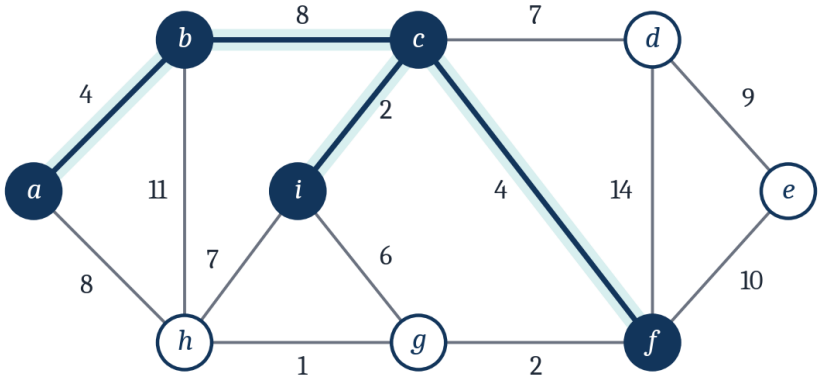
$$Q = \{\, f\colon 4/c,\ \ g\colon 6/i,\ \ d\colon 7/c,\ \ h\colon 7/i,\ \ e\colon \infty \,\}$$



Gambar 11.12 Langkah 4: (c, i) berbobot 2 adalah jalan keluar termurah dari $\{a, b, c\}$, sehingga i bergabung. Ini juga langkah saat kunci h *diperbaiki*, dari 8 menjadi 7.

Langkah 5 (Gambar 11.13). Keluarkan f: 4, komitmenkan (c, f) berbobot 4, total 18. Dari f: $key[g]$ membaik dari $6/i$ menjadi $2/f$, dan $key[e] \leftarrow 10/f$. ($w(f, d) = 14$ lebih buruk daripada $key[d] = 7$, sehingga d tidak tersentuh.)

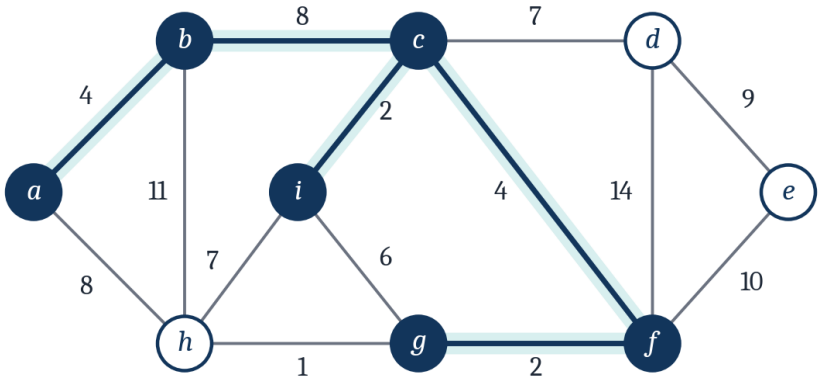
$$Q = \{\, g\colon 2/f,\ \ d\colon 7/c,\ \ h\colon 7/i,\ \ e\colon 10/f \,\}$$



Gambar 11.13 Langkah 5: (c, f) berbobot 4 memasukkan f. Pohonnya kini $\{a, b, c, i, f\}$ dan kunci g turun dari 6 menjadi 2.

Langkah 6 (Gambar 11.14). Keluarkan g: 2, komitmenkan (f, g) berbobot 2, total 20. Dari g: $key[h]$ membaik lagi, dari $7/i$ menjadi $1/g$, sebab $w(g, h) = 1$.

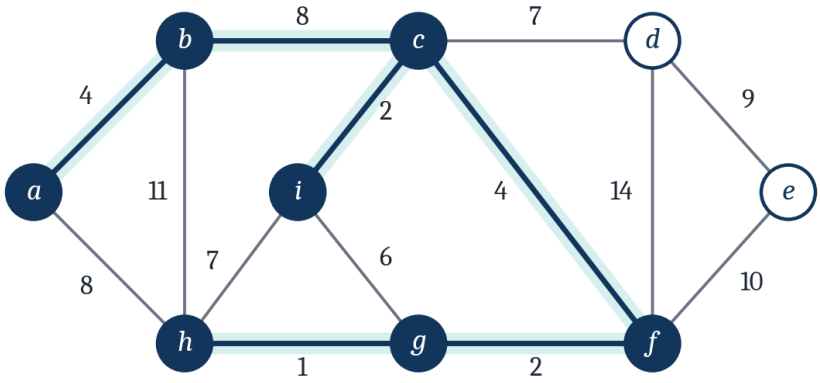
$$Q = \{\, h\colon 1/g,\ \ d\colon 7/c,\ \ e\colon 10/f \,\}$$



Gambar 11.14 Langkah 6: (f, g) berbobot 2 memasukkan g. Simpul h kini sudah tiga kali kuncinya diturunkan — 8, lalu 7, lalu 1 — dan persis untuk itulah `DecreaseKey` ada.

Langkah 7 (Gambar 11.15). Keluarkan h: 1, komitmenkan (g, h) berbobot 1, total 21. Dari h: tetangga-tetangganya a, b, i semuanya sudah berada di dalam pohonnya, sehingga tidak ada yang berubah.

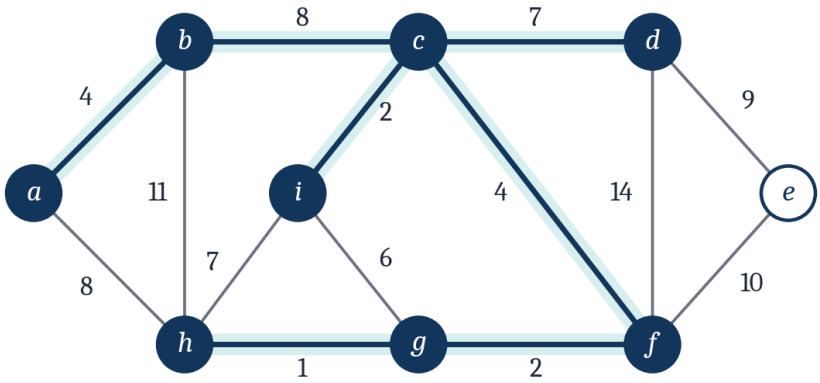
$$Q = \{\, d\colon 7/c,\ \ e\colon 10/f \,\}$$



Gambar 11.15 Langkah 7: sisi termurah di seluruh grafnya, (g, h) berbobot 1, baru diambil sekarang — pada langkah ketujuh. Prim mengambil sisi menurut urutan awannya yang tumbuh menjangkau mereka, bukan menurut urutan bobot.

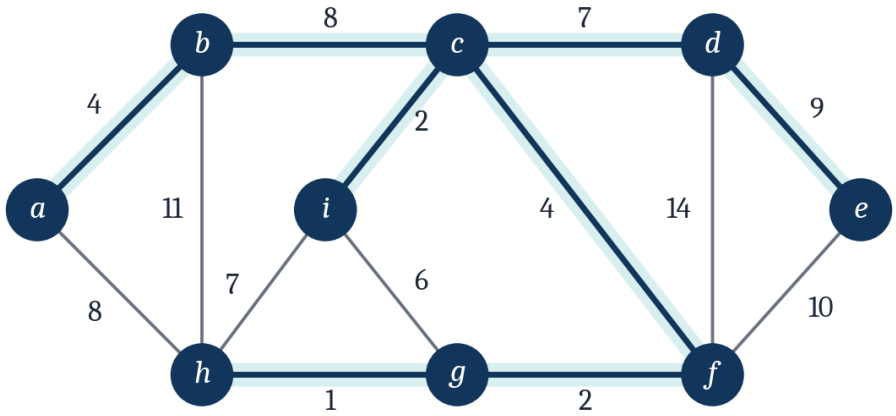
Langkah 8 (Gambar 11.16). Keluarkan d: 7, komitmenkan (c, d) berbobot 7, total 28. Dari d: $key[e]$ membaik dari $10/f$ menjadi $9/d$, sebab $w(d, e) = 9 < 10$.

$$Q = \{\, e\colon 9/d \,\}$$



Gambar 11.16 Langkah 8: (c, d) berbobot 7 memasukkan d, dan kunci e membaik dari 10 menjadi 9.

Langkah 9. Keluarkan e: 9, komitmenkan (d, e) berbobot 9, total 37. Q kosong dan algoritmanya berhenti.



Gambar 11.17 Langkah 9 — gambar terakhirnya. Kesembilan simpulnya hitam dan kedelapan sisi yang tersorot membentuk pohon merentang minimumnya: (a, b), (b, c), (c, i), (c, f), (f, g), (g, h), (c, d), (d, e), dengan bobot total 37.

Seluruh jalannya dalam satu halaman:

Tabel 11.7 Jalan lengkap Prim–Jarník dari **Gambar 11.9** sampai **Gambar 11.17**, dengan antrean prioritasnya dicetak sesudah tiap langkah sebagai `simpul:kunci/π`. Sembilan pengeluaran, delapan sisi, bobot total 37. Barisan bobot yang dikomitmenkan adalah 4, 8, 2, 4, 2, 1, 7, 9 — tidak terurut, sebab Prim tidak mengurutkan.

Langkah	Keluarkan	*key*	Sisi yang dikomitmenkan	Total	*Q* Sesudahnya
1	a	0	— (akar)	0	b:4/a, h:8/a, sisanya ∞
2	b	4	(a, b)	4	c:8/b, h:8/a, sisanya ∞
3	c	8	(b, c)	12	i:2/c, f:4/c, d:7/c, h:8/a, e:∞, g:∞
4	i	2	(c, i)	14	f:4/c, g:6/i, d:7/c, h:7/i, e:∞
5	f	4	(c, f)	18	g:2/f, d:7/c, h:7/i, e:10/f
6	g	2	(f, g)	20	h:1/g, d:7/c, e:10/f
7	h	1	(g, h)	21	d:7/c, e:10/f
8	d	7	(c, d)	28	e:9/d
9	e	9	(d, e)	37	(kosong)

YANG PERLU DIPERHATIKAN. Bobot yang dikomitmenkan Prim adalah 4, 8, 2, 4, 2, 1, 7, 9 — kacau-balau urutannya. Prim tidak pernah memandang grafnya sebagai keseluruhan; pada tiap langkah ia hanya memandang batas awannya sendiri. Sisi terberat yang diambilnya (9) diambil paling akhir, dan yang teringan (1) diambil ketujuh, semata-mata karena saat itulah awannya menjangkau mereka.

11.9.6 Mengapa Prim benar

Teorema. `MST-Prim` mengembalikan sebuah pohon merentang minimum.

Bukti. Kita tunjukkan bahwa Prim adalah `MoreSpecific-MST` dengan pilihan potongan tertentu, lalu kita kutip §11.8.3.

Misalkan A himpunan sisi $\{(\pi[v], v) : v \notin Q, v \neq r\}$ — sisi-sisi yang sudah dikomitmenkan — dan misalkan $S = V \setminus Q$ himpunan simpul yang sudah dikeluarkan. Tiga hal yang perlu diperiksa:

1. *A adalah pohon yang merentang S.* Dengan induksi. Pada mulanya $S = \varnothing$ dan $A = \varnothing$. Ketika u dikeluarkan, sisi yang ditambahkan adalah $(\pi[u], u)$, dan $\pi[u] \in S$ (algoritmanya hanya pernah menyetel $\pi[v] \leftarrow u$ pada saat u masuk ke S), sementara u belum berada di S. Jadi satu simpul

baru dan satu sisi baru ditempelkan pada pohon yang sudah ada: tetap sebuah pohon, kini merentang $S \cup \{u\}$.

2. *Potongan* $(S, V - S)$ *menghormati* A. Menurut (1) tiap sisi A kedua ujungnya berada di S, sehingga tidak ada sisi A yang melintas.
3. *Sisi yang ditambahkan adalah sisi ringan bagi potongan itu.* Menurut kedua invarian §11.9.2: $key[v]$ adalah sisi termurah dari v ke S, sehingga $\min_{v \in Q} key[v]$ adalah sisi termurah yang melintasi potongannya, dan `ExtractMin` mengembalikan v yang mencapainya.

Invarian-invarian itu sendiri berlaku sebab baris 08–11 memulihkannya segera sesudah tiap pengeluaran: satu-satunya sisi lintas yang baru sesudah u bergabung ke S adalah sisi-sisi yang keluar dari u, dan masing-masing dibandingkan dengan $key[v]$ yang sedang berlaku.

Jadi setiap sisi yang ditambahkan Prim aman bagi A menurut Akibat 11.6, dan menurut argumen terminasi §11.8.3 nilai akhir A — yang punya $|V| - 1$ sisi dan merupakan pohon merentang — adalah pohon merentang minimum. ■

11.9.7 Waktu jalannya

Hitunglah operasi antrean prioritasnya, sebab di situlah seluruh ongkosnya berada.

- `ExtractMin` (baris 07) dipanggil tepat sekali tiap simpul: sebanyak $|V|$ kali.
- `ModifyKey` (baris 11) dipanggil paling banyak sekali tiap pemeriksaan sebuah sisi. Sepanjang seluruh jalannya, baris 08–09 memeriksa tiap senarai ketetanggaan sekali, dan $\sum_u |Adj[u]| = 2|E|$ pada graf tak berarah. Jadi `ModifyKey` berjalan $O(E)$ kali.

Dengan menulis $T(\text{op})$ untuk biaya satu operasi, totalnya adalah

$$\text{Waktu} = |V| \cdot T(\text{ExtractMin}) + O(E) \cdot T(\text{ModifyKey}). \tag{11.5}$$

Dengan **tumpukan biner**, kedua operasinya berongkos $O(\lg V)$, sehingga Persamaan (11.5) menjadi

$$\text{Waktu} = O(V \lg V) + O(E \lg V) = O((V + E) \lg V) = O(E \lg V), \tag{11.6}$$

langkah terakhirnya karena G terhubung, sehingga $|E| \geq |V| - 1$ dan suku E yang mendominasi.

11.9.8 Tiga antrean dibandingkan

Pilihan antrean prioritasnya mengubah batas Persamaan (11.5), sebagaimana diperlihatkan **Tabel 11.8**:

Tabel 11.8 Waktu jalan Prim dengan tiga antrean prioritas. Baris *Fibonacci heap* adalah alasan struktur itu ada: ia membuat `DecreaseKey` menjadi gratis dalam arti teramortisasi, dan `DecreaseKey` adalah operasi yang paling sering dilakukan Prim.

Q	T(ExtractMin)	T(DecreaseKey)	Total
larik tak terurut	$O(V)$	$O(1)$	$O(V^2)$
tumpukan biner	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ teramortisasi	$O(1)$ teramortisasi	$O(V \lg V + E)$

Membaca tabelnya sebagai nasihat rekayasa:

- **Larik tak terurut** paling baik ketika grafnya **padat**. Dengan $|E| = \Theta(V^2)$, $O(V^2)$ mengalahkan $O(E \lg V) = O(V^2 \lg V)$. Ia juga hal yang paling sederhana untuk ditulis, dan untuk $|V|$ sampai

beberapa ribu ia sering paling cepat menurut jam dinding sebab ia menyentuh memorinya secara linear.

- **Tumpukan biner** paling baik untuk kasus jarang yang biasa, $|E| = O(V)$ sampai $O(V \lg V)$, dan itulah yang diberikan `std::priority_queue` kepada Anda secara cuma-cuma.
- ***Fibonacci heap*** mencapai batas yang paling baik secara asimtotik, $O(V \lg V + E)$, dan ia melakukannya dengan membuat $O(E)$ pemanggilan `DecreaseKey` berongkos $O(1)$ masing-masing secara *teramortisasi* — artinya walaupun satu pemanggilan tertentu bisa lambat, sembarang barisan berisi m pemanggilan berongkos $O(m)$ seluruhnya. **Persis klaim semacam itulah yang keberadaannya hendak dibuktikan oleh Bab 14**, dan metode potensial yang dikembangkan di sana adalah cara orang membuktikannya. Dalam praktik faktor konstantanya cukup besar sehingga ia jarang menang di bawah graf berjuta-juta sisi.

> **JEBAKAN.** Batas $O(E \lg V)$ dan $O(V \lg V + E)$ sama-sama mengandaikan antrean prioritasnya mendukung `DecreaseKey` atas elemen sembarang — yang memerlukan sebuah pegangan dari tiap simpul ke posisinya di dalam *heap*. `std::priority_queue` tidak menyediakan pegangan semacam itu. Program di bawah memakai akal-akalan bakunya, yang diterangkan di §11.9.10, dan ongkosnya $O(E \lg E)$, bukan $O(E \lg V)$. Karena $|E| \leq |V|^2$, kita punya $\lg E \leq 2 \lg V$, sehingga keduanya hanya berbeda sebanyak faktor konstanta — tetapi ada baiknya Anda tahu yang mana yang sudah Anda implementasikan.

11.9.9 Program selengkapnya

Kode 11.3 Prim–Jarník dengan tumpukan biner, menelusuri antreannya pada tiap langkah. Kompilasilah dengan `g++ -std=c++17 -O2 -o prim prim.cpp`.

```
// Prim-Jarnik dengan tumpukan biner, pada graf berbobot Subbab 11.9.5.
// Mencetak key[], pi[], dan isi Q pada setiap langkah.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
using namespace std;

const int INF = INT_MAX;

struct Edge { int to, w; };

struct Graph {
    int n;
    vector<string>       name;
    vector<vector<Edge>> adj;
    explicit Graph(const vector<string>& names)
        : n((int)names.size()), name(names), adj(names.size()) {}
    void addEdge(int u, int v, int w) {            // tak berarah: simpan dua arah
        adj[u].push_back({v, w});
        adj[v].push_back({u, w});
    }
};

// Sebuah entri di dalam heap. `key` didahulukan agar pembandingan baku pada pair
// mengurutkan menurut key; `greater<>` mengubah std::priority_queue menjadi min-heap.
using Item = pair<int, int>;                       // (key, simpul)

int primJarnik(const Graph& G, int r, vector<int>& pi, vector<int>& added, bool trace) {
    vector<int>  key(G.n, INF);
    vector<char> inQ(G.n, 1);                      // inQ[v] == 1 berarti v masih di dalam
Q
    pi.assign(G.n, -1);
    key[r] = 0;

    priority_queue<Item, vector<Item>, greater<Item>> Q;
    for (int v = 0; v < G.n; ++v) Q.push({key[v], v});
```

Kode 11.3 (Lanjutan) *Prim–Jarník dengan tumpukan biner, menelusuri antreannya pada tiap langkah. Kompilasilah dengan g++ -std=c++17 -O2 -o prim prim.cpp.*

```
    int total = 0, step = 0;
    while (!Q.empty()) {
        Item top = Q.top(); Q.pop();
        int u = top.second;
        if (!inQ[u] || top.first != key[u]) continue;   // salinan basi; lewati saja
        inQ[u] = 0;                                     // u bergabung ke pohonnya
        if (pi[u] != -1) { total += key[u]; added.push_back(u); }

        if (trace) {
            cout << "step " << setw(2) << ++step << ": extract " << G.name[u]
                 << "  key=" << (key[u] == INF ? string("INF") : to_string(key[u]));
            if (pi[u] != -1) cout << "  add edge (" << G.name[pi[u]] << "," << G.name[u]
                                  << ") w=" << key[u] << "  running total=" << total;
            cout << "\n";
        }

        for (const Edge& e : G.adj[u])
            if (inQ[e.to] && e.w < key[e.to]) {
                key[e.to] = e.w;
                pi[e.to]  = u;
                Q.push({key[e.to], e.to});         // DecreaseKey malas: dorong salinan
yang lebih baik
            }

        if (trace) {                               // cetak Q sebagai (simpul,key,pi)
            vector<int> rest;
            for (int v = 0; v < G.n; ++v) if (inQ[v]) rest.push_back(v);
            sort(rest.begin(), rest.end(),
                 [&](int a, int b) { return key[a] != key[b] ? key[a] < key[b] : a < b; });
            cout << "        Q = {";
            for (size_t i = 0; i < rest.size(); ++i) {
                int v = rest[i];
                cout << (i ? ", " : "") << G.name[v] << ":"
                     << (key[v] == INF ? string("INF") : to_string(key[v]));
                if (pi[v] != -1) cout << "/" << G.name[pi[v]];
            }
            cout << "}\n";
        }
    }
    return total;
}
```

Kode 11.3 (Lanjutan) *Prim–Jarník dengan tumpukan biner, menelusuri antreannya pada tiap langkah. Kompilasilah dengan g++ -std=c++17 -O2 -o prim prim.cpp.*

```
int main() {
    vector<string> names = { "a","b","c","d","e","f","g","h","i" };
    enum Vertex { Va, Vb, Vc, Vd, Ve, Vf, Vg, Vh, Vi };
    Graph G(names);
    G.addEdge(Va,Vb,4);  G.addEdge(Va,Vh,8);  G.addEdge(Vb,Vc,8);
    G.addEdge(Vb,Vh,11); G.addEdge(Vc,Vd,7);  G.addEdge(Vc,Vf,4);
    G.addEdge(Vc,Vi,2);  G.addEdge(Vd,Ve,9);  G.addEdge(Vd,Vf,14);
    G.addEdge(Ve,Vf,10); G.addEdge(Vf,Vg,2);  G.addEdge(Vg,Vh,1);
    G.addEdge(Vg,Vi,6);  G.addEdge(Vh,Vi,7);

    cout << "V = " << G.n << ", E = 14, root = a\n\n";
    vector<int> pi, added;
    int total = primJarnik(G, Va, pi, added, true);

    cout << "\nMST edges, in the order Prim added them:\n";
    int check = 0, cnt = 0;
    for (int v : added) {
        int w = 0;
        for (const Edge& e : G.adj[v]) if (e.to == pi[v]) w = e.w;
        cout << "  (" << G.name[pi[v]] << "," << G.name[v] << ") w=" << w << "\n";
        check += w; ++cnt;
    }
    cout << "  edge count = " << cnt << "  (|V|-1 = " << G.n - 1 << ")\n";
    cout << "  total weight = " << check << "   (accumulated during the run: "
         << total << ")\n";
    return 0;
}
```

Keluarannya:

```
V = 9, E = 14, root = a

step  1: extract a  key=0
        Q = {b:4/a, h:8/a, c:INF, d:INF, e:INF, f:INF, g:INF, i:INF}
step  2: extract b  key=4  add edge (a,b) w=4  running total=4
        Q = {c:8/b, h:8/a, d:INF, e:INF, f:INF, g:INF, i:INF}
step  3: extract c  key=8  add edge (b,c) w=8  running total=12
        Q = {i:2/c, f:4/c, d:7/c, h:8/a, e:INF, g:INF}
step  4: extract i  key=2  add edge (c,i) w=2  running total=14
        Q = {f:4/c, g:6/i, d:7/c, h:7/i, e:INF}
step  5: extract f  key=4  add edge (c,f) w=4  running total=18
        Q = {g:2/f, d:7/c, h:7/i, e:10/f}
step  6: extract g  key=2  add edge (f,g) w=2  running total=20
        Q = {h:1/g, d:7/c, e:10/f}
step  7: extract h  key=1  add edge (g,h) w=1  running total=21
        Q = {d:7/c, e:10/f}
step  8: extract d  key=7  add edge (c,d) w=7  running total=28
        Q = {e:9/d}
step  9: extract e  key=9  add edge (d,e) w=9  running total=37
        Q = {}

MST edges, in the order Prim added them:
  (a,b) w=4
  (b,c) w=8
  (c,i) w=2
  (c,f) w=4
  (f,g) w=2
  (g,h) w=1
  (c,d) w=7
  (d,e) w=9
  edge count = 8  (|V|-1 = 8)
  total weight = 37   (accumulated during the run: 37)
```

Bandingkan keluaran itu baris demi baris dengan **Tabel 11.7**: urutan pengeluarannya, setiap kuncinya, setiap pendahulunya, setiap jumlah berjalannya, dan bobot akhirnya 37 semuanya sesuai dengan penelusuran tangan dan dengan **Gambar 11.17**. Perhatikan pula bahwa seri pada langkah 3 dipecahkan demi c atas h — `std::priority_queue` dengan `greater<pair<int,int>>` membandingkan komponen keduanya ketika komponen pertamanya sama, dan c berindeks lebih kecil. Itu pilihan yang sama dengan yang diambil **Gambar 11.11**, dan itulah sebabnya penelusuran programnya dan gambar-gambarnya sebaris.

11.9.10 Catatan tentang `DecreaseKey` yang malas

`std::priority_queue` tidak dapat mengubah sebuah elemen di tempat, sehingga programnya tidak melakukannya. Sebagai gantinya, ketika sebuah kunci membaik, ia **mendorong salinan kedua** simpul itu dengan kunci yang lebih baik, lalu bersandar pada baris

```
if (!inQ[u] || top.first != key[u]) continue;   // salinan basi; lewati saja
```

untuk membuang salinan yang usang ketika salinan itu muncul ke permukaan. Ini pola *penghapusan malas* (*lazy deletion*) yang baku, dan ia layak dipahami alih-alih dihafalkan.

- Ia **benar** sebab salinan terbaik sebuah simpul selalu berkunci terkecil di antara salinan-salinannya, sehingga ia muncul lebih dahulu; pada saat itu simpulnya masih berada di Q dan `top.first`-nya sama dengan `key[u]` yang sedang berlaku. Setiap salinan berikutnya entah mendapati `inQ[u] == 0` (simpulnya sudah bergabung) atau kunci yang tidak cocok, lalu dibuang.
- Ia berongkos **ruang**: antreannya boleh jadi menampung sampai $O(E)$ entri, bukan $O(V)$.
- Ia berongkos **waktu**: tiap dorongan dan tiap pengambilannya $O(\lg E)$, bukan $O(\lg V)$, dan banyaknya $O(E)$, sehingga menjadi $O(E \lg E)$. Sebagaimana dicatat di §11.9.8, itu berada dalam faktor konstanta dari $O(E \lg V)$.

Alternatifnya — tumpukan biner terindeks yang menyimpan, untuk tiap simpul, posisinya saat ini di dalam larik *heap*-nya — memberikan $O(E \lg V)$ Persamaan (11.6) persis seperti di buku teks dan memakai ruang $O(V)$. Soal 11.13 meminta Anda menulisnya.

11.10 Kruskal

11.10.1 Gagasannya: sebuah hutan yang melebur

Joseph Kruskal menerbitkan algoritmanya pada 1956 [7], setahun lebih dahulu daripada Prim. Algoritma ini dapat dilukiskan dalam tiga baris:

Algoritma berbasis sisi. Tambahkan sisi-sisinya satu per satu, menurut urutan bobot yang menaik. Algoritmanya memelihara A — sebuah hutan berisi pohon-pohon. Sebuah sisi diterima bila ia menghubungkan simpul-simpul dari pohon yang berbeda.

Pertentangannya dengan Prim sempurna. Prim memelihara satu pohon dan bertanya *simpul mana berikutnya?* Kruskal memelihara banyak pohon dan bertanya *apakah sisi ini berguna?* Ia mengurutkan setiap sisi di dalam grafnya sekali, lalu menyusuri senarai terurut itu dari yang termurah ke yang termahal, menerima sebuah sisi jika dan hanya jika kedua ujungnya saat itu berada di pohon yang berbeda. Bila keduanya berada di pohon yang sama, sisi itu akan menutup sebuah siklus, sehingga ia dibuang.

> **DENGAN BAHASA SEHARI-HARI.** Bentangkan semua calon jalan di atas meja, yang termurah di atas. Ambil satu per satu. Bila sebuah jalan menyambungkan dua desa yang sudah punya

> rute di antara keduanya, buanglah — ia berlebih. Kalau tidak, bangunlah. Ketika Anda sudah membangun $|V| - 1$ jalan, berhentilah: semuanya terhubung dan tidak ada yang terbuang.

Kebenarannya mengikuti dari Akibat 11.6, sebagaimana akan kita tunjukkan di §11.10.11. Tetapi lebih dahulu: algoritmanya perlu menjawab, dengan murah, pertanyaan *apakah kedua simpul ini berada di pohon yang sama?* Dan ia perlu menjawabnya $|E|$ kali. Persoalan struktur data itulah seluruh kesulitan implementasinya, dan ia layak dikerjakan dengan benar.

11.10.2 Himpunan saling lepas, dari nol

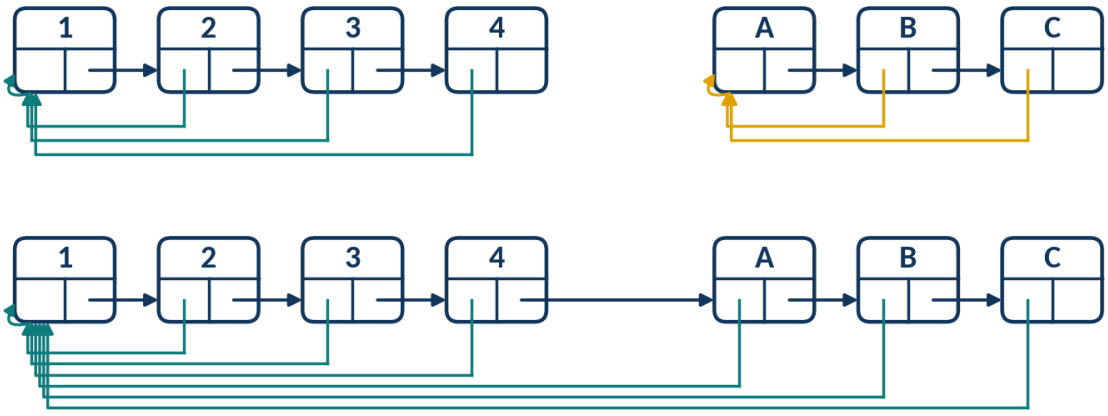
Kita perlu memelihara sebuah **partisi** atas V: sekumpulan himpunan yang saling lepas berpasangan dan gabungannya adalah V. Pada mulanya tiap simpul sendirian; seiring sisi-sisinya diterima, himpunannya melebur. Diperlukan tiga operasi:

> **DEFINISI.** Sebuah struktur **himpunan saling lepas** (*disjoint set*, atau *union–find*) memelihara sebuah partisi $S = \{S_1, S_2, \dots\}$ dan mendukung: • MakeSet(S, x): $S \leftarrow S \cup \{\{x\}\}$ — tambahkan sebuah himpunan baru yang hanya memuat x; • Union(S_i, S_j): $S \leftarrow S - \{S_i, S_j\} \cup \{S_i \cup S_j\}$ — ganti dua himpunan dengan gabungannya; • FindSet(S, x): kembalikan $S_i \in S$ tunggal dengan $x \in S_i$.

Dalam praktik `FindSet` tidak mengembalikan himpunannya melainkan sebuah **wakil** darinya — satu anggota yang ditunjuk, dan anggota yang sama itu untuk setiap elemen himpunannya. Dengan demikian "apakah u dan v berada di himpunan yang sama?" berubah menjadi `FindSet(u) == FindSet(v)`, sebuah pembandingan dua bilangan bulat.

11.10.3 Percobaan 1: himpunan sebagai senarai

Implementasi pertama, dan implementasi yang baik untuk dilihat sebelum yang lebih baik, menyatakan tiap himpunan sebagai sebuah **senarai berkait**, dengan setiap elemennya menunjuk balik ke elemen pertama, yang berperan sebagai wakilnya. **Gambar 11.18** menggambarkannya.



Gambar 11.18 Himpunan saling lepas sebagai senarai. **Atas:** dua himpunan, $\{1,2,3,4\}$ dan $\{A, B, C\}$; di dalam masing-masingnya, tiap elemen membawa penunjuk balik ke elemen pertama, yang merupakan wakilnya. **Bawah:** hasil penggabungan keduanya — senarai yang lebih pendek sudah ditempelkan dan setiap elemennya diarahkan ulang ke wakil yang baru, yaitu 1.

Ongkosnya mudah dibaca dari gambar itu:

- `FindSet(x)` berongkos $O(1)$: ikuti satu penunjuk.
- `Union(u,v)` harus mengarahkan ulang setiap elemen salah satu senarai ke kepala senarai satunya. Bila kita selalu memindahkan senarai yang **lebih kecil** ke dalam yang lebih besar —

"tambahkan senarai yang lebih kecil ke senarai yang lebih besar" — ongkosnya $O(\min\{|C(u)|, |C(v)|\})$, dengan $C(x)$ adalah himpunan yang memuat x.

Sebuah argumen pencacahan yang indah kemudian membatasi biaya total semua penggabungannya. Misalkan $t(v)$ banyaknya kali simpul v dipindahkan ke senarai baru. **Setiap kali v dipindahkan, ukuran himpunan yang memuat v sekurang-kurangnya berlipat dua** — sebab v berada di himpunan yang lebih kecil dari keduanya, sehingga himpunan yang ia datangi sekurang-kurangnya dua kali lebih besar daripada himpunan yang ia tinggalkan. Sebuah himpunan dapat berlipat dua paling banyak $\lg|V|$ kali sebelum ia memuat segalanya, sehingga

$$t(v) \leq \lg|V| \quad \text{untuk setiap } v,$$

dan kerja total seluruh penggabungan sepanjang jalannya adalah

$$\sum_{v \in V} t(v) \;\leq\; |V|\lg|V| \;=\; O(V\lg V). \tag{11.7}$$

Batas Persamaan (11.7) itu sudah cukup baik bagi Kruskal, sebagaimana akan ditunjukkan §11.10.12. Tetapi ada struktur yang lebih baik, dan itulah yang dipakai semua orang.

11.10.4 Percobaan 2: himpunan sebagai pohon

Nyatakan tiap himpunan sebagai sebuah **pohon**, disimpan sebagai larik `parent[]`. Tiap elemen menunjuk ke induknya; akarnya menunjuk ke dirinya sendiri dan merupakan wakilnya.

- `MakeSet(x)`: `parent[x] = x`. $O(1)$.
- `FindSet(x)`: berjalanlah naik dari x ke akarnya. Biaya = kedalaman x.
- `Union(x,y)`: cari kedua akarnya lalu buat salah satunya menunjuk ke yang lain. $O(1)$ sesudah kedua pencariannya.

Sedgewick dan Wayne [8] menyebut ini representasi *quick-union*, dan ia layak dilihat dalam bentuknya yang rusak sebelum diperbaiki. `Union` kini murah secara sepele, tetapi `FindSet` berongkos sedalam pohonnya — dan implementasi yang naif dapat membangun pohon berkedalaman $n-1$: gabungkan 1 ke dalam 0, lalu 2 ke dalam 1, lalu 3 ke dalam 2, dan Anda memperoleh sebuah lintasan, yang di atasnya tiap pencarian berongkos $\Theta(n)$. Dua akal-akalan yang saling bebas memperbaiki hal ini, dan bersama-sama keduanya memperbaikinya secara menakjubkan.

11.10.5 Akal-akalan 1: gabung menurut peringkat

Simpanlah, untuk tiap akar, sebuah bilangan bulat `rank` — sebuah batas atas bagi tinggi pohonnya. Ketika menggabungkan, **gantungkanlah pohon berperingkat lebih kecil di bawah akar yang berperingkat lebih besar.** Bila peringkatnya sama, gantungkan salah satunya di bawah yang lain lalu naikkan peringkat yang bertahan sebanyak satu. **Pseudokode 11.6** menguraikannya.

Pseudokode 11.6 Membuat himpunan bersinggel, dan gabung menurut peringkat: akar berperingkat lebih kecil digantungkan di bawah akar berperingkat lebih besar.

```
MAKE-SET(x)
1  parent[x] = x
2  rank[x] = 0

UNION(x, y)
1  rx = FIND-SET(x)
2  ry = FIND-SET(y)
3  if rx == ry
4      return                              // sudah menyatu
5  if rank[rx] < rank[ry]
6      swap rx and ry
7  parent[ry] = rx
8  if rank[rx] == rank[ry]
9      rank[rx] = rank[rx] + 1
```

> **LEMA 11.7.** Dengan gabung menurut peringkat, sebuah pohon yang akarnya berperingkat k memuat sekurang-kurangnya 2^k elemen. Akibatnya setiap peringkat paling besar $\lfloor \lg n \rfloor$, dan setiap pohon bertinggi paling besar $\lfloor \lg n \rfloor$, sehingga `FindSet` berongkos $O(\lg n)$.

Bukti, dengan induksi atas k. Akar berperingkat 0 adalah pohon berisi satu elemen, dan $2^0 = 1$. Sebuah akar mencapai peringkat $k \geq 1$ hanya pada saat dua akar berperingkat $(k-1)$ digabungkan; menurut induksi masing-masing pohon itu memuat sekurang-kurangnya 2^{k-1} elemen, sehingga gabungannya memuat sekurang-kurangnya $2^{k-1} + 2^{k-1} = 2^k$. Karena sebuah pohon paling banyak memuat n elemen, $2^k \leq n$, sehingga $k \leq \lg n$. ■

Ini saja sudah mengubah kasus terburuknya dari $\Theta(n)$ per pencarian menjadi $O(\lg n)$. Perhatikan *mengapa*: menggantungkan pohon yang pendek di bawah yang tinggi berarti tinggi pohon yang tinggi itu sama sekali tidak bertambah, dan satu-satunya cara menambah tinggi adalah menyatukan dua pohon yang peringkatnya persis sama — yang, menurut lemanya, menuntut jumlah penghuninya berlipat dua.

11.10.6 Akal-akalan 2: pemampatan lintasan

Akal-akalan kedua tidak berongkos apa-apa dan hampir gratis untuk ditulis. **Setiap kali `FindSet(x)` berjalan dari x naik ke akarnya, arahkanlah ulang setiap simpul yang dilewatinya langsung ke akarnya. Pseudokode 11.7** mengerjakannya.

Pseudokode 11.7 Pencarian dengan pemampatan lintasan: dalam perjalanan kembali dari akar, setiap simpul pada lintasannya diarahkan ulang ke akar.

```
FIND-SET(x)
1  if parent[x] != x
2      parent[x] = FIND-SET(parent[x])
3  return parent[x]
```

Tiga baris, dan rekursinya yang mengerjakan segalanya: pemanggilan rekursifnya mengembalikan akarnya, dan pengisiannya menyimpan akar itu. Pencarian berikutnya atas simpul mana pun pada lintasan itu berongkos satu langkah.

> **DENGAN BAHASA SEHARI-HARI.** Orang pertama yang bertanya "siapa yang berwenang di sini?" harus menyusuri seluruh rantai komandonya. Dalam perjalanan turunnya, ia memberitahukan jawabannya secara langsung kepada semua orang yang dilewatinya. Tidak seorang pun pada lintasan itu perlu berjalan lagi.

Penelusuran pada **Tabel 11.9** memperlihatkan kedua akal-akalan itu bekerja pada delapan elemen. Amati baris `rank` untuk melihat gabung menurut peringkat, dan amati `parent` berubah

selama *pencarian* di bagian akhirnya untuk melihat pemampatan lintasan meratakan pohonnya menjadi kedalaman 1.

Tabel 11.9 Sebuah hutan himpunan saling lepas dengan gabung menurut peringkat dan pemampatan lintasan, dicetak sesudah tiap operasi. Entri yang berubah dicetak tebal. Dua hal yang perlu diperhatikan: peringkat sebuah akar hanya naik ketika dua peringkat yang sama bertemu, dan dalam barisan ini ketujuh penggabungan sungguhan semuanya mempertemukan peringkat yang sama — menjadi 1 pada baris 2–5, menjadi 2 pada baris 6–7, dan menjadi 3 pada baris 8; dan `Union(1,5)` yang "tanpa efek" itu toh tetap mengubah `parent[5]` dari 4 menjadi 0, sebab kedua pencarian yang dilakukannya memampatkan lintasannya dalam perjalanan. Sesudah sapuan pencarian terakhirnya, setiap elemen menunjuk langsung ke akarnya dan peringkatnya — yang memang hanya pernah menjadi batas *atas* bagi tingginya — dibiarkan basi, dan itu tidak berbahaya.

Operasi	`parent` (elemen 0…7)	`rank`	Himpunan
sesudah `MakeSet`	0 1 2 3 4 5 6 7	0 0 0 0 0 0 0 0	8
`Union(0,1)`	0 **0** 2 3 4 5 6 7	**1** 0 0 0 0 0 0 0	7
`Union(2,3)`	0 0 2 **2** 4 5 6 7	1 0 **1** 0 0 0 0 0	6
`Union(4,5)`	0 0 2 2 4 **4** 6 7	1 0 1 0 **1** 0 0 0	5
`Union(6,7)`	0 0 2 2 4 4 6 **6**	1 0 1 0 1 0 **1** 0	4
`Union(0,2)`	0 0 **0** 2 4 4 6 6	**2** 0 1 0 1 0 1 0	3
`Union(4,6)`	0 0 0 2 4 4 **4** 6	2 0 1 0 **2** 0 1 0	2
`Union(0,4)`	0 0 0 2 **0** 4 4 6	**3** 0 1 0 2 0 1 0	1
`Union(1,5)` (tanpa efek)	0 0 0 2 0 **0** 4 6	3 0 1 0 2 0 1 0	1
sesudah `find(0)`…`find(7)`	**0 0 0 0 0 0 0 0**	3 0 1 0 2 0 1 0	1

11.10.7 Mengapa biaya teramortisasinya hampir konstan

Dengan kedua akal-akalan itu, biaya barisan berisi m operasi atas n elemen adalah

$$O(m \cdot \alpha(n)), \tag{11.8}$$

dengan α adalah **fungsi Ackermann balikan**. Buktinya berasal dari Tarjan [9] — monografnya [10] mengembangkan strukturnya beserta analisisnya sepanjang satu buku; ia tidak pendek, dan ia persis jenis argumen yang dikembangkan **Bab 14** — Anda tidak dapat membatasi satu `FindSet` tunggal dengan $\alpha(n)$, sebab satu pencarian sungguh-sungguh dapat berongkos $\Theta(\lg n)$; Anda hanya dapat membatasi *rata-ratanya atas seluruh barisan*, dan merata-ratakan kasus terburuk atas sebuah barisan itulah arti analisis amortisasi. Metode potensial Bab 14 adalah alatnya.

Yang penting pada Persamaan (11.8) adalah betapa mustahil lambatnya pertumbuhan α, sebagaimana diperlihatkan **Tabel 11.10**. (Fungsi A_k pada baris terakhir adalah fungsi jenis Ackermann yang dipakai untuk mendefinisikan α di [1]; yang kita perlukan hanyalah ukurannya.)

Tabel 11.10 Bagaimana fungsi Ackermann balikan tumbuh. Banyaknya atom di alam semesta teramati kira-kira 2^{266}, sehingga untuk setiap persoalan himpunan saling lepas yang akan pernah dijalankan di atas perangkat keras fisik, $\alpha(n) \leq 4$.

n	$\alpha(n)$
$0 \leq n \leq 2$	0
$n = 3$	1
$4 \leq n \leq 7$	2
$8 \leq n \leq 2047$	3
$2048 \leq n \leq A_4(1)$, bilangan yang jauh lebih besar daripada 2^{2048}	4

APA ARTINYA DALAM PRAKTIK. $\alpha(n) \leq 4$ untuk setiap n yang akan pernah Anda jumpai. Jadi hutan himpunan saling lepas dengan gabung menurut peringkat dan pemampatan lintasan melaksanakan tiap operasinya dalam *paling banyak empat kali* biaya sebuah operasi berwaktu konstan, secara teramortisasi — dan biasanya jauh kurang dari itu, sebab kebanyakan pencariannya mendarat di simpul yang sudah termampatkan. Anda boleh memperlakukan `FindSet` dan `Union` sebagai $O(1)$ dalam taksiran rekayasa, sambil tahu bahwa batas yang jujur adalah $O(\alpha(n))$ dan bahwa ia tidak benar-benar konstan. (Fredman dan Saks membuktikan pada 1989 bahwa waktu teramortisasi $\Omega(\alpha(n))$ per operasi tak terhindarkan bagi struktur data *apa pun* yang menyelesaikan persoalan ini, bahkan dalam model *cell-probe* yang murah hati [11], jadi inilah ujung jalannya.)

11.10.8 Program himpunan saling lepasnya

Kode 11.4 adalah strukturnya sendiri, yang mencetak lariknya sesudah tiap operasi supaya **Tabel 11.9** dapat diperiksa baris demi baris.

Kode 11.4 Sebuah hutan himpunan saling lepas dengan gabung menurut peringkat dan pemampatan lintasan. Kompilasilah dengan `g++ -std=c++17 -O2 -o dsu dsu.cpp`.

```
// Hutan himpunan saling lepas dengan gabung menurut peringkat dan pemampatan lintasan,
// mencetak larik parent sesudah tiap operasi supaya strukturnya terlihat.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;

class DisjointSets {
public:
    explicit DisjointSets(int n) : parent(n), rank_(n, 0), sets(n) {
        for (int i = 0; i < n; ++i) parent[i] = i;   // MakeSet: tiap elemen sendirian
    }

    // FindSet dengan pemampatan lintasan: sesudah pemanggilannya, tiap simpul pada
lintasan
    // dari x ke akarnya menunjuk langsung ke akarnya.
    int findSet(int x) {
        if (parent[x] != x) parent[x] = findSet(parent[x]);
        return parent[x];
    }

    // Gabung menurut peringkat: pohon yang lebih pendek digantung di bawah yang lebih
tinggi,
    // sehingga tingginya hanya bertambah bila dua pohon berperingkat sama disatukan.
    bool unite(int x, int y) {
        int rx = findSet(x), ry = findSet(y);
        if (rx == ry) return false;                  // sudah menyatu
        if (rank_[rx] < rank_[ry]) swap(rx, ry);
        parent[ry] = rx;                             // pohon ry menggantung di bawah rx
        if (rank_[rx] == rank_[ry]) ++rank_[rx];     // peringkat sama: tinggi bertambah 1
        --sets;
        return true;
    }

    int count() const { return sets; }
    int rankOf(int x) const { return rank_[x]; }
    const vector<int>& parents() const { return parent; }

private:
    vector<int> parent, rank_;
    int         sets;
};
```

Kode 11.4 (Lanjutan) *Sebuah hutan himpunan saling lepas dengan gabung menurut peringkat dan pemampatan lintasan. Kompilasilah dengan g++ -std=c++17 -O2 -o dsu dsu.cpp.*

```
void dump(const string& label, const DisjointSets& D, int n) {
    cout << left << setw(16) << label << right << " parent =";
    for (int i = 0; i < n; ++i) cout << setw(3) << D.parents()[i];
    cout << "   rank =";
    for (int i = 0; i < n; ++i) cout << setw(2) << D.rankOf(i);
    cout << "   sets=" << D.count() << "\n";
}

int main() {
    const int n = 8;
    DisjointSets D(n);
    cout << "elements 0..7, each in its own set\n";
    dump("after MakeSet", D, n);

    struct Op { int a, b; };
    vector<Op> ops = { {0,1}, {2,3}, {4,5}, {6,7}, {0,2}, {4,6}, {0,4}, {1,5} };
    for (const Op& o : ops) {
        bool merged = D.unite(o.a, o.b);
        dump("Union(" + to_string(o.a) + "," + to_string(o.b) + ")"
             + (merged ? "" : " noop"), D, n);
    }

    cout << "\nevery element now reports the same representative:\n  ";
    for (int i = 0; i < n; ++i) cout << "find(" << i << ")=" << D.findSet(i) << " ";
    cout << "\n";
    dump("after finds", D, n);

    cout << "\nwitness that path compression flattens: a chain built by hand\n";
    DisjointSets C(6);
    // Paksa pohon yang agak dalam dengan menggabung menurut peringkat yang menaik.
    C.unite(0,1); C.unite(2,3); C.unite(0,2); C.unite(4,5); C.unite(0,4);
    dump("before find(5)", C, 6);
    cout << "find(5) = " << C.findSet(5) << "\n";
    dump("after find(5)", C, 6);
    return 0;
}
```

Keluarannya:

```
elements 0..7, each in its own set
after MakeSet    parent =  0  1  2  3  4  5  6  7   rank = 0 0 0 0 0 0 0 0   sets=8
Union(0,1)       parent =  0  0  2  3  4  5  6  7   rank = 1 0 0 0 0 0 0 0   sets=7
Union(2,3)       parent =  0  0  2  2  4  5  6  7   rank = 1 0 1 0 0 0 0 0   sets=6
Union(4,5)       parent =  0  0  2  2  4  4  6  7   rank = 1 0 1 0 1 0 0 0   sets=5
Union(6,7)       parent =  0  0  2  2  4  4  6  6   rank = 1 0 1 0 1 0 1 0   sets=4
Union(0,2)       parent =  0  0  0  2  4  4  6  6   rank = 2 0 1 0 1 0 1 0   sets=3
Union(4,6)       parent =  0  0  0  2  4  4  4  6   rank = 2 0 1 0 2 0 1 0   sets=2
Union(0,4)       parent =  0  0  0  2  0  4  4  6   rank = 3 0 1 0 2 0 1 0   sets=1
Union(1,5) noop  parent =  0  0  0  2  0  0  4  6   rank = 3 0 1 0 2 0 1 0   sets=1

every element now reports the same representative:
  find(0)=0 find(1)=0 find(2)=0 find(3)=0 find(4)=0 find(5)=0 find(6)=0 find(7)=0
after finds      parent =  0  0  0  0  0  0  0  0   rank = 3 0 1 0 2 0 1 0   sets=1

witness that path compression flattens: a chain built by hand
before find(5)   parent =  0  0  0  2  0  4   rank = 2 0 1 0 1 0   sets=1
find(5) = 0
after find(5)    parent =  0  0  0  2  0  0   rank = 2 0 1 0 1 0   sets=1
```

Ketiga baris terakhirnya adalah pemampatan lintasan yang tertangkap basah: sebelum pemanggilannya, elemen 5 menunjuk ke 4, yang menunjuk ke 0. Sesudah satu `find(5)`, elemen 5 menunjuk langsung ke 0, dan ia tidak akan pernah menyusuri lintasan itu lagi.

11.10.9 Algoritmanya

Dengan strukturnya di tangan, algoritma Kruskal adalah kesembilan baris **Pseudokode 11.8**:

Pseudokode 11.8 Algoritma Kruskal: pindai sisi-sisinya dari yang teringan sampai yang terberat dan simpan setiap sisi yang menghubungkan dua pohon hutan yang berbeda.

```
MST-KRUSKAL(G, w)
01  A = {}
02  for each vertex v in G.V
03      MAKE-SET(v)
04  sort the edges of G.E into non-decreasing order by weight w
05  for each edge (u, v) in G.E, taken in non-decreasing order by weight
06      if FIND-SET(u) != FIND-SET(v)
07          A = A union {(u, v)}
08          UNION(u, v)
09  return A
```

Baris 02–03 memulai dengan $|V|$ himpunan bersinggel — sebuah hutan berisi $|V|$ simpul terpencil, yang merupakan hutan awal yang sah. Baris 04 mengurutkan. Baris 05–08 menyusuri sisi-sisi yang sudah terurut; baris 06 adalah uji siklusnya, dan ia persis berbunyi "apakah kedua simpul ini sudah terhubung?"

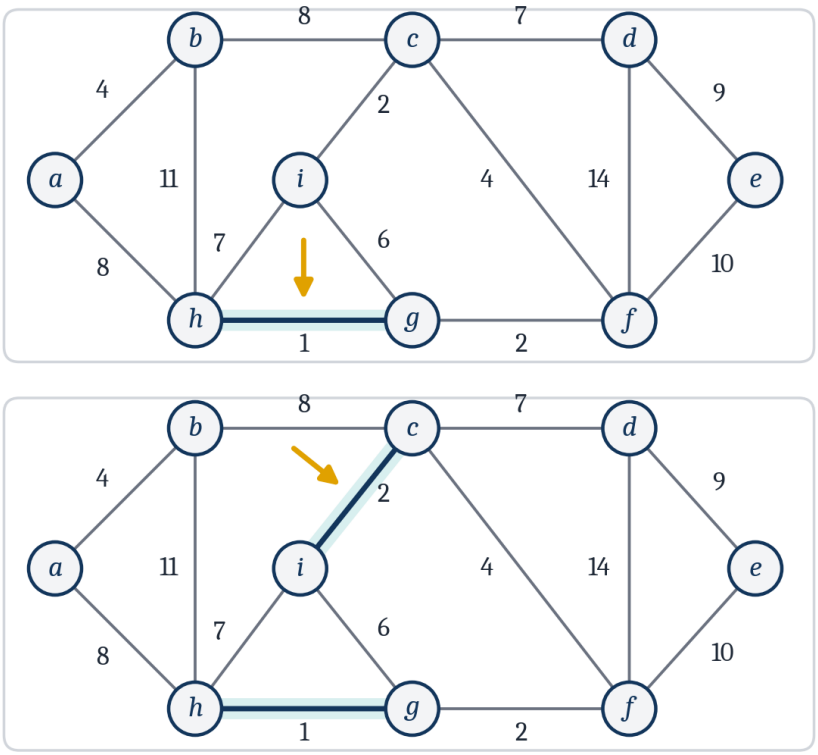
11.10.10 Contohnya ditelusuri, sisi demi sisi

Penelusurannya memakan empat belas panel — satu untuk tiap sisi grafnya. Kita mengambilnya dua-dua. Grafnya adalah graf sembilan simpul yang sama dengan **Gambar 11.9**, sehingga kedua algoritmanya dapat langsung dibandingkan. Senarai sisi terurutnya adalah

$$(g,h)1,\ (c,i)2,\ (f,g)2,\ (a,b)4,\ (c,f)4,\ (g,i)6,\ (c,d)7,\ (h,i)7,\ (a,h)8,\ (b$$

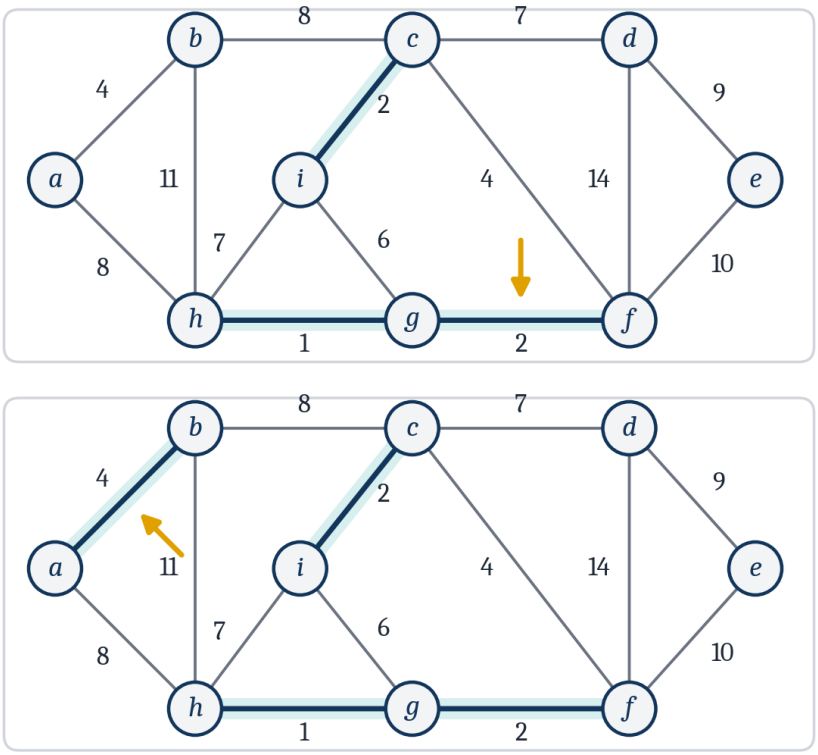
Pada tiap gambar di bawah, sebuah **anak panah** menandai sisi yang sedang ditimbang; bila ia kemudian tersorot, berarti ia diterima.

Sisi 1 dan 2. Hutannya sembilan singel: $\{a\}, \{b\}, \ldots, \{i\}$. Sisi (g,h) berbobot 1: `Find(g)`$=g \neq h=$`Find(h)`, berbeda, jadi terima. Total 1. Kini (c,i) berbobot 2: `Find(c)`$=c \neq i$, terima. Total 3.



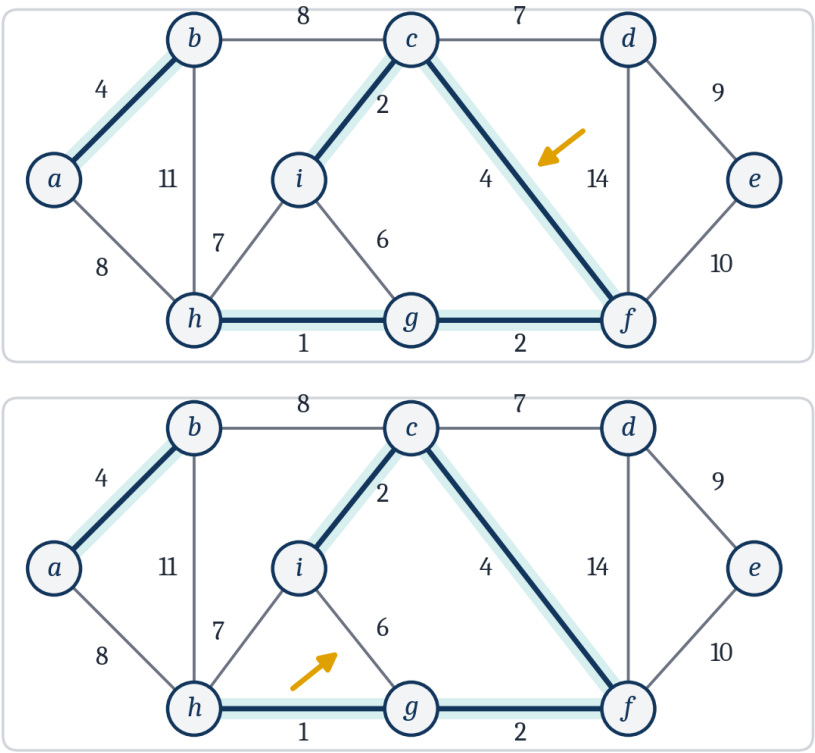
Gambar 11.19 Dua langkah Kruskal yang pertama. **Atas:** (g, h) berbobot 1 — sisi termurah di seluruh grafnya — diterima. **Bawah:** (c, i) berbobot 2 diterima. Tidak seperti Prim, Kruskal memulai dengan sisi termurah secara global, di mana pun ia kebetulan berada.

Sisi 3 dan 4 (Gambar 11.20). Hutan $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f\}, \{g, h\}$. Sisi (f, g) berbobot 2: `Find(f)` $= f$, `Find(g)` $= g$, berbeda, terima — dan kini $\{f, g, h\}$ menjadi satu pohon. Total 5. Sisi (a, b) berbobot 4: keduanya singel, terima. Total 9.



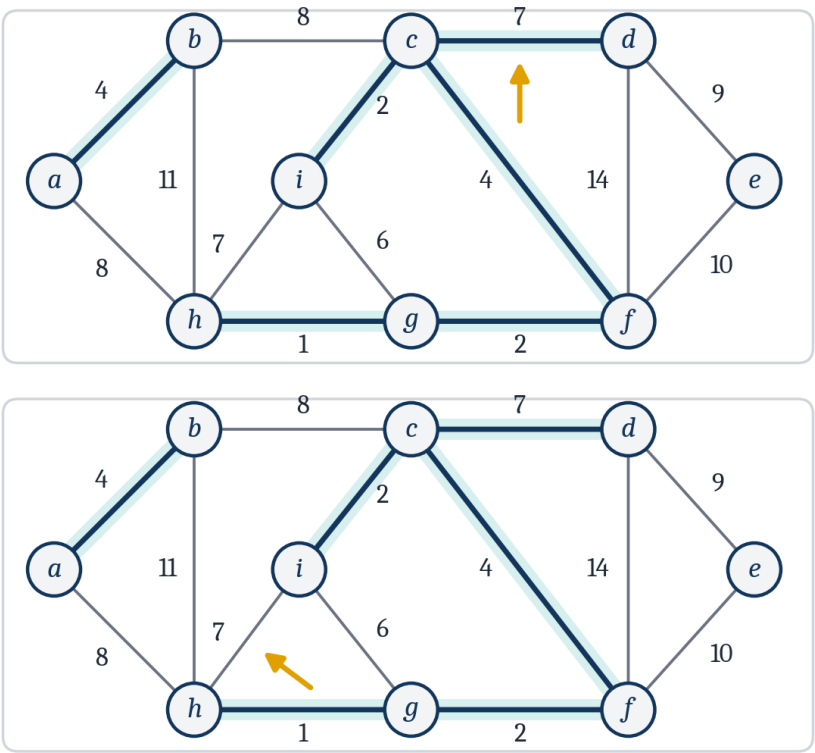
Gambar 11.20 Atas: (f, g) berbobot 2 diterima, meleburkan $\{f\}$ dengan $\{g, h\}$. **Bawah:** (a, b) berbobot 4 diterima. Perhatikan bahwa hutannya kini punya empat pohon terpisah yang tumbuh serentak — sesuatu yang tidak pernah dilakukan Prim.

Sisi 5 dan 6 (Gambar 11.21). Hutan $\{a, b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$. Sisi (c, f) berbobot 4: `Find(c)`$=c$, `Find(f)`$=g$ — berbeda, terima, melebur menjadi $\{c, f, g, h, i\}$. Total 13. Lalu sisi (g, i) berbobot 6: `Find(g)`$=c$ dan `Find(i)`$=c$ — **sama**. Ditolak: g dan i sudah terhubung, lewat rute $g - f - c - i$.



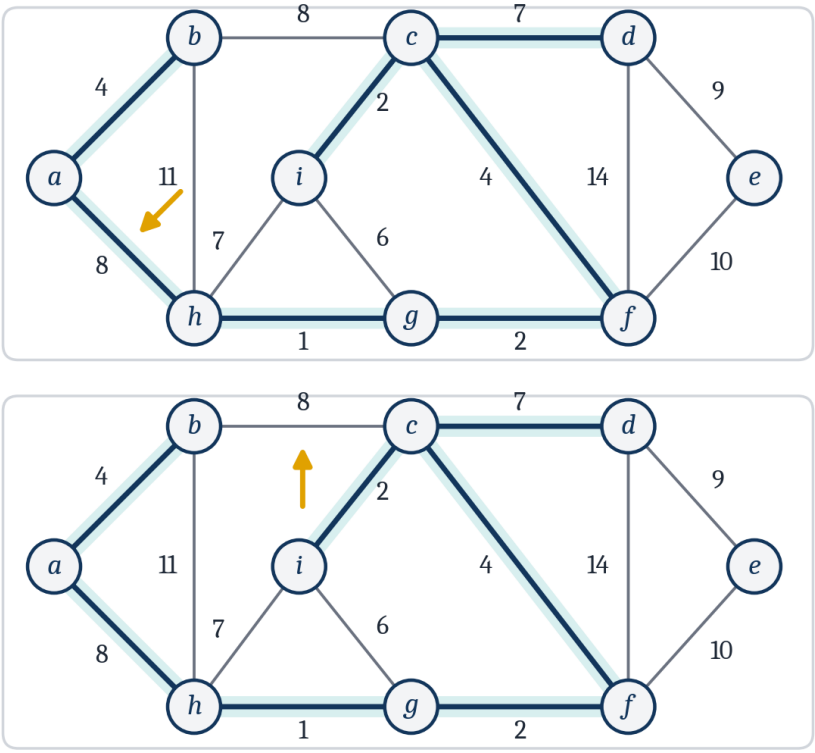
Gambar 11.21 Atas: (c, f) berbobot 4 diterima, meleburkan dua pohon menjadi $\{c, f, g, h, i\}$. **Bawah:** penolakan yang pertama — (g, i) berbobot 6 *tidak* tersorot, sebab g dan i sudah berada di satu pohon dan sisi itu akan menutup sebuah siklus.

Sisi 7 dan 8 (Gambar 11.22). Hutan $\{a, b\}, \{c, f, g, h, i\}, \{d\}, \{e\}$. Sisi (c, d) berbobot 7: himpunannya berbeda, terima; total 20. Sisi (h, i) berbobot 7: `Find(h)`$=$`Find(i)`$=c$, tolak.



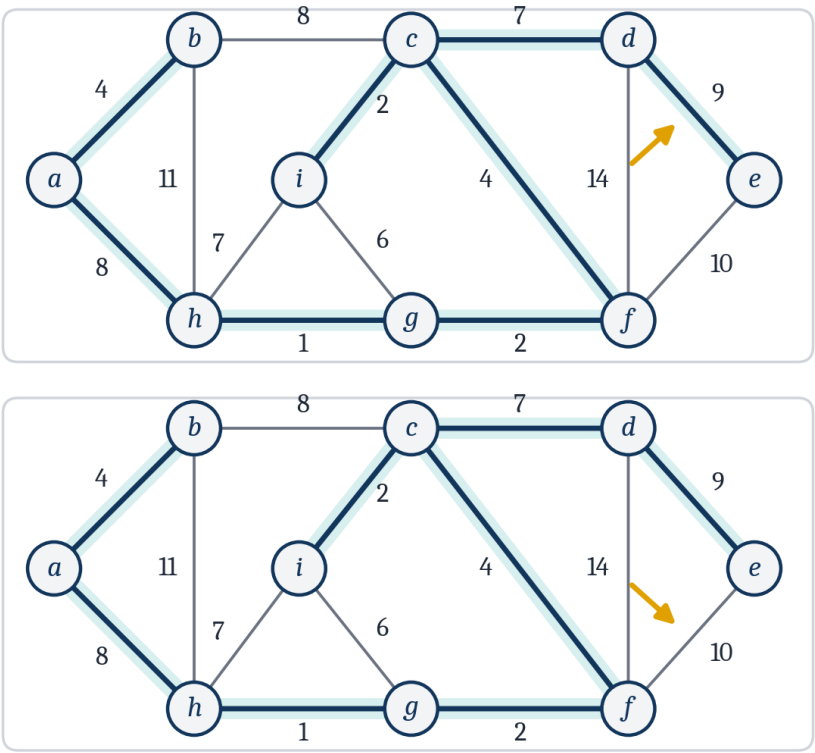
Gambar 11.22 Atas: (c, d) berbobot 7 diterima. **Bawah:** (h, i), yang juga berbobot 7, ditolak. Dua sisi berbobot sama, nasib berlawanan — keputusannya sama sekali tidak berkaitan dengan bobotnya dan sepenuhnya berkaitan dengan partisinya.

Sisi 9 dan 10. Hutan $\{a, b\}, \{c, d, f, g, h, i\}, \{e\}$. Sisi (a, h) berbobot 8: `Find(a)`$=a$, `Find(h)`$=c$, berbeda, terima; total 28; kedua pohon besarnya melebur. Sisi (b, c) berbobot 8: kini `Find(b)`$=$`Find(c)`$=c$, tolak.



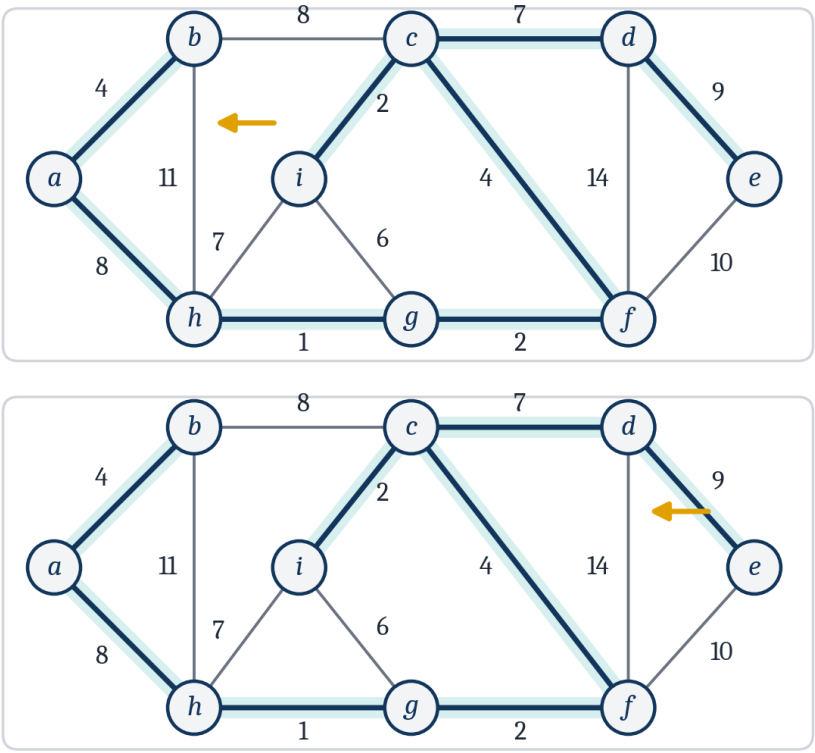
Gambar 11.23 Atas: (a, h) berbobot 8 diterima. **Bawah:** (b, c) berbobot 8 ditolak. **Inilah langkah saat jawaban Kruskal menyimpang dari jawaban Prim:** Prim mengambil (b, c) dan tidak pernah menimbang (a, h); Kruskal melakukan kebalikannya. Kedua pohonnya berbobot 37 (lihat §11.12).

Sisi 11 dan 12 (Gambar 11.24). Hutan $\{a, b, c, d, f, g, h, i\}, \{e\}$. Sisi (d, e) berbobot 9: e masih sendirian, terima; total 37, dan hutannya menjadi satu pohon tunggal. Sisi (e, f) berbobot 10: kini himpunannya sama, tolak.



Gambar 11.24 **Atas:** (d, e) berbobot 9 diterima — sisi kedelapan sekaligus yang terakhir, yang membawa simpul terpencil e masuk ke pohonnya. **Bawah:** (e, f) berbobot 10 ditolak.

Sisi 13 dan 14. Hutannya sudah berupa satu pohon tunggal dengan sembilan simpul, sehingga setiap sisi yang tersisa pasti ditolak. (b, h) berbobot 11: ditolak. (d, f) berbobot 14: ditolak.



Gambar 11.25 Dua panel terakhir: (b, h) berbobot 11 dan (d, f) berbobot 14 sama-sama ditolak, dan kedelapan sisi yang tersorot adalah pohon merentang minimum yang sudah jadi, dengan bobot total 37 — bobot yang sama dengan jawaban Prim pada **Gambar 11.17**, lewat himpunan sisi yang berbeda.

Jalannya dalam satu halaman:

Tabel 11.11 Jalan lengkap Kruskal dari **Gambar 11.19** sampai **Gambar 11.25**. Empat belas sisi ditimbang, delapan diterima, enam ditolak, bobot total 37. Kolom `Find` menunjukkan wakil yang sungguh-sungguh dikembalikan hutan himpunan saling lepasnya; perhatikan bagaimana c menjadi wakil komponen yang tumbuh itu lalu tetap demikian.

#	Sisi	w	`Find` tiap ujungnya	Keputusan	Total	Hutan sesudahnya
1	(g,h)	1	g,h	terima	1	a,b,c,d,e,f,**gh**,i
2	(c,i)	2	c,i	terima	3	a,b,**ci**,d,e,f,gh
3	(f,g)	2	f,g	terima	5	a,b,ci,d,e,**fgh**
4	(a,b)	4	a,b	terima	9	**ab**,ci,d,e,fgh
5	(c,f)	4	c,g	terima	13	ab,**cfghi**,d,e
6	(g,i)	6	c,c	**tolak**	13	ab,cfghi,d,e
7	(c,d)	7	c,d	terima	20	ab,**cdfghi**,e
8	(h,i)	7	c,c	**tolak**	20	ab,cdfghi,e
9	(a,h)	8	a,c	terima	28	**abcdfghi**,e
10	(b,c)	8	c,c	**tolak**	28	abcdfghi,e
11	(d,e)	9	c,e	terima	37	**abcdefghi**
12	(e,f)	10	c,c	**tolak**	37	abcdefghi
13	(b,h)	11	c,c	**tolak**	37	abcdefghi
14	(d,f)	14	c,c	**tolak**	37	abcdefghi

Delapan penerimaan dan enam penolakan — dan $8 = |V| - 1$, sebagaimana dituntut Teorema 11.3. Perhatikan bahwa setiap sisi mulai dari #12 dan seterusnya ditolak: begitu hutannya menjadi satu pohon tunggal, tidak ada lagi yang dapat ditambahkan, sehingga implementasi yang praktis berhenti segera sesudah ia menerima $|V| - 1$ sisi alih-alih memindai ekor senarainya.

11.10.11 Mengapa Kruskal benar

Teorema. `MST-Kruskal` mengembalikan sebuah pohon merentang minimum.

Bukti. Sekali lagi kita tunjukkan bahwa ia `MoreSpecific-MST` yang menyamar. Misalkan A himpunan sisi yang sudah diterima. Dua pengamatan:

A selalu merupakan hutan. Sebuah sisi hanya diterima ketika kedua ujungnya terletak di komponen A yang berbeda, dan menambahkan sebuah sisi di antara dua komponen yang berbeda tidak mungkin menciptakan siklus. Jadi A asiklik sepanjang jalannya, dan komponen-komponennya persis adalah himpunan-himpunan pada struktur himpunan saling lepasnya.

Setiap sisi yang diterima aman bagi A. Andaikan sisi (u, v) diterima. Misalkan C_u komponen A yang memuat u, lalu ambil potongan $S = C_u$, $V - S$ = selebihnya. Maka: - Potongan itu **menghormati** A: tiap sisi A terletak seluruhnya di dalam satu komponen, dan C_u adalah satu komponen yang utuh, sehingga tidak ada sisi A yang melintas. - (u, v) **melintasi**nya, sebab $v \notin C_u$ (justru karena itulah sisinya diterima). - (u, v) adalah sisi **ringan** baginya. Inilah langkah kuncinya. Andaikan suatu sisi (x, y) yang melintasi potongan itu memiliki $w(x, y) < w(u, v)$. Maka (x, y) muncul benar-benar lebih awal di dalam senarai terurutnya, sehingga Kruskal sudah menimbangnya. Apa yang dilakukannya ketika itu? Pada saat (x, y) ditimbang, entah kedua ujungnya berada di komponen yang berbeda — dan dalam hal itu ia diterima, dan x dan y sudah berada di komponen A yang sama sejak saat itu, yang bertentangan dengan kenyataan bahwa (x, y) melintasi potongannya *sekarang* (satu ujung di C_u, satu di luar) — atau kedua ujungnya sudah berada di komponen yang sama, dan dalam hal itu keduanya masih demikian, sehingga sekali lagi (x, y) tidak mungkin melintasi potongannya sekarang. Bagaimanapun juga, kontradiksi. Jadi tidak ada sisi lintas yang lebih murah daripada (u, v); ia ringan.

Menurut Akibat 11.6, $A \cup \{(u, v)\}$ masih termuat di dalam suatu MST. Dan ketika pemindaiannya berakhir, A punya $|V| - 1$ sisi: ia tidak mungkin punya kurang, sebab bila dua komponen masih tersisa, suatu sisi graf terhubung G akan menyambungkan keduanya, dan sisi itu — yang ditimbang pada suatu titik dalam pemindaiannya — pasti sudah diterima. Jadi A adalah pohon merentang yang termuat di dalam sebuah MST, sehingga sama dengannya. ■

> **DI MANA PENGURUTANNYA MEMBAYAR DIRINYA SENDIRI.** Seluruh argumen sisi ringan di atas bersandar pada satu kalimat: *sisi lintas mana pun yang lebih murah pasti sudah ditimbang lebih awal.* Itu benar hanya karena sisi-sisinya diproses menurut urutan bobot yang tak-menurun. Buanglah pengurutannya dan Kruskal bukan sekadar menjadi lebih lambat — ia menjadi salah.

11.10.12 Waktu jalannya

Hitunglah ketiga tahapnya:

- **Inisialisasi.** $|V|$ pemanggilan `MakeSet`, masing-masing $O(1)$: $O(V)$.
- **Pengurutan.** $\Theta(E \lg E)$ dengan pengurutan pembandingan mana pun.
- **Pemindaian.** $O(E)$ pemanggilan `FindSet` (dua tiap sisi) dan paling banyak $|V| - 1$ pemanggilan `Union`.

Dengan hutan himpunan saling lepas, $O(E)$ operasi itu berongkos $O(E\,\alpha(V))$ seluruhnya, yang untuk semua keperluan praktis berarti $O(E)$. Jadi totalnya

$$O(V) + \Theta(E \lg E) + O(E\,\alpha(V)) = O(E \lg E). \tag{11.9}$$

Pertanyaan yang wajar adalah mengapa Persamaan (11.9) sama dengan $\Theta(E \lg V)$, dan pertanyaan itu layak dijawab. Karena G sederhana dan terhubung,

$$|V| - 1 \le |E| \le \binom{|V|}{2} < |V|^2. \tag{11.10}$$

Dengan mengambil logaritma batas atas Persamaan (11.10): $\lg|E| < \lg|V|^2 = 2\lg|V|$. Dan dari batas bawahnya, $\lg|E| \ge \lg(|V| - 1)$, yang berarti $\lg|V| - o(1)$. Jadi

$$\frac{1}{2}\lg|E| \;<\; \lg|V| \;\leq\; \lg|E| + 1,$$

yaitu $\lg E = \Theta(\lg V)$. Kalikan dengan $|E|$: $\Theta(E\lg E) = \Theta(E\lg V)$. **Kedua batas itu adalah batas yang sama**, yang berarti Kruskal dan Prim-dengan-tumpukan-biner berongkos asimtotik identik, sehingga pilihan di antara keduanya harus dijatuhkan atas dasar yang lain — §11.11.

> **KE MANA WAKTUNYA PERGI.** Biaya Kruskal *didominasi oleh pengurutannya*. Bagian penelusuran grafnya pada dasarnya linear. Jadi bila sisi-sisinya sudah datang dalam keadaan terurut — atau bila bobotnya berupa bilangan bulat kecil sehingga Anda dapat mengurutkannya dengan *counting sort* dalam $O(E + W)$ — Kruskal turun ke $O(E\,\alpha(V))$, yang nyaris linear. Itu akal-akalan yang sungguhan, dan Soal 11.14 memakainya.

11.10.13 Program selengkapnya

Kode 11.5 Algoritma Kruskal dengan hutan himpunan saling lepas yang benar, menelusuri partisinya sebelum tiap keputusan. Kompilasilah dengan `g++ -std=c++17 -O2 -o kruskal kruskal.cpp`.

```
// Algoritma Kruskal dengan hutan himpunan saling lepas (gabung menurut peringkat +
// pemampatan lintasan), pada graf berbobot Subbab 11.9.5. Tiap sisi yang ditimbang
// dilaporkan, diterima atau ditolak, berikut komponennya sebelum keputusannya.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <map>
using namespace std;

class DisjointSets {
public:
    explicit DisjointSets(int n) : parent(n), rank_(n, 0) {
        for (int i = 0; i < n; ++i) parent[i] = i;
    }
    int findSet(int x) {                              // dengan pemampatan lintasan
        if (parent[x] != x) parent[x] = findSet(parent[x]);
        return parent[x];
    }
    bool unite(int x, int y) {                        // gabung menurut peringkat
        int rx = findSet(x), ry = findSet(y);
        if (rx == ry) return false;
        if (rank_[rx] < rank_[ry]) swap(rx, ry);
        parent[ry] = rx;
        if (rank_[rx] == rank_[ry]) ++rank_[rx];
        return true;
    }
private:
    vector<int> parent, rank_;
};

struct Edge {
    int u, v, w;
    bool operator<(const Edge& o) const { return w < o.w; }   // menurut bobot tak-menurun
};
```

Kode 11.5 (Lanjutan) *Algoritma Kruskal dengan hutan himpunan saling lepas yang benar, menelusuri partisinya sebelum tiap keputusan. Kompilasilah dengan g++ -std=c++17 -O2 -o kruskal kruskal.cpp.*

```
// Mencetak partisi saat ini sebagai himpunan blok, agar penelusurannya menunjukkan mengapa
// sebuah sisi diterima atau ditolak.
string partitionOf(DisjointSets& D, const vector<string>& name) {
    map<int, string> blocks;
    for (int v = 0; v < (int)name.size(); ++v) blocks[D.findSet(v)] += name[v];
    string s = "{";
    bool first = true;
    for (auto& kv : blocks) { if (!first) s += ","; s += kv.second; first = false; }
    return s + "}";
}

int main() {
    vector<string> name = { "a","b","c","d","e","f","g","h","i" };
    enum Vertex { Va, Vb, Vc, Vd, Ve, Vf, Vg, Vh, Vi };
    vector<Edge> edges = {
        {Va,Vb,4},  {Va,Vh,8},  {Vb,Vc,8},  {Vb,Vh,11}, {Vc,Vd,7},
        {Vc,Vf,4},  {Vc,Vi,2},  {Vd,Ve,9},  {Vd,Vf,14}, {Ve,Vf,10},
        {Vf,Vg,2},  {Vg,Vh,1},  {Vg,Vi,6},  {Vh,Vi,7}
    };
    const int n = (int)name.size();

    // Baris 04: urutkan sisi-sisi E menurut bobot tak-menurun.
    // stable_sort menjaga bobot yang sama pada urutan penulisannya di atas,
    // sehingga jalannya dapat diulang dan dapat dibandingkan dengan gambarnya.
    stable_sort(edges.begin(), edges.end());

    cout << "edges in sorted order:\n ";
    for (const Edge& e : edges) cout << " (" << name[e.u] << name[e.v] << ")" << e.w;
    cout << "\n\n";

    DisjointSets D(n);
    vector<Edge> mst;                                   // hutan yang sedang dibangun
    int total = 0, step = 0;
    for (const Edge& e : edges) {
        cout << "forest " << partitionOf(D, name) << "\n";
        int ru = D.findSet(e.u), rv = D.findSet(e.v);
        bool accept = (ru != rv);
        cout << "  edge " << setw(2) << ++step << ": (" << name[e.u] << "," << name[e.v]
             << ") w=" << setw(2) << e.w
             << "  Find(" << name[e.u] << ")=" << name[ru]
             << " Find(" << name[e.v] << ")=" << name[rv]
             << (accept ? "  DIFFERENT -> accept" : "  SAME -> reject");
        if (accept) { D.unite(e.u, e.v); mst.push_back(e); total += e.w;
                      cout << ", total=" << total; }
        cout << "\n";
    }
```

Kode 11.5 (Lanjutan) *Algoritma Kruskal dengan hutan himpunan saling lepas yang benar, menelusuri partisinya sebelum tiap keputusan. Kompilasilah dengan g++ -std=c++17 -O2 -o kruskal kruskal.cpp.*

```
    cout << "\nMST edges, in the order Kruskal accepted them:\n";
    for (const Edge& e : mst)
        cout << "  (" << name[e.u] << "," << name[e.v] << ") w=" << e.w << "\n";
    cout << "  edge count = " << mst.size() << "  (|V|-1 = " << n - 1 << ")\n";
    cout << "  total weight = " << total << "\n";
    cout << "  components at the end = "
         << [&]{ int c = 0; for (int v = 0; v < n; ++v) if (D.findSet(v) == v) ++c; return
c; }()
         << "\n";
    return 0;
}
```

Keluarannya, diringkas pada keputusan-keputusannya (jalan lengkapnya mencetak hutannya sebelum masing-masing):

```
edges in sorted order:
  (gh)1 (ci)2 (fg)2 (ab)4 (cf)4 (gi)6 (cd)7 (hi)7 (ah)8 (bc)8 (de)9 (ef)10 (bh)11 (df)14

  edge  1: (g,h) w= 1  Find(g)=g Find(h)=h  DIFFERENT -> accept, total=1
  edge  2: (c,i) w= 2  Find(c)=c Find(i)=i  DIFFERENT -> accept, total=3
  edge  3: (f,g) w= 2  Find(f)=f Find(g)=g  DIFFERENT -> accept, total=5
  edge  4: (a,b) w= 4  Find(a)=a Find(b)=b  DIFFERENT -> accept, total=9
  edge  5: (c,f) w= 4  Find(c)=c Find(f)=g  DIFFERENT -> accept, total=13
  edge  6: (g,i) w= 6  Find(g)=c Find(i)=c  SAME -> reject
  edge  7: (c,d) w= 7  Find(c)=c Find(d)=d  DIFFERENT -> accept, total=20
  edge  8: (h,i) w= 7  Find(h)=c Find(i)=c  SAME -> reject
  edge  9: (a,h) w= 8  Find(a)=a Find(h)=c  DIFFERENT -> accept, total=28
  edge 10: (b,c) w= 8  Find(b)=c Find(c)=c  SAME -> reject
  edge 11: (d,e) w= 9  Find(d)=c Find(e)=e  DIFFERENT -> accept, total=37
  edge 12: (e,f) w=10  Find(e)=c Find(f)=c  SAME -> reject
  edge 13: (b,h) w=11  Find(b)=c Find(h)=c  SAME -> reject
  edge 14: (d,f) w=14  Find(d)=c Find(f)=c  SAME -> reject

MST edges, in the order Kruskal accepted them:
  (g,h) w=1
  (c,i) w=2
  (f,g) w=2
  (a,b) w=4
  (c,f) w=4
  (c,d) w=7
  (a,h) w=8
  (d,e) w=9
  edge count = 8  (|V|-1 = 8)
  total weight = 37
  components at the end = 1
```

Setiap barisnya cocok dengan **Tabel 11.11** dan setiap keputusannya cocok dengan keempat belas panelnya. Baris `components at the end = 1` adalah pemeriksaan mekanis bahwa jawabannya sungguh-sungguh *merentang*.

11.11 Prim lawan Kruskal

11.11.1 Kedua jawabannya, dibandingkan

Kedua algoritmanya berjalan pada graf sembilan simpul yang sama dan keduanya mengembalikan pohon berbobot 37. Tetapi bukan pohon yang sama:

$$T_{\text{Prim}} = \{(a,b),(b,c),(c,i),(c,f),(f,g),(g,h),(c,d),(d,e)\}$$
$$T_{\text{Kruskal}} = \{(g,h),(c,i),(f,g),(a,b),(c,f),(c,d),(a,h),(d,e)\}$$

Tujuh dari delapan sisinya dipakai bersama. Keduanya berbeda tepat di satu tempat: Prim mengambil (b,c) berbobot 8, Kruskal mengambil (a,h), yang juga berbobot 8. Satu penukaran itu dapat dilacak ke seri pada **Gambar 11.11** dan ke penolakan pada **Gambar 11.23**, dan §11.12 menerangkan mengapa keduanya sah.

11.11.2 Jarang dan padat

Ingat kembali Persamaan (11.10): $|V| - 1 \le |E| < |V|^2$. Sebuah graf secara longgar disebut **jarang** bila $|E|$ dekat ke ujung bawah rentang itu dan **padat** bila ia dekat ke ujung atasnya.

§11.10.12 menunjukkan kedua batasnya identik secara formal — $O(E\lg E) = O(E\lg V)$ — sehingga perbedaannya terletak pada konstanta dan pada apa yang *disentuh* masing-masing algoritma.

- Biaya Kruskal adalah satu pengurutan atas $|E|$ butir ditambah satu pemindaian yang nyaris linear. Pengurutan bersifat berurutan, ramah-*cache*, dan $|E|$ kecil pada graf yang jarang.
- Biaya Prim adalah $|V|$ pengeluaran *heap* ditambah sampai $|E|$ pembaruan kunci, dengan akses acak ke dalam struktur ketetanggaannya. Itu berarti mengejar penunjuk, dan itu terbayar ketika $|E|$ besar sebab Prim tidak pernah mengurutkan sisi-sisi berat yang tidak akan pernah diperlukannya.
- Bila grafnya **sangat** padat, versi larik Prim pada $O(V^2)$ mengalahkan keduanya, sebab $O(V^2) < O(V^2\lg V)$ dan ia sama sekali tidak memakai *heap*.

11.11.3 Diukur

Kode 11.6 menjalankan keduanya pada graf contoh sembilan simpul, lalu pada seribu graf terhubung acak untuk memeriksa bahwa keduanya selalu sepakat, lalu pada satu graf jarang besar dan satu graf padat besar dengan jam berjalan.

Kode 11.6 Prim dan Kruskal diukur satu terhadap yang lain, termasuk uji kesepakatan teracak atas seribu graf. Kompilasilah dengan `g++ -std=c++17 -O2 -o compare compare.cpp`.

```
// Prim dan Kruskal berdampingan. Keduanya dijalankan pada graf Subbab 11.9.5 lalu
// pada seribu graf terhubung acak, dan bobot totalnya dibandingkan.
#include <iostream>
#include <iomanip>
#include <vector>
#include <queue>
#include <algorithm>
#include <random>
#include <climits>
#include <chrono>
using namespace std;

struct Edge { int u, v, w; };

// ---------- Prim-Jarnik, tumpukan biner ----------
long long prim(int n, const vector<Edge>& edges) {
    // Ketetanggaan dalam satu larik datar (compressed sparse row): start[u]..start[u+1]-1
    // adalah entri milik u di nbr/wt. Satu alokasi, bukan n alokasi.
    vector<int> start(n + 1, 0), nbr(2 * edges.size()), wt(2 * edges.size());
    for (const Edge& e : edges) { ++start[e.u]; ++start[e.v]; }
    for (int u = 0, acc = 0; u <= n; ++u) { int c = (u < n ? start[u] : 0); start[u] = acc;
acc += c; }
    { vector<int> fill(start.begin(), start.end() - 1);
      for (const Edge& e : edges) {
          nbr[fill[e.u]] = e.v; wt[fill[e.u]++] = e.w;
          nbr[fill[e.v]] = e.u; wt[fill[e.v]++] = e.w;
      } }
    vector<int>  key(n, INT_MAX);
    vector<char> inQ(n, 1);
    key[0] = 0;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    Q.push({0, 0});
    long long total = 0;
    int taken = 0;
    while (!Q.empty()) {
        auto [k, u] = Q.top(); Q.pop();
        if (!inQ[u] || k != key[u]) continue;
        inQ[u] = 0; total += k; ++taken;
        for (int j = start[u]; j < start[u + 1]; ++j) {
            int v = nbr[j], w = wt[j];
            if (inQ[v] && w < key[v]) { key[v] = w; Q.push({w, v}); }
        }
    }
    return taken == n ? total : -1;                    // -1 berarti tak terhubung
}
```

Kode 11.6 (Lanjutan) *Prim dan Kruskal diukur satu terhadap yang lain, termasuk uji kesepakatan teracak atas seribu graf. Kompilasilah dengan g++ -std=c++17 -O2 -o compare compare.cpp.*

```
// ---------- Kruskal, hutan himpunan saling lepas ----------
struct DSU {
    vector<int> p, r;
    explicit DSU(int n) : p(n), r(n, 0) { for (int i = 0; i < n; ++i) p[i] = i; }
    int find(int x) { return p[x] == x ? x : p[x] = find(p[x]); }
    bool unite(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return false;
        if (r[x] < r[y]) swap(x, y);
        p[y] = x;
        if (r[x] == r[y]) ++r[x];
        return true;
    }
};

long long kruskal(int n, vector<Edge> edges) {
    sort(edges.begin(), edges.end(), [](const Edge& a, const Edge& b) { return a.w < b.w;
});
    DSU D(n);
    long long total = 0;
    int taken = 0;
    for (const Edge& e : edges)
        if (D.unite(e.u, e.v)) { total += e.w; ++taken; }
    return taken == n - 1 ? total : -1;
}

int main() {
    // Graf contoh: a=0 b=1 c=2 d=3 e=4 f=5 g=6 h=7 i=8
    vector<Edge> example = {
        {0,1,4}, {0,7,8}, {1,2,8}, {1,7,11}, {2,3,7}, {2,5,4}, {2,8,2},
        {3,4,9}, {3,5,14}, {4,5,10}, {5,6,2}, {6,7,1}, {6,8,6}, {7,8,7}
    };
    cout << "the example graph: V=9, E=" << example.size() << "\n";
    cout << "  Prim    total = " << prim(9, example)    << "\n";
    cout << "  Kruskal total = " << kruskal(9, example) << "\n";
    cout << "  agree: " << (prim(9, example) == kruskal(9, example) ? "yes" : "NO") <<
"\n\n";
```

Kode 11.6 (Lanjutan) *Prim dan Kruskal diukur satu terhadap yang lain, termasuk uji kesepakatan teracak atas seribu graf. Kompilasilah dengan g++ -std=c++17 -O2 -o compare compare.cpp.*

```
    mt19937 rng(20251111);
    int disagreements = 0, trials = 1000;
    for (int t = 0; t < trials; ++t) {
        int n = 5 + (int)(rng() % 26);                  // 5..30 simpul
        vector<Edge> es;
        for (int v = 1; v < n; ++v)                     // pohon acak menjaganya tetap
terhubung
            es.push_back({(int)(rng() % v), v, 1 + (int)(rng() % 20)});
        int extra = (int)(rng() % (2 * n + 1));
        for (int k = 0; k < extra; ++k) {
            int a = (int)(rng() % n), b = (int)(rng() % n);
            if (a != b) es.push_back({a, b, 1 + (int)(rng() % 20)});
        }
        if (prim(n, es) != kruskal(n, es)) ++disagreements;
    }
    cout << trials << " random connected graphs (5..30 vertices, weights 1..20)\n";
    cout << "  disagreements between Prim and Kruskal: " << disagreements << "\n\n";

    // Pewaktuan pada satu graf padat yang besar dan satu graf jarang yang besar.
    auto build = [&](int n, long long m) {
        vector<Edge> es;
        for (int v = 1; v < n; ++v) es.push_back({(int)(rng() % v), v, 1 + (int)(rng() %
1000)});
        while ((long long)es.size() < m) {
            int a = (int)(rng() % n), b = (int)(rng() % n);
            if (a != b) es.push_back({a, b, 1 + (int)(rng() % 1000)});
        }
        return es;
    };
    cout << fixed << setprecision(1);
    cout << left << setw(10) << "shape" << setw(10) << "V" << setw(12) << "E"
         << setw(12) << "Prim ms" << setw(12) << "Kruskal ms" << "weights equal\n";
    struct Shape { const char* tag; int n; long long m; };
    for (Shape s : { Shape{"sparse", 200000, 400000}, Shape{"dense", 2000, 1000000} }) {
        vector<Edge> es = build(s.n, s.m);
        auto t0 = chrono::steady_clock::now();
        long long wp = prim(s.n, es);
        auto t1 = chrono::steady_clock::now();
        long long wk = kruskal(s.n, es);
        auto t2 = chrono::steady_clock::now();
        cout << left << setw(10) << s.tag << setw(10) << s.n << setw(12) << es.size()
             << setw(12) << chrono::duration<double, milli>(t1 - t0).count()
             << setw(12) << chrono::duration<double, milli>(t2 - t1).count()
             << (wp == wk ? "yes" : "NO") << "\n";
    }
    return 0;
}
```

Keluarannya:

```
the example graph: V=9, E=14
  Prim    total = 37
  Kruskal total = 37
  agree: yes

1000 random connected graphs (5..30 vertices, weights 1..20)
  disagreements between Prim and Kruskal: 0

shape     V         E           Prim ms     Kruskal ms  weights equal
sparse    200000    400000      104.9       26.3        yes
dense     2000      1000000     39.9        58.7        yes
```

Ada tiga hal yang layak diambil dari situ. Pertama, pemeriksaan silangnya: kedua algoritmanya mengembalikan 37 pada graf contoh, dan keduanya sepakat pada seluruh seribu graf acaknya — yang memang dijamin oleh Teorema 11.5 dan merupakan uji regresi yang baik bagi implementasi apa pun yang Anda tulis. Kedua, bentuk grafnya berperilaku sebagaimana diramalkan: pada graf yang jarang Kruskal kira-kira empat kali lebih cepat, dan pada graf yang padat Prim kira-kira sepertiga lebih cepat. Ketiga, keempat angka milidetik itu adalah satu kali jalan, di satu mesin, dengan satu kompilator. Jalankan programnya lagi dan angkanya akan bergeser beberapa persen; jalankan di mesin Anda sendiri dan ia akan bergeser lebih banyak. *Rasio*-nyalah fakta yang dapat dipindahkan — Kruskal kira-kira empat kali lebih cepat pada graf yang jarang, Prim kira-kira sepertiga lebih cepat pada yang padat — dan bahkan rasionya pun akan berubah bila Anda mengubah sebaran bobotnya atau tata letak ketetanggaannya. Jangan pernah mengutip sebuah pewaktuan tanpa menyebutkan apa yang menghasilkannya, dan jangan pernah membiarkan sebuah uji menegaskan satu angka pewaktuan.

11.11.4 Tabel keputusannya

Tabel 11.12 mengumpulkan nasihatnya, dengan alasannya di tiap baris, supaya pilihannya dapat dipertahankan, bukan sekadar dijatuhkan.

Tabel 11.12 Memilih di antara kedua algoritmanya. Dua baris pertamanya meliputi sebagian besar kasus nyata; selebihnya adalah kasus-kasus yang jawabannya tidak gamblang.

Bila…	Pakai	Sebab
E dekat ke V (jarang: jaringan jalan, jaring renggang)	**Kruskal**	pengurutannya atas sedikit butir dan ramah-*cache*
E dekat ke $\|V\|^2$ (padat: matriks jarak semua pasangan)	**Prim**, tumpukan biner	ia tidak pernah mengurutkan banyak sisi yang tak akan dipakainya
$\|E\| = \Theta(V^2)$ *dan* V kecil	**Prim**, larik tak terurut	$O(V^2)$ dengan konstanta mungil, dan sepele untuk ditulis
sisi-sisinya sudah datang terurut, atau bobotnya bilangan bulat kecil	**Kruskal**	buang pengurutannya; sisanya $O(E\,\alpha(V))$, nyaris linear
grafnya diberikan sebagai matriks ketetanggaan	**Prim**, larik	memindai satu baris matriks *adalah* `ExtractMin`-nya
grafnya tidak muat di memori dan datang sebagai aliran sisi	**Kruskal**	ia satu sapuan atas sisi terurut; pengurutan eksternal persoalan yang sudah selesai
Anda harus menangani graf **tak terhubung** dan menginginkan *hutan* merentang minimum	**Kruskal**	ia sama sekali tidak perlu diubah; ia cukup berhenti sebelum $\|V\| - 1$ sisi
Anda memerlukan pohonnya berakar di simpul tertentu, lengkap dengan induknya	**Prim**	π adalah pohon berakarnya, cuma-cuma
$\|E\| = \Theta(V\lg V)$ atau lebih padat dan V berjuta-juta	**Prim**, *Fibonacci heap*	$O(V\lg V + E)$ adalah batas terbaik yang diketahui

RINGKASAN YANG JUJUR. Kleinberg dan Tardos [12] menegaskan bahwa kedua algoritmanya adalah perwujudan satu gagasan yang sama, dan bahwa yang Anda pilih adalah implementasinya, bukan gagasannya. Untuk kebanyakan persoalan salah satunya sudah memadai, dan yang sebaiknya Anda pakai adalah yang dapat Anda tulis dengan benar dari ingatan di bawah tekanan waktu. Biasanya Kruskal-lah yang demikian: urutkan, lalu gelung sepuluh baris atas struktur himpunan saling lepas yang sudah Anda punya. Raihlah Prim ketika grafnya padat, ketika ia diserahkan kepada Anda sebagai matriks, atau ketika Anda menginginkan pohonnya berakar.

11.12 Bobot yang Sama, dan Kapan MST-nya Tunggal

11.12.1 Seri memecah algoritmanya, bukan jawabannya

Kita sudah melihat mekanismenya dua kali. Pada **Gambar 11.11** Prim menghadapi c: 8 dan h: 8 dan harus memilih salah satu; pada **Gambar 11.23** Kruskal menghadapi (a, h)8 dan (b, c)8 dan menerima mana pun yang lebih dahulu di dalam urutan terurutnya. Tidak satu pun pilihan itu salah, dan alasannya adalah kata *suatu* di dalam Teorema 11.5: ketika dua sisi lintas seri sebagai yang teringan, **masing-masingnya terletak di dalam suatu MST**, dan MST-MST itu boleh jadi berbeda tetapi berbobot sama.

Jadi dengan hadirnya seri, *pohon* yang dikembalikan sebuah algoritma bergantung pada rincian implementasinya: pembanding yang Anda berikan kepada `sort`, apakah *heap* Anda berupa *max-heap* atas kunci yang dinegatifkan atau *min-heap*, urutan Anda menuliskan sisi-sisinya. *Bobotnya* tidak bergantung pada itu. Itulah jaminannya, dan itulah yang harus dijadikan sasaran pengujian.

> **JEBAKAN BAGI RANGKAIAN UJI.** Jangan pernah menegaskan bahwa sebuah implementasi MST mengembalikan himpunan sisi tertentu. Tegaskanlah bahwa (a) hasilnya punya $|V| - 1$ sisi, (b) ia terhubung dan asiklik, serta (c) bobotnya sama dengan bobot acuan. **Kode 11.6** menguji persis (c), pada seribu graf, dan itulah sebabnya ia menemukan nol ketidaksepakatan alih-alih seribu.

11.12.2 Bobot yang berbeda-beda berakibat MST yang tunggal

> **TEOREMA 11.8.** Bila semua bobot sisi sebuah graf terhubung G **berbeda-beda**, maka G punya tepat satu pohon merentang minimum.

Bukti. Andaikan T_1 dan T_2 keduanya MST dan $T_1 \neq T_2$. Maka ada sisi yang berada di salah satunya dan tidak di yang lain; di antara semua sisi semacam itu, misalkan e yang bobotnya **terkecil**, dan katakanlah tanpa mengurangi keumuman $e \in T_1 \setminus T_2$.

Tambahkan e ke T_2. Menurut §11.4.2 hal itu menciptakan tepat satu siklus C, dan C memuat e. Nah, C tidak mungkin terletak seluruhnya di dalam T_1, sebab T_1 asiklik. Jadi C memuat suatu sisi $f \notin T_1$, dengan $f \neq e$ dan $f \in T_2$. Maka $f \in T_2 \setminus T_1$, yang menjadikan f salah satu sisi yang atasnya e dipilih sebagai yang minimal. Karena bobotnya berbeda-beda dan e adalah sisi minimum semacam itu,

$$w(e) < w(f).$$

Sekarang bentuklah $T_2{}' = (T_2 \cup \{e\}) \setminus \{f\}$. Menghapus sebuah sisi dari siklus tunggal C meninggalkan sebuah pohon merentang (siklusnya terputus, cacah simpulnya tak berubah, cacah sisinya kembali ke $|V| - 1$, dan keterhubungannya selamat sebab sisa C menyediakan rute yang hilang). Dan

$$w(T_2{}') = w(T_2) + w(e) - w(f) < w(T_2).$$

Itu bertentangan dengan keminimuman T_2. Maka pasangan $T_1 \neq T_2$ semacam itu tidak ada. ■

Perhatikan bahwa kebalikannya **salah**. Sebuah graf boleh punya bobot berulang dan tetap punya MST yang tunggal — bila sisi-sisi yang seri itu tidak pernah bersaing untuk potongan yang sama. Graf lengkap pada tiga simpul dengan bobot 1, 1, 5 punya MST yang tunggal (kedua sisi berbobot 1), meskipun ada seri.

11.12.3 Menguji ketunggalannya

Inilah sebuah uji yang eksak, dan ia hanya beberapa baris bila Kruskal sudah tersedia.

> **KRITERIA.** Misalkan T sebuah MST dari G. Maka T adalah *satu-satunya* MST dari G jika dan hanya jika, untuk setiap sisi $e \in T$, pohon merentang minimum dari $G - e$ (grafnya dengan e dihapus) benar-benar lebih berat daripada T.

Mengapa. ($\Leftarrow$) Andaikan ada MST lain $T' \neq T$. Pilihlah $e \in T \setminus T'$, yang tidak kosong sebab kedua pohonnya berbeda dan keduanya punya $|V| - 1$ sisi. Maka T' adalah pohon merentang $G - e$ yang berbobot $w(T)$, sehingga $\mathrm{MST}(G - e) \leq w(T)$; dan $\mathrm{MST}(G - e) \geq w(T)$ selalu berlaku, sebab $G - e$ punya lebih sedikit pohon untuk dipilih daripada G. Jadi $\mathrm{MST}(G - e) = w(T)$ — tidak benar-benar lebih berat. ($\Rightarrow$) Sebaliknya, bila $\mathrm{MST}(G - e) = w(T)$ untuk suatu $e \in T$, maka pohon itu adalah MST dari G yang tidak memuat e, sehingga berbeda dari T. ■

Menjalankan uji itu pada graf contoh sembilan simpul — Kruskal dipanggil delapan kali, sekali untuk tiap sisi pohonnya — memberikan:

Tabel 11.13 Uji ketunggalan §11.12.3 diterapkan pada graf contoh sembilan simpul, dihitung dengan memanggil Kruskal dari **Kode 11.5** sekali untuk tiap sisi pohonnya dengan sisi itu dihapus. Tepat satu barisnya berbobot sama, sehingga MST-nya tidak tunggal — dan pohon alternatif yang dihasilkan baris ketujuhnya adalah $\{(a,b)4, (b,c)8, (c,d)7, (c,f)4, (c,i)2, (d,e)9, (f,g)2, (g,h)1\}$, yang persis merupakan jawaban Prim dari **Gambar 11.17**.

Sisi yang dihapus	Bobot terbaik tanpanya	Kesimpulan
(g,h) 1	43	lebih berat — ada di setiap MST
(c,i) 2	41	lebih berat — ada di setiap MST
(f,g) 2	41	lebih berat — ada di setiap MST
(a,b) 4	41	lebih berat — ada di setiap MST
(c,f) 4	39	lebih berat — ada di setiap MST
(c,d) 7	40	lebih berat — ada di setiap MST
(a,h) 8	**37**	**sama** — ada MST lain tanpanya
(d,e) 9	38	lebih berat — ada di setiap MST

Jadi graf contoh itu punya **tepat dua** pohon merentang minimum, masing-masing berbobot 37, dan kedua algoritma bab ini kebetulan menemukan satu-satu. (Ujinya memperlihatkan bahwa MST-nya tidak tunggal; bahwa jumlahnya tidak lebih dari dua menyusul karena hanya sisi (a,h) yang dapat dibuang tanpa ongkos, dan satu-satunya pengganti yang sama bobotnya adalah (b,c) — pencacahan paksa atas semua pohon merentangnya menegaskan hal itu.) Itu bukan kebetulan yang direkayasa demi contohnya; itulah yang terjadi ketika sebuah graf punya bobot seri yang bersaing melintasi potongan yang sama, dan itulah ilustrasi paling bersih yang mungkin bagi kata *suatu* di dalam Teorema 11.5. (Program yang sama, dijalankan dengan pengurutannya dibalik, mengembalikan pohon merentang **maksimum**, berbobot 71 — Kruskal bekerja sama baiknya secara terbalik, sebab sifat potongannya punya dual yang eksak untuk maksimum. Soal 11.12.)

11.13 Profitina dalam Praktik

Profitina adalah **platform visibilitas AI (GEO)**: ia memindai sebuah situs web, menilai seberapa siap isinya untuk *dikutip* oleh mesin jawaban AI, menjelaskan tiap sinyalnya, dan membantu memperbaiki apa yang ditemukannya. Ia berjalan di produksi pada `app.profitina.com`, sebagai aplikasi Python di atas PostgreSQL, dengan inferensi model lokal melalui Ollama dan Qwen2.5. Dua gagasan dari bab ini memikul beban di dalamnya.

11.13.1 Pipeline pemindaiannya adalah sebuah DAG, dan penjadwalnya mengurutkannya secara topologis

Satu kali pemindaian situs bukan satu pengukuran, melainkan beberapa lusin, dan pengukuran-pengukuran itu saling bergantung. Mengambil sebuah halaman harus mendahului menguraikannya. Penguraian harus mendahului sinyal keterekstrakan isi, sinyal data terstruktur, dan sinyal tautan internal. Mengkanonkan URL harus mendahului menyingkirkan halaman yang nyaris identik, yang harus mendahului agregat per halaman mana pun. Menyelesaikan `robots.txt` harus mendahului keputusan halaman mana yang memenuhi syarat sama sekali. Dan skor kesiapan akhirnya bergantung pada setiap sinyal di bawahnya.

Itu adalah DAG kebergantungan, dan ia ditangani persis sebagaimana diuraikan §11.3. Graf tahapannya diperiksa terhadap siklus pada saat mulai — sebuah siklus berarti ada sinyal yang bergantung pada dirinya sendiri, yang merupakan kutu konfigurasi dan harus gagal dengan lantang alih-alih macet — lalu diurutkan secara topologis untuk memberikan urutan eksekusi yang sah. Algoritma Kahn-lah yang tepat di sini, dengan alasan yang ada di **Tabel 11.5**: himpunan siapnya merupakan antrean kerja yang wajar, sehingga tahapan yang seluruh kebergantungannya sudah terpenuhi dapat diserahkan kepada pekerja seketika dan secara paralel, dan tahapan baru menjadi siap seiring hasilnya berdatangan. Varian DFS-nya akan memberikan satu urutan berurutan tunggal dan membuang keparalelan yang justru sedang diberitahukan DAG-nya kepada Anda.

11.13.2 Pohon merentang minimum di dalam deteksi hampir-duplikat

Sebuah situs besar punya banyak halaman yang nyaris sama: satu templat produk yang diisi produk yang berbeda-beda, sebuah arsip berhalaman, sebuah tampilan cetak. Menilai masing-masingnya secara terpisah memboroskan kerja dan menyimpangkan agregatnya. Jadi halaman yang nyaris identik harus dikelompokkan.

Perlakukan tiap halaman sebagai simpul dan tiap pasangan sebagai sisi yang berbobot *jarak* — bilangan kecil ketika dua halaman nyaris identik, besar ketika keduanya berbeda. Bangunlah pohon merentang minimum graf itu lalu hapuslah setiap sisi yang lebih berat daripada sebuah ambang. Yang tersisa adalah sehimpunan komponen, dan komponen-komponen itulah kelompok duplikatnya. Ini **pengelompokan pautan tunggal** (*single-linkage clustering*), dan kaitannya eksak: kelompok yang dihasilkan dengan memotong sebuah MST pada ambang τ persis merupakan kelompok pautan tunggal pada τ. Kruskal bahkan membuatnya lebih langsung lagi — ia *adalah* pengelompokan pautan tunggal, dijalankan dari sisi termurah ke atas, dan menghentikannya lebih awal di τ memberikan pengelompokannya tanpa pernah menyelesaikan pohonnya.

11.13.3 Studi kasus yang dikerjakan: Profitina Laundry

Contoh yang dapat dijalankan selalu **Profitina Laundry**, studi kasus untuk pengajaran, tidak pernah produk yang sesungguhnya. Profitina Laundry punya sepuluh cabang di seputar Surabaya. Dua pertanyaan, satu dari tiap paruh bab ini.

Pertanyaan 1. Perusahaannya menginginkan jaringan antar-jemput sendiri supaya sehelai pakaian yang dititipkan di cabang mana pun dapat mencapai cabang mana pun — untuk mengonsolidasikan muatan, memindahkan limpahan, atau mengantar order pelanggan ke cabang yang lebih dekat dengan rumahnya. Tim operasi sudah menempuh delapan belas rute calon antarpasangan cabang dan mencatat jarak masing-masingnya. Mengoperasikan sebuah rute memakan biaya per kilometer yang ditempuh, setiap hari, selamanya. **Rute mana yang**

sebaiknya dioperasikan? Menurut §11.5.2 inilah persoalan MST persis: jaringannya harus merentang kesepuluh cabangnya, siklus mana pun di dalamnya memuat sebuah rute yang dapat dibuang tanpa memutus apa pun, dan kita menghendaki jarak total yang terkecil.

Pertanyaan 2. Sebuah order cucian melewati sembilan tahap dengan kendala pendahuluan yang nyata, dan tiap tahapnya memakan waktu sekian menit yang sudah diketahui. **Dalam urutan apa tahap-tahapnya boleh dijalankan, dan paling cepat berapa lama sebuah order selesai?** Bagian pertamanya adalah pengurutan topologis; bagian keduanya adalah rantai terpanjang yang menembus DAG-nya, yang — karena simpul-simpulnya keluar dalam urutan topologis — dapat dihitung dalam sapuan yang sama, dengan satu relaksasi tiap sisi. (Rantai terpanjang itu adalah *lintasan kritis* (*critical path*): barisan tahap yang menentukan waktu putar minimum sebuah order. Bab 13 menanganinya secara semestinya sebagai persoalan lintasan terpendek pada DAG; di sini ia hanya tambahan dua baris pada algoritma Kahn.)

Kode 11.7 Studi kasus Profitina Laundry: sebuah pohon merentang minimum atas sepuluh cabang, dan sebuah pengurutan topologis berikut lintasan kritis atas sembilan tahap order. Kompilasilah dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp`.

```
// Profitina Laundry: dua persoalan dari satu bab.
//   Bagian 1 - himpunan tautan antar-jemput termurah yang menjaga sepuluh cabang terhubung
//            (sebuah pohon merentang minimum), diselesaikan Kruskal dan diperiksa Prim.
//   Bagian 2 - urutan yang sah untuk menjalankan sembilan tahap sebuah order cucian
//            (sebuah pengurutan topologis), dan panjang rantai terpanjangnya.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
using namespace std;

// ======================== Bagian 1: jaringan antar-jemputnya ================
struct Link { int a, b, km100; };                 // jarak dalam ratusan meter

struct DSU {
    vector<int> parent, rank_;
    explicit DSU(int n) : parent(n), rank_(n, 0) {
        for (int i = 0; i < n; ++i) parent[i] = i;
    }
    int find(int x) { return parent[x] == x ? x : parent[x] = find(parent[x]); }
    bool unite(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return false;
        if (rank_[x] < rank_[y]) swap(x, y);
        parent[y] = x;
        if (rank_[x] == rank_[y]) ++rank_[x];
        return true;
    }
};

int primTotal(int n, const vector<Link>& links) {
    vector<vector<pair<int,int>>> adj(n);
    for (const Link& l : links) {
        adj[l.a].push_back({l.b, l.km100});
        adj[l.b].push_back({l.a, l.km100});
    }
    vector<int> key(n, INT_MAX); vector<char> inQ(n, 1);
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    key[0] = 0; Q.push({0, 0});
    int total = 0, taken = 0;
    while (!Q.empty()) {
        auto [k, u] = Q.top(); Q.pop();
        if (!inQ[u] || k != key[u]) continue;
```

Kode 11.7 (Lanjutan) *Studi kasus Profitina Laundry: sebuah pohon merentang minimum atas sepuluh cabang, dan sebuah pengurutan topologis berikut lintasan kritis atas sembilan tahap order. Kompilasilah dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
        inQ[u] = 0; total += k; ++taken;
        for (auto [v, w] : adj[u]) if (inQ[v] && w < key[v]) { key[v] = w; Q.push({w, v});
}
    }
    return taken == n ? total : -1;
}

void part1() {
    vector<string> branch = { "Keputih", "Gebang", "Klampis", "Manyar", "Ngagel",
                              "Darmo", "Wonokromo", "Rungkut", "Gubeng", "Kertajaya" };
    // Jarak jalan yang sungguh-sungguh ditempuh tim operasi, dalam ratusan meter.
    vector<Link> links = {
        {0,1,12}, {0,7,18}, {1,2,9},  {1,9,7},  {2,3,14}, {2,9,6},
        {3,4,11}, {3,8,8},  {4,5,10}, {4,6,13}, {5,6,7},  {6,7,22},
        {7,9,25}, {8,9,5},  {8,4,9},  {0,9,20}, {5,8,16}, {1,7,15}
    };
    const int n = (int)branch.size();

    int allLinks = 0;
    for (const Link& l : links) allLinks += l.km100;

    sort(links.begin(), links.end(),
         [](const Link& x, const Link& y) { return x.km100 < y.km100; });

    DSU D(n);
    vector<Link> chosen;
    int total = 0;
    cout << "Part 1 - cheapest shuttle network over " << n << " branches, "
         << links.size() << " candidate links\n";
    cout << "  link                                  km    decision\n";
    for (const Link& l : links) {
        bool take = (D.find(l.a) != D.find(l.b));
        if (take) { D.unite(l.a, l.b); chosen.push_back(l); total += l.km100; }
        cout << "  " << left << setw(12) << branch[l.a] << "- " << setw(12) << branch[l.b]
             << right << setw(8) << fixed << setprecision(1) << l.km100 / 10.0
             << "    " << (take ? "lay it" : "skip (already reachable)") << "\n";
    }
    cout << "  links laid        : " << chosen.size() << "  (needed: " << n - 1 << ")\n";
    cout << "  network length    : " << total / 10.0 << " km\n";
    cout << "  every link laid   : " << allLinks / 10.0 << " km\n";
    cout << "  saved             : " << (allLinks - total) / 10.0 << " km ("
         << 100.0 * (allLinks - total) / allLinks << "% of the candidate length)\n";
    int byPrim = primTotal(n, links);
    cout << "  Prim's answer     : " << byPrim / 10.0 << " km -- "
         << (byPrim == total ? "identical" : "DIFFERENT") << "\n\n";
}
```

Kode 11.7 (Lanjutan) *Studi kasus Profitina Laundry: sebuah pohon merentang minimum atas sepuluh cabang, dan sebuah pengurutan topologis berikut lintasan kritis atas sembilan tahap order. Kompilasilah dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
// ======================== Bagian 2: alur kerja ordernya ====================
void part2() {
    vector<string> stage = { "intake", "weigh", "price", "sort", "wash",
                             "dry", "press", "fold", "handover" };
    vector<int>    minutes = { 4, 3, 2, 6, 35, 40, 12, 8, 3 };
    enum { INTAKE, WEIGH, PRICE, SORT, WASH, DRY, PRESS, FOLD, HANDOVER };
    const int n = (int)stage.size();
    vector<vector<int>> next_(n);
    auto edge = [&](int u, int v) { next_[u].push_back(v); };
    edge(INTAKE, WEIGH);  edge(INTAKE, SORT);
    edge(WEIGH, PRICE);   edge(PRICE, HANDOVER);
    edge(SORT, WASH);     edge(WASH, DRY);
    edge(DRY, PRESS);     edge(DRY, FOLD);
    edge(PRESS, FOLD);    edge(FOLD, HANDOVER);

    vector<int> indeg(n, 0);
    for (int u = 0; u < n; ++u) for (int v : next_[u]) ++indeg[v];
    queue<int> ready;
    for (int u = 0; u < n; ++u) if (indeg[u] == 0) ready.push(u);

    vector<int> order;
    while (!ready.empty()) {
        int u = ready.front(); ready.pop();
        order.push_back(u);
        for (int v : next_[u]) if (--indeg[v] == 0) ready.push(v);
    }

    cout << "Part 2 - a legal order for the " << n << " stages\n  ";
    for (int u : order) cout << stage[u] << (u == order.back() ? "\n" : " -> ");

    // Rantai terpanjang: relaksasikan tiap tahap menurut urutan topologis. Inilah
    // saat paling awal tiap tahap dapat selesai bila tak ada yang menunggu mesin.
    vector<int>    finish(n, 0);
    vector<int>    from(n, -1);
    for (int u : order) {
        if (finish[u] == 0) finish[u] = minutes[u];
        for (int v : next_[u])
            if (finish[u] + minutes[v] > finish[v]) {
                finish[v] = finish[u] + minutes[v];
                from[v]   = u;
            }
    }
    int last = (int)(max_element(finish.begin(), finish.end()) - finish.begin());
    cout << "  earliest finish per stage:\n";
    for (int u : order)
        cout << "    " << left << setw(10) << stage[u] << right << setw(4)
```

Kode 11.7 (Lanjutan) *Studi kasus Profitina Laundry: sebuah pohon merentang minimum atas sepuluh cabang, dan sebuah pengurutan topologis berikut lintasan kritis atas sembilan tahap order. Kompilasilah dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
             << finish[u] << " min\n";
    vector<string> chain;
    for (int v = last; v != -1; v = from[v]) chain.push_back(stage[v]);
    reverse(chain.begin(), chain.end());
    cout << "  longest chain (" << finish[last] << " min): ";
    for (size_t i = 0; i < chain.size(); ++i) cout << (i ? " -> " : "") << chain[i];
    cout << "\n";
}

int main() { part1(); part2(); return 0; }
```

Keluarannya:

```
Part 1 - cheapest shuttle network over 10 branches, 18 candidate links
  link                                  km    decision
  Gubeng      - Kertajaya        0.5    lay it
  Klampis     - Kertajaya        0.6    lay it
  Gebang      - Kertajaya        0.7    lay it
  Darmo       - Wonokromo        0.7    lay it
  Manyar      - Gubeng           0.8    lay it
  Gebang      - Klampis          0.9    skip (already reachable)
  Gubeng      - Ngagel           0.9    lay it
  Ngagel      - Darmo            1.0    lay it
  Manyar      - Ngagel           1.1    skip (already reachable)
  Keputih     - Gebang           1.2    lay it
  Ngagel      - Wonokromo        1.3    skip (already reachable)
  Klampis     - Manyar           1.4    skip (already reachable)
  Gebang      - Rungkut          1.5    lay it
  Darmo       - Gubeng           1.6    skip (already reachable)
  Keputih     - Rungkut          1.8    skip (already reachable)
  Keputih     - Kertajaya        2.0    skip (already reachable)
  Wonokromo   - Rungkut          2.2    skip (already reachable)
  Rungkut     - Kertajaya        2.5    skip (already reachable)
  links laid        : 9  (needed: 9)
  network length    : 7.9 km
  every link laid   : 22.7 km
  saved             : 14.8 km (65.2% of the candidate length)
  Prim's answer     : 7.9 km -- identical

Part 2 - a legal order for the 9 stages
  intake -> weigh -> sort -> price -> wash -> dry -> press -> fold -> handover
  earliest finish per stage:
    intake       4 min
    weigh        7 min
    sort        10 min
    price        9 min
    wash        45 min
    dry         85 min
    press       97 min
    fold       105 min
    handover   108 min
  longest chain (108 min): intake -> sort -> wash -> dry -> press -> fold -> handover
```

Bacalah Bagian 1 sebagaimana seorang manajer membacanya. Sembilan tautan dioperasikan dari delapan belas yang disurvei, dan sembilan persis sama dengan $|V| - 1$ untuk sepuluh cabang. Jaringannya sepanjang 7,9 km dibandingkan 22,7 km bila semuanya dioperasikan — pengurangan 65 % jarak antar-jemput harian, dengan setiap cabang tetap dapat dicapai dari setiap cabang lainnya. Separuh dari tautan yang dilewatkan justru yang murah (0,9 km, 1,1 km) tetapi kebetulan berlebih, dan itulah seluruh pokoknya: kriterianya bukan kemurahan melainkan *kegunaan*, dan struktur himpunan saling lepaslah yang membedakan keduanya. Prim, dijalankan pada data yang sama, mengembalikan 7,9 km — pemeriksaan silangnya.

Bacalah Bagian 2 sebagaimana seorang perencana operasi membacanya. Urutan yang dihasilkannya adalah satu dari banyak urutan yang sah, dan lintasan kritisnya adalah `intake → sort → wash → dry → press → fold → handover`, 108 menit. Perhatikan tahap mana yang *tidak* berada padanya: `weigh` dan `price` selesai pada menit ke-7 dan ke-9 lalu menunggu 99 menit untuk linennya. Jadi membeli timbangan kedua tidak akan memperpendek apa pun, sedangkan memangkas lima menit dari siklus pengeringnya akan memperpendek segalanya sebanyak lima menit. Itulah nilai praktis lintasan kritis, dan ia jatuh begitu saja dari sebuah pengurutan topologis.

11.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. Mengacaukan pohon merentang minimum dengan pohon lintasan terpendek. MST meminimumkan bobot *total* jaringannya. Pohon Dijkstra meminimumkan jarak *dari satu sumber ke tiap simpul*. Keduanya sasaran yang berbeda dan biasanya pohon yang berbeda. Pada MST Prim untuk graf contoh (**Gambar 11.17**), rute pohon dari a ke e adalah $a-b-c-d-e$, berongkos $4+8+7+9=28$, sedangkan $a-h-g-f-e$ berongkos $8+1+2+10=21$. *Hindarilah dengan* bertanya apa yang sedang Anda minimumkan sebelum memilih algoritmanya: sebuah jumlah atas seluruh pohonnya, atau sebuah jarak ke tiap simpul.

2. Menuliskan `key[u] + w(u,v) < key[v]` di dalam Prim. Itu relaksasi Dijkstra; relaksasi Prim adalah `w(u,v) < key[v]`. Di dalam MST tidak ada panjang lintasan yang terakumulasi. *Hindarilah dengan* mengingat bahwa $key[v]$ adalah bobot satu sisi, tidak pernah bobot sebuah lintasan.

3. Melupakan pengujian `v in Q` di dalam Prim. Tanpanya, sebuah simpul yang sudah berada di pohonnya dapat diturunkan kuncinya, π-nya ditimpa, dan sebuah sisi kembar ditambahkan. *Hindarilah dengan* memperlakukan keanggotaan Q sebagai satu-satunya sumber kebenaran tentang siapa yang berada di dalam pohonnya.

4. Tidak mengurutkan di dalam Kruskal, atau mengurutkan menurun. §11.10.11 menunjukkan bukti kebenarannya bersandar sepenuhnya pada "sisi lintas mana pun yang lebih murah pasti sudah ditimbang". Urutan menurun memberi Anda pohon merentang *maksimum*, yang merupakan hal yang baik untuk diinginkan dan bencana untuk dihasilkan tanpa sengaja. *Hindarilah dengan* mencetak lima bobot pertama senarai terurut Anda selama pengembangannya.

5. Memakai larik biasa atau `set<int>` untuk uji komponen Kruskal. Memindai sebuah komponen untuk melihat apakah v ada di dalamnya berongkos $O(V)$ tiap sisi, yang mengubah $O(E\lg E)$ menjadi $O(VE)$. *Hindarilah dengan* memakai hutan himpunan saling lepas. Panjangnya lima belas baris.

6. Menghilangkan gabung menurut peringkat *atau* pemampatan lintasan lalu mengharapkan α. Pemampatan lintasan sendirian memberikan $O(\lg n)$ teramortisasi; gabung menurut peringkat sendirian memberikan $O(\lg n)$ kasus terburuk. Hanya keduanya bersama-sama yang memberikan $O(\alpha(n))$. *Hindarilah dengan* menuliskan keduanya sebagai satu kelas, sebagaimana pada **Kode 11.4**, sehingga Anda tidak mungkin punya yang satu tanpa yang lain.

7. Menegaskan sebuah MST tertentu di dalam pengujian. Dengan bobot yang seri ada beberapa jawaban yang benar (**Tabel 11.13** menampilkan dua). *Hindarilah dengan* menegaskan *bobotnya*, cacah sisinya $|V|-1$, dan keterhubungannya.

8. Mendorong ke belakang senarai di dalam pengurutan topologis DFS. Itu memberikan waktu penyelesaian yang menaik, yang merupakan kebalikan urutan topologisnya. *Hindarilah dengan* memeriksa satu sisi keluarannya dengan tangan sebelum memercayainya.

9. Mencatat sebuah simpul pada waktu penemuan alih-alih pada waktu penyelesaian. Urutan penemuan bukan urutan topologis; pada **Tabel 11.3** ia dimulai `shirt, tie, jacket`, sedangkan `jacket` punya dua pendahulu yang belum terdaftar. *Hindarilah dengan* menempatkan `push_front` sesudah gelung atas $Adj[u]$, tidak pernah sebelumnya.

10. Mengandaikan pengurutan topologis itu tunggal. Ia hampir tidak pernah tunggal; kedua algoritma §11.3 sama sekali tidak sepakat pada graf yang sama. *Hindarilah dengan* memverifikasi sifatnya, bukan untainya.

11. Menerapkan teorema DAG pada graf tak berarah tanpa mengabaikan sisi kedatangannya. Jebakan §11.2.5. *Hindarilah dengan* mewariskan induknya ke dalam rekursinya.

12. Menjalankan Prim pada graf tak terhubung lalu memercayai jawabannya. Prim berhenti ketika antreannya kehabisan simpul yang dapat dicapai lalu diam-diam mengembalikan pohon komponen yang memuat akarnya. *Hindarilah dengan* mencacah simpul yang dikeluarkan lalu membandingkannya dengan $|V|$ — **Kode 11.6** mengembalikan -1 persis karena alasan ini.

13. Melimpah pada bobotnya. Menjumlahkan $|V| - 1$ bobot sisi yang masing-masingnya sampai 10^9 melimpahkan `int` 32-bit begitu $|V| = 4$: tiga sisi semacam itu saja sudah berjumlah $3 \times 10^9 > 2^{31} - 1 \approx 2.1 \times 10^9$. *Hindarilah dengan* mengakumulasi totalnya di dalam `long long`, sebagaimana dilakukan **Kode 11.6**.

11.15 Ringkasan Bab

- Sebuah **graf berarah asiklik** adalah graf berarah tanpa siklus. Ia memodelkan pendahuluan: kebergantungan pembangunan program, prasyarat mata kuliah, jadwal tugas, kebergantungan instruksi.
- **Teorema 11.1 (teorema DAG):** G asiklik jika dan hanya jika sebuah pencarian mendalam-dahulu atas G tidak menghasilkan sisi balik. Buktinya punya dua paruh — sisi balik menutup sebuah siklus bersama lintasan pohonnya; dan sebuah siklus tidak dapat bersembunyi, sebab simpul siklus yang pertama ditemukan melihat lintasan putih sampai berkeliling. Karena itu deteksi siklus berongkos $\Theta(V + E)$.
- Sebuah **pengurutan topologis** adalah urutan linear simpul-simpulnya yang di dalamnya setiap sisi menunjuk maju. Ia ada persis untuk DAG, dan ia hampir tidak pernah tunggal.
- Dua algoritma, keduanya $\Theta(V + E)$: daftarkan simpul-simpulnya menurut **waktu penyelesaian yang menurun** dari sebuah DFS (benar sebab $f[v] < f[u]$ untuk setiap sisi (u, v), dibuktikan dengan argumen tiga kasus); atau berulang kali keluarkan sebuah simpul yang **berderajat masuk 0** (Kahn), yang mendeteksi siklus melalui kekurangan $|L| < |V|$.
- Sebuah **pohon merentang** memuat setiap simpul, terhubung, dan tidak punya siklus; menurut **Teorema 11.3** ia punya tepat $|V| - 1$ sisi, dan menambahkan sisi lain mana pun menciptakan tepat satu siklus.
- Sebuah **pohon merentang minimum** adalah pohon merentang yang bobot totalnya terkecil. Pencarian paksa sia-sia: graf lengkap dengan n simpul punya n^{n-2} pohon merentang.
- **Teorema 11.4 (substruktur optimal):** menghapus sebuah sisi dari sebuah MST meninggalkan dua pohon, masing-masingnya sebuah MST bagi upagraf yang diinduksi oleh simpul-simpulnya sendiri. Dibuktikan dengan gunting-dan-tempel.
- **Teorema 11.5 (sifat potongan):** untuk potongan $(S, V - S)$ mana pun, sebuah sisi lintas yang berbobot minimum termasuk ke dalam *suatu* MST. Dibuktikan dengan pertukaran: lintasan yang disediakan MST-nya harus melintasi potongannya di suatu tempat, seberangan itu tidak mungkin lebih murah daripada seberangan kita, jadi tukarlah. **Akibat 11.6** memperluasnya ke potongan mana pun yang menghormati jawaban parsial A — dan itulah yang mengubah teoremanya menjadi sebuah algoritma.
- `Generic-MST` berulang kali menambahkan sebuah **sisi aman**; invarian gelung "A dapat diperluas menjadi sebuah MST" membuatnya benar. Satu-satunya kebebasannya adalah potongan mana yang dipakai, dan kebebasan itulah perbedaan antara kedua algoritmanya.
- **Prim–Jarník** menumbuhkan satu pohon dari sebuah akar, dengan memelihara $key[v]$ = sisi termurah dari v ke dalam pohonnya dan $\pi[v]$ = ujung seberangnya, di dalam sebuah antrean

prioritas. $O(V^2)$ dengan larik, $O(E\lg V)$ dengan tumpukan biner, $O(V\lg V + E)$ dengan *Fibonacci heap*.

- **Kruskal** mengurutkan semua sisi lalu menerima tiap sisi yang kedua ujungnya terletak di komponen yang berbeda, memakai **hutan himpunan saling lepas** dengan **gabung menurut peringkat** (Lema 11.7: peringkat k berarti sekurang-kurangnya 2^k elemen, sehingga tingginya $\leq \lg n$) dan **pemampatan lintasan**. Totalnya $O(E\lg E) = O(E\lg V)$, didominasi pengurutannya; operasi himpunannya berongkos $O(E\,\alpha(V))$, yang secara efektif linear.
- Keduanya selalu menghasilkan *bobot* yang sama, dan menghasilkan *pohon* yang sama hanya ketika MST-nya tunggal. **Teorema 11.8:** bobot yang berbeda-beda berakibat MST yang tunggal. Kebalikannya tidak berlaku.
- Pilihlah Kruskal untuk graf yang jarang, untuk aliran, dan untuk bobot yang sudah terurut atau berupa bilangan bulat kecil; pilihlah Prim untuk graf yang padat, untuk matriks ketetanggaan, dan ketika Anda menginginkan pohonnya berakar.

11.15.1 Menengok ke depan

Bab 13 mengajukan pertanyaan graf berbobot yang satunya lagi — rute termurah antara dua tempat — dan Anda akan menjumpai algoritma Dijkstra, yang begitu mirip dengan Prim sehingga bedanya hanya satu tanda +, dan yang merupakan algoritma berbeda untuk menyelesaikan persoalan yang berbeda. Ia juga menyelesaikan persoalan lintasan terpendek pada DAG dalam $\Theta(V + E)$ dengan merelaksasi sisi-sisinya menurut urutan topologis yang dihitung di sini, dan itulah versi jujur dari lintasan kritis yang kita hitung pada **Kode 11.7**. **Bab 14** berutang kepada kita bukti bahwa pemampatan lintasan bersama gabung menurut peringkat berongkos $O(\alpha(n))$ teramortisasi, dan metode potensial yang dikembangkannya juga yang membenarkan baris *Fibonacci heap* pada **Tabel 11.8**. **Bab 15** membawa pertanyaan lintasan terpendek itu ke semua pasangan sekaligus.

11.16 Soal Latihan

11.1 Jalankan pengurutan topologis DFS dengan tangan pada DAG bersimpul $\{1,2,3,4,5\}$ dan bersisi $1 \to 2$, $1 \to 3$, $2 \to 4$, $3 \to 4$, $4 \to 5$, dengan mengunjungi simpul pada gelung luarnya menurut urutan indeks dan tiap senarai ketetanggaannya menaik. Berikan d, f, dan urutan keluarannya.

11.2 Ada berapa urutan topologis yang berbeda pada DAG Soal 11.1? Daftarkan semuanya.

11.3 Terangkan mengapa pengujian "ada sisi balik" melaporkan siklus pada *setiap* graf tak berarah yang punya sekurang-kurangnya satu sisi, lalu berikan perbaikan satu barisnya. Tuliskan fungsi rekursif yang sudah diperbaiki itu.

11.4 Buktikan bahwa setiap DAG berhingga punya sekurang-kurangnya satu simpul berderajat masuk 0, dan sekurang-kurangnya satu berderajat keluar 0. Di bagian mana §11.3.11 memakai yang pertama?

11.5 Jalankan Prim pada graf contoh sembilan simpul dengan akar h, bukan a. Berikan urutan pengeluarannya dan bobot totalnya. Apakah pohonnya sama dengan **Gambar 11.17**?

11.6 Pada **Tabel 11.11**, sisi 6 adalah (g, i) berbobot 6 dan ia ditolak. Sebutkan siklus yang akan diciptakannya, lalu periksa bahwa setiap sisi lain pada siklus itu lebih ringan daripada 6.

11.7 Buktikan: graf terhubung dengan n simpul dan tepat $n - 1$ sisi adalah sebuah pohon. (Inilah kebalikan yang dipakai di dalam bukti Teorema 11.4.)

11.8 Ambillah segitiga $a - b$ berbobot 1, $b - c$ berbobot 1, $a - c$ berbobot 1,5. Hitunglah (i) MST-nya, dan (ii) pohon lintasan terpendek yang berakar di a. Tunjukkan keduanya berbeda, lalu tunjukkan satu baris Prim yang harus diubah untuk menghasilkan yang kedua.

11.9 Memakai kriteria §11.12.3, tunjukkan dengan tangan bahwa graf bersimpul $\{a, b, c\}$ dan bersisi $a - b = 1$, $b - c = 1$, $a - c = 1$ punya tiga MST yang berbeda, dan bahwa tidak ada graf dengan bobot yang semuanya berbeda dapat punya dua.

11.10 MST terbaik kedua dari G adalah pohon merentang yang berbobot minimum di antara semua pohon merentang selain MST-nya. Berikan algoritma $O(VE)$, lalu berikan alasan bahwa pohon terbaik kedua itu berbeda dari yang terbaik tepat sebanyak satu penukaran sisi.

11.11 Leher botol sebuah pohon merentang adalah bobot sisi terberatnya. Buktikan bahwa pohon merentang minimum juga meminimumkan leher botolnya atas semua pohon merentang. Apakah kebalikannya benar?

11.12 Nyatakan dan buktikan dual Teorema 11.5 untuk pohon merentang **maksimum**, lalu sebutkan satu baris **Kode 11.5** yang harus diubah. Periksalah jawaban Anda dengan mencari pohon merentang maksimum graf contoh dengan tangan.

11.13 Gantilah `DecreaseKey` yang malas pada **Kode 11.3** dengan **tumpukan biner terindeks**: simpanlah `pos[v]`, yaitu indeks v di dalam larik *heap*-nya, yang diperbarui oleh setiap penyaringan. Berikan waktu jalan yang dihasilkannya, dan sebutkan berapa banyak ruang yang dihematnya.

11.14 Andaikan setiap bobot sisinya bilangan bulat di dalam $[1, W]$ dengan $W = O(E)$. Tunjukkan cara menjalankan Kruskal dalam waktu $O(E\,\alpha(V))$, lalu sebutkan apa yang berubah pada bukti kebenarannya (kalau ada).

11.15 Misalkan G terhubung dengan fungsi bobot w, dan misalkan $w'(u, v) = w(u, v) + c$ untuk sebuah konstanta $c > 0$. Buktikan bahwa setiap MST di bawah w adalah MST di bawah w'. Lalu tunjukkan dengan contoh tiga simpul bahwa pernyataan padanannya untuk *lintasan terpendek* salah.

11.16 Benar atau salah, dengan bukti atau contoh penyangkal: (a) sisi teringan G berada di setiap MST; (b) sisi terberat G tidak berada di satu MST pun; (c) bila sebuah sisi merupakan satu-satunya sisi teringan yang melintasi suatu potongan, ia berada di setiap MST.

11.17 Ubahlah Kruskal supaya ia menghitung **hutan merentang minimum** dari graf yang mungkin tak terhubung. Apa yang harus berubah di dalam kodenya, dan apa yang harus berubah di dalam bukti §11.10.11?

11.18 Terangkan, dengan kata-kata Anda sendiri dan tanpa membuktikan batas α-nya, mengapa gabung menurut peringkat sendirian memberikan $O(\lg n)$ tiap operasi, mengapa pemampatan lintasan sendirian juga memberikan $O(\lg n)$ teramortisasi, dan mengapa keduanya bersama-sama jauh lebih baik daripada masing-masingnya. Bab mana yang membuktikan batas gabungannya?

11.17 Penyelesaian Soal Latihan

11.1 DFS dari 1: $d[1] = 1$; lalu 2 ($d = 2$); lalu 4 ($d = 3$); lalu 5 ($d = 4$, $f = 5$); $f[4] = 6$; $f[2] = 7$; kembali di 1, pergi ke 3 ($d = 8$); 4 sudah hitam, sehingga $f[3] = 9$; $f[1] = 10$. Waktunya: 1: 1/10, 2: 2/7, 3: 8/9, 4: 3/6, 5: 4/5. f menurun: 1,3,2,4,5. Periksa: $1 \to 2$ (posisi $1 < 3$) ✓, $1 \to 3$ ($1 < 2$) ✓, $2 \to 4$ ($3 < 4$) ✓, $3 \to 4$ ($2 < 4$) ✓, $4 \to 5$ ✓.

11.2 Dua. Simpul 1 harus paling dahulu dan 5 paling akhir; 4 harus menyusul baik 2 maupun 3; sehingga satu-satunya kebebasannya adalah urutan 2 dan 3. Urutannya adalah 1,2,3,4,5 dan 1,3,2,4,5.

11.3 Pada graf tak berarah setiap sisi $\{u, v\}$ muncul baik di $Adj[u]$ maupun di $Adj[v]$. Ketika datang ke v dari u, pencariannya langsung memeriksa (v, u) lalu mendapati u kelabu — tetapi itu sisi yang sama dilihat dari ujung seberangnya, bukan sebuah siklus. Perbaikannya adalah mewariskan induknya ke dalam rekursi lalu melewatinya *sekali*, persis seperti yang dilakukan fungsi `hasCycle` pada **Kode 11.8**.

Menyetel `parent = -1` sesudah lewatan pertamanya itulah yang membuat sisi *sejajar* antara u dan v dilaporkan sebagai siklus dengan benar, sebab ia memang siklus. **Kode 11.8** menyandingkan uji yang rusak dan yang sudah diperbaiki pada tujuh graf kecil.

Kode 11.8 Uji berarah yang rusak dan uji tak berarah yang sudah diperbaiki, diukur terhadap tujuh graf yang jawabannya sudah diketahui. Kompilasilah dengan `g++ -std=c++17 -O2 -o undirected undirected.cpp`.

```
// Soal 11.3: deteksi siklus pada graf TAK BERARAH.
// Uji berarah "tetangga kelabu berarti siklus" menyala pada setiap sisi,
// sebab tiap sisi muncul di senarai ketetanggaan kedua ujungnya. Perbaikannya adalah
// mengabaikan sisi tempat kita datang -- sekali saja, agar sisi sejajar tetap terhitung.
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct Graph {
    int n;
    vector<vector<int>> adj;
    explicit Graph(int n_) : n(n_), adj(n_) {}
    void addEdge(int u, int v) { adj[u].push_back(v); adj[v].push_back(u); }
};

// Uji yang rusak: laporkan siklus setiap kali seorang tetangga sudah pernah terlihat.
bool brokenHasCycle(const Graph& G, vector<char>& seen, int u) {
    seen[u] = 1;
    for (int v : G.adj[u]) {
        if (seen[v]) return true;                    // menyala pada sisi kedatangannya
juga
        if (brokenHasCycle(G, seen, v)) return true;
    }
    return false;
}

// Perbaikannya: lewati sisi kedatangannya tepat sekali. Menyetel parent ke -1 sesudah
// lewatan pertama berarti salinan kedua dari sisi yang sama (sisi sejajar) tetap
// dilaporkan sebagai siklus, dan itu betul.
bool hasCycle(const Graph& G, vector<char>& seen, int u, int parent) {
    seen[u] = 1;
    for (int v : G.adj[u]) {
        if (v == parent) { parent = -1; continue; }   // lewati sisi kedatangannya sekali
saja
        if (seen[v]) return true;                    // sisi balik yang sungguhan
        if (hasCycle(G, seen, v, u)) return true;
    }
    return false;
}
```

Kode 11.8 (Lanjutan) *Uji berarah yang rusak dan uji tak berarah yang sudah diperbaiki, diukur terhadap tujuh graf yang jawabannya sudah diketahui. Kompilasilah dengan g++ -std=c++17 -O2 -o undirected undirected.cpp.*

```
bool anyCycle(const Graph& G, bool repaired) {
    vector<char> seen(G.n, 0);
    for (int u = 0; u < G.n; ++u)
        if (!seen[u]) {
            if (repaired ? hasCycle(G, seen, u, -1) : brokenHasCycle(G, seen, u))
                return true;
        }
    return false;
}

int main() {
    struct Case { string name; int n; vector<pair<int,int>> es; bool truth; };
    vector<Case> cases = {
        { "single edge 0-1",            2, {{0,1}},                       false },
        { "path 0-1-2-3",               4, {{0,1},{1,2},{2,3}},           false },
        { "triangle 0-1-2-0",           3, {{0,1},{1,2},{2,0}},           true  },
        { "star, center 0",             4, {{0,1},{0,2},{0,3}},           false },
        { "two parallel edges 0=1",     2, {{0,1},{0,1}},                 true  },
        { "forest: 0-1 and 2-3",        4, {{0,1},{2,3}},                 false },
        { "tree plus one chord",        5, {{0,1},{1,2},{2,3},{3,4},{4,1}}, true }
    };
    cout << "graph                        truth   broken   repaired\n";
    int brokenWrong = 0, repairedWrong = 0;
    for (const Case& c : cases) {
        Graph G(c.n);
        for (auto [u, v] : c.es) G.addEdge(u, v);
        bool b = anyCycle(G, false), r = anyCycle(G, true);
        if (b != c.truth) ++brokenWrong;
        if (r != c.truth) ++repairedWrong;
        cout << "  " << c.name;
        for (size_t k = c.name.size(); k < 28; ++k) cout << ' ';
        cout << (c.truth ? "cycle  " : "none   ")
             << (b ? "cycle  " : "none   ") << "  "
             << (r ? "cycle" : "none")
             << (r == c.truth ? "" : "   <-- WRONG") << "\n";
    }
    cout << "broken version wrong on   " << brokenWrong << " of " << cases.size() << "
graphs\n";
    cout << "repaired version wrong on " << repairedWrong << " of " << cases.size() << "
graphs\n";
    return 0;
}
```

Keluarannya:

```
graph                        truth   broken   repaired
  single edge 0-1             none   cycle    none
  path 0-1-2-3                none   cycle    none
  triangle 0-1-2-0            cycle  cycle    cycle
  star, center 0              none   cycle    none
  two parallel edges 0=1      cycle  cycle    cycle
  forest: 0-1 and 2-3         none   cycle    none
  tree plus one chord         cycle  cycle    cycle
broken version wrong on   4 of 7 graphs
repaired version wrong on 0 of 7 graphs
```

Versi yang rusak mengatakan “cycle” pada setiap graf yang punya sisi sama sekali, sehingga ia benar hanya secara kebetulan pada ketiga graf yang memang punya siklus.

11.4 Andaikan setiap simpul berderajat masuk ≥ 1. Mulailah dari mana saja lalu berjalanlah *mundur* menyusuri sebuah sisi masuk; Anda selalu dapat melanjutkan, sehingga jalannya tak

berhingga, dan karena V berhingga suatu simpul pasti berulang — yang memberikan sebuah siklus. Kontradiksi. Pernyataan derajat keluarnya adalah argumen yang sama dengan berjalan maju. Hasil derajat-masuk-0 itu dipakai secara tersirat di awal §11.3.11: ia menjamin himpunan siap R tidak kosong pada baris 2, sehingga gelung Kahn sungguh-sungguh dapat dimulai.

11.5 Dari h: keluarkan h (kunci 0); lalu g lewat (h, g)=1; f lewat (g, f)=2; c lewat (f, c)=4; i lewat (c, i)=2; d lewat (c, d)=7; kini a dan b sama-sama berkunci 8 (dari h dan dari c), dan serinya dimenangkan a lewat (h, a)=8; mengeluarkan a menurunkan kunci b menjadi 4, sehingga b menyusul lewat (a, b)=4; terakhir e lewat (d, e)=9. Urutan pengeluarannya adalah $h, g, f, c, i, d, a, b, e$. Total $1 + 2 + 4 + 2 + 7 + 8 + 4 + 9 = 37$. Pohonnya adalah $\{(g, h), (f, g), (c, f), (c, i), (a, h), (a, b), (c, d), (d, e)\}$ — pohon **Kruskal**, bukan pohon Prim-dari-a. Bobotnya sama, sebagaimana dijamin Teorema 11.5; pohonnya bergantung pada akarnya sebab akarnya mengubah potongan mana yang dipakai algoritmanya.

11.6 Pada saat (g, i) ditimbang, komponennya adalah $\{c, f, g, h, i\}$ dan siklus yang akan tercipta adalah $g - i - c - f - g$, yang memakai (g, i)=6, (i, c)=2, (c, f)=4, (f, g)=2. Setiap sisi lainnya berbobot 2, 4, 2 — semuanya lebih ringan daripada 6, dan itu persis yang dijamin urutan terurut Kruskal: sisi mana pun yang sudah diterima ditimbang lebih awal, sehingga bobotnya tidak lebih besar.

11.7 Misalkan G terhubung dengan n simpul dan $n - 1$ sisi, dan andaikan ia punya siklus. Hapuslah satu sisi siklus itu: grafnya tetap terhubung (sisa siklusnya memasok rute yang hilang) dan kini punya $n - 2$ sisi. Ulangi sampai tidak ada siklus yang tersisa; hasilnya terhubung dan asiklik, sehingga sebuah pohon, sehingga punya $n - 1$ sisi menurut Teorema 11.3. Tetapi kita sudah menghapus sekurang-kurangnya satu sisi, sehingga ia paling banyak punya $n - 2$. Kontradiksi. Jadi G asiklik, dan karena terhubung sekaligus asiklik, ia sebuah pohon. ■

11.8 (i) MST: ambillah $a - b$ (1) dan $b - c$ (1), bobot 2. (ii) Pohon lintasan terpendek dari a: $a - b$ berongkos 1; $a - c$ langsung berongkos 1,5, dibandingkan 2 lewat b — sehingga pohonnya $\{a - b,\ a - c\}$, bobot 2,5. Pohon yang berbeda. Baris yang berubah adalah baris 09 Prim: mengganti $w(u, v) < key[v]$ dengan $key[u] + w(u, v) < key[v]$ mengubahnya menjadi Dijkstra.

11.9 Dengan ketiga bobotnya 1, masing-masing dari ketiga pohon merentangnya (buang satu sisi) berbobot 2, sehingga ketiganya MST. Kriterianya sepakat: menghapus sisi pohon mana pun tetap meninggalkan sebuah MST berbobot 2. Untuk bagian keduanya, inilah Teorema 11.8: bila $T_1 \neq T_2$ keduanya minimum, mengambil sisi teringan e pada selisih simetrisnya lalu menukarnya ke pohon satunya benar-benar memperbaiki pohon itu, yang mustahil. Keberbedaan bobotnyalah yang membuat ketaksamaan $w(e) < w(f)$ menjadi tegas.

11.10 *Cara sederhana.* Hitunglah sebuah MST T sekali. Untuk masing-masing dari $|V| - 1$ sisi $e \in T$, hitung ulang MST dari $G - e$, lalu simpan hasil yang teringan. Bila sisi-sisinya diurutkan sekali lalu urutannya dipakai ulang, tiap penghitungan ulang adalah satu pindaian Kruskal berongkos $O(E\,\alpha(V))$, sehingga seluruh cara ini berongkos $O(E\lg E + VE\,\alpha(V))$ — dalam faktor $\alpha(V) \leq 4$ dari sasarannya.

Cara $O(VE)$. Untuk setiap simpul u, telusurilah pohon T dari u dengan satu pencarian mendalam-dahulu dan catat, untuk setiap v, $\max[u][v]$, yaitu sisi terberat pada lintasan pohon dari u ke v; itu $O(V)$ per u dan $O(V^2)$ seluruhnya. Menambahkan sisi bukan-pohon (u, v) ke T menutup tepat satu siklus (§11.4.2), dan cara termurah untuk memutusnya lagi adalah membuang sisi terberat lain pada siklus itu, yaitu $\max[u][v]$. Jadi bobot terbaik kedua adalah $w(T) + \min_{(u,v)\notin T}(w(u, v) - w(\max[u][v]))$, yang ditemukan dalam $O(E)$. Totalnya $O(E\lg E + V^2)$, yaitu $O(VE)$ sebab $|E| \geq |V| - 1$.

Klaim penukarannya. Misalkan T' sebuah pohon terbaik kedua. Dengan argumen pertukaran §11.7.3 yang diterapkan berulang-ulang, T' dapat diubah menjadi T lewat penukaran sisi tunggal yang tak satu pun menaikkan bobotnya. Perhatikan pohon terakhir sebelum T dalam barisan itu: ia berbeda dari T oleh satu penukaran, bobotnya paling besar $w(T')$, dan ia bukan T, sehingga menurut definisi terbaik kedua bobotnya sekurang-kurangnya $w(T')$. Karena itu ia sendiri adalah pohon terbaik kedua, satu penukaran dari T.

11.11 Misalkan T sebuah MST dan T' sembarang pohon merentang dengan leher botol $\beta' < \beta$, yaitu leher botol T. Misalkan $e \in T$ sebuah sisi berbobot β. Menghapus e membelah T menjadi dua komponen, yang mendefinisikan sebuah potongan. T' terhubung, sehingga ia punya sisi f yang melintasi potongan itu, dan $w(f) \leq \beta' < \beta = w(e)$. Maka $(T \setminus \{e\}) \cup \{f\}$ adalah pohon merentang yang bobotnya benar-benar lebih kecil — kontradiksi. Jadi tidak ada pohon merentang yang leher botolnya lebih kecil daripada leher botol sebuah MST. Kebalikannya **salah**: pohon merentang berleher-botol-minimum tidak harus sebuah MST. Ambillah segitiga $a - b = 1$, $b - c = 5$, $a - c = 5$: pohon $\{ab, bc\}$ dan $\{ab, ac\}$ sama-sama berleher botol 5 dan berbobot 6, sedangkan $\{bc, ac\}$ berleher botol 5 dan berbobot 10 — leher botol minimum, bukan bobot minimum.

11.12 *Dual Teorema 11.5:* untuk potongan mana pun, sebuah sisi lintas berbobot **maksimum** termasuk ke dalam suatu pohon merentang maksimum. Buktinya adalah pertukaran yang sama dengan ketaksamaannya dibalik: sisi lintas (x, y) pada lintasan pohonnya memenuhi $w(x, y) \leq w(u, v)$, sehingga penukarannya tidak menurunkan bobotnya. Pada **Kode 11.5**, ubahlah `return w < o.w;` menjadi `return w > o.w;`. Pada graf contoh, pohon merentang maksimumnya adalah $\{(d, f)14, (b, h)11, (e, f)10, (a, h)8, (b, c)8, (c, d)7, (h, i)7, (g, i)6\}$, berbobot $14 + 11 + 10 + 8 + 8 + 7 + 7 + 6 = 71$. (Setara dengan itu: negatifkan setiap bobotnya lalu jalankan Kruskal biasa.)

11.13 Simpanlah `heap` sebagai larik simpul dan `pos[v]` sebagai indeks v di dalamnya; setiap penukaran di dalam `siftUp`/`siftDown` memperbarui kedua entri `pos`. Maka `decreaseKey(v, k)` menjadi `key[v] = k; siftUp(pos[v]);`, yang $O(\lg V)$. *Heap*-nya kini paling banyak menampung $|V|$ entri alih-alih $O(E)$, sehingga ruangnya turun dari $O(E)$ menjadi $O(V)$, dan waktu jalannya menjadi persis $|V| \cdot O(\lg V) + O(E) \cdot O(\lg V) = O(E \lg V)$, bukan $O(E \lg E)$. Karena $\lg E < 2 \lg V$, perbaikannya berupa faktor konstanta pada waktunya dan faktor $|E|/|V|$ yang sungguhan pada ruangnya.

11.14 Gantilah pengurutan pembandingannya dengan *counting sort* atas bobotnya: alokasikan $W + 1$ ember, jatuhkan tiap sisi ke ember $w(u, v)$, lalu rangkaikan. Itu berongkos $\Theta(E + W) = \Theta(E)$ ketika $W = O(E)$. Sisa algoritmanya tak berubah, sehingga totalnya menjadi $\Theta(E) + O(E\,\alpha(V)) = O(E\,\alpha(V))$. **Tidak ada yang berubah pada bukti kebenarannya**: buktinya hanya menuntut sisi-sisinya diproses menurut urutan bobot yang tak-menurun, dan *counting sort* menghasilkan persis urutan itu.

11.15 Misalkan T sembarang pohon merentang. Ia punya tepat $|V| - 1$ sisi menurut Teorema 11.3, sehingga $w'(T) = w(T) + (|V| - 1)c$ — konstanta yang sama ditambahkan kepada *setiap* pohon merentang. Menambahkan sebuah konstanta kepada setiap elemen sebuah himpunan tidak mengubah elemen mana yang terkecil, sehingga peminimumnya berimpit. Untuk lintasan terpendek argumennya gagal sebab lintasan yang berbeda punya cacah sisi yang berbeda. Ambil $a - b = 1$, $b - c = 1$, $a - c = 3$. Lintasan terpendek a ke c adalah $a - b - c$, berongkos 2. Sekarang tambahkan $c = 2$ ke setiap bobotnya: $a - b = 3$, $b - c = 3$, $a - c = 5$, dan lintasan terpendek a ke c menjadi sisi langsungnya, berongkos 5 lawan 6. Lintasannya berubah.

11.16 (a) **Salah sebagaimana dinyatakan, benar bila sisi teringannya tunggal.** Bila dua sisi seri sebagai yang teringan, masing-masing berada di suatu MST tetapi mungkin tidak keduanya berada di semuanya: segitiga serba-1 pada Soal 11.9 punya tiga MST, dan masing-masingnya meniadakan satu dari ketiga sisi teringannya. Bila minimumnya tunggal, ambil $S = \{u\}$ untuk salah satu ujungnya: sisinya ringan bagi potongan itu, dan menurut versi tegas argumen pertukarannya ia berada di setiap MST. (b) **Salah.** Pada sebuah pohon, setiap sisi — termasuk yang terberat — berada di dalam MST tunggalnya, sebab hanya ada satu pohon merentang. (c) **Benar.** Ketunggalannya membuat ketaksamaan pada Langkah 5 §11.7.3 menjadi tegas, sehingga MST mana pun yang meniadakan sisi itu dapat diperbaiki secara tegas, dan itu mustahil.

11.17 Di dalam kodenya: tidak ada. Kruskal sudah menerima persis sisi-sisi yang menyambungkan komponen yang berbeda, sehingga pada graf tak terhubung ia berakhir dengan hutan berisi k pohon dan $|V| - k$ sisi, dengan k adalah banyaknya komponen terhubungnya. Satu-satunya yang berubah adalah *pemeriksaannya*: tegaskan $|A| = |V| - k$ alih-alih $|V| - 1$, dan berhentilah lebih awal sesudah $|V| - k$ penerimaan alih-alih $|V| - 1$. Di dalam bukti §11.10.11, alinea terakhirnya harus dibaca ulang: "bila dua komponen masih tersisa, suatu sisi G akan menyambungkan keduanya" hanya benar di dalam sebuah komponen terhubung G, sehingga kesimpulannya menjadi "A yang dibatasi pada tiap komponen G adalah pohon merentang komponen itu, dan ia minimum menurut argumen sisi aman yang sama yang diterapkan di dalamnya".

11.18 *Gabung menurut peringkat sendirian:* menurut Lema 11.7, akar berperingkat k memiliki sekurang-kurangnya 2^k elemen, sehingga tidak ada peringkat yang melampaui $\lg n$ dan tidak ada pohon yang lebih tinggi daripada $\lg n$; karena itu setiap `FindSet` menyusuri paling banyak $\lg n$ tautan. Itu batas **kasus terburuk** dan ia ketat — pohon seimbang bergaya binomial sungguh-sungguh dapat setinggi itu. *Pemampatan lintasan sendirian:* tiap pencarian meratakan lintasan yang dilaluinya, sehingga tidak ada simpul yang dapat menjadi mahal dua kali berturut-turut; dirata-ratakan atas sebuah barisan, ongkosnya $O(\lg n)$ teramortisasi, tetapi satu pencarian tunggal masih bisa $\Theta(n)$ sebab tidak ada yang mencegah penggabungan membangun rantai panjang sejak awal. *Bersama-sama:* gabung menurut peringkat mencegah pohonnya *menjadi* tinggi, dan pemampatan lintasan menghancurkan tinggi berapa pun yang sempat muncul, pada saat seseorang membayar untuk menyusurinya. Tidak satu pun mekanisme itu sendirian mencegah penyusuran mahal yang berulang; bersama-sama, tiap satuan tinggi hanya dapat dibayar sebanyak jumlah yang terbatas, dan pembukuan yang membuat hal itu presisi memberikan batas teramortisasi $O(\alpha(n))$. **Bab 14** membuktikannya, dengan metode potensial; bukti aslinya adalah bukti Tarjan [9], dan Fredman serta Saks [11] menunjukkan bahwa batasnya tidak dapat diperbaiki.

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962.

[3] A. Cayley, "A theorem on trees," *Quarterly Journal of Pure and Applied Mathematics*, vol. 23, pp. 376–378, 1889.

[4] V. Jarník, "O jistém problému minimálním" ["About a certain minimal problem"], *Práce Moravské Přírodovědecké Společnosti*, vol. 6, no. 4, pp. 57–63, 1930.

[5] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, Nov. 1957.

[6] R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem," *Annals of the History of Computing*, vol. 7, no. 1, pp. 43–57, Jan. 1985.

[7] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, Feb. 1956.

[8] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[9] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, Apr. 1975.

[10] R. E. Tarjan, *Data Structures and Network Algorithms*. Philadelphia, PA, USA: SIAM, 1983.

[11] M. L. Fredman and M. E. Saks, "The cell probe complexity of dynamic data structures," in *Proc. 21st Annual ACM Symposium on Theory of Computing (STOC)*, Seattle, WA, USA, 1989, pp. 345–354.

[12] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

Bab 12
Evaluasi dan Pembahasan III: Algoritma Greedy dan Graf

Bab ini menghentikan sejenak langkah maju buku ini dan memantapkan segala sesuatu yang telah dibangun oleh Bab 9, 10, dan 11: algoritma *greedy* (rakus) beserta argumen pertukarannya, pemilihan aktivitas, kode Huffman, kedua *knapsack*, dan matroid; pencarian melebar dan pencarian mendalam beserta segala yang dihitung keduanya; graf berarah taksiklik dan pengurutan topologis; serta pohon merentang minimum menurut Prim dan menurut Kruskal, bersama hutan himpunan saling lepas yang membuat Kruskal cepat. Tidak ada teori baru di dalamnya. Sebagai gantinya, bab ini mengerjakan persoalan — satu proyek perancangan yang utuh dan dua puluh tiga persoalan lain, masing-masing diselesaikan dari nol, dengan setiap antrean dicetak, setiap cap waktu dibubuhkan, setiap penyatuan dicatat, dan setiap bilangan antara dituliskan. Di akhir bab Anda akan telah membawa sebuah persoalan nyata dari halaman kosong sampai program jadi yang terukur dalam sembilan tahap (persoalan, model, pemilihan algoritma, pseudokode, kompleksitas, pembuktian dengan tangan, implementasi, evaluasi, kesimpulan). Anda akan telah menyanggah sebuah aturan rakus dengan contoh sanggahan dan membuktikan aturan lain dengan argumen pertukaran, membangun kode Huffman penggabungan demi penggabungan dan menggambar pohonnya, menjalankan algoritma rakus matroid pada sebuah persoalan penjadwalan dan memeriksanya terhadap setiap upahimpunan, serta menelusuri pencarian melebar, pencarian mendalam pada kedua jenis graf, algoritma Kahn, Kruskal dengan larik-larik himpunan saling lepasnya, dan Prim baik dengan larik maupun dengan tumpukan biner. Setiap persoalan ditutup dengan varian *uji diri* beserta jawabannya. Tiga puluh latihan singkat, serangkaian soal latihan dengan penyelesaian lengkap, dan dua belas program C++17 yang lengkap melengkapi bab ini; setiap program dikompilasi dengan `g++ -std=c++17 -O2 -Wall` dan dikutip bersama keluaran yang benar-benar dicetaknya.

12.1 Posisi Kita Saat Ini

12.1.1 Apa yang telah dibangun Bab 9, 10, dan 11

Tiga bab telah membekali kita dengan perangkat alat kedua, dan ada baiknya kita mendaftar isinya sebelum mulai memakainya, sebab setiap persoalan dalam bab ini meraih salah satu alat tersebut, dan menyebut nama alatnya sudah separuh pekerjaan.

Dari **Bab 9** kita memperoleh strategi *greedy*: membangun penyelesaian satu pilihan tak tertarik kembali demi satu pilihan, selalu mengambil opsi yang tampak terbaik saat itu juga. Kita memperoleh kenyataan yang tidak mengenakkan bahwa strategi ini sering gagal (sistem koin §9.3), dan dua bahan yang membuatnya berhasil — **sifat pilihan rakus** dan ***optimal substructure*** (substruktur optimal) (§9.5). Kita telah membuktikan kedua bahan itu untuk pemilihan aktivitas dengan *argumen pertukaran* (§9.4), melihat kerakusan menang pada *knapsack* pecahan dan kalah pada *knapsack* 0-1 (§9.6), membangun kode bebas awalan Huffman yang optimal (§9.7), dan berkenalan dengan **matroid**, struktur yang dalam satu teorema menjelaskan mengapa sekeluarga algoritma rakus itu benar (§9.8).

Dari **Bab 10** kita memperoleh graf: kosakatanya (simpul, sisi, derajat, lintasan, siklus, komponen, graf bipartit), tiga cara menyimpan graf beserta biaya masing-masing (§10.5), dan kedua pencarian yang mendasar. **Pencarian melebar** menjelajah dalam cincin-cincin berjarak makin jauh dan menghitung cacah lompatan terpendek (§10.7); **pencarian mendalam** menyelam sedalam-dalamnya, membubuhkan pada setiap simpul waktu temu dan waktu selesai, serta menggolongkan setiap sisi sebagai sisi pohon, mundur, maju, atau silang (§10.8). Dengan keduanya kita dapat menemukan komponen, mendeteksi siklus, dan menguji kebipartitan dalam waktu linear (§10.9).

Dari **Bab 11** kita memperoleh **graf berarah taksiklik** dan teorema DAG (§11.2), **pengurutan topologis** menurut waktu selesai pencarian mendalam dan menurut algoritma derajat-masuk Kahn (§11.3), serta **pohon merentang minimum**: sifat potongan yang membuat kerakusan aman (§11.7), **Prim–Jarník**, yang menumbuhkan satu pohon (§11.9), dan **Kruskal**, yang menggabungkan sebuah hutan dengan bantuan struktur himpunan saling lepas yang disetel dengan penyatuan menurut peringkat dan pemampatan lintasan (§11.10). Semuanya mengikuti penyajian Cormen dkk. [1].

12.1.2 Peta pengulangan

Tabel 12.1 mendaftar setiap gagasan Bab 9 sampai 11 yang dapat ditanyakan sebuah persoalan, subbab yang mengajarkannya, pertanyaan yang khas, dan tempat di bab ini sebuah persoalan dari jenis itu persis dikerjakan. Pakailah dalam dua arah: sebelum mulai, untuk menemukan lubang Anda sendiri; sesudahnya, ketika sebuah latihan di §12.7 menjebak Anda, untuk menemukan halaman yang membetulkannya.

Tabel 12.1 Setiap gagasan Bab 9 sampai 11, subbab yang mengajarkannya, jenis pertanyaan yang dimunculkannya, dan tempat di bab ini persoalan dari jenis itu dikerjakan tuntas.

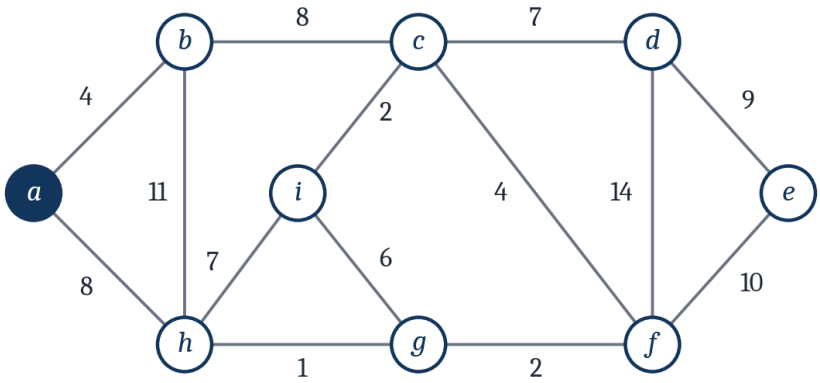
Gagasan	Diajarkan di	Pertanyaan yang khas	Dikerjakan di sini
Apa yang membuat sebuah algoritma *greedy*	§9.2	Definisikan sifat pilihan rakus	§12.3.2
Kegagalan *greedy*: menukar uang kembalian	§9.3	Berilah sistem koin tempat kerakusan kalah, beserta jumlahnya	§12.3.2
Pemilihan aktivitas, empat aturan	§9.4.3	Aturan mana yang keliru, dan pada contoh apa	§12.3.3
Argumen pertukaran	§9.4.7	Buktikan aturan waktu-selesai-terawal itu optimal	§12.3.3, §12.3.4
Optimal substructure	§9.4.4, §9.5.3	Nyatakan dan buktikan untuk sebuah persoalan tertentu	§12.3.3, §12.3.6
Knapsack pecahan lawan 0-1	§9.6	Mengapa kerakusan berhasil pada yang satu dan tidak pada yang lain	§12.3.6
Konstruksi dan biaya Huffman	§9.7	Bangunlah pohonnya, hitunglah $B(T)$	§12.3.5
Matroid, penjadwalan waktu satuan	§9.8	Kebebasan, sifat pertukaran, jadwal rakusnya	§12.3.7
Kosakata dan penyimpanan graf	§10.4, §10.5	Derajat, lintasan, siklus; biaya daftar lawan matriks	§12.4.2
Pencarian melebar	§10.7	Telusurilah; buktikan $d[v] = \delta(s, v)$	§12.2.8, §12.4.2
Pencarian mendalam, cap waktu	§10.8	Telusurilah; teorema kurung	§12.4.3, §12.4.4
Penggolongan sisi	§10.8.10	Golongkan tiap busur; golongan mana yang dapat dimiliki graf tak berarah	§12.4.3, §12.4.4
Komponen, siklus, kebipartitan	§10.9	Warnai sebuah graf dengan dua warna; sebutkan siklus ganjilnya	§12.4.5–§12.4.7
DAG dan teorema DAG	§11.2	Buktikan: taksiklik bila dan hanya bila DFS tak menemukan sisi mundur	§12.5.2
Pengurutan topologis, kedua algoritmanya	§11.3	Hasilkan urutan dengan masing-masingnya; katakan mengapa keduanya berbeda	§12.2.8, §12.5.3, §12.5.4
Pohon merentang, $\|V\| - 1$ sisi	§11.4	Cacahlah sisinya; cacahlah pohonnya	§12.2.5, §12.4.5
Sifat potongan	§11.7	Nyatakan, buktikan, terapkan	§12.6.5
Prim–Jarník	§11.9	Telusurilah dengan antrean prioritasnya	§12.6.3, §12.6.4
Kruskal dan hutan himpunan saling lepas	§11.10	Telusurilah dengan `Find` dan `Union`	§12.2.8, §12.6.2

Tabel 12.1 (Lanjutan) *Setiap gagasan Bab 9 sampai 11, subbab yang mengajarkannya, jenis pertanyaan yang dimunculkannya, dan tempat di bab ini persoalan dari jenis itu dikerjakan tuntas.*

Gagasan	Diajarkan di	Pertanyaan yang khas	Dikerjakan di sini
Penyatuan menurut peringkat, pemampatan lintasan	§11.10.5–7	Gambarlah hutannya sesudah tiap operasi	§12.6.6
Ketunggalan pohon merentang minimum	§11.12	Apakah ia tunggal? Hasilkan yang kedua	§12.6.7

Jika ada satu gambar yang membawa seluruh Bab 11, gambar itu adalah graf berbobot bersimpul sembilan yang dipakai untuk setiap penelusuran pohon merentang minimum di sana, direproduksi sebagai **Gambar 12.1**. Jawabannya — pohon berbobot 37 — layak dihafalkan,

sebab itulah tolok ukur yang dapat Anda pakai untuk memeriksa implementasi Prim atau Kruskal mana pun dalam tiga puluh detik. Persoalan Terkerjakan T di §12.6.4 menjalankan Prim padanya sekali lagi, kali ini dengan tumpukan biner, dan tiba pada 37 dari dua simpul awal yang berbeda.



Gambar 12.1 Graf berbobot dengan sembilan simpul dan empat belas sisi yang dipakai untuk setiap penelusuran pohon merentang minimum di Bab 11. Pohon merentang minimumnya berbobot 37, dan ada dua pohon berbeda dengan bobot itu — dan itulah sebabnya §11.12 ada.

12.1.3 Susunan bab ini

Bab ini terdiri atas empat lapis.

1. **Satu proyek perancangan, dikerjakan dari awal sampai akhir** (Subbab 12.2). Sebuah persoalan nyata untuk Profitina Laundry, studi kasus pengajaran buku ini, dibawa melewati sembilan tahap: persoalannya dinyatakan, datanya dimodelkan sebagai graf, algoritmanya dipilih dengan menimbang alternatif yang disebut namanya, pseudokodenya, kompleksitasnya diramalkan lebih dahulu, rancangannya dibuktikan dengan tangan pada data yang sesungguhnya, implementasinya, evaluasinya, dan kesimpulannya. Proyek ini memakai pencarian melebar, algoritma Kruskal dengan hutan himpunan saling lepas, dan pengurutan topologis Kahn di atas satu himpunan data, dan ia memperlihatkan arti "menerapkan sebuah algoritma" ketika tak seorang pun memberi tahu Anda algoritma mana yang berlaku.
2. **Dua puluh tiga persoalan terkerjakan lainnya** (Subbab 12.3 sampai 12.6), dikelompokkan menurut bab: enam tentang algoritma *greedy* (A sampai F), enam tentang menelusuri graf (G sampai L), lima tentang DAG dan pengurutan topologis (M sampai Q), dan enam tentang pohon merentang minimum (R sampai W). Masing-masing disusun di atas data yang belum pernah dipakai buku ini, sehingga mengerjakannya benar-benar latihan dan bukan sekadar mengingat kembali, dan setiap kelompok ditutup dengan satu program lengkap yang menghitung setiap jawaban dalam kelompok itu.
3. **Tiga puluh latihan uji diri singkat** (Subbab 12.7), masing-masing satu baris, meliputi setiap gagasan dalam **Tabel 12.1**.
4. **Penutup yang biasa**: di mana gagasan yang sama muncul dalam sebuah produk nyata (Subbab 12.8), kesalahan yang sering terjadi (Subbab 12.9), ringkasan (Subbab 12.10), lima belas soal latihan (Subbab 12.11), dan penyelesaian lengkapnya (Subbab 12.12).

CARA MEMAKAI BAB INI. Bacalah pernyataan setiap persoalan, lalu tutuplah penyelesaiannya dengan selembar kertas dan cobalah sendiri selama sepuluh menit sebelum melanjutkan membaca. Penyelesaian yang telah Anda gumuli selama sepuluh menit akan dipahami dengan cara yang tidak pernah dicapai oleh penyelesaian yang hanya Anda baca. Varian *uji diri* di akhir setiap persoalan disediakan supaya Anda dapat menguji pemahaman itu pada sesuatu yang belum pernah Anda lihat dikerjakan; jawabannya menyusul seketika, jadi tutuplah juga.

12.2 Sebuah Proyek Perancangan, Dikerjakan dari Awal sampai Akhir

12.2.1 Apa itu proyek perancangan, dan kesembilan tahapnya

Soal di buku teks memberi tahu Anda apa yang harus dihitung. Persoalan nyata tidak. Ia datang sebagai pertanyaan usaha — *zona mana yang dapat kita layani, jalan mana yang harus kita operasikan, secepat apa sebuah kantong dapat diputar tuntas?* — dan langkah pertama yang paling sulit adalah **mengenalinya** sebagai contoh dari sesuatu yang pernah diajarkan kepada Anda. Keterampilan itu jarang dilatih oleh soal-soal pendek, sebab soal pendek sudah melakukan pengenalan itu untuk Anda. Subbab ini melatihnya pada satu persoalan, dari halaman kosong sampai program yang terukur.

Pekerjaannya terbagi dalam sembilan tahap, dan masing-masing menjawab satu pertanyaan. **Tabel 12.2** mendaftarnya; anak-anak subbab berikutnya mengerjakannya secara berurutan.

Tabel 12.2 Kesembilan tahap sebuah proyek perancangan, pertanyaan yang dijawab masing-masing tahap, dan bukti yang dihasilkan masing-masingnya. Tahap 1 sampai 6 berlangsung sebelum ada kode apa pun.

Tahap	Pertanyaan yang dijawabnya	Apa yang harus dihasilkannya
1. Persoalan	*Apa persisnya yang ditanyakan?*	Pertanyaan-pertanyaannya, dengan kata-kata sederhana, beserta masukan dan keluarannya yang disebut namanya
2. Model	*Objek matematis apakah ini?*	Datanya dinyatakan sebagai graf, himpunan, atau barisan, dengan setiap lambang didefinisikan
3. Pemilihan algoritma	*Algoritma yang mana, dan mengapa bukan yang lain?*	Algoritmanya disebut namanya, dan sekurang-kurangnya satu alternatif ditolak beserta alasannya
4. Pseudokode	*Apa yang akan dikerjakan program, baris demi baris?*	Prosedur bernomor yang cukup tepat untuk ditelusuri dengan tangan
5. Kompleksitas	*Berapa ongkosnya, sebelum kita menjalankannya?*	Waktu jalan setiap fase, diramalkan lebih dahulu
6. Pembuktian dengan tangan	*Apakah rancangannya berhasil pada data yang sesungguhnya?*	Algoritmanya ditelusuri dengan tangan pada masukan yang sebenarnya, dengan setiap langkah dituliskan
7. Implementasi	*Apakah ia berjalan, dan apakah ia algoritma yang kita rancang?*	Program lengkap yang keluarannya dapat dibandingkan dengan penelusuran tangan
8. Evaluasi	*Bagaimana kita tahu ia benar, dan bagaimana kita tahu ia cepat?*	Bukti kebenaran pada banyak masukan, dan waktu terukur yang dibandingkan dengan ramalannya
9. Kesimpulan	*Apa yang kita pelajari, dan apa berikutnya?*	Angka-angka utamanya, satu keterbatasan nyata, satu langkah berikutnya yang spesifik

Dua ciri daftar ini layak diberi satu kalimat masing-masing. Pertama, **enam dari sembilan tahap berlangsung sebelum sebaris kode pun ditulis**. Itu bukan birokrasi: rancangan yang diargumentasikan sebelum programnya ada adalah argumen tentang apa yang *seharusnya* dikerjakan program, sedangkan rancangan yang ditulis sesudahnya sekadar menguraikan apa yang *kebetulan* dikerjakannya, dan tidak dapat menangkap kekeliruan. Kedua, **tahap 8 tidak boleh dilewati**. Program yang belum pernah dijalankan terhadap apa pun selain contoh masukannya sendiri belum terbukti berhasil; ia baru terbukti dapat berjalan.

12.2.2 Memilih persoalan yang layak diberi algoritma

Sebelum kesembilan tahap itu dimulai, ada satu pilihan: *apakah persoalan ini memang layak diberi algoritma?* Kegagalan yang jauh paling lazim dalam proyek perancangan pertama adalah algoritma yang sekadar *hiasan*. Sebuah permainan kecil yang menyenangkan memanggil

pencarian melebar pada kisi 5×5 untuk memindahkan satu *sprite*. Algoritmanya hadir; ia tidak *memikul beban*. Pada kisi 5×5 metode apa pun berhasil, jadi tahap rancangan tak punya apa pun untuk diargumentasikan dan tahap evaluasi tak punya apa pun untuk diukur.

Uji yang harus dikenakan pada gagasan mana pun panjangnya satu kalimat.

> **WAWASAN KUNCI: UJI PEMIKUL BEBAN.** *Seandainya masukannya membesar seribu kali lipat, apakah metode naifnya masih akan berhasil?* Jika ya, algoritmanya hiasan dan rancangannya takkan punya apa pun untuk dikatakan. Jika tidak, Anda punya persoalan yang sungguh-sungguh, dan rancangannya menulis dirinya sendiri: inilah metode naifnya, inilah sebabnya ia mati, inilah algoritma yang tidak mati.

Tabel 12.3 mengenakan uji itu pada enam gagasan dari jenis yang sungguh-sungguh diusulkan orang ketika diminta membangun sesuatu dengan algoritma graf.

Tabel 12.3 Enam calon proyek yang dikenai uji pemikul beban. Empat lolos. Dua yang tidak lolos bukanlah program yang buruk; keduanya persoalan *perancangan* yang lemah, sebab keduanya tidak memberi rancangan dan evaluasi apa pun untuk diperdebatkan.

Gagasan	Algoritma	Metode naif	Selamatkah pada ×1000?	Vonis
Sprite berjalan di labirin 5×5	BFS	coba setiap lintasan	ya, dengan mudah	hiasan
Rute jemputan di jalan-jalan kota	BFS, MST	coba setiap upahimpunan jalan	tidak — 2^E	**pemikul beban**
Minesweeper "buka wilayah kosong"	*flood fill* (DFS/BFS)	pindai ulang papannya tiap klik	di batas	lemah
Perencana prasyarat mata ajar	pengurutan topologis	tebak urutannya, lalu periksa	tidak — $n!$ urutan	**pemikul beban**
Jaringan serat termurah untuk n kantor	Kruskal atau Prim	cacah semua pohon merentang	tidak — Cayley: n^{n-2}	**pemikul beban**
Jejaring sosial "siapa di lingkaran saya"	komponen, BFS	kueri keterjangkauan berulang	tidak — $O(V(V+E))$ per tumpak	**pemikul beban**

Proyek yang dikerjakan di subbab ini adalah baris kelima yang disilangkan dengan yang kedua: **Rute Laundry**, perencana penjemputan dan pengantaran hari-yang-sama untuk Profitina Laundry. Ia dekat dengan gagasan yang paling umum dari semuanya — *aplikasi yang menghitung rute pengantaran terbaik antara dua lokasi* — supaya Anda dapat melihat persis betapa jauh lebih banyak yang dapat dikatakan tentang gagasan itu daripada yang biasanya dikatakan percobaan pertama.

12.2.3 Tahap 1: persoalannya, dinyatakan

Profitina Laundry mengoperasikan satu gerai di Sukolilo, dekat kampus ITS, dan melayani kelurahan-kelurahan di sekitarnya dengan kurir bermotor. Tiga pertanyaan tiba pada pagi yang sama.

1. **Zona mana yang dapat dijanjikan penjemputan dua lompatan**, dan mana yang sama sekali tak terjangkau?
2. Jika perusahaannya menginginkan **jaringan antar-jemput sendiri** supaya kantong mana pun dapat berpindah dari zona terlayani mana pun ke zona terlayani mana pun, **jalan mana yang harus dioperasikannya** — mengingat bahwa setiap kilometer yang dioperasikan memakan biaya setiap hari, selamanya?

3. Sesudah itu sebuah kantong melewati serangkaian tahap pemrosesan tetap dengan kendala urutan. **Dalam urutan apa saja tahap-tahap itu boleh dijalankan, dan secepat apa satu kantong dapat diputar tuntas?**

Masukannya adalah daftar zona, daftar jalan beserta panjangnya, daftar tahap pemrosesan beserta durasinya, dan daftar kendala urutan antartahap. Keluarannya adalah cacah lompatan per zona, sebuah himpunan jalan, sebuah urutan tahap, dan sebuah waktu putar.

12.2.4 Tahap 2: modelnya

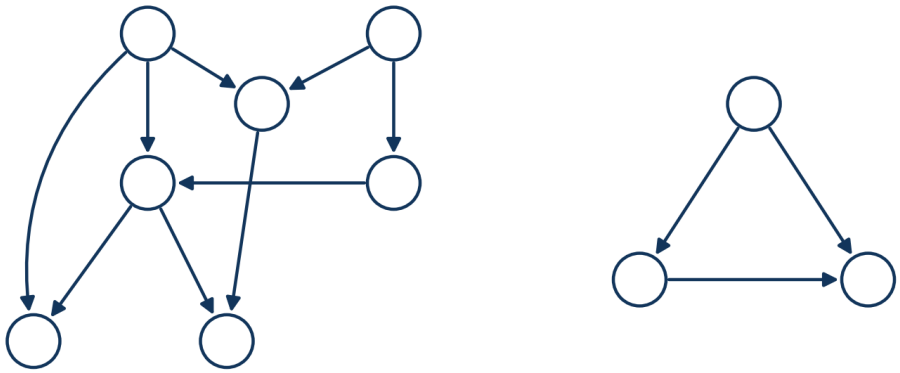
Ketiga pertanyaan itu persoalan graf, dan satu himpunan data menjawab ketiganya.

Jaringan jalannya adalah **graf tak berarah** $G = (V, E)$: simpulnya adalah keempat belas zona, dan $\{u, v\} \in E$ bila seorang kurir dapat berkendara langsung di antara keduanya. Untuk pertanyaan 1 sisinya tak berbobot, sebab setiap kali berkendara berongkos waktu kurang lebih sama. Untuk pertanyaan 2 tiap sisi membawa bobot $w(u, v)$, yaitu jarak jalannya dalam ratusan meter. **Tabel 12.4** adalah datanya.

Tabel 12.4 Ketujuh belas jalan pada contoh Rute Laundry, dengan jarak dalam ratusan meter. Empat belas zona; enam belas jalan terletak di dalam satu komponen dan yang ketujuh belas, Bulak–Kenjeran, adalah pulau tersendiri.

Jalan	w	Jalan	w	Jalan	w
Sukolilo–Menur	8	Nginden–Manyar	6	Mulyosari–Kalijudan	5
Sukolilo–Keputih	6	Klampis–Manyar	3	Kalijudan–DukuhSetro	8
Menur–Gebang	7	Klampis–Mulyosari	13	Manyar–DukuhSetro	9
Menur–Semolowaru	9	Keputih–Kejawan	12	Kejawan–Mulyosari	7
Keputih–Gebang	5	Gebang–Klampis	10	Bulak–Kenjeran	4
Semolowaru–Nginden	4	Semolowaru–Klampis	11		

Alur pemrosesannya adalah **graf berarah**: simpulnya adalah kesepuluh tahap, dan busur $x \to y$ berarti *x harus selesai sebelum y boleh mulai*. Ia harus taksiklik, atau spesifikasinya mustahil dipenuhi — dan justru itulah sesuatu yang semestinya diperiksa programnya alih-alih diandaikan. **Gambar 12.2** memperlihatkan apa arti taksiklik — termasuk satu kasus yang tampak seperti gelung padahal bukan.



Gambar 12.2 Dua graf berarah taksiklik. Di kiri, ikutilah anak panahnya dari mana pun dan Anda akan kehabisan anak panah. Di kanan, tiga anak panah membentuk segitiga yang tetap taksiklik, sebab arahnya tidak berputar searah mengelilinginya; balikkan anak panah bawahnya dan ketiganya akan menutup sebuah siklus berarah, dan spesifikasi alur kerja yang berbentuk seperti itu tidak dapat dijalankan dalam urutan apa pun.

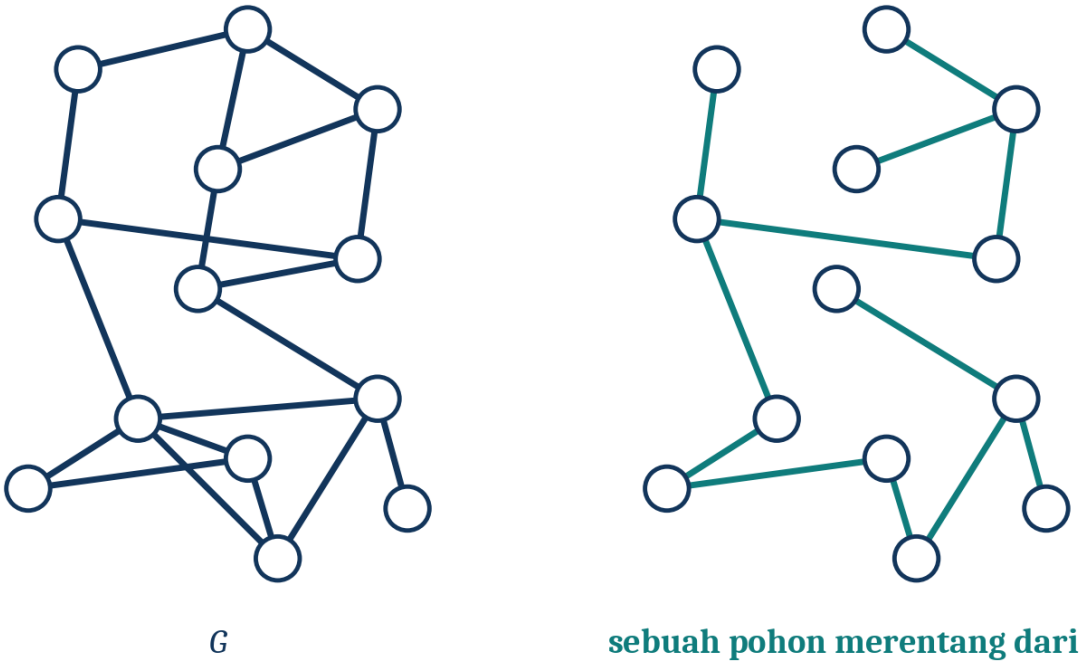
12.2.5 Tahap 3: algoritmanya dipilih, dan alternatif yang ditolak

Pertanyaan 1: pencarian melebar dari gerainya. Grafnya tak berbobot, jadi banyaknya lompatan dari gerai ke sebuah zona persis sama dengan jarak melebar $d[v]$, dan BFS menghitung semuanya dalam satu lintasan seharga $\Theta(V + E)$ (§10.7.7). Zona mana pun yang tertinggal dengan $d[v] = \infty$ berada di komponen terhubung yang lain; itu bukan persoalan penjadwalan melainkan persoalan gerai kedua.

Ditolak: algoritma Dijkstra (Bab 13). Ia benar di sini, tetapi ia berongkos $O((V + E)\lg V)$ untuk menghitung sesuatu yang diperoleh BFS dalam waktu linear, sebab dengan bobot satuan antrean prioritasnya merosot menjadi antrean biasa. Pakailah perkakas yang lebih lemah ketika yang lebih lemah itu mencukupi.

Ditolak juga: menghitung jarak antara semua pasangan zona, seharga $\Theta(V^3)$ dengan metode Floyd–Warshall. Kita punya satu sumber, bukan V sumber.

Pertanyaan 2: Kruskal dengan hutan himpunan saling lepas. Algoritma Kruskal [2] adalah jawabannya. Kita menginginkan subgraf perentang yang terhubung, taksiklik, dan berbobot total minimum — yaitu definisi pohon merentang minimum (§11.5). **Gambar 12.3** memperlihatkan arti "pohon merentang" itu.



Gambar 12.3 Sebuah pohon merentang. **Kiri:** sebuah graf. **Kanan:** sebuah pohon merentangnya — semua simpul hadir, masih terhubung, tanpa siklus; tiga belas simpul dan dua belas sisi, yaitu aturan $|V| - 1$ pada §11.4.2. Tulang punggung antar-jemput pertanyaan 2 persis benda ini, dipilih seringan mungkin.

Ditolak: mencacah pohon merentangnya lalu memungut yang teringan. Banyaknya pohon merentang graf lengkap K_n atas n simpul diberikan oleh rumus Cayley,

$$\tau(K_n) = n^{n-2}, \tag{12.1}$$

jadi menurut Persamaan (12.1) graf lengkap bersimpul dua belas saja sudah punya $12^{10} =$ 61 917 364 224 pohon merentang. Ini lolos uji pemikul beban §12.2.2 dengan jarak berlebih.

Ditolak: Prim–Jarník [3]. Ia sama benarnya dan memberikan pohon berbobot sama (§11.11). Kruskal dipilih sebab datanya tiba sebagai **daftar sisi** — tim operasional mencatat satu baris per

jalan yang dilewati — dan Kruskal melahap daftar sisi secara langsung, sedangkan Prim harus lebih dahulu membangun struktur ketetanggaan. Pada graf jarang kedua biayanya, $O(E\lg E)$ dan $O(E\lg V)$, berselisih paling banyak sebuah faktor tetapan, jadi faktor penentunya adalah format masukannya, dan itulah argumen yang harus dituliskan.

Pertanyaan 3: pengurutan topologis derajat-masuk Kahn [4]. Urutan kerja yang sah persis sebuah urutan topologis DAG prasyaratnya (§11.3). Kahn dipilih ketimbang metode mendalam sebab ia mendeteksi spesifikasi siklik *secara gratis*: jika ia mengeluarkan kurang dari $|V|$ tahap, ada siklus, dan tahap-tahap yang gagal dikeluarkannya persis tahap-tahap yang berada pada atau di hilir siklus itu (§11.3.13). Metode mendalam memerlukan uji sisi mundur terpisah untuk mengatakan hal yang sama.

Lintasan yang sama juga memberikan **lintasan kritisnya**: karena simpulnya keluar dalam urutan topologis, satu pelonggaran per busur menghitung rantai terpanjangnya, dan rantai itulah waktu putar minimumnya. Dengan menulis $dur[v]$ untuk durasi tahap v dan $ef[v]$ untuk waktu selesai terawalnya,

$$ef[v] = dur[v] + \max(\{0\} \cup \{\, ef[u] : u \to v \,\}). \tag{12.2}$$

Persamaan (12.2) berkata: sebuah tahap tidak dapat selesai lebih awal daripada durasinya sendiri sesudah prasyaratnya yang paling akhir selesai, dan tahap tanpa prasyarat selesai sesudah durasinya sendiri saja. Karena Kahn mengeluarkan setiap u sebelum v mana pun dengan $u \to v$, setiap $ef[u]$ di ruas kanan sudah final saat v memerlukannya.

12.2.6 Tahap 4: pseudokodenya

Pseudokode 12.1 adalah seluruh rancangannya dalam sebelas baris. Setiap baris bernomor akan punya rumah di dalam program §12.2.9.

Pseudokode 12.1 Rancangan Rute Laundry: pencarian melebar dan komponen untuk wilayah layanan, Kruskal di atas hutan himpunan saling lepas untuk tulang punggungnya, dan algoritma Kahn beserta lintasan kritisnya untuk alur kerjanya.

```
RUTE-LAUNDRY(zones, roads, stages, precedences, outlet)
 1  d, pi  = BFS(zones, roads, outlet)            // pertanyaan 1
 2  comp   = COMPONENTS(zones, roads)             // keterjangkauan
 3  sort roads by weight, ascending               // pertanyaan 2
 4  MAKE-SET(z) for every zone z in comp[outlet]
 5  T = empty set
 6  for each road (u, v, w) in weight order
 7      if FIND-SET(u) != FIND-SET(v)
 8          T = T + {(u, v)};  UNION(u, v)
 9  order, ef = KAHN(stages, precedences)         // pertanyaan 3
10  if |order| < |stages| then report a cyclic specification
11  return d, pi, comp, T, order, max(ef)
```

12.2.7 Tahap 5: kompleksitasnya, dihitung lebih dahulu

Misalkan $n = |V|$ zona, $m = |E|$ jalan, s tahap, dan p kendala urutan. **Tabel 12.5** adalah ramalannya — ditulis sebelum programnya ada, supaya evaluasinya punya sesuatu untuk diuji.

Tabel 12.5 Biaya Rute Laundry yang diramalkan, fase demi fase, ditulis sebelum ada kode yang dikompilasi. Langkah pengurutannya mendominasi, dan §12.2.10 mengukur apakah itulah yang sungguh-sungguh terjadi.

Fase	Biaya	Mengapa
Membangun daftar ketetanggaan	$\Theta(n+m)$	satu lintasan atas daftar sisinya
BFS dari gerainya	$\Theta(n+m)$	tiap simpul diantrekan sekali, tiap sisi dipindai dua kali (§10.7.7)
Komponen terhubung	$\Theta(n+m)$	satu BFS per komponen, himpunan sisinya saling lepas
Mengurutkan jalannya	$\Theta(m\lg m)$	pengurutan berbasis pembandingan; inilah yang mendominasi
m kali `Find` dan $n-1$ kali `Union`	$O(m\,\alpha(n))$	penyatuan menurut peringkat dengan pemampatan lintasan (§11.10.7)
Kahn beserta lintasan kritisnya	$\Theta(s+p)$	tiap busur dilonggarkan sekali
Total	$\Theta(m\lg m+n+s+p)$	

Menjumlahkan baris-barisnya memberikan totalnya

$$T(n,m,s,p)=\Theta(m\lg m+n+s+p), \tag{12.3}$$

dengan α pada baris kelima adalah fungsi Ackermann terbalik §11.10.7, yang paling besar bernilai 4 untuk n mana pun yang akan pernah dijalankan pada perangkat keras fisik. Jadi kerja himpunan saling lepasnya pada hakikatnya linear, dan pada jaringan jalan yang terhubung (dengan $m \geq n-1$) Persamaan (12.3) sekadar $\Theta(m\lg m)$ ditambah alur kerjanya: pengurutannyalah yang sungguh-sungguh menjadi leher botol.

12.2.8 Tahap 6: rancangannya dibuktikan dengan tangan, sebelum ada kode

Inilah tahap yang paling sering dilewati dan paling layak dikerjakan. **Kerjakan algoritmanya dengan tangan pada data yang sesungguhnya, dan simpanlah penelusurannya.** Jika penelusuran tangan dan programnya kelak tidak sepakat, salah satunya keliru, dan Anda akan tahu dalam satu sore alih-alih pada malam sebelum programnya diperlukan.

Pertanyaan 1 dengan tangan. Pencarian melebar dari Sukolilo, daftar ketetanggaannya dalam urutan jalannya dicatat. Awali dengan $d[\text{Sukolilo}]=0$ dan yang lain semuanya ∞; antrekan Sukolilo.

- **Keluarkan Sukolilo.** Tetangganya Menur dan Keputih putih: $d=1$ masing-masing. Antrean: (Menur:1, Keputih:1).
- **Keluarkan Menur.** Sukolilo hitam. Gebang dan Semolowaru putih: $d=2$. Antrean: (Keputih:1, Gebang:2, Semolowaru:2).
- **Keluarkan Keputih.** Sukolilo hitam, Gebang sudah kelabu — *lewati; sentuhan pertama yang menang.* Kejawan putih: $d=2$. Antrean: (Gebang:2, Semolowaru:2, Kejawan:2).
- **Keluarkan Gebang.** Klampis putih: $d=3$. Antrean: (Semolowaru:2, Kejawan:2, Klampis:3).
- **Keluarkan Semolowaru.** Nginden putih: $d=3$; Klampis sudah kelabu. Antrean: (Kejawan:2, Klampis:3, Nginden:3).
- **Keluarkan Kejawan.** Mulyosari putih: $d=3$. Antrean: (Klampis:3, Nginden:3, Mulyosari:3).
- **Keluarkan Klampis.** Manyar putih: $d=4$; Mulyosari kelabu. Antrean: (Nginden:3, Mulyosari:3, Manyar:4).

- **Keluarkan Nginden.** Manyar kelabu. Tak ada yang baru.
- **Keluarkan Mulyosari.** Kalijudan putih: $d = 4$. Antrean: (Manyar:4, Kalijudan:4).
- **Keluarkan Manyar.** DukuhSetro putih: $d = 5$. Antrean: (Kalijudan:4, DukuhSetro:5).
- **Keluarkan Kalijudan.** DukuhSetro kelabu. Tak ada yang baru.
- **Keluarkan DukuhSetro.** Semua tetangganya hitam. Antrean kosong; gelungnya berakhir.

Dua belas simpul dikeluarkan, dan Bulak serta Kenjeran sama sekali tak pernah disentuh. **Tabel 12.6** mengumpulkan jawabannya.

Tabel 12.6 Cincin-cincin melebar di sekeliling gerai Sukolilo, dihitung dengan tangan. Enam zona terletak di dalam janji dua lompatan. Dua zona tak terjangkau, dan itu harus dilaporkan programnya sebagai fakta usaha alih-alih sebagai galat.

d	Zona
0	Sukolilo
1	Menur, Keputih
2	Gebang, Semolowaru, Kejawan
3	Nginden, Klampis, Mulyosari
4	Manyar, Kalijudan
5	DukuhSetro
∞	Bulak, Kenjeran

Pertanyaan 2 dengan tangan. Urutkan keenam belas jalan komponen utamanya menurut bobot lalu jalankan Kruskal, dengan seri diputus oleh urutan kedua ujungnya dalam daftar zona. **Tabel 12.7** adalah seluruh jalannya.

Tabel 12.7 Kruskal pada jalan-jalan Rute Laundry, dengan tangan. Sebelas sisi diterima untuk dua belas zona — persis $|V| - 1$ — dengan total 69 ratus meter, atau 6,9 km. Lima jalan ditolak sebab kedua ujungnya sudah tersambung.

#	Jalan	w	Vonis	Total berjalan
1	Klampis–Manyar	3	terima	3
2	Semolowaru–Nginden	4	terima	7
3	Keputih–Gebang	5	terima	12
4	Mulyosari–Kalijudan	5	terima	17
5	Sukolilo–Keputih	6	terima	23
6	Nginden–Manyar	6	terima	29
7	Menur–Gebang	7	terima	36
8	Kejawan–Mulyosari	7	terima	43
9	Sukolilo–Menur	8	**tolak** — kedua ujungnya sudah dalam satu pohon	43
10	Kalijudan–DukuhSetro	8	terima	51
11	Menur–Semolowaru	9	terima	60
12	Manyar–DukuhSetro	9	terima	69
13–16	Gebang–Klampis 10, Semolowaru–Klampis 11, Keputih–Kejawan 12, Klampis–Mulyosari 13		semuanya ditolak	69

Dua hal patut diperhatikan. Pertama, penolakan di langkah 9 sama sekali tak ada urusannya dengan bobot 8: sisi berikutnya persis, yang juga berbobot 8, diterima. Keputusannya hanya bergantung pada partisinya, yaitu pada zona mana yang sudah tersambung. Kedua, menjalankan

keenam belas jalannya akan berongkos $8 + 6 + 7 + 9 + 5 + 12 + 10 + 4 + 11 + 3 + 13 + 5 + 8 + 9 + 6 + 7 = 123$, jadi tulang punggungnya menghemat $123 - 69 = 54$, yaitu $54/123 = 43.9\%$ jarak harian. Persentase itulah kalimat yang berhak ada di dalam kesimpulannya.

Pertanyaan 3 dengan tangan. Kesepuluh tahapnya, dengan durasi dalam menit, adalah intake 2, weigh 3, sort 5, pretreat 8, wash 30, dry 45, press 12, fold 10, pack 6, dan invoice 4; kendala urutannya adalah intake→weigh, weigh→sort, weigh→invoice, sort→pretreat, sort→wash, pretreat→wash, wash→dry, dry→press, press→fold, fold→pack, dan invoice→pack. Derajat masuknya bermula pada intake 0, weigh 1, sort 1, pretreat 1, wash 2, dry 1, press 1, fold 1, pack 2, invoice 1, jadi hanya *intake* yang berderajat masuk nol dan antreannya bermula dengannya seorang diri. **Tabel 12.8** adalah jalannya.

Tabel 12.8 Algoritma Kahn pada kesepuluh tahap alur kerjanya. Sepuluh tahap dikeluarkan dalam sepuluh putaran, jadi spesifikasinya taksiklik dan urutannya sah. Pada putaran 4 *invoice* dikeluarkan sebelum *pretreat*, bukan karena mendesak, tetapi karena antreannya masuk-pertama-keluar-pertama dan invoice diantrekan lebih dahulu.

Putaran	Keluarkan	Busurnya mengurangi	Baru mencapai nol, diantrekan
1	intake	weigh 1 → 0	weigh
2	weigh	sort 1 → 0, invoice 1 → 0	sort, invoice
3	sort	pretreat 1 → 0, wash 2 → 1	pretreat
4	invoice	pack 2 → 1	—
5	pretreat	wash 1 → 0	wash
6	wash	dry 1 → 0	dry
7	dry	press 1 → 0	press
8	press	fold 1 → 0	fold
9	fold	pack 1 → 0	pack
10	pack	—	—

Menerapkan Persamaan (12.2) saat tiap tahap dikeluarkan memberikan waktu selesai terawal setiap tahap: intake 2; weigh $2 + 3 = 5$; sort $5 + 5 = 10$; invoice $5 + 4 = 9$; pretreat $10 + 8 = 18$; wash $\max(10,18) + 30 = 48$; dry $48 + 45 = 93$; press $93 + 12 = 105$; fold $105 + 10 = 115$; pack $\max(115,9) + 6 = 121$. Rantai terpanjangnya adalah

$$\text{intake} \to \text{weigh} \to \text{sort} \to \text{pretreat} \to \text{wash} \to \text{dry} \to \text{press} \to \text{fold} \to \text{pack},$$

yang berongkos $2 + 3 + 5 + 8 + 30 + 45 + 12 + 10 + 6 = 121$ menit. *Invoice* tidak berada di atasnya, dan itulah temuan usaha yang berguna: penerbitan faktur dapat ditunda sampai $115 - 9 = 106$ menit tanpa menunda kantongnya.

12.2.9 Tahap 7: implementasinya

Sekarang, dan baru sekarang, programnya. **Kode 12.1** adalah implementasi yang lengkap: ia dikompilasi dengan satu perintah, berjalan tanpa argumen, dan mencetak jawaban untuk masing-masing dari ketiga pertanyaannya. Bacalah ia beriringan dengan **Pseudokode 12.1** — setiap baris bernomor di sana punya rumah di sini.

Kode 12.1 Rute Laundry: pencarian melebar dan komponen, Kruskal dengan hutan himpunan saling lepas, dan pengurutan topologis Kahn beserta lintasan kritisnya. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o router router.cpp` lalu jalankan `./router`.

```
// Rute Laundry -- proyek perancangan untuk Profitina Laundry, dikerjakan tuntas.
// Tiga algoritma dari Bab 10 dan 11, atas data satu hari kerja:
//   Bagian A  pencarian melebar atas jaringan jalan: berapa lompatan
//             tiap zona dari gerai, dan zona mana yang tak dicapai kurir?
//   Bagian B  Kruskal dengan hutan himpunan saling lepas: himpunan tautan
//             antar-jemput termurah yang menjaga tiap zona tetap terhubung.
//   Bagian C  pengurutan topologis Kahn atas alur kerja, beserta lintasan
//             kritisnya -- secepat-cepatnya satu kantong dapat dituntaskan.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
using namespace std;

// ---------------------------------------------- jaringan jalan antarzona
static const vector<string> ZONE = {
    "Sukolilo", "Menur", "Keputih", "Gebang", "Semolowaru", "Nginden",
    "Klampis", "Manyar", "Mulyosari", "Kalijudan", "Kejawan", "DukuhSetro",
    "Bulak", "Kenjeran"
};
struct Road { int a, b, hm; };                  // hm = jarak dalam ratusan meter
static const vector<Road> ROAD = {
    {0,1,8}, {0,2,6}, {1,3,7}, {1,4,9}, {2,3,5}, {2,10,12}, {3,6,10},
    {4,5,4}, {4,6,11}, {5,7,6}, {6,7,3}, {6,8,13}, {8,9,5}, {9,11,8},
    {7,11,9}, {10,8,7}, {12,13,4}
};

// ----------------------------------------------------- Bagian A: BFS
struct Bfs { vector<int> d, pi; };

Bfs bfs(const vector<vector<int>>& adj, int s) {
    Bfs R;
    R.d.assign(adj.size(), INT_MAX);
    R.pi.assign(adj.size(), -1);
    R.d[s] = 0;
    queue<int> Q; Q.push(s);
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        for (int v : adj[u])
            if (R.d[v] == INT_MAX) { R.d[v] = R.d[u] + 1; R.pi[v] = u; Q.push(v); }
    }
    return R;
}
```

Kode 12.1 (Lanjutan) *Rute Laundry: pencarian melebar dan komponen, Kruskal dengan hutan himpunan saling lepas, dan pengurutan topologis Kahn beserta lintasan kritisnya. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o router router.cpp lalu jalankan ./router.*

```
string routeTo(const Bfs& R, int v) {                    // telusuri pi mundur, lalu balik
    if (R.d[v] == INT_MAX) return "-- no route --";
    vector<int> p;
    for (int x = v; x != -1; x = R.pi[x]) p.push_back(x);
    reverse(p.begin(), p.end());
    string s;
    for (size_t i = 0; i < p.size(); ++i) { if (i) s += " -> "; s += ZONE[p[i]]; }
    return s;
}

// -------------------------------- Bagian B: hutan himpunan saling lepas
struct DSU {
    vector<int> parent, rnk;
    explicit DSU(int n) : parent(n), rnk(n, 0) {
        for (int i = 0; i < n; ++i) parent[i] = i;
    }
    int find(int x) { return parent[x] == x ? x : parent[x] = find(parent[x]); }
    bool unite(int x, int y) {                            // penyatuan menurut peringkat
        x = find(x); y = find(y);
        if (x == y) return false;
        if (rnk[x] < rnk[y]) swap(x, y);
        parent[y] = x;
        if (rnk[x] == rnk[y]) ++rnk[x];
        return true;
    }
};

// ------------------------------------------- Bagian C: DAG alur kerjanya
static const vector<string> STAGE = {
    "intake", "weigh", "sort", "pretreat", "wash",
    "dry", "press", "fold", "pack", "invoice"
};
static const vector<int> MINUTES = { 2, 3, 5, 8, 30, 45, 12, 10, 6, 4 };
static const vector<pair<int,int>> PREC = {
    {0,1}, {1,2}, {1,9}, {2,3}, {2,4}, {3,4},
    {4,5}, {5,6}, {6,7}, {7,8}, {9,8}
};

int main() {
    const int n = (int)ZONE.size();
    vector<vector<int>> adj(n);
    for (const Road& r : ROAD) { adj[r.a].push_back(r.b); adj[r.b].push_back(r.a); }
```

Kode 12.1 (Lanjutan) *Rute Laundry: pencarian melebar dan komponen, Kruskal dengan hutan himpunan saling lepas, dan pengurutan topologis Kahn beserta lintasan kritisnya. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o router router.cpp lalu jalankan ./router.*

```
    cout << "=== Part A: service area, by breadth-first search from Sukolilo ===\n";
    Bfs R = bfs(adj, 0);
    cout << left << setw(12) << "zone" << right << setw(6) << "hops" << "   route\n";
    for (int v = 0; v < n; ++v) {
        cout << left << setw(12) << ZONE[v] << right << setw(6);
        if (R.d[v] == INT_MAX) cout << "inf"; else cout << R.d[v];
        cout << "   " << routeTo(R, v) << "\n";
    }
    int twoHop = 0;
    for (int v = 0; v < n; ++v) if (R.d[v] <= 2) ++twoHop;
    cout << "zones within the two-hop promise: " << twoHop << " of " << n << "\n";

    // komponen terhubung, lewat BFS berulang
    vector<int> comp(n, -1); int nc = 0;
    for (int v = 0; v < n; ++v) if (comp[v] == -1) {
        queue<int> Q; Q.push(v); comp[v] = nc;
        while (!Q.empty()) { int u = Q.front(); Q.pop();
            for (int w : adj[u]) if (comp[w] == -1) { comp[w] = nc; Q.push(w); } }
        ++nc;
    }
    cout << "connected components: " << nc << "\n";
    for (int c = 0; c < nc; ++c) {
        cout << "  component " << c << ":";
        for (int v = 0; v < n; ++v) if (comp[v] == c) cout << " " << ZONE[v];
        cout << "\n";
    }

    cout << "\n=== Part B: the shuttle backbone, by Kruskal ===\n";
    // hanya komponen yang memuat gerai dapat dilayani satu tulang punggung
    vector<Road> cand;
    for (const Road& r : ROAD) if (comp[r.a] == 0) cand.push_back(r);
    sort(cand.begin(), cand.end(), [](const Road& x, const Road& y) {
        if (x.hm != y.hm) return x.hm < y.hm;
        if (x.a != y.a) return x.a < y.a;
        return x.b < y.b;
    });
    int inComp = 0; for (int v = 0; v < n; ++v) if (comp[v] == 0) ++inComp;
    DSU dsu(n);
    int total = 0, taken = 0, examined = 0;
    cout << setw(4) << "w" << "  edge                      verdict\n";
    for (const Road& r : cand) {
        ++examined;
        bool ok = dsu.unite(r.a, r.b);
        cout << setw(4) << r.hm << "  " << left << setw(24)
             << (ZONE[r.a] + "-" + ZONE[r.b]) << right
             << (ok ? "ACCEPT" : "reject (cycle)") << "\n";
```

Kode 12.1 (Lanjutan) *Rute Laundry: pencarian melebar dan komponen, Kruskal dengan hutan himpunan saling lepas, dan pengurutan topologis Kahn beserta lintasan kritisnya. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o router router.cpp lalu jalankan ./router.*

```
        if (ok) { total += r.hm; ++taken; }
    }
    cout << "edges examined " << examined << ", accepted " << taken
         << " (= " << inComp << " - 1), total " << total
         << " hm = " << fixed << setprecision(1) << total / 10.0 << " km\n";
    int all = 0; for (const Road& r : cand) all += r.hm;
    cout << "all candidate roads would cost " << all << " hm; the backbone saves "
         << all - total << " hm (" << setprecision(1)
         << 100.0 * (all - total) / all << "%)\n";

    cout << "\n=== Part C: the job pipeline, by Kahn ===\n";
    const int m = (int)STAGE.size();
    vector<vector<int>> dag(m); vector<int> indeg(m, 0);
    for (auto [u, v] : PREC) { dag[u].push_back(v); ++indeg[v]; }
    vector<int> ef(m, 0), pred(m, -1), order;
    queue<int> Q;
    for (int v = 0; v < m; ++v) if (indeg[v] == 0) Q.push(v);
    cout << "round  emit       in-degrees after the removal\n";
    int round = 0;
    while (!Q.empty()) {
        int u = Q.front(); Q.pop(); order.push_back(u);
        if (ef[u] == 0) ef[u] = MINUTES[u];                // sebuah tahap sumber
        for (int v : dag[u]) {
            if (ef[u] + MINUTES[v] > ef[v]) { ef[v] = ef[u] + MINUTES[v]; pred[v] = u; }
            if (--indeg[v] == 0) Q.push(v);
        }
        cout << setw(5) << ++round << "  " << left << setw(10) << STAGE[u] << right;
        for (int v = 0; v < m; ++v) cout << " " << indeg[v];
        cout << "\n";
    }
    cout << "emitted " << order.size() << " of " << m
         << (order.size() == (size_t)m ? "  -> acyclic, the order is legal\n"
                                       : "  -> a cycle exists\n");
    cout << "topological order:";
    for (int v : order) cout << " " << STAGE[v];
    cout << "\n";
    cout << "earliest finish, minute by minute:\n";
    for (int v : order)
        cout << "  " << left << setw(10) << STAGE[v] << right << setw(4) << MINUTES[v]
             << " min, done at " << setw(4) << ef[v] << "\n";
    int last = (int)(max_element(ef.begin(), ef.end()) - ef.begin());
    vector<int> crit;
    for (int x = last; x != -1; x = pred[x]) crit.push_back(x);
    reverse(crit.begin(), crit.end());
    cout << "critical path (" << ef[last] << " min):";
    for (size_t i = 0; i < crit.size(); ++i)
```

Kode 12.1 (Lanjutan) *Rute Laundry: pencarian melebar dan komponen, Kruskal dengan hutan himpunan saling lepas, dan pengurutan topologis Kahn beserta lintasan kritisnya. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o router router.cpp lalu jalankan ./router.*

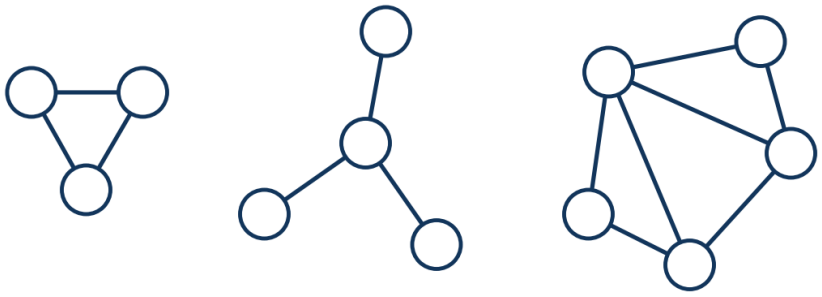
```
        cout << (i ? " -> " : " ") << STAGE[crit[i]];
    cout << "\n";
    return 0;
}
```

Menjalankannya menghasilkan persis ini, pertama untuk pertanyaan 1:

```
=== Part A: service area, by breadth-first search from Sukolilo ===
zone          hops   route
Sukolilo         0   Sukolilo
Menur            1   Sukolilo -> Menur
Keputih          1   Sukolilo -> Keputih
Gebang           2   Sukolilo -> Menur -> Gebang
Semolowaru       2   Sukolilo -> Menur -> Semolowaru
Nginden          3   Sukolilo -> Menur -> Semolowaru -> Nginden
Klampis          3   Sukolilo -> Menur -> Gebang -> Klampis
Manyar           4   Sukolilo -> Menur -> Gebang -> Klampis -> Manyar
Mulyosari        3   Sukolilo -> Keputih -> Kejawan -> Mulyosari
Kalijudan        4   Sukolilo -> Keputih -> Kejawan -> Mulyosari -> Kalijudan
Kejawan          2   Sukolilo -> Keputih -> Kejawan
DukuhSetro       5   Sukolilo -> Menur -> Gebang -> Klampis -> Manyar -> DukuhSetro
Bulak          inf   -- no route --
Kenjeran       inf   -- no route --
zones within the two-hop promise: 6 of 14
connected components: 2
  component 0: Sukolilo Menur Keputih Gebang Semolowaru Nginden Klampis Manyar Mulyosari
Kalijudan Kejawan DukuhSetro
  component 1: Bulak Kenjeran
```

Bandingkan dengan **Tabel 12.6**, yang dihitung dengan tangan satu halaman sebelumnya: setiap jaraknya sepakat, termasuk kedua tak hingganya. Kesepakatan itu adalah baris pertama evaluasinya, dan ia tak berongkos apa pun sebab penelusuran tangannya sudah ada.

Hasil komponennya layak dibaca dua kali. Graf dengan beberapa komponen tampak seperti **Gambar 12.4**: potongan-potongan tanpa sisi di antaranya, dan di dalam tiap potongan setiap simpul menjangkau setiap simpul lain.



Gambar 12.4 Graf dengan tiga komponen terhubung: sebuah segitiga, sebuah bintang, dan sepotong bersimpul lima. Tak ada sisi yang melintas di antaranya. Jaringan jalan Rute Laundry punya dua potongan semacam itu, dan yang kedua — Bulak dengan Kenjeran — adalah sebabnya janji dua lompatan tidak boleh diiklankan untuk seluruh kota.

Bagian B mencetak seluruh jalannya Kruskal, dan ia cocok dengan **Tabel 12.7** baris demi baris:

```
=== Part B: the shuttle backbone, by Kruskal ===
   w  edge                      verdict
   3  Klampis-Manyar          ACCEPT
   4  Semolowaru-Nginden      ACCEPT
   5  Keputih-Gebang          ACCEPT
   5  Mulyosari-Kalijudan     ACCEPT
   6  Sukolilo-Keputih        ACCEPT
   6  Nginden-Manyar          ACCEPT
   7  Menur-Gebang            ACCEPT
   7  Kejawan-Mulyosari       ACCEPT
   8  Sukolilo-Menur          reject (cycle)
   8  Kalijudan-DukuhSetro    ACCEPT
   9  Menur-Semolowaru        ACCEPT
   9  Manyar-DukuhSetro       ACCEPT
  10  Gebang-Klampis          reject (cycle)
  11  Semolowaru-Klampis      reject (cycle)
  12  Keputih-Kejawan         reject (cycle)
  13  Klampis-Mulyosari       reject (cycle)
edges examined 16, accepted 11 (= 12 - 1), total 69 hm = 6.9 km
all candidate roads would cost 123 hm; the backbone saves 54 hm (43.9%)
```

Dan Bagian C:

```
=== Part C: the job pipeline, by Kahn ===
round  emit       in-degrees after the removal
    1  intake     0 0 1 1 2 1 1 1 2 1
    2  weigh      0 0 0 1 2 1 1 1 2 0
    3  sort       0 0 0 0 1 1 1 1 2 0
    4  invoice    0 0 0 0 1 1 1 1 1 0
    5  pretreat   0 0 0 0 0 1 1 1 1 0
    6  wash       0 0 0 0 0 0 1 1 1 0
    7  dry        0 0 0 0 0 0 0 1 1 0
    8  press      0 0 0 0 0 0 0 0 1 0
    9  fold       0 0 0 0 0 0 0 0 0 0
   10  pack       0 0 0 0 0 0 0 0 0 0
emitted 10 of 10  -> acyclic, the order is legal
topological order: intake weigh sort invoice pretreat wash dry press fold pack
earliest finish, minute by minute:
  intake       2 min, done at    2
  weigh        3 min, done at    5
  sort         5 min, done at   10
  invoice      4 min, done at    9
  pretreat     8 min, done at   18
  wash        30 min, done at   48
  dry         45 min, done at   93
  press       12 min, done at  105
  fold        10 min, done at  115
  pack         6 min, done at  121
critical path (121 min): intake -> weigh -> sort -> pretreat -> wash -> dry -> press ->
fold -> pack
```

Tiga rincian di dalam cetakan itu layak diperhatikan.

`find` memampatkan dalam perjalanan pulang. Baris `return parent[x] == x ? x : parent[x] = find(parent[x]);` mengerjakan dua tugas sekaligus: ia menemukan wakilnya, dan dalam perjalanan naik keluar dari rekursinya ia mengarahkan ulang setiap simpul yang dilewatinya langsung ke akar. Satu penugasan itu adalah seluruh isi pemampatan lintasan (§11.10.6).

Sisi yang ditolak pun tetap mengubah hutannya. Pada penolakan `Sukolilo-Menur`, dua panggilan `find` berjalan sebelum perbandingannya gagal, dan keduanya memampatkan lintasannya. Orang yang hanya memandangi kolom terima/tolak takkan pernah melihatnya.

Uji alur kerjanya gratis. `emitted 10 of 10` adalah sertifikat ketaksiklikannya. Tak ada tambahan apa pun yang dihitung untuk memperolehnya; ia jatuh sendiri dari gelung Kahn. Itulah argumen yang dipakai §12.2.5 untuk memilih Kahn ketimbang metode mendalam, kini terlihat di dalam keluarannya.

12.2.10 Tahap 8: evaluasinya

Sebuah evaluasi menjawab tiga pertanyaan, dan tidak ada yang lain: **apakah ia benar, apakah ia menskala seperti yang diramalkan, dan apa untungnya bagi usahanya?** **Kode 12.2** menjawab dua yang pertama secara mekanis.

Kode 12.2 Perkakas evaluasinya: pemeriksaan silang Kruskal terhadap Prim, pemeriksaan pascakondisi pada BFS, dan satu putaran pengukuran waktu pada lima ukuran dengan waktu berlalunya dibagi biaya yang diramalkan. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o routereval routereval.cpp` lalu jalankan `./routereval`.

```
// Perangkat evaluasi untuk proyek perancangan Rute Laundry.
// Ia menjawab tiga pertanyaan yang wajib dijawab setiap evaluasi:
//   1. apakah implementasinya BENAR?  -- Kruskal lawan Prim, 2000 graf,
//                                        dan tiap pascakondisi diperiksa;
//   2. apakah ia MENSKALA seperti ramalan analisisnya? -- lima ukuran;
//   3. apa untungnya bagi USAHA?  -- tulang punggung lawan seluruh jalannya.
#include <iostream>
#include <iomanip>
#include <vector>
#include <queue>
#include <algorithm>
#include <random>
#include <chrono>
#include <climits>
#include <cmath>
using namespace std;
using Clock = chrono::steady_clock;

struct Edge { int a, b, w; };

// ---------------------------------------------------------------- Kruskal
struct DSU {
    vector<int> p, r;
    explicit DSU(int n) : p(n), r(n, 0) { for (int i = 0; i < n; ++i) p[i] = i; }
    int find(int x) { return p[x] == x ? x : p[x] = find(p[x]); }
    bool unite(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return false;
        if (r[x] < r[y]) swap(x, y);
        p[y] = x; if (r[x] == r[y]) ++r[x];
        return true;
    }
};

long long kruskal(int n, vector<Edge> e) {
    sort(e.begin(), e.end(), [](const Edge& x, const Edge& y) { return x.w < y.w; });
    DSU d(n); long long t = 0; int k = 0;
    for (const Edge& x : e) if (d.unite(x.a, x.b)) { t += x.w; if (++k == n - 1) break; }
    return k == n - 1 ? t : -1;
}
```

Kode 12.2 (Lanjutan) *Perkakas evaluasinya: pemeriksaan silang Kruskal terhadap Prim, pemeriksaan pascakondisi pada BFS, dan satu putaran pengukuran waktu pada lima ukuran dengan waktu berlalunya dibagi biaya yang diramalkan. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o routereval routereval.cpp lalu jalankan ./routereval.*

```
// ------------------------------------------------------------------ Prim
long long prim(int n, const vector<Edge>& e) {
    vector<vector<pair<int,int>>> adj(n);
    for (const Edge& x : e) { adj[x.a].push_back({x.b, x.w}); adj[x.b].push_back({x.a,
x.w}); }
    vector<int> key(n, INT_MAX); vector<char> in(n, 1);
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    key[0] = 0; Q.push({0, 0});
    long long t = 0; int taken = 0;
    while (!Q.empty()) {
        auto [k, u] = Q.top(); Q.pop();
        if (!in[u] || k != key[u]) continue;                // entri antrean yang basi
        in[u] = 0; t += k; ++taken;
        for (auto [v, w] : adj[u]) if (in[v] && w < key[v]) { key[v] = w; Q.push({w, v}); }
    }
    return taken == n ? t : -1;
}

// ------------------------------------------------------------------ BFS
vector<int> bfs(const vector<vector<int>>& adj, int s) {
    vector<int> d(adj.size(), INT_MAX);
    d[s] = 0; queue<int> Q; Q.push(s);
    while (!Q.empty()) { int u = Q.front(); Q.pop();
        for (int v : adj[u]) if (d[v] == INT_MAX) { d[v] = d[u] + 1; Q.push(v); } }
    return d;
}

int main() {
    mt19937 rng(20260515);

    cout << "=== Test 1: Kruskal and Prim must agree, on 2000 random graphs ===\n";
    int mismatch = 0;
    for (int t = 0; t < 2000; ++t) {
        int n = 5 + (int)(rng() % 25);
        vector<Edge> e;
        for (int v = 1; v < n; ++v)                         // pohon merentang acak dahulu
            e.push_back({(int)(rng() % v), v, 1 + (int)(rng() % 50)});
        int extra = (int)(rng() % (2 * n));
        for (int i = 0; i < extra; ++i) {
            int a = (int)(rng() % n), b = (int)(rng() % n);
            if (a != b) e.push_back({a, b, 1 + (int)(rng() % 50)});
        }
        if (kruskal(n, e) != prim(n, e)) ++mismatch;
    }
    cout << "mismatches: " << mismatch << " out of 2000\n";
```

Kode 12.2 (Lanjutan) *Perkakas evaluasinya: pemeriksaan silang Kruskal terhadap Prim, pemeriksaan pascakondisi pada BFS, dan satu putaran pengukuran waktu pada lima ukuran dengan waktu berlalunya dibagi biaya yang diramalkan. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o routereval routereval.cpp lalu jalankan ./routereval.*

```
    cout << "\n=== Test 2: the BFS postcondition, on 2000 random graphs ===\n";
    int broken = 0;
    for (int t = 0; t < 2000; ++t) {
        int n = 5 + (int)(rng() % 25);
        vector<vector<int>> adj(n);
        for (int v = 1; v < n; ++v) { int u = (int)(rng() % v);
            adj[u].push_back(v); adj[v].push_back(u); }
        int extra = (int)(rng() % (2 * n));
        for (int i = 0; i < extra; ++i) { int a = (int)(rng()%n), b = (int)(rng()%n);
            if (a != b) { adj[a].push_back(b); adj[b].push_back(a); } }
        vector<int> d = bfs(adj, 0);
        for (int u = 0; u < n; ++u) for (int v : adj[u])
            if (abs(d[u] - d[v]) > 1) ++broken;              // tak ada sisi melintasi dua
tingkat
    }
    cout << "edges spanning more than one BFS level: " << broken << "\n";

    cout << "\n=== Test 3: does it scale as the analysis says? ===\n";
    cout << setw(9) << "zones" << setw(10) << "roads"
         << setw(12) << "BFS ms" << setw(12) << "Kruskal ms"
         << setw(15) << "BFS/(V+E) us" << setw(18) << "Kruskal/(ElgE) us" << "\n";
    for (int n : {2000, 4000, 8000, 16000, 32000}) {
        int m = 4 * n;
        vector<Edge> e; vector<vector<int>> adj(n);
        for (int v = 1; v < n; ++v) {
            int u = (int)(rng() % v), w = 1 + (int)(rng() % 1000);
            e.push_back({u, v, w}); adj[u].push_back(v); adj[v].push_back(u);
        }
        while ((int)e.size() < m) {
            int a = (int)(rng()%n), b = (int)(rng()%n);
            if (a == b) continue;
            int w = 1 + (int)(rng() % 1000);
            e.push_back({a, b, w}); adj[a].push_back(b); adj[b].push_back(a);
        }
        auto t0 = Clock::now(); volatile int sink = bfs(adj, 0)[n-1]; (void)sink;
        auto t1 = Clock::now(); volatile long long s2 = kruskal(n, e); (void)s2;
        auto t2 = Clock::now();
        double bms = chrono::duration<double, milli>(t1 - t0).count();
        double kms = chrono::duration<double, milli>(t2 - t1).count();
        double VE = n + 2.0 * m, ElgE = m * log2((double)m);
        cout << setw(9) << n << setw(10) << m << fixed << setprecision(3)
             << setw(12) << bms << setw(12) << kms << setprecision(4)
             << setw(15) << 1000.0 * bms / VE
             << setw(18) << 1000.0 * kms / ElgE << "\n";
    }
    return 0;
}
```

Keluarannya:

```
=== Test 1: Kruskal and Prim must agree, on 2000 random graphs ===
mismatches: 0 out of 2000

=== Test 2: the BFS postcondition, on 2000 random graphs ===
edges spanning more than one BFS level: 0

=== Test 3: does it scale as the analysis says? ===
    zones     roads      BFS ms  Kruskal ms   BFS/(V+E) us Kruskal/(ElgE) us
     2000      8000       0.084       0.446         0.0047            0.0043
     4000     16000       0.158       0.921         0.0044            0.0041
     8000     32000       0.355       1.771         0.0049            0.0037
    16000     64000       0.742       3.731         0.0052            0.0037
    32000    128000       1.886       7.174         0.0065            0.0033
```

Sekarang bacalah, sebab angka yang dicetak tetapi tidak dibaca bukanlah evaluasi.

Uji 1 adalah pemeriksaan silang, bukan pembuktian. Prim dan Kruskal adalah dua algoritma berbeda yang harus sepakat tentang *bobot* pohon merentang minimumnya, walaupun belum tentu tentang sisi-sisinya (§11.12.1). Dua ribu kesepakatan adalah bukti kuat bahwa tak ada implementasi yang mengandung kutu sistematis. Ia bukan bukti bahwa keduanya benar: dua implementasi yang sama-sama keliru dapat saja sepakat. Katakanlah begitu — kejujuran tentang seberapa kuat bukti Anda adalah bagian dari bukti itu sendiri.

Uji 2 adalah pascakondisi, dan itulah jenis bukti yang lebih baik, sebab ia memeriksa sifat yang harus berlaku pada jawaban *yang sesungguhnya* alih-alih pada implementasi yang lain. Jika d adalah jarak melebarnya, maka untuk setiap sisi $\{u, v\}$

$$|\, d[u] - d[v] \,| \leq 1. \tag{12.4}$$

Persamaan (12.4) berlaku sebab sebuah sisi tidak dapat melompati satu tingkat: jika $d[v] \geq d[u] + 2$, maka ketika u dikeluarkan ia sudah akan menemukan v pada jarak $d[u] + 1$, sebuah kontradiksi. Nol pelanggaran pada dua ribu graf adalah hasil yang sungguh-sungguh.

Uji 3 adalah klaim penskalaannya, diuji. Rancangannya meramalkan $\Theta(V + E)$ untuk BFS dan $\Theta(E \lg E)$ untuk Kruskal. Kedua kolom terakhirnya membagi waktu terukurnya persis dengan ramalan itu. Jika ramalannya benar, kolom-kolom itu seharusnya tinggal kira-kira tetap sementara masukannya membesar enam belas kali lipat. Waktu mutlaknya bergantung pada mesinnya dan berubah beberapa puluh persen dari satu putaran ke putaran berikutnya, jadi kedua kolom nisbah itulah, bukan milidetiknya, yang membawa klaimnya. Kolom Kruskal berbunyi 0.0043,0.0041,0.0037,0.0037,0.0033: pada hakikatnya datar, dan sedikit melayang ke *bawah* sebab biaya awal yang tetap tersebar atas kerja yang lebih banyak. BFS berbunyi 0.0047,0.0044,0.0049,0.0052,0.0065: datar pada tiga ukuran pertama lalu menanjak, sekitar 38% antara masukan terkecil dan terbesar.

Tanjakan itu bukan kutu dan bukan kegagalan analisisnya, dan menjelaskannya dengan benar adalah tanda evaluasi yang memahami angka-angkanya sendiri.

> **WAWASAN KUNCI: MENGAPA ALGORITMA LINEAR BERHENTI TAMPAK LINEAR.** $\Theta(V + E)$ mencacah *operasi*. Ia tidak mencacah *cache miss*. Pada $|V| = 32\,000$ dan $|E| = 128\,000$ struktur ketetanggaannya tak lagi muat di dalam *cache* tercepat prosesornya, jadi tiap akses memori yang jumlahnya sama itu berongkos lebih banyak nanodetik daripada sebelumnya. Cacah operasinya masih linear; *waktu per operasi*-nyalah yang naik. Evaluasi yang jujur melaporkan keduanya, dan meramalkan bahwa nisbahnya akan mendatar lagi begitu seluruh strukturnya keluar dari *cache*.

Jawaban usahanya melengkapi evaluasi itu: tulang punggungnya 6,9 km terhadap 12,3 km jalan calon, yaitu pengurangan 43,9% jarak yang ditempuh tiap hari, dan enam dari keempat belas zona duduk di dalam janji dua lompatan sementara dua zona sama sekali tak dapat dilayani. Ketiga angka itu, bukan waktunya, yang dibaca pemilik binatunya.

12.2.11 Tahap 9: kesimpulannya

Satu paragraf, empat kalimat. Ia menyatakan apa yang dibangun, berapa angkanya, di mana batasnya, dan apa berikutnya.

> **KESIMPULAN.** Rute Laundry memodelkan persoalan penjemputan Profitina Laundry sebagai tiga persoalan graf di atas satu himpunan data, dan menyelesaikannya dengan pencarian melebar, algoritma Kruskal di atas hutan himpunan saling lepas, dan pengurutan topologis Kahn, dengan biaya total yang diramalkan $\Theta(m \lg m)$ yang didominasi pengurutan sisinya. Pada contoh empat belas zonanya, ia menempatkan enam zona di dalam janji dua lompatan, mengenali dua zona yang sama sekali tak dicapai rute kurir mana pun, memilih tulang punggung sebelas jalan sepanjang 6,9 km terhadap 12,3 km jalan calon — pengurangan 43,9% — dan memperlihatkan lintasan kritis 121 menit menyusuri alur cuci dengan kelonggaran 106 menit pada tahap penerbitan faktur. Keterbatasan utamanya adalah model lompatan tak berbobotnya: berkendara melewati Gebang pukul 07.30 tidak berongkos sama dengan berkendara yang sama pukul 14.00, jadi janji dua lompatan itu adalah batas bawah waktu tempuh dan bukan jaminan. Langkah berikutnya yang wajar karena itu adalah Bab 13: ganti cacah lompatannya dengan bobot waktu tempuh dan ganti BFS dengan algoritma Dijkstra, lalu jalankan ulang perkakas evaluasi yang sama terhadap jawaban yang baru.

Perhatikan apa yang *tidak* dikerjakan paragraf itu. Ia tidak meringkas algoritmanya — tahap-tahap sebelumnya sudah melakukannya. Ia tidak meminta maaf. Ia menyebut satu keterbatasan yang nyata dan satu langkah berikutnya yang spesifik, dan itulah seluruh isi sebuah kesimpulan.

12.2.12 Lima jebakan proyek perancangan, dan satu uji diri

1. **Algoritma hiasan.** Mengenakan BFS pada papan yang begitu kecil sehingga metode apa pun berhasil. Kenakan uji pemikul beban §12.2.2 *sebelum* Anda mulai membangun.
2. **Rancangan yang ditulis terakhir.** Ia terbaca sebagai uraian alih-alih argumen, dan ia takkan memuat alternatif yang ditolak. Tulislah tahap 1 sampai 6 lebih dahulu, dan simpanlah versi yang Anda tulis lebih dahulu itu.
3. **Evaluasi yang hilang.** Program yang hanya dijalankan pada contoh masukannya sendiri belum terbukti benar. Dua jam kerja — orakel gaya kasar atau pascakondisi, beberapa ribu masukan acak, pengukuran waktu pada lima ukuran — mengubah pernyataan menjadi bukti.
4. **Pengukuran yang tidak dibaca.** Tabel waktu yang dicetak tanpa nisbahnya terhadap ramalan, atau dengan nisbah tetapi tanpa komentar tentang pergeserannya, adalah data, bukan evaluasi.
5. **Masukan merosot yang tak diperiksa.** Apa yang dikerjakan program pada graf kosong, satu simpul tunggal, graf yang tak terhubung, sebuah "DAG" yang siklik? **Kode 12.1** menjawab dua yang terakhir: ia mencetak dua komponen alih-alih ambruk, dan ia mencetak `emitted 10 of 10` alih-alih mengandaikan ketaksiklikan. Masing-masingnya satu baris kode dan satu kalimat dalam tulisan laporannya.

> **UJI DIRI.** (a) Tim operasional mengaspal ulang jalan Sukolilo–Menur, dan panjangnya turun dari 8 menjadi 5 ratus meter. Jalankan ulang Kruskal pada **Tabel 12.4**: jalan mana yang keluar dari tulang punggung, mana yang masuk, dan berapa total barunya? (b) Terpisah dari itu, penerbitan faktur dipindahkan ke sistem yang lebih lambat dan kini memakan 110 menit alih-alih 4. Apa yang terjadi pada waktu putar dan pada lintasan kritisnya?

Jawaban. (a) Sukolilo–Menur (5) kini diperiksa ketiga, seri dengan Keputih–Gebang dan Mulyosari–Kalijudan, dan diterima. Menur–Gebang (7) lalu ditolak, sebab Menur dan Gebang sudah tersambung lewat Sukolilo dan Keputih. Tulang punggungnya tetap sebelas jalan dan berbobot $69 - 7 + 5 = 67$ ratus meter, 6,7 km. (b) Menurut Persamaan (12.2), invoice kini selesai pada $5 + 110 = 115$, persis bersamaan dengan fold, jadi pack selesai pada $\max(115,115) + 6 = 121$: waktu putarnya **tidak berubah**, tetapi kini ada **dua** lintasan kritis, yang lama dan intake → weigh → invoice → pack. Kelonggaran 106 menit itu terpakai habis; satu menit tambahan pada penerbitan faktur akan menunda setiap kantong.

12.3 Persoalan Terkerjakan Lanjutan I: Algoritma Greedy

12.3.1 Susunan persoalan-persoalan terkerjakan

Subbab 12.3 sampai 12.6 mengerjakan dua puluh tiga persoalan, berlabel A sampai W dan dikelompokkan menurut bab yang menjadi sumbernya. **Tabel 12.9** mendaftar semuanya, supaya pembaca yang sedang mengulang satu bagian yang lemah dapat langsung menuju ke sana.

Tabel 12.9 Kedua puluh tiga persoalan terkerjakan Subbab 12.3 sampai 12.6, subbab bab ini tempat masing-masing dikerjakan, dan alat dari Bab 9 sampai 11 yang menyelesaikannya.

Persoalan	Topik	Subbab	Alat utamanya, dan tempat ia diajarkan
A	Menukar uang: satu aturan rakus disanggah, satu lagi dibuktikan	12.3.2	kegagalan *greedy* (§9.3); argumen pertukaran (§9.4.7)
B	Pemilihan aktivitas, dijalankan dan dibuktikan	12.3.3	waktu selesai terawal; sifat pilihan rakus (§9.4)
C	Meminimumkan keterlambatan maksimum	12.3.4	pertukaran pembalikan bersebelahan
D	Kode Huffman, dibangun langkah demi langkah	12.3.5	algoritma Huffman; $B(T)$ dengan dua cara (§9.7)
E	Kedua *knapsack*	12.3.6	rakus pecahan lawan pemrograman dinamis 0-1 (§9.6)
F	Matroid yang bekerja: tugas berwaktu satuan dengan tenggat	12.3.7	algoritma rakus matroid (§9.8)
G	Pencarian melebar, ditelusuri	12.4.2	BFS, d dan π (§10.7)
H	Pencarian mendalam pada digraf, setiap busur digolongkan	12.4.3	cap waktu, golongan sisi, teorema kurung (§10.8)
I	Pencarian mendalam pada graf tak berarah	12.4.4	tabel waktu temu dan selesai; hanya sisi pohon dan mundur (§10.8.10)
J	Komponen terhubung	12.4.5	BFS berulang; peringkat sirkuit (§10.9.1)
K	Kebipartitan, dan sertifikatnya	12.4.6	pewarnaan dua warna; siklus ganjil (§10.9.3)
L	Pendeteksian siklus, tak berarah dan berarah	12.4.7	aturan induk; aturan kelabu (§10.9.2, §11.2)
M	Apakah ia DAG?	12.5.2	teorema DAG (§11.2.3)
N	Pengurutan topologis menurut waktu selesai	12.5.3	f menurun (§11.3.4)
O	Pengurutan topologis dengan algoritma Kahn	12.5.4	derajat masuk dan sebuah antrean (§11.3.9)
P	Mencacah urutan topologis	12.5.5	rekursi atas himpunan; lintasan Hamilton
Q	Lintasan kritis	12.5.6	pelonggaran dalam urutan topologis
R	Kruskal, dengan penelusuran himpunan saling lepas	12.6.2	`Find`, `Union`, peringkat (§11.10)
S	Prim, dengan larik kunci	12.6.3	key dan π (§11.9)
T	Prim, dengan tumpukan biner	12.6.4	`DecreaseKey` malas (§11.9.10)
U	Sifat potongan dan sifat siklus	12.6.5	argumen pertukaran pada pohon (§11.7)
V	Penyatuan menurut peringkat dan pemampatan lintasan	12.6.6	batas peringkat (§11.10.5–7)
W	Apakah pohon merentang minimumnya tunggal?	12.6.7	uji penghapusan; sifat siklus (§11.12)

Setiap persoalan berbentuk sama, supaya Anda selalu tahu sedang berada di mana:

- **Persoalan** — pernyataannya, dengan setiap bilangan yang Anda perlukan.
- **Yang diuji** — alatnya, dan di bagian mana buku ini alat itu diperkenalkan.
- **Penyelesaian** — langkah demi langkah, dengan setiap nilai antara dituliskan, dan jebakan yang menjerat orang disebut namanya di tempat ia muncul.
- **Jawaban** — hasilnya, dinyatakan dengan terang.

- **Program** — setiap kelompok persoalan ditutup dengan program C++17 lengkap yang menghitung setiap jawaban dalam kelompok itu, dikutip bersama keluaran yang benar-benar dicetaknya; persoalan A, F, I, dan T punya program sendiri.
- **Uji diri** — sebuah varian untuk Anda kerjakan sendiri, beserta jawabannya.

Subbab ini mengerjakan enam persoalan tentang algoritma *greedy*. Yang pertama memperlihatkan kerakusan gagal, empat berikutnya memperlihatkannya berhasil dan membuktikan mengapa, dan yang keenam memperlihatkan teori umumnya — matroid — yang menjelaskan sekeluarga keberhasilan sekaligus.

12.3.2 Persoalan Terkerjakan A — menukar uang: satu aturan rakus disanggah, satu lagi dibuktikan

Persoalan. Sebuah kios binatu memberikan uang kembalian dalam bentuk token bernilai 1, 7, dan 10. Aturan yang kentara, **Pseudokode 12.2**, berbunyi *selalu serahkan token terbesar yang masih muat*.

Pseudokode 12.2 Aturan rakus penukaran uang: berulang-ulang serahkan koin terbesar yang tidak melebihi sisa jumlah yang masih terutang.

```
GREEDY-CHANGE(coins, x)          // koin terurut sehingga c_1 > c_2 > ... > c_k = 1
1  count = 0
2  for i = 1 to k
3      while x >= c_i
4          x = x - c_i
5          count = count + 1
6  return count
```

(a) Jalankan aturan itu pada $x = 14$. (b) Temukan banyaknya token minimum yang sesungguhnya untuk $x = 14$. (c) Untuk jumlah $1 \le x \le 30$ yang mana aturan itu gagal, dan berapa yang terkecil? (d) Dengan memakai kedua bahan §9.5, jelaskan bahan mana yang hilang. (e) Buktikan bahwa aturan yang sama *memang* optimal untuk token {1,5,10}.

Yang diuji. Bahwa aturan rakus adalah sebuah *klaim* dan memerlukan bukti (§9.2.2); bahwa satu contoh sanggahan sudah membunuhnya (§9.3.3); dan bahwa argumen pertukaran dapat membuktikannya ketika ia benar (§9.4.7).

Penyelesaian.

(a) Jalannya aturan rakus pada 14. Token terbesar yang tidak melebihi 14 adalah 10: serahkan, sisa 4. 7 tidak muat ke dalam 4. Lalu 1, empat kali: sisa 3, 2, 1, 0. Totalnya: $10 + 1 + 1 + 1 + 1$, **lima token**.

(b) Minimum sesungguhnya untuk 14. Dua token 7 membentuk 14 dengan tepat. Satu token tidak dapat membentuk 14, sebab tak ada token bernilai 14. Jadi minimumnya **dua token**, dan kerakusan memakai dua setengah kali lipatnya.

(c) Setiap kegagalan sampai 30. Untuk mengetahui minimum sesungguhnya bagi setiap jumlah kita memerlukan lebih dari kecerdikan; kita memerlukan metode yang pasti benar. Misalkan $M(x)$ adalah token tersedikit yang membayar x dengan tepat. Token terakhir yang diserahkan adalah suatu $c \le x$, dan sisanya, $x - c$, harus dibayar secara optimal pula, jadi

$$M(0) = 0, \qquad M(x) = 1 + \min_{\substack{c \in \text{koin} \\ c \le x}} M(x - c) \quad (x \ge 1). \tag{12.5}$$

Persamaan (12.5) adalah rekurensi *optimal substructure* pemrograman dinamis, dan menghitungnya untuk $x = 1,2,\ldots,30$ secara berurutan berongkos $30 \times 3 = 90$ evaluasi. Mengerjakan beberapa yang pertama dengan tangan: $M(1) = 1$, $M(2) = 2$, ..., $M(6) = 6$ (hanya token 1 yang muat); $M(7) = 1 + \min(M(6), M(0)) = 1 + 0 = 1$; $M(10) = 1 + \min(M(9), M(3), M(0)) = 1$; $M(14) = 1 + \min(M(13), M(7), M(4)) = 1 + \min(4,1,4) = 2$. **Kode 12.3** mengerjakan sisanya dan mencetak jawaban kerakusan di samping optimumnya.

Kode 12.3 Penukaran uang secara rakus dibandingkan dengan optimum pemrograman dinamis, untuk setiap jumlah dari 1 sampai 30 dengan token {1,7,10}, disusul dua audit ringkasan. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o change change.cpp`.

```
// Persoalan Terkerjakan A: menukar uang dengan koin {1, 7, 10}.
// Untuk setiap jumlah dari 1 sampai 30 program ini membandingkan aturan greedy
// ("koin terbesar yang muat") dengan optimum sejati hasil pemrograman dinamis,
// mencetak kedua multihimpunan koinnya, dan melaporkan jumlah terkecil tempat greedy
kalah.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <climits>
using namespace std;

string showCoins(const vector<int>& c) {
    string s;
    for (size_t i = 0; i < c.size(); ++i) { if (i) s += "+"; s += to_string(c[i]); }
    return s;
}

vector<int> greedy(const vector<int>& coins, int x) {      // koin dalam urutan menaik
    vector<int> used;
    for (int k = (int)coins.size() - 1; k >= 0; --k)
        while (x >= coins[k]) { x -= coins[k]; used.push_back(coins[k]); }
    return used;
}

// best[x] = koin tersedikit untuk membayar x; last[x] = satu koin dari pembayaran optimal
void optimal(const vector<int>& coins, int maxX, vector<int>& best, vector<int>& last) {
    best.assign(maxX + 1, INT_MAX); last.assign(maxX + 1, 0);
    best[0] = 0;
    for (int x = 1; x <= maxX; ++x)
        for (int c : coins)
            if (c <= x && best[x - c] != INT_MAX && best[x - c] + 1 < best[x]) {
                best[x] = best[x - c] + 1; last[x] = c;
            }
}

vector<int> rebuild(const vector<int>& last, int x) {
    vector<int> used;
    while (x > 0) { used.push_back(last[x]); x -= last[x]; }
    return used;
}
```

Kode 12.3 (Lanjutan) *Penukaran uang secara rakus dibandingkan dengan optimum pemrograman dinamis, untuk setiap jumlah dari 1 sampai 30 dengan token \{1, 7, 10\}, disusul dua audit ringkasan. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o change change.cpp.*

```
void audit(const vector<int>& coins, int maxX, bool table) {
    vector<int> best, last;
    optimal(coins, maxX, best, last);
    cout << "coins {";
    for (size_t i = 0; i < coins.size(); ++i) cout << (i ? ", " : "") << coins[i];
    cout << "}, amounts 1.." << maxX << "\n";
    if (table) cout << "  x  greedy              #g  optimum             #o\n";
    int firstBad = 0, bad = 0;
    for (int x = 1; x <= maxX; ++x) {
        vector<int> g = greedy(coins, x), o = rebuild(last, x);
        bool lose = g.size() > o.size();
        if (lose) { ++bad; if (!firstBad) firstBad = x; }
        if (table)
            cout << setw(3) << x << "  " << left << setw(18) << showCoins(g) << right
                 << setw(4) << g.size() << "  " << left << setw(18) << showCoins(o)
                 << right << setw(4) << o.size() << (lose ? "   <- greed loses" : "") <<
"\n";
    }
    cout << "  amounts on which greed loses: " << bad << " of " << maxX;
    if (firstBad) cout << "; the smallest is " << firstBad << "\n";
    else cout << "; greed is optimal on every amount tested\n";
}

int main() {
    audit({1, 7, 10}, 30, true);
    cout << "\n";
    audit({1, 5, 8}, 100, false);                          // sistem untuk bagian uji diri
    audit({1, 5, 10, 25}, 100, false);                     // sebuah sistem kanonis
    return 0;
}
```

```
coins {1, 7, 10}, amounts 1..30
  x  greedy              #g  optimum            #o
  1  1                    1  1                    1
  2  1+1                  2  1+1                  2
  3  1+1+1                3  1+1+1                3
  4  1+1+1+1              4  1+1+1+1              4
  5  1+1+1+1+1            5  1+1+1+1+1            5
  6  1+1+1+1+1+1          6  1+1+1+1+1+1          6
  7  7                    1  7                    1
  8  7+1                  2  1+7                  2
  9  7+1+1                3  1+1+7                3
 10  10                   1  10                   1
 11  10+1                 2  1+10                 2
 12  10+1+1               3  1+1+10               3
 13  10+1+1+1             4  1+1+1+10             4
 14  10+1+1+1+1           5  7+7                  2   <- greed loses
 15  10+1+1+1+1+1         6  1+7+7                3   <- greed loses
 16  10+1+1+1+1+1+1       7  1+1+7+7              4   <- greed loses
 17  10+7                 2  7+10                 2
 18  10+7+1               3  1+7+10               3
 19  10+7+1+1             4  1+1+7+10             4
 20  10+10                2  10+10                2
 21  10+10+1              3  1+10+10              3
 22  10+10+1+1            4  1+1+10+10            4
 23  10+10+1+1+1          5  1+1+1+10+10          5
 24  10+10+1+1+1+1        6  7+7+10               3   <- greed loses
 25  10+10+1+1+1+1+1      7  1+7+7+10             4   <- greed loses
 26  10+10+1+1+1+1+1+1    8  1+1+7+7+10           5   <- greed loses
 27  10+10+7              3  7+10+10              3
 28  10+10+7+1            4  1+7+10+10            4
 29  10+10+7+1+1          5  1+1+7+10+10          5
 30  10+10+10             3  10+10+10             3
  amounts on which greed loses: 6 of 30; the smallest is 14

coins {1, 5, 8}, amounts 1..100
  amounts on which greed loses: 47 of 100; the smallest is 10
coins {1, 5, 10, 25}, amounts 1..100
  amounts on which greed loses: 0 of 100; greed is optimal on every amount tested
```

Kerakusan kalah pada persis enam jumlah sampai 30 — 14, 15, 16, 24, 25, dan 26 — dan yang terkecil adalah **14**. Polanya tampak di tabel: setiap kali jumlahnya 4, 5, atau 6 lebih dari kelipatan 10, dua token 7 mengalahkan satu token 10 ditambah setumpuk token 1.

(d) Bahan yang hilang. Optimal substructure hadir — Persamaan (12.5) *adalah* pernyataan *optimal substructure*, dan itulah sebabnya pemrograman dinamis memperoleh jawaban yang benar. Yang hilang adalah **sifat pilihan rakus**: *tidak* benar bahwa suatu pembayaran optimal atas 14 memuat token terbesar yang muat, sebab satu-satunya pembayaran optimal, $7 + 7$, sama sekali tidak memuat 10. Kerakusan terikat pada token 10 dan tak pernah dapat menariknya kembali.

(e) Bukti bahwa kerakusan optimal untuk $\{1,5,10\}$. Ambillah pembayaran optimal mana pun atas x yang memakai a token sepuluh, b token lima, dan e token satu, jadi $x = 10a + 5b + e$ dengan $a + b + e$ sekecil mungkin.

- $e \leq 4$: jika tidak, gantilah lima token satu dengan satu token lima, menghemat empat token — bertentangan dengan keoptimalan.
- $b \leq 1$: jika tidak, gantilah dua token lima dengan satu token sepuluh, menghemat satu token.
- Karena itu token lima dan token satu bersama-sama bernilai $5b + e \leq 5 + 4 = 9 < 10$.

Jadi $10a \leq x < 10a + 10$, yang memaksa $a = \lfloor x/10 \rfloor$ — persis banyaknya token sepuluh yang diambil kerakusan. Sisanya, $x - 10a < 10$, lalu dibayar dengan $b = \lfloor (x - 10a)/5 \rfloor$ token lima

(sebab $e \leq 4$ memaksanya) dan sisanya dengan token satu, yang lagi-lagi persis dikerjakan kerakusan. Setiap pembayaran optimal *adalah* pembayaran rakus. ■

Argumen yang sama patah untuk {1,7,10} pada butir kedua: dua token 7 bernilai 14, dan tak ada token tunggal bernilai 14, jadi tak ada pertukaran yang memendekkan pembayaran yang memuat dua token 7 — dan tepat di situlah contoh sanggahannya tinggal.

Seberapa jauh orang harus mencari sebelum menyatakan sebuah sistem koin aman bagi kerakusan? Tidak jauh. Kozen dan Zaks membuktikan bahwa jika kerakusan gagal sama sekali, ia sudah gagal pada suatu jumlah yang lebih kecil daripada jumlah kedua koin terbesar [5]; untuk {1,7,10} batas itu $10 + 7 = 17$, dan memang kegagalan pertamanya adalah 14. Pearson kemudian memberikan uji berwaktu polinomial yang memutuskan pertanyaan itu bagi sistem koin mana pun [6]. Sistem yang di atasnya kerakusan optimal bagi setiap jumlah disebut **kanonis**.

Jawaban. Kerakusan membayar 14 dengan lima token padahal dua mencukupi; ia gagal pada 14, 15, 16, 24, 25, 26 dalam rentang 1 sampai 30. Sifat pilihan rakuslah yang gagal. Untuk {1,5,10} sebuah argumen pertukaran membuktikan kerakusan optimal, dan audit atas sistem AS {1,5,10,25} tidak menemukan satu kegagalan pun sampai 100.

> **UJI DIRI.** Untuk token {1,5,8}, temukan jumlah terkecil tempat kerakusan gagal, dan berikan kedua pembayarannya.
>
> *Jawaban.* $x = 10$. Kerakusan membayar $8 + 1 + 1$ (tiga token); optimumnya $5 + 5$ (dua). Audit kedua **Kode 12.3** membenarkan 10 sebagai kegagalan terkecil, dan mendapati bahwa kerakusan kalah pada 47 dari jumlah-jumlah 1 sampai 100.

12.3.3 Persoalan Terkerjakan B — pemilihan aktivitas, dijalankan dan dibuktikan

Persoalan. Sepuluh pekerjaan berebut satu mesin pengering industri. Pekerjaan a_i menduduki pengering itu sepanjang selang setengah terbuka $[s_i, f_i)$:

Tabel 12.10 Kesepuluh pekerjaan pengering pada Persoalan Terkerjakan B, didaftar menurut urutan waktu selesai yang tidak menurun.

i	1	2	3	4	5	6	7	8	9	10
s_i	0	2	1	5	4	8	7	10	9	12
f_i	3	4	6	7	9	11	12	13	14	16

(a) Periksalah bahwa pekerjaan dalam **Tabel 12.10** didaftar menurut urutan waktu selesai yang tidak menurun, dan berikan durasi masing-masing. (b) Jalankan aturan rakus waktu-selesai-terawal dengan tangan, dengan memperlihatkan setiap perbandingan, dan nyatakan himpunan yang terpilih. (c) Jalankan aturan waktu-mulai-terawal dengan tangan dan nyatakan jawabannya. (d) Nyatakan dan buktikan sifat pilihan rakus untuk persoalan ini dengan argumen pertukaran.

Yang diuji. Apakah Anda dapat menjalankan aturan rakus secara mekanis *dan* membuktikannya benar (§9.4.7). Bagian (b) adalah hafalan; bagian (d) adalah satu-satunya tempat dalam persoalan ini yang menuntut bukti, dan bagian itulah yang terpenting.

Penyelesaian.

(a) Pemeriksaannya, dan durasinya. Waktu selesainya berbunyi 3,4,6,7,9,11,12,13,14,16. Masing-masing tegas lebih besar daripada yang sebelumnya, jadi daftarnya terurut — syarat yang lebih kuat daripada yang diperlukan algoritmanya, yang hanya menuntut tidak menurun. Pengurangan memberikan durasinya, yang dicatat **Tabel 12.11** bersama banyaknya pekerjaan lain yang

berbenturan dengan masing-masingnya (diperlukan ketika aturan benturan-tersedikit §9.4.3 dicoba di dalam program):

Tabel 12.11 Kesepuluh pekerjaan pengering beserta durasinya dan banyaknya pekerjaan lain yang berbenturan dengan masing-masingnya. Pekerjaan a_1, a_2, a_4, dan a_{10} seri dengan benturan tersedikit, yaitu dua.

i	1	2	3	4	5	6	7	8	9	10
$f_i - s_i$	3	2	5	2	5	3	5	3	5	4
benturan	2	2	4	2	4	4	4	4	4	2

(b) Waktu selesai terawal, dengan tangan. **Pseudokode 12.3** adalah aturan iteratif §9.4.11. Ia memelihara satu peubah, F, yaitu saat pengeringnya bebas berikutnya, yang bermula di 0, dan ia menyusuri daftarnya menurut urutan indeks — yang merupakan urutan waktu selesai, dan itulah sebabnya mengurutkan lebih dahulu itu penting — sambil mengambil sebuah pekerjaan kapan pun $s_i \geq F$.

Pseudokode 12.3 Aturan waktu-selesai-terawal untuk pemilihan aktivitas, sebagai satu sapuan atas pekerjaan yang sudah terurut menurut waktu selesai.

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)   // pekerjaan terurut sehingga f_1 <= f_2 <= ... <= f_n
A = {a_1}
F = f_1
for i = 2 to n
    if s_i >= F
        A = A + {a_i}
        F = f_i
return A
```

- $i = 1$: $s_1 = 0 \geq 0 = F$. **Ambil** a_1. $F \leftarrow f_1 = 3$. Terpilih: $\{a_1\}$.
- $i = 2$: $s_2 = 2 \geq 3$? Tidak. Lewati.
- $i = 3$: $s_3 = 1 \geq 3$? Tidak. Lewati.
- $i = 4$: $s_4 = 5 \geq 3$? Ya. **Ambil** a_4. $F \leftarrow 7$. Terpilih: $\{a_1, a_4\}$.
- $i = 5$: $s_5 = 4 \geq 7$? Tidak. Lewati.
- $i = 6$: $s_6 = 8 \geq 7$? Ya. **Ambil** a_6. $F \leftarrow 11$. Terpilih: $\{a_1, a_4, a_6\}$.
- $i = 7$: $s_7 = 7 \geq 11$? Tidak. $i = 8$: $10 \geq 11$? Tidak. $i = 9$: $9 \geq 11$? Tidak.
- $i = 10$: $s_{10} = 12 \geq 11$? Ya. **Ambil** a_{10}. $F \leftarrow 16$.

Jawabannya: $\{a_1, a_4, a_6, a_{10}\}$, berukuran **4**, menduduki pengeringnya sepanjang [0,3), [5,7), [8,11), dan [12,16). Sepuluh perbandingan, satu per pekerjaan — sapuannya $\Theta(n)$ begitu daftarnya terurut.

(c) Waktu mulai terawal, dengan tangan. Urutkan menurut waktu mulai: $a_1(0), a_3(1), a_2(2), a_5(4), a_4(5), a_7(7), a_6(8), a_9(9), a_8(10), a_{10}(12)$.

- Ambil $a_1 = [0,3)$. Buang semua yang tumpang tindih dengannya: $a_3 = [1,6)$ dan $a_2 = [2,4)$ keduanya begitu.
- Dari yang tersisa, yang paling awal mulai adalah $a_5 = [4,9)$. **Ambil ia.** Buang $a_4 = [5,7)$, $a_7 = [7,12)$, dan $a_6 = [8,11)$, yang semuanya tumpang tindih dengan [4,9).
- Dari yang tersisa, yang paling awal mulai adalah $a_9 = [9,14)$. **Ambil ia.** Buang $a_8 = [10,13)$ dan $a_{10} = [12,16)$.
- Tak ada yang tersisa.

Jawabannya: $\{a_1, a_5, a_9\}$, berukuran **3**. Aturannya **keliru**, dan contoh ini membuktikannya, sebab bagian (b) menghasilkan empat. Satu pekerjaan panjang a_5, yang terpilih karena kebetulan mulai lebih awal, memusnahkan tiga pilihan berikutnya.

(d) Sifat pilihan rakus, dibuktikan. Tuliskan $S_k = \{a_i \in S: s_i \geq f_k\}$ untuk himpunan pekerjaan yang masih dapat berjalan sesudah a_k selesai.

> **TEOREMA (pilihan rakus).** Misalkan S_k tidak kosong dan misalkan a_m adalah pekerjaan di S_k dengan waktu selesai paling awal. Maka a_m termasuk di dalam *suatu* himpunan pekerjaan saling sesuai berukuran maksimum yang diambil dari S_k.

Bukti. Misalkan A_k adalah upahimpunan saling sesuai berukuran maksimum mana pun dari S_k, dan misalkan a_j adalah pekerjaan di A_k yang paling awal selesai. Ada dua kasus.

Kasus 1: $a_j = a_m$. Maka $a_m \in A_k$ dan tak ada yang perlu dibuktikan.

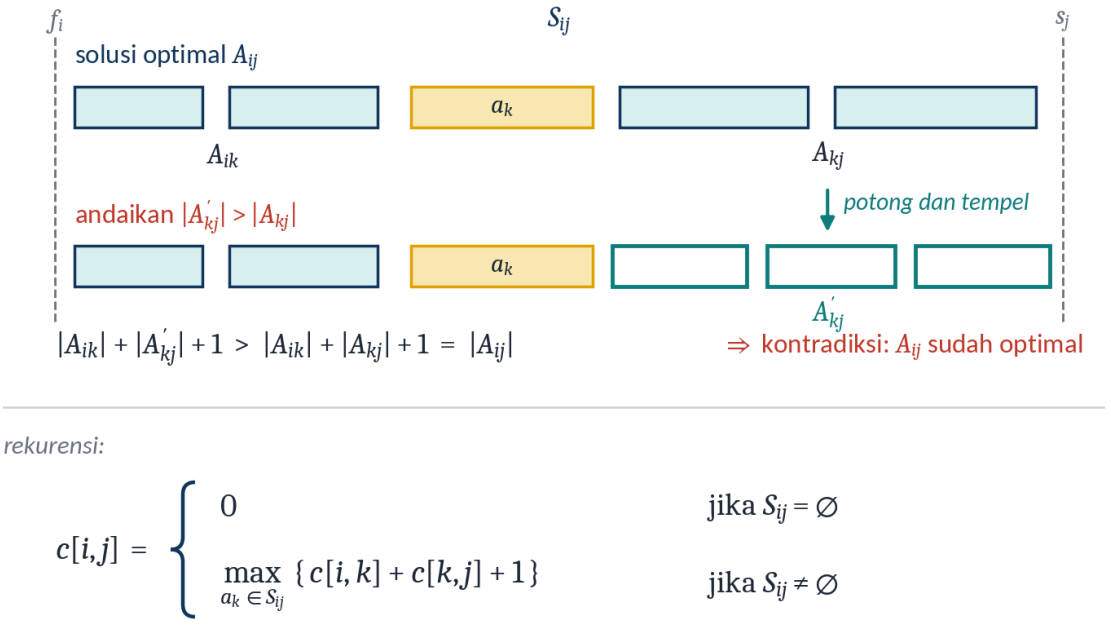
Kasus 2: $a_j \neq a_m$. Bentuklah

$$A'_k = (A_k \setminus \{a_j\}) \cup \{a_m\}. \tag{12.6}$$

Kita harus memeriksa bahwa himpunan A'_k pada Persamaan (12.6) masih saling sesuai. Pekerjaan-pekerjaan di $A_k \setminus \{a_j\}$ saling sesuai sebab A_k begitu. Setiap satunya mulai pada atau sesudah f_j, sebab a_j adalah yang paling awal selesai di A_k dan pekerjaannya saling lepas. Dan a_m dipilih sebagai yang paling awal selesai di seluruh S_k, jadi $f_m \leq f_j$. Digabungkan, setiap pekerjaan di $A_k \setminus \{a_j\}$ mulai pada atau sesudah $f_j \geq f_m$, jadi tak satu pun tumpang tindih dengan a_m. Karena itu A'_k saling sesuai.

Akhirnya $|A'_k| = |A_k|$, sebab kita membuang satu pekerjaan dan menambahkan satu. Jadi A'_k juga merupakan upahimpunan saling sesuai berukuran maksimum dari S_k, dan ia memuat a_m. ■

Gambar penyulihan ini adalah **Gambar 12.5**: seandainya ada himpunan yang tegas lebih baik bagi sub-persoalan yang tersisa, menyulihkannya kembali akan bertentangan dengan keoptimalan yang kita jadikan titik tolak.



Gambar 12.5 Teorema pemilihan aktivitas dirampungkan lewat kontradiksi, bersama rekurensi pemrograman dinamis untuk $c[i,j]$ yang dihasilkan separuh *optimal substructure* argumennya. Penyulihan yang digambar di atasnya — himpunan A'_{kj} yang lebih besar ditempelkan menggantikan A_{kj} — adalah kembaran potong-dan-tempel dari pertukaran pada Persamaan (12.6): di sana satu pekerjaan ditukar dengan yang rakus dan ukurannya tetap; di sini potongan yang lebih besar akan memperbesar keseluruhannya, dan keoptimalan melarang hal itu.

PERINGATAN: JEBAKAN DI BAGIAN (d). Teoremanya berkata *suatu* penyelesaian optimal memuat pilihan rakusnya, bukan *setiap* penyelesaian optimal. Pada contoh ini persis, $\{a_2, a_4, a_6, a_{10}\}$ juga optimal dan tidak memuat a_1. Bukti yang bermula "misalkan A adalah penyelesaian optimalnya" sudah tersesat, sebab ia mengandaikan optimumnya tunggal. Tulislah "misalkan A_k adalah *sebuah* upahimpunan saling sesuai berukuran maksimum".

Cara lain. Jawaban yang sama menyusul dari pemrograman dinamis. Definisikan $c[i,j]$ sebagai ukuran himpunan saling sesuai terbesar yang diambil dari S_{ij}, yaitu pekerjaan yang muat tegas di antara a_i dan a_j; maka $c[i,j] = \max_{a_k \in S_{ij}}\{c[i,k] + c[k,j] + 1\}$, yang berongkos $\Theta(n^3)$ untuk diisi. Teorema rakusnya berkata bahwa k yang memaksimumkan itu selalu yang paling awal selesai, jadi max-nya mengerut menjadi satu calon tunggal dan tabelnya mengerut menjadi satu sapuan. Mengetahui rekurensinya sekaligus mengetahui mengapa ia tak diperlukan bernilai lebih daripada salah satu saja.

Jawaban. Waktu selesai terawal memilih $\{a_1, a_4, a_6, a_{10}\}$, berukuran 4, yang optimal; waktu mulai terawal hanya memilih 3. Pertukaran Persamaan (12.6) membuktikan bahwa pekerjaan yang paling awal selesai selalu termasuk dalam suatu himpunan optimal.

Varian uji diri di bawah ini juga memakai kesebelas aktivitas §9.4.2, yang diperlihatkan lagi sebagai **Gambar 12.6**.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Gambar 12.6 Kesebelas aktivitas §9.4.2 beserta waktu mulai dan waktu selesainya, sudah terurut sehingga $f_1 \leq f_2 \leq \cdots \leq f_{11}$.

UJI DIRI. (a) Jalankan aturan *waktu mulai paling akhir dahulu* pada **Tabel 12.10**: berulang-ulang ambillah pekerjaan tersisa yang paling akhir mulai dan buanglah segala yang tumpang tindih dengannya. Apakah jawabannya optimal, dan mengapa? (b) Jalankan waktu selesai terawal pada kesebelas aktivitas **Gambar 12.6**.

Jawaban. (a) Menurut waktu mulai, menurun: $a_{10}(12)$, $a_8(10)$, $a_9(9)$, $a_6(8)$, $a_7(7)$, $a_4(5)$, $a_5(4)$, $a_2(2)$, $a_3(1)$, $a_1(0)$. Ambil $a_{10} = [12,16)$; a_8 dan a_9 tumpang tindih dengannya. Ambil $a_6 = [8,11)$; a_7 tumpang tindih dengannya. Ambil $a_4 = [5,7)$; a_5 tumpang tindih dengannya. Ambil $a_2 = [2,4)$; a_3 dan a_1 tumpang tindih dengannya. Hasilnya $\{a_2, a_4, a_6, a_{10}\}$, berukuran 4 — **optimal**, dan itulah optimum *yang lain* yang disebut dalam peringatan di atas. Ia pasti optimal: balikkan waktunya (petakan setiap t ke $-t$), maka waktu-mulai-paling-akhir-dahulu menjadi waktu-selesai-terawal-dahulu pada contoh cerminannya, yang dicakup teoremanya. (b) $\{a_1, a_4, a_8, a_{11}\}$, berukuran 4; $\{a_2, a_4, a_9, a_{11}\}$ sama optimalnya.

12.3.4 Persoalan Terkerjakan C — meminimumkan keterlambatan maksimum

Persoalan. Enam pekerjaan harus berjalan satu per satu pada satu mesin, semuanya tersedia pada saat 0. Pekerjaan J_i memerlukan p_i satuan pemrosesan dan bertenggat d_i:

Tabel 12.12 Keenam pekerjaan pada Persoalan Terkerjakan C, beserta waktu pemrosesan dan tenggatnya.

Pekerjaan	J_1	J_2	J_3	J_4	J_5	J_6
p_i	3	2	1	4	3	2
d_i	6	8	9	9	14	15

Jika sebuah pekerjaan selesai pada saat C_i, **keterlambatannya** adalah $L_i = C_i - d_i$ (negatif bila ia lebih awal), dan tujuannya adalah

$$L_{\max} = \max_{1 \le i \le n} (C_i - d_i) \;\rightarrow\; \min. \tag{12.7}$$

(a) Jadwalkan pekerjaan-pekerjaan **Tabel 12.12** menurut *tenggat terawal dahulu* dan hitunglah setiap C_i, setiap L_i, dan $L_{\max}$. (b) Kerjakan hal yang sama untuk *waktu pemrosesan terpendek dahulu*, lalu bandingkan. (c) Buktikan bahwa tenggat terawal dahulu meminimumkan Persamaan (12.7), dengan argumen pertukaran atas pembalikan bersebelahan.

Yang diuji. Apakah Anda dapat membangun sendiri sebuah argumen pertukaran pada persoalan yang belum pernah Anda lihat dibuktikan. Bukti pemilihan aktivitas §9.4.7 adalah teladannya; yang ini saudaranya, dan bagian (c) adalah jantungnya. Persoalan ini beserta pembuktiannya dibahas oleh Kleinberg dan Tardos [7].

Penyelesaian.

(a) Tenggat terawal dahulu. Urutkan menurut d_i: $J_1(6), J_2(8), J_3(9), J_4(9), J_5(14), J_6(15)$. Mesinnya tak pernah menganggur, jadi C_i adalah jumlah berjalan waktu pemrosesannya; **Tabel 12.13** mengerjakannya sampai tuntas.

Tabel 12.13 Tenggat terawal dahulu pada keenam pekerjaan. Hanya J_4 yang terlambat, sebesar satu satuan, jadi $L_{\max} = 1$.

Urutan	Pekerjaan	p_i	C_i	d_i	$L_i = C_i - d_i$
1	J_1	3	3	6	-3
2	J_2	2	5	8	-3
3	J_3	1	6	9	-3
4	J_4	4	10	9	$+1$
5	J_5	3	13	14	-1
6	J_6	2	15	15	0

$$L_{\max} = \max\{-3, -3, -3, +1, -1, 0\} = \mathbf{1}.$$

(b) Waktu pemrosesan terpendek dahulu. Urutkan menurut p_i: $J_3(1), J_2(2), J_6(2), J_1(3), J_5(3), J_4(4)$, lalu ulangi hitungannya seperti **Tabel 12.14**.

Tabel 12.14 Waktu pemrosesan terpendek dahulu pada keenam pekerjaan yang sama. $L_{\max} = 6$, enam kali lebih buruk daripada tenggat terawal dahulu, walaupun *total* waktu selesainya lebih kecil.

Urutan	Pekerjaan	p_i	C_i	d_i	L_i
1	J_3	1	1	9	-8
2	J_2	2	3	8	-5
3	J_6	2	5	15	-10
4	J_1	3	8	6	$+2$
5	J_5	3	11	14	-3
6	J_4	4	15	9	$+6$

$L_{\max} = 6$. Perbandingannya itulah intinya: mengurutkan menurut kunci yang salah bukanlah kesalahan kecil. Waktu-pemrosesan-terpendek-dahulu justru optimal bagi tujuan yang *berbeda* — meminimumkan rata-rata waktu selesai — dan kedua tujuan itu di sini berselisih sebesar faktor enam.

(c) Buktinya. Katakan sebuah jadwal punya sebuah **pembalikan** jika ada pekerjaan i yang dijadwalkan sebelum pekerjaan j padahal $d_i > d_j$.

Langkah 1: ada jadwal optimal tanpa waktu menganggur. Membuang celah menganggur memajukan setiap pekerjaan berikutnya, yang hanya dapat menurunkan waktu selesai dan karena itu menurunkan keterlambatan.

Langkah 2: menukar sepasang bersebelahan yang terbalik tidak menaikkan $L_{\max}$. Misalkan pekerjaan i berjalan persis sebelum pekerjaan j, dengan $d_i > d_j$, dan segala yang mendahului i selesai pada saat t. Sebelum penukaran,

$$C_i = t + p_i, \qquad C_j = t + p_i + p_j.$$

Sesudah penukaran,

$$C'_j = t + p_j, \qquad C'_i = t + p_i + p_j.$$

Tak ada pekerjaan lain yang bergerak, jadi hanya L_i dan L_j yang dapat berubah.

Untuk j: $L'_j = t + p_j - d_j \leq t + p_i + p_j - d_j = L_j$, sebab $p_i \geq 0$. Jadi j tidak lebih terlambat daripada sebelumnya.

Untuk i: $L'_i = t + p_i + p_j - d_i$. Bandingkan ia dengan keterlambatan *lama* milik j, $L_j = t + p_i + p_j - d_j$. Kedua ungkapan itu hanya berbeda pada tenggat yang dikurangkan, dan $d_i > d_j$, jadi

$$L'_i = t + p_i + p_j - d_i \ < \ t + p_i + p_j - d_j = L_j. \tag{12.8}$$

Menurut Persamaan (12.8) dan baris sebelumnya, $\max(L'_i, L'_j) \leq \max(L_i, L_j)$, dan karena tak ada lagi yang berubah, $L_{\max}$ tidak naik.

Langkah 3: jadwal apa pun dapat diubah menjadi jadwal tenggat terawal lewat penukaran bersebelahan atas pasangan yang terbalik. Ini persis *bubble sort*: jadwal tanpa pembalikan terurut menurut tenggat, dan selama ada pembalikan pasti ada pasangan terbalik yang *bersebelahan* (jika i mendahului j dengan $d_i > d_j$ tetapi setiap pasangan bersebelahan di antaranya sudah berurutan, tenggatnya akan tidak menurun dari i ke j, bertentangan dengan $d_i > d_j$). Paling banyak $\binom{6}{2} = 15$ penukaran mencukupi di sini, dan $\binom{n}{2}$ pada umumnya.

Menyatukan ketiga langkahnya: bertolak dari jadwal optimal mana pun, gelembungkan ia menjadi urutan tenggat; menurut Langkah 2 tak ada penukaran yang menaikkan $L_{\max}$, jadi jadwal tenggat terawal sekurang-kurangnya sama baiknya. Karena itu ia optimal. ■

Jebakannya. Jadwal yang hanya berbeda pada urutan pekerjaan bertenggat *sama* punya $L_{\max}$ yang sama, jadi optimum "yang itu" tidaklah tunggal. Di sini J_3 dan J_4 sama-sama bertenggat $d = 9$, dan menukar keduanya memberikan $L_{\max} = 1$ lagi, oleh pekerjaan yang berbeda. Gaya kasar atas seluruh $6! = 720$ urutan, di dalam program §12.3.8, membenarkan baik nilainya maupun kegandaannya: optimumnya 1 dan **20** dari 720 urutan mencapainya.

Jawaban. Tenggat terawal dahulu memberikan $L_{\max} = 1$, yang optimal; waktu pemrosesan terpendek dahulu memberikan 6.

> **UJI DIRI.** Ubahlah d_4 dari 9 menjadi 7 lalu ulangi bagian (a). Pekerjaan mana yang kini terlambat, dan apakah $L_{\max}$ masih optimal?
>
> *Jawaban.* Urutannya menjadi $J_1(6), J_4(7), J_2(8), J_3(9), J_5(14), J_6(15)$ dengan $C = 3,7,9,10,13,15$ dan $L = -3,0,+1,+1,-1,0$. Kini J_2 dan J_3 masing-masing terlambat satu, dan J_4 tepat waktu. $L_{\max} = 1$, dan menurut bagian (c) ia optimal; gaya kasar atas ke-720 urutan mendapati bahwa 4 di antaranya mencapainya.

12.3.5 Persoalan Terkerjakan D — kode Huffman, dibangun langkah demi langkah

Persoalan. Sebuah log mesin berisi 100.000 kejadian memakai tujuh simbol, dengan frekuensi dalam ribuan berikut: R 38, W 22, D 15, F 12, P 8, X 4, Q 1. (a) Berapa bit yang diperlukan kode panjang-tetap, dan berapa ongkos seluruh lognya? (b) Jalankan algoritma Huffman, dengan memperlihatkan antreannya sebelum setiap penggabungan dan label setiap simpul yang tercipta, lalu gambarlah pohonnya. (c) Berikan kata kode beserta panjangnya untuk tiap simbol, dan hitunglah ongkos $B(T)$ dua kali — sekali dengan menjumlahkan atas daun-daunnya dan sekali dengan menjumlahkan label simpul-simpul dalamnya. (d) Nyatakan penghematannya dalam persen, dan jelaskan mengapa tak ada kode bebas awalan yang dapat lebih baik pada data ini.

Yang diuji. Ketelitian mekanis, dan kedua cara yang saling bebas untuk menghitung $B(T)$ (§9.7.5). Bagian (c) meminta keduanya dengan sengaja: jika keduanya tidak sepakat, ada kekeliruan hitung, dan Anda dapat menemukannya sendiri.

Penyelesaian.

(a) Panjang tetap. Tujuh simbol memerlukan $\lceil \lg 7 \rceil$ bit. Karena $\lg 4 = 2$ dan $\lg 8 = 3$, kita punya $2 < \lg 7 < 3$, jadi pembulatan ke atasnya 3. Periksalah lebih dahulu bahwa frekuensinya berjumlah benar: $38 + 22 + 15 + 12 + 8 + 4 + 1 = 100$ ribu, jadi lognya memuat 100.000 kejadian, sesuai yang dinyatakan. Pada tiga bit masing-masing,

$$3 \times 100\,000 = 300\,000 \text{ bit} = 300 \text{ kbit}.$$

(b) Huffman, penggabungan demi penggabungan. **Pseudokode 12.4** adalah prosedur §9.7.6. Antrean Q adalah antrean prioritas-min atas frekuensi; tiap penggabungan membuang kedua pohon terkecil dan menyisipkan sebuah simpul dalam baru yang labelnya jumlah keduanya.

Pseudokode 12.4 Algoritma Huffman: $n - 1$ penggabungan atas kedua pohon terjarang, tiap penggabungan menciptakan satu simpul dalam.

```
HUFFMAN(C)                  // C: simbol-simbolnya, masing-masing dengan frekuensi c.freq
1  n = |C|
2  Q = C                    // antrean prioritas-min berkunci freq
3  for i = 1 to n - 1
4      allocate a new node z
5      x = EXTRACT-MIN(Q)
6      y = EXTRACT-MIN(Q)
7      z.left = x;  z.right = y
8      z.freq = x.freq + y.freq
9      INSERT(Q, z)
10 return EXTRACT-MIN(Q)    // akar pohonnya
```

Awalnya antrean memuat tujuh daun. **Tabel 12.15** mencatat setiap penggabungan.

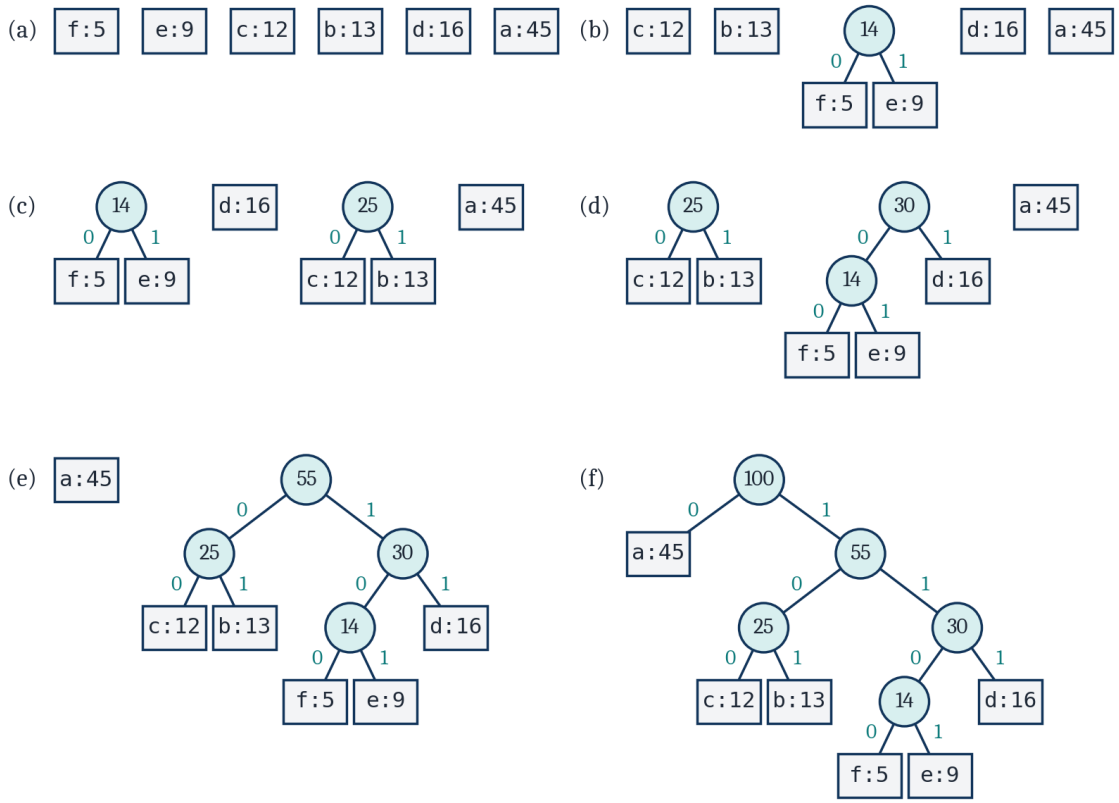
Tabel 12.15 Keenam penggabungan algoritma Huffman pada frekuensi lognya. Enam penggabungan untuk tujuh simbol, sebagaimana dituntut $|C| - 1$. Label simpul terakhirnya, 100, adalah frekuensi totalnya — sebuah pemeriksaan yang berguna.

Gabung	Antrean sebelumnya (terurut)	Dibuang	Simpul baru
1	1, 4, 8, 12, 15, 22, 38	Q 1, X 4	5
2	5, 8, 12, 15, 22, 38	5, P 8	13
3	12, 13, 15, 22, 38	F 12, 13	25
4	15, 22, 25, 38	D 15, W 22	37
5	25, 37, 38	25, 37	62
6	38, 62	R 38, 62	100

Ikutilah penggabungannya dan pohonnya menyusun dirinya sendiri. Penggabungan 1 menjadikan Q dan X bersaudara di bawah simpul berlabel 5. Penggabungan 2 menggantungkan simpul itu di samping P di bawah 13. Penggabungan 3 menggantungkan 13 di samping F di bawah 25. Penggabungan 4 menjadikan D dan W bersaudara di bawah 37 — sebuah pohon kecil tersendiri, sebab pada saat itu 15 dan 22 adalah dua butir terkecil di dalam antrean. Penggabungan 5 menyatukan kedua subpohon 25 dan 37 di bawah 62, dan penggabungan 6 meletakkan R di samping 62 di bawah akarnya, 100. Digambar dengan anak yang diambil pertama di kiri (label sisi 0) dan yang kedua di kanan (label sisi 1), pohonnya adalah:

```
        (100)
    0/      \1
  R:38      (62)
          0/      \1
        (25)       (37)
    0/    \1     0/    \1
  F:12    (13)  D:15   W:22
          0/    \1
        (5)    P:8
    0/    \1
  Q:1     X:4
```

Gambar 12.7 memperlihatkan proses yang sama, panel demi panel, pada data bersimbol enam §9.7; bentuk antreannya di tiap langkah sejenis persis dengan **Tabel 12.15**.



Gambar 12.7 Penelusuran enam panel atas algoritma Huffman pada data bersimbol enam §9.7. Tiap panel memperlihatkan antreannya terurut menaik menurut frekuensi; pada tiap langkah kedua pohon berfrekuensi terendah digabungkan, daunnya digambar sebagai persegi panjang dan simpul dalamnya sebagai lingkaran.

(c) Buku kodenya, dan $B(T)$ dihitung dua kali. Kata kode sebuah simbol adalah barisan label sisi pada lintasan dari akar turun ke daunnya, jadi panjangnya adalah kedalaman daun itu, $d_T(c)$. Membaca pohonnya memberikan **Tabel 12.16**.

Tabel 12.16 Buku kode Huffman untuk lognya, beserta ongkos tiap simbol. Kedua simbol terjarang, Q dan X, memperoleh kedua kata kode terpanjang dan hanya berbeda pada bit terakhirnya — keduanya bersaudara, dan itulah yang dilakukan penggabungan pertama terhadap keduanya.

Simbol	Frek (k)	Kata kode	Panjang	Frek × panjang
R	38	`0`	1	38
W	22	`111`	3	66
D	15	`110`	3	45
F	12	`100`	3	36
P	8	`1011`	4	32
X	4	`10101`	5	20
Q	1	`10100`	5	5
total	**100**			**242**

Cara 1, atas daunnya. Ongkos sebuah pohon adalah banyaknya bit yang dibelanjakannya untuk seluruh berkas,

$$B(T) = \sum_{c \in C} c\,.\,freq \cdot d_T(c), \tag{12.9}$$

dan menurut Persamaan (12.9), $B(T) = 38 + 66 + 45 + 36 + 32 + 20 + 5 = 242$ kbit.

Cara 2, atas simpul dalamnya. Label tiap simpul dalam adalah frekuensi total di bawahnya, dan tiap simbol melewati persis satu simpul dalam per bit kata kodenya, jadi jumlah label dalamnya mencacah setiap bit persis sekali:

$$B(T) = \sum_{z \text{ dalam}} z\,.\,freq = 5 + 13 + 25 + 37 + 62 + 100 = 242 \text{ kbit.} \tag{12.10}$$

Kedua cara itu, Persamaan (12.9) dan Persamaan (12.10), sepakat, jadi hitungannya benar.

(d) Penghematannya, dan keoptimalannya. $300 - 242 = 58$ kbit dihemat, yaitu

$$\frac{58}{300} = 0.1933\ldots = \mathbf{19.33\%}.$$

Tak ada kode bebas awalan yang dapat lebih baik, menurut teorema §9.7.10 dan argumen asli Huffman [8]: pilihan rakusnya aman (ada pohon optimal yang di dalamnya kedua simbol terjarang bersaudara pada kedalaman maksimum), dan persoalannya punya *optimal substructure* (mengganti kedua simbol itu dengan satu simbol berfrekuensi gabungannya menghasilkan contoh yang lebih kecil yang optimumnya memperpanjang diri menjadi optimum contoh aslinya). Induksi atas $|C|$ lalu memberikan keoptimalan atas *seluruh* kode bebas awalan.

> **PERINGATAN: JEBAKANNYA.** "Optimal" di sini berarti optimal *di antara kode bebas awalan yang memberikan bilangan bulat bit kepada tiap simbol, secara bebas dari konteksnya*. Pengodean aritmetik mengalahkan Huffman pada data ini dengan memakai bit pecahan, dan model yang meramalkan simbol berikutnya dari simbol sebelumnya mengalahkan keduanya. "Huffman adalah kompresi terbaik yang mungkin" itu keliru; "Huffman optimal di antara kode bebas awalan per simbol" itu benar.

Gambar 12.8 membandingkan kode panjang-tetap dengan kode panjang-berubah pada berkas bersimbol enam §9.7 — perhitungan yang sama persis dengan bagian (a) dan (c), pada enam simbol alih-alih tujuh.

	a	b	c	d	e	f
Frekuensi (dalam ribuan)	45	13	12	16	9	5
Kata kode panjang tetap	000	001	010	011	100	101
Kata kode panjang berubah	0	101	100	111	1101	1100

Gambar 12.8 Persoalan pengodean karakter §9.7: berkas berisi 100.000 karakter atas abjad a–f, dengan kode panjang-tetap berongkos 300.000 bit dan kode panjang-berubah berongkos 224.000. Bangun argumennya persis seperti Persoalan Terkerjakan D; hanya abjadnya yang berganti.

Jawaban. Kata kodenya R `0`, W `111`, D `110`, F `100`, P `1011`, X `10101`, Q `10100`; $B(T) = 242$ kbit terhadap 300 kbit panjang-tetap, penghematan 19,33%, yang optimal di antara kode bebas awalan. Program §12.3.8 mencetak pohon yang sama dan menyandikan lalu mengurai sebuah pesan 13 simbol pulang-pergi.

> **UJI DIRI.** (a) Uraikan untai bit `10110100111110` dengan buku kode **Tabel 12.16**. (b) Misalkan frekuensi Q adalah 6 ribu alih-alih 1. Bangunlah ulang kodenya: berapa panjang kata kode yang baru, dan berapa $B(T)$?
>
> *Jawaban.* (a) Bacalah dari kiri ke kanan, memotong setiap kali sebuah kata kode lengkap: `1011` P, `0` R, `100` F, `111` W, `110` D — pesannya **PRFWD**. Karena kodenya bebas awalan, tak pernah ada pilihan tentang di mana harus memotong. (b) Totalnya kini 105. Penggabungannya adalah $4 + 6 = 10$, $8 + 10 = 18$, $12 + 15 = 27$, $18 + 22 = 40$, $27 + 38 = 65$, $40 + 65 = 105$. Panjangnya menjadi R 2, W 2, D 3, F 3, P 3, X 4, Q 4, dan $B(T) = 38 \cdot 2 + 22 \cdot 2 + 15 \cdot 3 + 12 \cdot 3 + 8 \cdot 3 + 4 \cdot 4 + 6 \cdot 4 = 265$ kbit, yang dibenarkan label dalamnya: $10 + 18 + 27 + 40 + 65 + 105 = 265$. Perhatikan bahwa R kehilangan kata kode satu bitnya: ia tidak lagi *cukup* sering dibandingkan dengan segala yang lain.

12.3.6 Persoalan Terkerjakan E — kedua *knapsack*

Persoalan. Empat barang ditawarkan; ranselnya memuat 10 kg.

Tabel 12.17 Keempat barang pada Persoalan Terkerjakan E.

Barang	1	2	3	4
nilai v_i	7	6	9	9
berat w_i	3	3	5	5

(a) Hitunglah kerapatan nilai tiap barang dalam **Tabel 12.17** lalu urutkan. (b) Selesaikan persoalan *pecahan* secara rakus, dengan memperlihatkan pecahan yang diambil dari tiap barang. (c) Jalankan aturan rakus yang sama pada persoalan *0-1*, dan nyatakan nilai yang diperoleh. (d) Isilah tabel pemrograman dinamis $c[i, w]$ untuk $0 \le i \le 4$, $0 \le w \le 10$, dan bacalah penyelesaian 0-1 yang optimal. (e) Seberapa jauh, dalam persen, kerakusan meleset pada persoalan 0-1, dan bahan §9.5 mana yang tidak dimiliki persoalan 0-1?

Yang diuji. Pembedaan paling penting di seluruh Bab 9 (§9.6): bahwa sebuah aturan rakus dapat terbukti optimal pada satu persoalan dan buruk pada kembarannya yang nyaris serupa.

Penyelesaian.

(a) Kerapatannya. $v_1/w_1 = 7/3 = 2.333$, $v_2/w_2 = 6/3 = 2.000$, $v_3/w_3 = 9/5 = 1.800$, $v_4/w_4 = 9/5 = 1.800$. Terurut: barang 1, barang 2, lalu barang 3 dan 4 seri.

(b) Persoalan pecahannya. Ambillah barang utuh menurut urutan kerapatan selama muat, lalu isi ruang sisanya dengan pecahan barang berikutnya.

- Barang 1: 3 kg muat di dalam 10. Ambil utuh. Ruang sisa $10 - 3 = 7$. Nilai 7.
- Barang 2: 3 kg muat di dalam 7. Ambil utuh. Ruang sisa $7 - 3 = 4$. Nilai $7 + 6 = 13$.
- Barang 3: 5 kg **tidak** muat di dalam 4. Ambil pecahan $4/5 = 0.8$ darinya, bernilai $0.8 \times 9 = 7.2$. Ruang sisa 0. Nilai $13 + 7.2 = \mathbf{20.2}$.

Itu optimal bagi persoalan pecahannya: ranselnya terisi persis penuh, dan tiap kilogram di dalamnya sekurang-kurangnya serapat tiap kilogram yang ditinggalkan, jadi tak ada penukaran yang dapat memperbaikinya.

(c) Aturan yang sama pada persoalan 0-1. Pecahan dilarang, jadi barang 3 begitu saja dilewati.

- Barang 1: muat. Ambil. Berat 3, nilai 7.
- Barang 2: muat. Ambil. Berat 6, nilai 13.
- Barang 3: 5 kg ke dalam ruang 4 kg — tidak. Lewati.
- Barang 4: 5 kg ke dalam ruang 4 kg — tidak. Lewati.

Nilai rakus 0-1: **13**, memakai hanya 6 dari 10 kg. Empat kilogram kapasitas terbuang, dan pemborosan itulah seluruh kegagalannya.

(d) Tabel pemrograman dinamisnya. Misalkan $c[i, w]$ adalah nilai terbaik yang dapat diperoleh dari i barang pertama dengan kapasitas w. Barang i entah ditinggalkan, entah diambil lalu sisa kapasitasnya diisi secara optimal dari $i - 1$ barang pertama:

$$c[i, w] = \begin{cases} 0 & i = 0 \text{ atau } w = 0, \\ c[i-1, w] & w_i > w, \\ \max(c[i-1, w],\ v_i + c[i-1,\ w - w_i]) & \text{selainnya.} \end{cases} \tag{12.11}$$

Isilah Persamaan (12.11) baris demi baris ke dalam **Tabel 12.18**. Baris 1 (barang $v = 7, w = 3$): nol sampai $w = 3$, lalu 7 di mana-mana. Baris 2 (barang $v = 6, w = 3$): tak berubah di bawah $w = 6$; pada $w = 6$, $\max(7,\ 6 + c[1,3]) = \max(7,13) = 13$. Baris 3 (barang $v = 9, w = 5$): pada $w = 5$, $\max(7,\ 9 + c[2,0]) = 9$; pada $w = 8$, $\max(13,\ 9 + c[2,3]) = \max(13,16) = 16$. Baris 4 (barang $v = 9, w = 5$): pada $w = 10$, $\max(16,\ 9 + c[3,5]) = \max(16,18) = 18$.

Tabel 12.18 Tabel *knapsack* 0-1 $c[i, w]$ untuk Persoalan Terkerjakan E. Lima puluh lima sel, masing-masing satu perbandingan tunggal, jadi seluruh tabelnya berongkos $\Theta(nW) = \Theta(4 \times 10)$. Jawabannya adalah pojok kanan bawahnya, 18.

$w=$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
$i = 1$	0	0	0	7	7	7	7	7	7	7	7
$i = 2$	0	0	0	7	7	7	13	13	13	13	13
$i = 3$	0	0	0	7	7	9	13	13	16	16	16
$i = 4$	0	0	0	7	7	9	13	13	16	16	**18**

Bacalah penyelesaiannya kembali dengan menyusuri kolom terakhirnya ke atas. $c[4,10] = 18 \neq 16 = c[3,10]$, jadi **barang 4 masuk**; turunlah ke $c[3,10 - 5] = c[3,5] = 9$. Sekarang $c[3,5] = 9 \neq$

$7 = c[2,5]$, jadi **barang 3 masuk**; turunlah ke $c[2,0] = 0$ lalu berhenti. Himpunan optimalnya $\{3,4\}$, berat $5 + 5 = 10$, nilai $9 + 9 = \mathbf{18}$. Gaya kasar atas seluruh $2^4 = 16$ upahimpunan membenarkannya.

(e) Kelesetannya, dan bahan yang hilang. Kerakusan memperoleh 13 padahal 18 tersedia:

$$\frac{18 - 13}{18} = \frac{5}{18} = 0.2777 \ldots = \mathbf{27.8\%} \text{ dari optimumnya hilang.}$$

Bahan yang tidak dimiliki persoalan 0-1 adalah **sifat pilihan rakus** (§9.5.2): *tidak* benar bahwa suatu penyelesaian optimal memuat barang terapat. Di sini optimumnya yang tunggal $\{3,4\}$ sama sekali tidak memuat barang 1. *Optimal substructure* memang ia miliki — justru itulah yang dimanfaatkan Persamaan (12.11) — dan itulah sebabnya pemrograman dinamis berhasil sedangkan kerakusan tidak.

> **WAWASAN KUNCI: MENGAPA PERSOALAN PECAHANNYA BERBEDA, DALAM SATU KALIMAT.** Pada persoalan pecahan Anda selalu dapat mengisi ranselnya persis penuh, jadi "ambil yang terapat lebih dahulu" tak pernah memboroskan kapasitas; pada persoalan 0-1 sebuah barang entah muat entah tidak, jadi mengambil barang yang rapat dapat mengandaskan kapasitas yang hanya dapat dipakai barang yang kurang rapat. Barang 1 di sini rapat, kecil, dan mematikan: ia meninggalkan 4 kg yang tak dapat dimuati apa pun.

Jawaban. Optimum pecahan 20,2 lewat kerakusan; rakus 0-1 memperoleh 13 terhadap optimum 0-1 yang sesungguhnya, 18 (barang 3 dan 4), kelesetan 27,8%.

> **UJI DIRI.** Susutkan ranselnya menjadi 8 kg. Berikan optimum pecahannya, nilai rakus 0-1-nya, dan optimum 0-1-nya.
>
> *Jawaban.* Pecahan: barang 1 dan 2 utuh (6 kg, nilai 13), lalu 2/5 barang 3, bernilai 3.6: **16,6**. Rakus 0-1: barang 1 dan 2, nilai **13**, 2 kg terbuang. Optimum 0-1: **16**, lewat $\{1,3\}$ atau sama baiknya $\{1,4\}$ (masing-masing berat 8); bacalah dari baris 4, kolom 8 **Tabel 12.18**. Kerakusan kini kehilangan $3/16 = 18.75\%$.

12.3.7 Persoalan Terkerjakan F — matroid yang bekerja: tugas berwaktu satuan dengan tenggat

Persoalan. Pada suatu pagi yang sibuk, stasiun setrika binatu menerima tujuh pesanan satu jam. Pesanan a_i harus siap paling lambat pada akhir jam ke-d_i; jika terlambat, binatu mengembalikan uang sebesar w_i ribu rupiah. Hanya satu pesanan yang dapat disetrika pada satu waktu.

Tabel 12.19 Ketujuh pesanan setrika pada Persoalan Terkerjakan F: masing-masing memakan satu jam, dan pesanan yang terlambat berongkos sebesar pengembalian uangnya.

Pesanan	a_1	a_2	a_3	a_4	a_5	a_6	a_7
tenggat d_i	2	4	1	3	2	4	6
pengembalian w_i	55	45	40	35	30	20	10

(a) Jelaskan mengapa sekumpulan pesanan dapat semuanya diselesaikan tepat waktu bila dan hanya bila, untuk setiap t, paling banyak t di antaranya bertenggat t atau lebih awal. (b) Periksalah kedua aksioma matroid pada struktur ini, dan ilustrasikan sifat pertukarannya pada sepasang himpunan yang konkret. (c) Jalankan algoritma rakus matroid pada **Tabel 12.19** dan berikan jadwal serta total pengembalian uangnya. (d) Benarkan jawabannya dengan gaya kasar.

Yang diuji. Matroid §9.8: definisinya, uji kebebasan untuk penjadwalan waktu satuan (§9.8.6), dan teorema bahwa algoritma rakus optimal pada matroid berbobot mana pun (§9.8.5), yang dalam bentuk ini berasal dari Edmonds [9].

Penyelesaian.

(a) Uji kebebasannya. Sebutlah sebuah himpunan pesanan A **bebas** jika semuanya dapat diselesaikan tepat waktu. Untuk himpunan A dan jam t, misalkan $N_t(A)$ adalah banyaknya pesanan di A yang tenggatnya t atau lebih awal. Maka

$$A \text{ bebas} \Leftrightarrow N_t(A) \leq t \quad \text{untuk } t = 0,1,\dots,n. \tag{12.12}$$

Mengapa Persamaan (12.12) *perlu:* jika lebih dari t pesanan jatuh tempo paling lambat jam ke-t, maka dalam t jam pertama paling banyak t di antaranya dapat disetrika, jadi salah satunya terlambat. *Mengapa ia cukup:* setrikalah pesanan-pesanan A menurut urutan tenggatnya. Pesanan pada posisi ke-k dalam daftar itu punya sekurang-kurangnya k pesanan (dirinya sendiri dan yang mendahuluinya) yang jatuh tempo paling lambat pada tenggatnya sendiri d, jadi $k \leq N_d(A) \leq d$: ia selesai pada jam ke-$k \leq d$, tepat waktu.

(b) Kedua aksioma matroid. Himpunan dasarnya adalah ketujuh pesanan, dan himpunan-himpunan bebasnya adalah yang lolos Persamaan (12.12).

- **Turun-temurun.** Jika A bebas dan $B \subseteq A$, maka setiap cacah $N_t(B) \leq N_t(A) \leq t$, jadi B bebas. Membuang pesanan hanya dapat membuat jadwal lebih mudah.
- **Pertukaran.** Jika A dan B bebas dan $|A| < |B|$, ada suatu $x \in B \setminus A$ yang menjaga $A \cup \{x\}$ tetap bebas. Bukti umumnya ada di §9.8.6; inilah sifat itu bekerja pada sepasang himpunan. Ambillah $A = \{a_1, a_3\}$ (tenggat 2 dan 1: $N_1 = 1$, $N_2 = 2$, bebas) dan $B = \{a_3, a_4, a_5\}$ (tenggat 1, 3, 2: $N_1 = 1$, $N_2 = 2$, $N_3 = 3$, bebas). Maka $B \setminus A = \{a_4, a_5\}$. Menambahkan a_5 memberikan tenggat 2, 1, 2 dan $N_2 = 3 > 2$: **tidak** bebas. Menambahkan a_4 memberikan tenggat 2, 1, 3 dan $N_1 = 1$, $N_2 = 2$, $N_3 = 3$: bebas. Jadi pertukarannya berhasil dengan $x = a_4$ — sifat itu menjanjikan bahwa *suatu* unsur $B \setminus A$ berhasil, bukan bahwa setiap unsurnya berhasil.

(c) Algoritma rakusnya. **Pseudokode 12.5** adalah algoritma rakus §9.8.5, dikhususkan untuk matroid ini: timbanglah pesanan-pesanannya menurut urutan pengembalian uang yang menurun, dan pertahankan masing-masing yang penambahannya membiarkan himpunannya tetap bebas. Menjaga pesanan dengan pengembalian terbesar tetap tepat waktu sama dengan membuat total pengembalian pesanan yang terlambat sekecil mungkin.

Pseudokode 12.5 Algoritma rakus matroid untuk tugas berwaktu satuan dengan tenggat dan penalti: pertahankan pesanan termahal tetap tepat waktu kapan pun kebebasannya mengizinkan.

```
GREEDY-SCHEDULE(a, d, w, n)
sort the orders so that w_1 >= w_2 >= ... >= w_n
A = empty set
for i = 1 to n
    if A + {a_i} satisfies N_t <= t for every t
        A = A + {a_i}
schedule A in order of deadline, then the other orders in any order
return the schedule, and the refund sum of the orders not in A
```

Pesanan-pesanannya sudah dalam urutan pengembalian menurun di **Tabel 12.19**. **Tabel 12.20** menelusuri gelungnya; setiap baris memperlihatkan cacah $N_1, \dots, N_7$ himpunan *percobaan* $A \cup \{a_i\}$, dengan cacah yang melanggar Persamaan (12.12) ditandai.

Tabel 12.20 Algoritma rakus matroid pada ketujuh pesanan. Cacah yang tercetak tebal melanggar aturan $N_t \leq t$. Dua pesanan ditolak, dan pengembalian uang yang dibayar adalah jumlah keduanya, $30 + 20 = 50$.

Timbang	N_1	N_2	N_3	N_4	N_5	N_6	N_7	Vonis	A Sesudahnya
a_1 (55)	0	1	1	1	1	1	1	pertahankan	$\{a_1\}$
a_2 (45)	0	1	1	2	2	2	2	pertahankan	$\{a_1, a_2\}$
a_3 (40)	1	2	2	3	3	3	3	pertahankan	$\{a_1, a_2, a_3\}$
a_4 (35)	1	2	3	4	4	4	4	pertahankan	$\{a_1, a_2, a_3, a_4\}$
a_5 (30)	1	**3**	**4**	**5**	5	5	5	**tolak**	tak berubah
a_6 (20)	1	2	3	**5**	5	5	5	**tolak**	tak berubah
a_7 (10)	1	2	3	4	4	5	5	pertahankan	$\{a_1, a_2, a_3, a_4, a_7\}$

Perhatikan penolakan a_5: dengannya, tiga pesanan (a_1, a_3, a_5) akan jatuh tempo paling lambat jam ke-2, dan dua jam tidak dapat menyetrika tiga pesanan. Penolakan a_6 lebih halus: lima pesanan akan jatuh tempo paling lambat jam ke-4. Pesanan a_7, yang termurah dari semuanya, dipertahankan, sebab tenggatnya 6 masih menyisakan ruang.

Jadwalnya menyetrika pesanan yang dipertahankan menurut urutan tenggat — a_3 (jam 1, tenggat 1), a_1 (jam 2, tenggat 2), a_4 (jam 3, tenggat 3), a_2 (jam 4, tenggat 4), a_7 (jam 5, tenggat 6) — lalu kedua pesanan yang ditolak, yang terlambat apa pun yang kita lakukan: a_5 pada jam 6 dan a_6 pada jam 7. Pengembalian uangnya $30 + 20 = \mathbf{50}$ ribu rupiah, dari kemungkinan 235.

(d) Gaya kasar. **Kode 12.4** mencetak penelusuran yang sama lalu mencoba seluruh $2^7 = 128$ upahimpunan, memegang yang bebas dengan bobot pengembalian terbesar.

Kode 12.4 Algoritma rakus matroid untuk tugas berwaktu satuan dengan tenggat dan pengembalian uang, dengan cacah kebebasannya dicetak di setiap langkah dan pemeriksaan gaya kasar atas seluruh 128 upahimpunan. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o tasks tasks.cpp`.

```
// Persoalan Terkerjakan F: tugas berwaktu satuan dengan tenggat dan penalti, diselesaikan
// dengan algoritma greedy matroid. Tujuh pesanan setrika satu jam di binatu;
// pesanan i harus siap paling lambat jam d[i], atau binatu mengembalikan w[i] ribu rupiah.
// Program mencetak uji kebebasan N_t(A) <= t di setiap langkah, jadwal dalam
// bentuk yang-awal-dahulu, dan memeriksa jawabannya terhadap semua 2^7 upahimpunan.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <numeric>
using namespace std;

static const vector<int> D = { 2, 4, 1, 3, 2, 4, 6 };        // tenggat, dalam jam
static const vector<int> W = { 55, 45, 40, 35, 30, 20, 10 };  // pengembalian uang, ribu
rupiah
static const int n = (int)D.size();

// N[t] = banyaknya tugas dalam himpunan yang tenggatnya t atau lebih awal, t = 0..n
vector<int> counts(const vector<int>& set) {
    vector<int> N(n + 1, 0);
    for (int i : set) for (int t = D[i]; t <= n; ++t) ++N[t];
    return N;
}
bool independent(const vector<int>& set) {
    vector<int> N = counts(set);
    for (int t = 0; t <= n; ++t) if (N[t] > t) return false;
    return true;
}
string name(const vector<int>& set) {
    string s = "{";
    for (size_t k = 0; k < set.size(); ++k) s += (k ? ", a" : "a") + to_string(set[k] + 1);
    return s + "}";
}

int main() {
    vector<int> order(n); iota(order.begin(), order.end(), 0);
    stable_sort(order.begin(), order.end(), [](int a, int b) { return W[a] > W[b]; });
    cout << "task   d   w\n";
    for (int i = 0; i < n; ++i) cout << "  a" << i + 1 << setw(4) << D[i] << setw(4) <<
W[i] << "\n";
```

Kode 12.4 (Lanjutan) *Algoritma rakus matroid untuk tugas berwaktu satuan dengan tenggat dan pengembalian uang, dengan cacah kebebasannya dicetak di setiap langkah dan pemeriksaan gaya kasar atas seluruh 128 upahimpunan. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o tasks tasks.cpp.*

```
    vector<int> A;
    cout << "\nconsider   N_1..N_" << n << " of the trial set   verdict\n";
    for (int i : order) {
        vector<int> trial = A; trial.push_back(i); sort(trial.begin(), trial.end());
        vector<int> N = counts(trial);
        bool ok = independent(trial);
        cout << "  a" << i + 1 << "      ";
        for (int t = 1; t <= n; ++t) cout << setw(2) << N[t] << (N[t] > t ? "!" : " ");
        cout << "   " << (ok ? "keep  " : "REJECT") << "  ";
        if (ok) A = trial;
        cout << "A = " << name(A) << "\n";
    }
    int early = 0, late = 0; vector<int> L;
    for (int i = 0; i < n; ++i) {
        if (find(A.begin(), A.end(), i) != A.end()) early += W[i];
        else { late += W[i]; L.push_back(i); }
    }
    cout << "early set " << name(A) << ", weight " << early
         << "; late set " << name(L) << ", penalty " << late << "\n";

    vector<int> sched = A;                                    // tugas awal menurut
tenggat,
    stable_sort(sched.begin(), sched.end(), [](int a, int b) { return D[a] < D[b]; });
    for (int i : L) sched.push_back(i);                        // lalu tugas yang
terlambat
    cout << "schedule:";
    for (int s = 0; s < n; ++s) {
        int i = sched[s];
        cout << " a" << i + 1 << (s + 1 <= D[i] ? "" : "(late)");
    }
    cout << "\n";

    int best = -1, bestMask = 0, ties = 0;                    // gaya kasar atas 2^n
upahimpunan
    for (int mask = 0; mask < (1 << n); ++mask) {
        vector<int> set; int w = 0;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) { set.push_back(i); w += W[i]; }
        if (!independent(set)) continue;
        if (w > best) { best = w; bestMask = mask; ties = 1; } else if (w == best) ++ties;
    }
    vector<int> bs; for (int i = 0; i < n; ++i) if (bestMask >> i & 1) bs.push_back(i);
    int total = accumulate(W.begin(), W.end(), 0);
    cout << "brute force over " << (1 << n) << " subsets: best early weight " << best
         << " by " << name(bs) << " (" << ties << " optimal set" << (ties > 1 ? "s" : "")
         << "), minimum penalty " << total - best << "\n";
    return 0;
}
```

```
task   d   w
  a1   2  55
  a2   4  45
  a3   1  40
  a4   3  35
  a5   2  30
  a6   4  20
  a7   6  10

consider   N_1..N_7 of the trial set   verdict
  a1       0  1  1  1  1  1  1    keep    A = {a1}
  a2       0  1  1  2  2  2  2    keep    A = {a1, a2}
  a3       1  2  2  3  3  3  3    keep    A = {a1, a2, a3}
  a4       1  2  3  4  4  4  4    keep    A = {a1, a2, a3, a4}
  a5       1  3! 4! 5! 5  5  5    REJECT  A = {a1, a2, a3, a4}
  a6       1  2  3  5! 5  5  5    REJECT  A = {a1, a2, a3, a4}
  a7       1  2  3  4  4  5  5    keep    A = {a1, a2, a3, a4, a7}
early set {a1, a2, a3, a4, a7}, weight 185; late set {a5, a6}, penalty 50
schedule: a3 a1 a4 a2 a7 a5(late) a6(late)
brute force over 128 subsets: best early weight 185 by {a1, a2, a3, a4, a7} (1 optimal
set), minimum penalty 50
```

Gaya kasarnya sepakat: himpunan tepat waktu yang terbaik adalah $\{a_1, a_2, a_3, a_4, a_7\}$, ia satu-satunya himpunan optimal, dan pengembalian minimumnya 50. **Gambar 12.9** memperlihatkan contoh tujuh tugas §9.8.7, yang pada contoh itu algoritma yang sama mempertahankan himpunan berbobot 230 tetap tepat waktu dan membayar penalti 50; kedua contoh itu berbeda, dan hanya kebetulan kedua penaltinya sama-sama 50.

Tugas

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Gambar 12.9 Contoh §9.8.7 untuk penjadwalan tugas berwaktu satuan dengan tenggat dan penalti: tujuh tugas dengan tenggat d_i dan penalti w_i. Algoritma rakus matroid menjadwalkan lebih awal sebuah himpunan berbobot 230 dan membayar penalti 50.

Ongkosnya. Pengurutan berongkos $\Theta(n\lg n)$. Tiap uji kebebasan, bila dikerjakan seperti di dalam program dengan mencacah ulang, berongkos $O(n^2)$, jadi gelungnya $O(n^3)$ sebagaimana tertulis; menyimpan cacah N_t di dalam sebuah larik dan hanya memperbarui entri $t \geq d_i$ menurunkan tiap uji menjadi $O(n)$ dan seluruh algoritmanya menjadi $O(n^2)$, yaitu batas §9.8.5; struktur himpunan saling lepas menurunkannya lebih jauh lagi, menjadi $O(n\lg n)$.

Jawaban. Pertahankan $\{a_1, a_2, a_3, a_4, a_7\}$ tepat waktu, disetrika dalam urutan a_3, a_1, a_4, a_2, a_7; pesanan a_5 dan a_6 terlambat; pengembalian uangnya 50 ribu rupiah, yang optimal menurut teorema matroid dan menurut gaya kasar.

> **UJI DIRI.** Seorang pelanggan setuju mengambil a_5 belakangan, jadi tenggatnya bergeser dari 2 ke 5. Jalankan ulang algoritma rakusnya. Pesanan mana yang kini terlambat, dan berapa pengembalian uangnya?
>
> *Jawaban.* a_1 sampai a_4 dipertahankan seperti sebelumnya. a_5 (tenggat 5) kini dipertahankan: cacahnya menjadi $N_1 = 1$, $N_2 = 2$, $N_3 = 3$, $N_4 = 4$, $N_5 = 5$, semuanya dalam batas. a_6 (tenggat

> 4) ditolak, sebab N_4 akan menjadi 5. a_7 (tenggat 6) dipertahankan, dengan $N_6 = 6$. Hanya a_6 yang terlambat, dan pengembalian uangnya turun menjadi **20**.

12.3.8 Persoalan B sampai E sebagai satu program

Kode 12.5 menjawab Persoalan B sampai E secara mekanis, termasuk orakel gaya kasar yang menyertifikasi jawaban-jawaban rakusnya dan pohon Huffman §12.3.5.

Kode 12.5 Persoalan Terkerjakan B sampai E, dihitung: pemilihan aktivitas dengan empat aturan terhadap gaya kasar; tenggat terawal dahulu terhadap seluruh 720 urutan; Huffman beserta pohonnya, $B(T)$ dengan kedua cara, dan perjalanan pulang-pergi; serta *knapsack*, pecahan dan 0-1. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp` lalu jalankan `./greedy`.

```
// Persoalan Terkerjakan B sampai E Bab 12 -- algoritma greedy, dihitung.
//   B  pemilihan aktivitas: empat aturan, dan optimum sejati dengan gaya kasar
//   C  meminimumkan keterlambatan maksimum: tenggat terawal, dibandingkan dengan semua 6!
urutan
//   D  Huffman atas log mesin bersimbol tujuh, dengan perjalanan pulang-pergi
//   E  knapsack, pecahan dan 0-1, greedy dan pemrograman dinamis
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <map>
#include <algorithm>
#include <numeric>
#include <functional>
#include <climits>
using namespace std;

// ============================================ B: pemilihan aktivitas
struct Act { int i, s, f; };
static const vector<Act> ACT = {
    {1,0,3}, {2,2,4}, {3,1,6}, {4,5,7}, {5,4,9},
    {6,8,11}, {7,7,12}, {8,10,13}, {9,9,14}, {10,12,16}
};
bool compatible(const Act& a, const Act& b) { return a.f <= b.s || b.f <= a.s; }

string show(const vector<int>& v) {
    string s = "{";
    for (size_t i = 0; i < v.size(); ++i) { if (i) s += ", "; s += "a" + to_string(v[i]); }
    return s + "}";
}

// penggerak umum "ambil terbaik yang tersisa, buang yang berbenturan"
vector<int> greedyBy(function<bool(const Act&, const Act&)> better) {
    vector<Act> left = ACT; vector<int> chosen;
    while (!left.empty()) {
        const Act* best = &left[0];
        for (const Act& a : left) if (better(a, *best)) best = &a;
        Act pick = *best; chosen.push_back(pick.i);
        vector<Act> keep;
        for (const Act& a : left) if (a.i != pick.i && compatible(a, pick))
keep.push_back(a);
        left = keep;
    }
    sort(chosen.begin(), chosen.end());
    return chosen;
}
```

Kode 12.5 (Lanjutan) *Persoalan Terkerjakan B sampai E, dihitung: pemilihan aktivitas dengan empat aturan terhadap gaya kasar; tenggat terawal dahulu terhadap seluruh 720 urutan; Huffman beserta pohonnya, B(T) dengan kedua cara, dan perjalanan pulang-pergi; serta knapsack, pecahan dan 0-1. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp lalu jalankan ./greedy.*

```
int conflictsOf(const Act& a, const vector<Act>& pool) {
    int c = 0; for (const Act& b : pool) if (b.i != a.i && !compatible(a, b)) ++c;
    return c;
}

void q1() {
    cout << "=== Problem B  activity selection ===\n";
    cout << "instance (already sorted by finish time):\n  i  ";
    for (const Act& a : ACT) cout << setw(4) << a.i;
    cout << "\n  s  ";
    for (const Act& a : ACT) cout << setw(4) << a.s;
    cout << "\n  f  ";
    for (const Act& a : ACT) cout << setw(4) << a.f;
    cout << "\n  d  ";
    for (const Act& a : ACT) cout << setw(4) << a.f - a.s;
    cout << "\n  c  ";
    for (const Act& a : ACT) cout << setw(4) << conflictsOf(a, ACT);
    cout << "\n";

    auto eft = greedyBy([](const Act& a, const Act& b){ return a.f < b.f; });
    auto est = greedyBy([](const Act& a, const Act& b){ return a.s < b.s; });
    auto sdu = greedyBy([](const Act& a, const Act& b){ return a.f - a.s < b.f - b.s; });
    vector<int> few;                                        // benturan tersedikit, dihitung
ulang tiap putaran
    {
        vector<Act> left = ACT;
        while (!left.empty()) {
            const Act* best = &left[0]; int bc = conflictsOf(left[0], left);
            for (const Act& a : left) { int c = conflictsOf(a, left);
                if (c < bc || (c == bc && a.i < best->i)) { bc = c; best = &a; } }
            Act pick = *best; few.push_back(pick.i);
            vector<Act> keep;
            for (const Act& a : left) if (a.i != pick.i && compatible(a, pick))
keep.push_back(a);
            left = keep;
        }
        sort(few.begin(), few.end());
    }
    cout << "  earliest finish time -> " << show(eft) << ", size " << eft.size() << "\n";
    cout << "  earliest start time  -> " << show(est) << ", size " << est.size() << "\n";
    cout << "  shortest duration    -> " << show(sdu) << ", size " << sdu.size() << "\n";
    cout << "  fewest conflicts     -> " << show(few) << ", size " << few.size() << "\n";
```

Kode 12.5 (Lanjutan) *Persoalan Terkerjakan B sampai E, dihitung: pemilihan aktivitas dengan empat aturan terhadap gaya kasar; tenggat terawal dahulu terhadap seluruh 720 urutan; Huffman beserta pohonnya, B(T) dengan kedua cara, dan perjalanan pulang-pergi; serta knapsack, pecahan dan 0-1. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp lalu jalankan ./greedy.*

```
    int n = (int)ACT.size(), bestSize = 0; vector<int> bestSet;
    for (int mask = 0; mask < (1 << n); ++mask) {          // gaya kasar atas 1024
upahimpunan
        vector<int> sel; bool ok = true;
        for (int i = 0; i < n && ok; ++i) if (mask >> i & 1) {
            for (int j : sel) if (!compatible(ACT[i], ACT[j-1])) { ok = false; break; }
            if (ok) sel.push_back(ACT[i].i);
        }
        if (ok && (int)sel.size() > bestSize) { bestSize = (int)sel.size(); bestSet = sel;
}
    }
    cout << "  brute force over all " << (1 << n) << " subsets -> optimum size "
         << bestSize << ", e.g. " << show(bestSet) << "\n";
}

// ============================================ C: meminimumkan keterlambatan maksimum
struct Job { string name; int p, d; };
static const vector<Job> JOB = {
    {"J1",3,6}, {"J2",2,8}, {"J3",1,9}, {"J4",4,9}, {"J5",3,14}, {"J6",2,15}
};
int maxLateness(const vector<int>& order, bool print = false) {
    int t = 0, L = INT_MIN;
    for (int k : order) { t += JOB[k].p; int l = t - JOB[k].d; L = max(L, l);
        if (print) cout << "    " << JOB[k].name << ": p=" << JOB[k].p
                        << " finishes at " << setw(3) << t << ", deadline " << setw(3)
                        << JOB[k].d << ", lateness " << setw(3) << l << "\n"; }
    return L;
}
void q2() {
    cout << "\n=== Problem C  minimizing maximum lateness ===\n";
    vector<int> edf(JOB.size()); iota(edf.begin(), edf.end(), 0);
    sort(edf.begin(), edf.end(), [](int a, int b){ return JOB[a].d < JOB[b].d; });
    cout << "  earliest deadline first:\n";
    int Ledf = maxLateness(edf, true);
    cout << "    max lateness = " << Ledf << "\n";
    vector<int> spt(JOB.size()); iota(spt.begin(), spt.end(), 0);
    stable_sort(spt.begin(), spt.end(), [](int a, int b){ return JOB[a].p < JOB[b].p; });
    cout << "  shortest processing time first:\n";
    int Lspt = maxLateness(spt, true);
    cout << "    max lateness = " << Lspt << "\n";
    vector<int> perm(JOB.size()); iota(perm.begin(), perm.end(), 0);
    int best = INT_MAX, orders = 0, ties = 0;
    do { int L = maxLateness(perm); ++orders;
         if (L < best) { best = L; ties = 1; } else if (L == best) ++ties;
    } while (next_permutation(perm.begin(), perm.end()));
    cout << "  brute force over all " << orders << " orders -> optimum " << best
         << ", attained by " << ties << " of them\n";
}
```

Kode 12.5 (Lanjutan) *Persoalan Terkerjakan B sampai E, dihitung: pemilihan aktivitas dengan empat aturan terhadap gaya kasar; tenggat terawal dahulu terhadap seluruh 720 urutan; Huffman beserta pohonnya, B(T) dengan kedua cara, dan perjalanan pulang-pergi; serta knapsack, pecahan dan 0-1. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp lalu jalankan ./greedy.*

```
// ============================================ D: Huffman
struct Node { int freq; char ch; int l = -1, r = -1; };
void q3() {
    cout << "\n=== Problem D  Huffman on the machine log ===\n";
    vector<pair<char,int>> F = {
        {'R',38}, {'W',22}, {'D',15}, {'F',12}, {'P',8}, {'X',4}, {'Q',1}
    };
    vector<Node> t;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    for (auto [c, f] : F) { t.push_back({f, c}); Q.push({f, (int)t.size() - 1}); }
    int step = 0, internalSum = 0;
    while (Q.size() > 1) {
        auto [f1, x] = Q.top(); Q.pop();
        auto [f2, y] = Q.top(); Q.pop();
        t.push_back({f1 + f2, 0, x, y});
        internalSum += f1 + f2;
        cout << "  merge " << ++step << ": " << setw(3) << f1 << " + " << setw(3) << f2
             << " = " << setw(3) << f1 + f2 << "\n";
        Q.push({f1 + f2, (int)t.size() - 1});
    }
    int root = Q.top().second;
    map<char,string> code;
    function<void(int,string)> walk = [&](int v, string pre) {
        if (t[v].l == -1) { code[t[v].ch] = pre.empty() ? "0" : pre; return; }
        walk(t[v].l, pre + "0"); walk(t[v].r, pre + "1");
    };
    walk(root, "");
    // gambar pohonnya: satu baris per simpul, menjorok sesuai kedalaman, label sisi di
depan
    cout << "  the tree (internal nodes in brackets, leaves as symbol:freq):\n";
    function<void(int,string,string)> draw = [&](int v, string indent, string edge) {
        cout << "    " << indent << edge;
        if (t[v].l == -1) { cout << t[v].ch << ":" << t[v].freq << "\n"; return; }
        cout << "(" << t[v].freq << ")\n";
        draw(t[v].l, indent + "   ", "0-"); draw(t[v].r, indent + "   ", "1-");
    };
    draw(root, "", "");
    int variable = 0, fixedBits = 0;
    cout << "  sym  freq  code        len  freq*len\n";
    for (auto [c, f] : F) {
        cout << "   " << c << setw(6) << f << "  " << left << setw(10) << code[c]
             << right << setw(4) << code[c].size() << setw(9) << f * (int)code[c].size() <<
"\n";
        variable += f * (int)code[c].size();
        fixedBits += f * 3;
    }
    cout << "  B(T) by summing over the leaves          = " << variable << " kbit\n";
```

Kode 12.5 (Lanjutan) *Persoalan Terkerjakan B sampai E, dihitung: pemilihan aktivitas dengan empat aturan terhadap gaya kasar; tenggat terawal dahulu terhadap seluruh 720 urutan; Huffman beserta pohonnya, B(T) dengan kedua cara, dan perjalanan pulang-pergi; serta knapsack, pecahan dan 0-1. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp lalu jalankan ./greedy.*

```
    cout << "  B(T) by summing the internal node labels = " << internalSum << " kbit\n";
    cout << "  fixed-length (3 bits, 7 symbols)         = " << fixedBits << " kbit\n";
    cout << "  saving = " << fixedBits - variable << " kbit = " << fixed << setprecision(2)
         << 100.0 * (fixedBits - variable) / fixedBits << "%\n";
    string msg = "RRWDRFPRWXQDR";
    string bits; for (char c : msg) bits += code[c];
    string back; int v = root;
    for (char b : bits) { v = (b == '0' ? t[v].l : t[v].r);
        if (t[v].l == -1) { back += t[v].ch; v = root; } }
    cout << "  round trip: \"" << msg << "\" -> " << bits.size() << " bits -> \"" << back
         << "\"  " << (back == msg ? "OK" : "FAILED") << "\n";
}

// ========================================== E: knapsack
struct Item { int v, w; };
static const vector<Item> IT = { {7,3}, {6,3}, {9,5}, {9,5} };
static const int CAP = 10;
void q4() {
    cout << "\n=== Problem E  the knapsack, capacity " << CAP << " ===\n";
    int n = (int)IT.size();
    vector<int> ord(n); iota(ord.begin(), ord.end(), 0);
    stable_sort(ord.begin(), ord.end(), [](int a, int b) {
        return (double)IT[a].v / IT[a].w > (double)IT[b].v / IT[b].w; });
    cout << "  item  v   w   v/w\n";
    for (int i : ord) cout << setw(6) << i + 1 << setw(4) << IT[i].v << setw(4) << IT[i].w
                          << setw(7) << fixed << setprecision(3)
                          << (double)IT[i].v / IT[i].w << "\n";
    double room = CAP, frac = 0;
    cout << "  fractional greedy:";
    for (int i : ord) {
        if (room <= 0) break;
        double take = min((double)IT[i].w, room);
        frac += take / IT[i].w * IT[i].v; room -= take;
        cout << "  item" << i + 1 << " x" << setprecision(2) << take / IT[i].w;
    }
    cout << "  -> value " << setprecision(2) << frac << "\n";
    int cap = CAP, gv = 0; string gs;
    for (int i : ord) if (IT[i].w <= cap) { cap -= IT[i].w; gv += IT[i].v; gs += " item" +
to_string(i+1); }
    cout << "  0-1 greedy by density:" << gs << " -> value " << gv
         << ", weight used " << CAP - cap << "\n";
    vector<vector<int>> c(n + 1, vector<int>(CAP + 1, 0));
    for (int i = 1; i <= n; ++i)
        for (int w = 0; w <= CAP; ++w)
            c[i][w] = (IT[i-1].w > w) ? c[i-1][w]
                                      : max(c[i-1][w], IT[i-1].v + c[i-1][w - IT[i-1].w]);
    cout << "  the dynamic-programming table c[i][w]:\n      w =";
```

Kode 12.5 (Lanjutan) *Persoalan Terkerjakan B sampai E, dihitung: pemilihan aktivitas dengan empat aturan terhadap gaya kasar; tenggat terawal dahulu terhadap seluruh 720 urutan; Huffman beserta pohonnya, B(T) dengan kedua cara, dan perjalanan pulang-pergi; serta knapsack, pecahan dan 0-1. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o greedy greedy.cpp lalu jalankan ./greedy.*

```
    for (int w = 0; w <= CAP; ++w) cout << setw(4) << w;
    cout << "\n";
    for (int i = 0; i <= n; ++i) {
        cout << "  i = " << i << "  ";
        for (int w = 0; w <= CAP; ++w) cout << setw(4) << c[i][w];
        cout << "\n";
    }
    string sel; int w = CAP;
    for (int i = n; i >= 1; --i) if (c[i][w] != c[i-1][w]) { sel = " item" + to_string(i) +
sel; w -= IT[i-1].w; }
    cout << "  0-1 optimum by dynamic programming = " << c[n][CAP] << ", taking" << sel <<
"\n";
    int bv = 0, bm = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        int tv = 0, tw = 0;
        for (int i = 0; i < n; ++i) if (mask >> i & 1) { tv += IT[i].v; tw += IT[i].w; }
        if (tw <= CAP && tv > bv) { bv = tv; bm = mask; }
    }
    cout << "  brute force over " << (1 << n) << " subsets = " << bv << " (mask " << bm <<
")\n";
    cout << "  greedy loses " << bv - gv << " of " << bv << " = " << setprecision(1)
         << 100.0 * (bv - gv) / bv << "% of the 0-1 optimum\n";
}

int main() { q1(); q2(); q3(); q4(); return 0; }
```

Keluaran lengkapnya, yang menyertifikasi setiap hitungan tangan di atas:

```
=== Problem B  activity selection ===
instance (already sorted by finish time):
  i     1   2   3   4   5   6   7   8   9  10
  s     0   2   1   5   4   8   7  10   9  12
  f     3   4   6   7   9  11  12  13  14  16
  d     3   2   5   2   5   3   5   3   5   4
  c     2   2   4   2   4   4   4   4   4   2
  earliest finish time -> {a1, a4, a6, a10}, size 4
  earliest start time  -> {a1, a5, a9}, size 3
  shortest duration    -> {a2, a4, a6, a10}, size 4
  fewest conflicts     -> {a1, a4, a6, a10}, size 4
  brute force over all 1024 subsets -> optimum size 4, e.g. {a1, a4, a6, a10}

=== Problem C  minimizing maximum lateness ===
  earliest deadline first:
    J1: p=3 finishes at   3, deadline   6, lateness  -3
    J2: p=2 finishes at   5, deadline   8, lateness  -3
    J3: p=1 finishes at   6, deadline   9, lateness  -3
    J4: p=4 finishes at  10, deadline   9, lateness   1
    J5: p=3 finishes at  13, deadline  14, lateness  -1
    J6: p=2 finishes at  15, deadline  15, lateness   0
    max lateness = 1
  shortest processing time first:
    J3: p=1 finishes at   1, deadline   9, lateness  -8
    J2: p=2 finishes at   3, deadline   8, lateness  -5
    J6: p=2 finishes at   5, deadline  15, lateness -10
    J1: p=3 finishes at   8, deadline   6, lateness   2
    J5: p=3 finishes at  11, deadline  14, lateness  -3
    J4: p=4 finishes at  15, deadline   9, lateness   6
    max lateness = 6
  brute force over all 720 orders -> optimum 1, attained by 20 of them

=== Problem D  Huffman on the machine log ===
  merge 1:   1 +   4 =   5
  merge 2:   5 +   8 =  13
  merge 3:  12 +  13 =  25
  merge 4:  15 +  22 =  37
  merge 5:  25 +  37 =  62
  merge 6:  38 +  62 = 100
  the tree (internal nodes in brackets, leaves as symbol:freq):
    (100)
       0-R:38
       1-(62)
          0-(25)
             0-F:12
             1-(13)
                0-(5)
                   0-Q:1
                   1-X:4
                1-P:8
          1-(37)
             0-D:15
```

```
              1-W:22
   sym  freq  code        len  freq*len
    R    38  0            1       38
    W    22  111          3       66
    D    15  110          3       45
    F    12  100          3       36
    P     8  1011         4       32
    X     4  10101        5       20
    Q     1  10100        5        5
   B(T) by summing over the leaves          = 242 kbit
   B(T) by summing the internal node labels = 242 kbit
   fixed-length (3 bits, 7 symbols)         = 300 kbit
   saving = 58 kbit = 19.33%
   round trip: "RRWDRFPRWXQDR" -> 34 bits -> "RRWDRFPRWXQDR"  OK

 === Problem E  the knapsack, capacity 10 ===
   item  v   w   v/w
      1   7   3  2.333
      2   6   3  2.000
      3   9   5  1.800
      4   9   5  1.800
   fractional greedy:  item1 x1.00  item2 x1.00  item3 x0.80  -> value 20.20
   0-1 greedy by density: item1 item2 -> value 13, weight used 6
   the dynamic-programming table c[i][w]:
       w =   0   1   2   3   4   5   6   7   8   9  10
   i = 0     0   0   0   0   0   0   0   0   0   0   0
   i = 1     0   0   0   7   7   7   7   7   7   7   7
   i = 2     0   0   0   7   7   7  13  13  13  13  13
   i = 3     0   0   0   7   7   9  13  13  16  16  16
   i = 4     0   0   0   7   7   9  13  13  16  16  18
   0-1 optimum by dynamic programming = 18, taking item3 item4
   brute force over 16 subsets = 18 (mask 12)
   greedy loses 5 of 18 = 27.8% of the 0-1 optimum
```

Perhatikan apa yang diperlihatkan blok pertamanya: pada contoh *ini*, tiga dari keempat aturan aktivitas mengembalikan himpunan berukuran 4 dan hanya waktu-mulai-terawal yang gagal. Justru itulah sebabnya bagian (d) Persoalan B menuntut bukti. Kesepakatan pada satu contoh bukanlah kebenaran; aturan durasi-terpendek kebetulan benar di sini dan keliru pada contoh tiga aktivitas di Soal Latihan 12.7.

12.4 Persoalan Terkerjakan Lanjutan II: Menelusuri Sebuah Graf

12.4.1 Kedua graf subbab ini

Sebagian besar persoalan dalam subbab ini memakai dua graf yang sama, jadi keduanya dinyatakan sekali di sini. Di sepanjang subbab, daftar ketetanggaan disimpan menurut **urutan abjad**, dan gelung luar pencarian mendalam mana pun menimbang simpul menurut **urutan abjad**. Kesepakatan itu penting: *jarak* yang dihitung sebuah BFS tidak bergantung padanya, tetapi pohon, cap waktu, dan golongan sisinya bergantung, dan sebuah penelusuran hanya dapat diperiksa terhadap penelusuran lain yang dibuat dengan kesepakatan yang sama.

G_1 adalah graf **tak berarah**, dengan $V = \{A, \ldots, I\}$ dan dua belas sisi

$$E_1 = \{AB,\ AC,\ BD,\ BE,\ CE,\ CF,\ DG,\ EG,\ EH,\ FH,\ GI,\ HI\}.$$

G_2 adalah graf **berarah**, dengan $V = \{A, \ldots, G\}$ dan sebelas busur

$$A \to B,\ A \to C,\ A \to D,\ B \to D,\ C \to D,\ C \to E,\ D \to A,\ E \to F,\ F \to E,\ G \to C,\ G \to F.$$

Ada baiknya menggambar keduanya di atas kertas sebelum melanjutkan membaca. G_1 tampak seperti kisi berbentuk belah ketupat dengan lima baris (A; B, C; D, E, F; G, H; I) dan setiap sisinya berjalan di antara baris-baris yang bertetangga; G_2 punya dua gelung kecil, $A \to B \to D \to A$ dan $E \rightleftarrows F$, yang akan ditemukan Persoalan H.

12.4.2 Persoalan Terkerjakan G — pencarian melebar, ditelusuri

Persoalan. (a) Berikan derajat setiap simpul G_1 dan periksalah lema jabat tangan. (b) Jalankan pencarian melebar dari A, dengan memberikan tabel satu baris per putaran gelung **while**: simpul yang dikeluarkan, apa yang ditemukannya, dan antrean sesudahnya. (c) Tabulasikan $d[v]$ dan $\pi[v]$ lalu nyatakan sebuah lintasan terpendek dari A ke I. (d) Ada berapa sisi pohon, dan mengapa haruslah sebanyak itu?

Yang diuji. Apakah Anda dapat menjalankan algoritma **Gambar 12.10** (§10.7.3) persis, termasuk bagian yang sering dilewati orang: *memeriksa* sisi menuju simpul yang sudah kelabu lalu dengan benar tidak mengerjakan apa pun.

```
BFS (G, s)
for each vertex u ∈ V[G] − {s}
    color[u] ← WHITE
    d[u] ← ∞
    π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← {s}
while Q ≠ ∅ do
    u ← head[Q]
    for each v ∈ Adj[u] do
        if color[v] = WHITE then
            color[v] ← GRAY
            d[v] ← d[u] + 1
            π[v] ← u
            ENQUEUE (Q, v)
    DEQUEUE (Q)
    color[u] ← BLACK
```

Inisialisasi setiap simpul kecuali s

Inisialisasi sumber dan antreannya

Tangani semua anak u sebelum menangani anak dari anak mana pun

Gambar 12.10 Pencarian melebar. Blok ambernya menyiapkan setiap simpul selain sumbernya; blok toskanya menyiapkan sumbernya dan antreannya; blok abu-abunya adalah pencariannya. Catatan di sebelah kanan menyatakan invariannya dengan kata-kata — *tangani semua anak u sebelum anak mana pun dari anak-anaknya* — dan kalimat itulah seluruh bedanya dari pencarian mendalam.

Penyelesaian.

(a) Derajatnya. $\deg(A) = 2$ (B, C), $\deg(B) = 3$ (A, D, E), $\deg(C) = 3$ (A, E, F), $\deg(D) = 2$ (B, G), $\deg(E) = 4$ (B, C, G, H), $\deg(F) = 2$ (C, H), $\deg(G) = 3$ (D, E, I), $\deg(H) = 3$ (E, F, I), $\deg(I) = 2$ (G, H). Lema jabat tangan (§10.4.3) berbunyi

$$\sum_{v \in V} \deg(v) = 2\,|E|, \tag{12.13}$$

sebab setiap sisi menyumbang satu kepada derajat masing-masing dari kedua ujungnya. Di sini ruas kiri Persamaan (12.13) adalah $2 + 3 + 3 + 2 + 4 + 2 + 3 + 3 + 2 = 24$ dan ruas kanannya $2 \times 12 = 24$. ✓

(b) Penelusurannya. Awali dengan $d[A] = 0$, yang lain semuanya ∞, lalu antrekan A; **Tabel 12.21** adalah hasilnya.

Tabel 12.21 Pencarian melebar atas G_1 dari A, satu baris per putaran gelung **while**. Sembilan simpul, sembilan langkah: setiap simpul diantrekan sekali dan dikeluarkan sekali. Jarak di dalam antreannya tak pernah menurun dari kiri ke kanan dan tak pernah merentang lebih dari satu nilai — yaitu invarian antrean §10.7.9.

Langkah	Keluarkan	Ditemukan, dengan d	Dilewati, dan sebabnya	Antrean sesudahnya
1	A	B: 1, C: 1	—	B: 1 C: 1
2	B	D: 2, E: 2	A hitam	C: 1 D: 2 E: 2
3	C	F: 2	A hitam; E **kelabu**	D: 2 E: 2 F: 2
4	D	G: 3	B hitam	E: 2 F: 2 G: 3
5	E	H: 3	B, C hitam; G **kelabu**	F: 2 G: 3 H: 3
6	F	—	C hitam; H kelabu	G: 3 H: 3
7	G	I: 4	D, E hitam	H: 3 I: 4
8	H	—	E, F hitam; I kelabu	I: 4
9	I	—	G, H hitam	(kosong)

Dua baris yang paling penting adalah 3 dan 5. Pada langkah 3, daftar C memuat E, dan E sudah kelabu dengan $d[E] = 2$; kita bisa saja menghitung $d[C] + 1 = 2$ dan memperoleh bilangan yang sama, tetapi algoritmanya tidak boleh menyentuh E, sebab **sentuhan pertama yang menang** dan menyentuh ulang akan mengubah $\pi[E]$. Pada langkah 5, daftar E memuat G, yang sudah kelabu pada $d = 3$; mengabaikannya itulah yang menjaga pohonnya tetap pohon.

(c) Lariknya, dan lintasannya. **Tabel 12.22** mengumpulkan keluarannya.

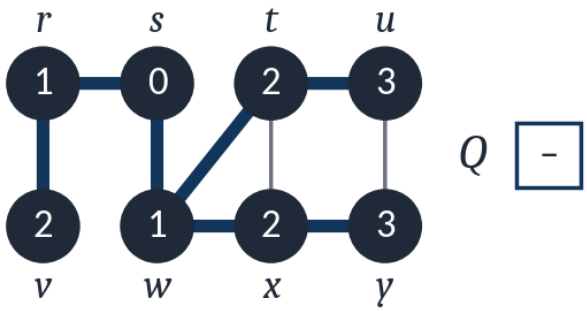
Tabel 12.22 Keluaran pencarian melebar atas G_1: jarak dan pendahulu. Membaca π mundur dari simpul mana pun lalu membaliknya memberikan sebuah lintasan terpendek, dan panjangnya selalu sama dengan d simpul itu.

v	A	B	C	D	E	F	G	H	I
$d[v]$	0	1	1	2	2	2	3	3	4
$\pi[v]$	–	A	A	B	B	C	D	E	G

Sebuah lintasan terpendek ke I: $\pi[I] = G$, $\pi[G] = D$, $\pi[D] = B$, $\pi[B] = A$. Dibalik, $A \to B \to D \to G \to I$, sepanjang $4 = d[I]$. ✓ Ada lintasan terpendek kedua, $A\,C\,E\,H\,I$, juga sepanjang 4; BFS mengembalikan salah satunya, bukan semuanya.

(d) Sisi pohonnya. Delapan — satu untuk setiap simpul kecuali sumbernya, sebab tiap simpul bukan-sumber ditemukan persis sekali dan memperoleh persis satu pendahulu. Dan $8 = 9 - 1 = |V| - 1$, yaitu cacah sisi sebuah pohon atas sembilan simpul (§11.4.2). **Gambar 12.11**

memperlihatkan fakta yang sama pada contoh bersimpul delapan §10.7.4: delapan simpul, tujuh sisi tebal.



Gambar 12.11 Pencarian melebar yang sudah selesai: setiap simpul hitam, antreannya kosong, tiap simpul membawa jaraknya dari s, dan ketujuh sisi tebalnya membentuk pohon melebar atas delapan simpul.

> **PERINGATAN: JEBAKANNYA.** *Jarak* yang dihitung sebuah BFS adalah sifat grafnya dan tidak bergantung pada urutan daftar ketetanggaannya. *Pohonnya*, larik π, dan isi antreannya semuanya bergantung. Jika π Anda berbeda dari milik orang lain, kalian mungkin sama-sama benar; jika d Anda berbeda, sekurang-kurangnya salah seorang dari kalian keliru.

Jawaban. Derajatnya berjumlah $24 = 2|E|$; $d = (0,1,1,2,2,2,3,3,4)$ untuk A sampai I; sebuah lintasan terpendek ke I adalah $A\ B\ D\ G\ I$; delapan sisi pohon.

> **UJI DIRI.** Jalankan BFS pada G_1 dari E. Berikan d dan π untuk setiap simpul, dan jarak terbesar dari E.
>
> *Jawaban.* d: A 2, B 1, C 1, D 2, E 0, F 2, G 1, H 1, I 2. π: $A \leftarrow B$, $B \leftarrow E$, $C \leftarrow E$, $D \leftarrow B$, $F \leftarrow C$, $G \leftarrow E$, $H \leftarrow E$, $I \leftarrow G$. Jarak terbesarnya **2**: setiap simpul berada paling jauh dua sisi dari E, yang menjadikan E simpul paling sentral di G_1 (dari A jarak terbesarnya 4).

12.4.3 Persoalan Terkerjakan H — pencarian mendalam pada digraf, setiap busur digolongkan

Persoalan. (a) Jalankan pencarian mendalam pada G_2 dan berikan $d[v]/f[v]$ untuk setiap simpul. (b) Uraikan hutan mendalamnya dan katakan berapa pohon yang dimilikinya. (c) Golongkan setiap satu dari kesebelas busurnya sebagai pohon, mundur, maju, atau silang, beserta uji warna yang menentukan masing-masingnya. (d) Tuliskan struktur kurungnya, dan katakan apa yang diberitahukannya tentang B dan C.

Yang diuji. Uji warna **Gambar 12.12** (§10.8.2), diterapkan tanpa ragu sebelas kali, dan kedisiplinan membubuhkan cap jam pada *kedua* kejadiannya.

```
DFS (G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
        then DFS-VISIT (u)

DFS-VISIT (u)
color[u] ← GRAY                    ▷ Simpul putih u ditemukan.
d[u] ← time                        ▷ Catat waktu penemuan.
time ← time + 1                    ▷ Majukan jam global.
for each v ∈ Adj[u]                ▷ Jelajahi semua sisi (u, v).
    do if color[v] = WHITE
        then DFS-VISIT (v)
color[u] ← BLACK                   ▷ Hitamkan u; ia selesai.
f[u] ← time                        ▷ Catat waktu selesai.
time ← time + 1                    ▷ Majukan jam global.
```

Gambar 12.12 Pencarian mendalam. Prosedur luarnya memutihkan setiap simpul, memulai jamnya dari nol, dan memanggil DFS-Visit pada setiap simpul yang masih putih — tiap panggilan semacam itu mengakari satu pohon hutannya. Prosedur dalamnya mengelabui u, membubuhkan $d[u]$, berekursi ke setiap tetangga putih, lalu menghitamkan u dan membubuhkan $f[u]$.

Penyelesaian.

(a) Penelusurannya. Daftar ketetanggaan menurut abjad: $\text{Adj}[A] = [B, C, D]$, $\text{Adj}[B] = [D]$, $\text{Adj}[C] = [D, E]$, $\text{Adj}[D] = [A]$, $\text{Adj}[E] = [F]$, $\text{Adj}[F] = [E]$, $\text{Adj}[G] = [C, F]$.

- A putih. Kunjungi: kelabu, $d[A] = 1$.
- $\text{Adj}[A]$, pertama B: putih. **Sisi pohon** $A \to B$. Kelabu, $d[B] = 2$.
- $\text{Adj}[B] = [D]$: putih. **Sisi pohon** $B \to D$. Kelabu, $d[D] = 3$.
- $\text{Adj}[D] = [A]$: A **kelabu**, jadi ia leluhur. **Sisi mundur** $D \to A$. Inilah algoritmanya menemukan siklus $A \to B \to D \to A$.
- Daftar D habis: hitam, $f[D] = 4$. Kembali ke B; habis: hitam, $f[B] = 5$. Kembali ke A.
- $\text{Adj}[A]$, berikutnya C: putih. **Sisi pohon** $A \to C$. Kelabu, $d[C] = 6$.
- $\text{Adj}[C]$, pertama D: **hitam**, dan $d[C] = 6 > 3 = d[D]$, jadi D bukan leluhur dan bukan keturunan C. **Sisi silang** $C \to D$.
- Berikutnya E: putih. **Sisi pohon** $C \to E$. Kelabu, $d[E] = 7$.
- $\text{Adj}[E] = [F]$: putih. **Sisi pohon** $E \to F$. Kelabu, $d[F] = 8$.
- $\text{Adj}[F] = [E]$: **kelabu**. **Sisi mundur** $F \to E$ — siklus dua $E \rightleftarrows F$.
- F hitam, $f[F] = 9$. E hitam, $f[E] = 10$. C hitam, $f[C] = 11$.
- Kembali di $\text{Adj}[A]$, entri terakhir D: **hitam**, dan $d[A] = 1 < 3 = d[D]$, jadi D *memang* keturunan A (dicapai lewat B). **Sisi maju** $A \to D$.

- A hitam, $f[A] = 12$. Pohon pertamanya rampung.
- Gelung luarnya mendapati B, C, D, E, F hitam dan G putih. Kunjungi G: kelabu, $d[G] = 13$. $\text{Adj}[G] = [C, F]$: keduanya hitam dengan d lebih kecil, jadi keduanya **sisi silang**. G hitam, $f[G] = 14$.

Tabel 12.23 Cap waktu dan pendahulu sesudah pencarian mendalam atas G_2. Jamnya berhenti di $14 = 2|V|$, dan keempat belas capnya adalah bilangan bulat 1 sampai 14, masing-masing terpakai sekali — itulah pemeriksaan pertama yang harus dijalankan pada jawaban DFS mana pun.

v	A	B	C	D	E	F	G
d/f	1/12	2/5	6/11	3/4	7/10	8/9	13/14
π	–	A	A	B	C	E	–

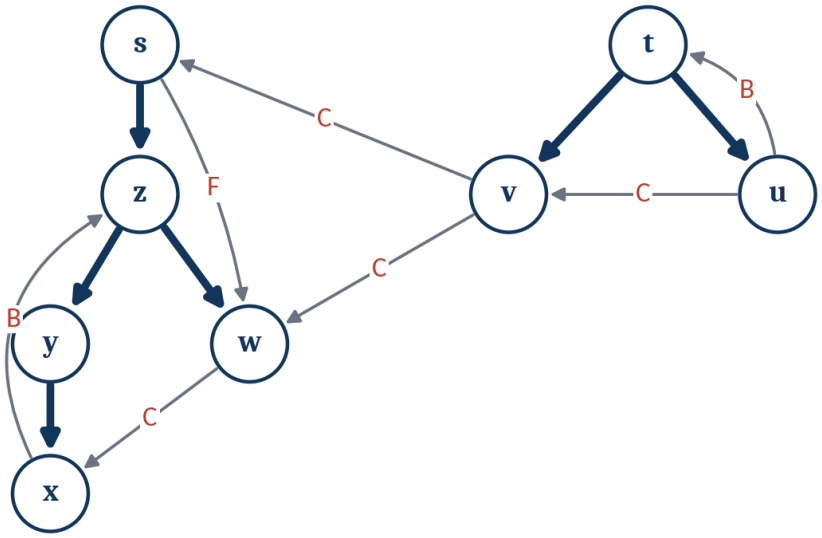
(b) Hutannya. Membaca **Tabel 12.23**, ada dua akar, A dan G — kedua simpul tanpa pendahulu — jadi hutannya punya **dua pohon**: $\{A, B, D, C, E, F\}$ dan $\{G\}$. Sisi pohonnya berjumlah $|V| -$ (banyaknya pohon) $= 7 - 2 = 5$, yang cocok dengan kelima yang ditemukan di atas.

(c) Penggolongannya. **Tabel 12.24** mendaftar setiap busur beserta uji yang menentukannya.

Tabel 12.24 Setiap busur G_2 digolongkan, beserta uji yang menentukannya. Lima pohon, dua mundur, satu maju, tiga silang — sebelas seluruhnya. Dua sisi mundur berarti grafnya tidak taksiklik.

Busur	Warna ujung jauhnya saat diperiksa	Golongan
$A \to B$	putih	pohon
$A \to C$	putih	pohon
$A \to D$	hitam, $d[A] < d[D]$	**maju**
$B \to D$	putih	pohon
$C \to D$	hitam, $d[C] > d[D]$	**silang**
$C \to E$	putih	pohon
$D \to A$	**kelabu**	**mundur**
$E \to F$	putih	pohon
$F \to E$	**kelabu**	**mundur**
$G \to C$	hitam, $d[G] > d[C]$	silang
$G \to F$	hitam, $d[G] > d[F]$	silang

Satu-satunya aturan yang harus dihafalkan ada di kolom tengahnya: **kelabu berarti mundur, putih berarti pohon, dan hitam terbelah menjadi maju atau silang menurut perbandingan waktu temunya. Gambar 12.13** membuat penggolongan yang sama menjadi kasatmata pada contoh yang lebih besar di §10.8.10 — sisi mundur melengkung *ke atas*, sisi maju menunjuk *ke bawah* melewati satu tingkat, sisi silang berjalan *menyamping*.



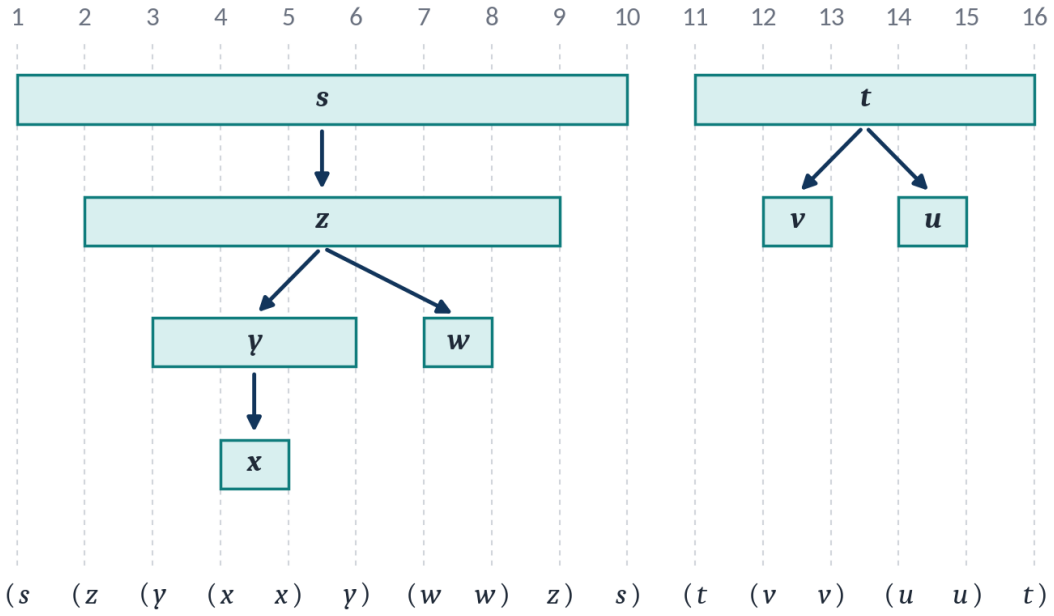
Gambar 12.13 Sebuah contoh yang lebih besar digambar ulang sebagai hutan mendalamnya, sisi pohonnya tebal dan menurun. Kini penggolongannya menjadi geometris: **B** melengkung kembali ke atas menuju leluhur, **F** menunjuk ke bawah melewati satu tingkat antara menuju keturunan, dan **C** berjalan menyamping di antara cabang atau di antara pohon.

(d) Struktur kurungnya. Menuliskan "(" pada tiap penemuan dan ")" pada tiap penyelesaian, menurut urutan jamnya:

$$(A\ (B\ (D\ D)\ B)\ (C\ (E\ (F\ F)\ E)\ C)\ A)\ (G\ G)$$

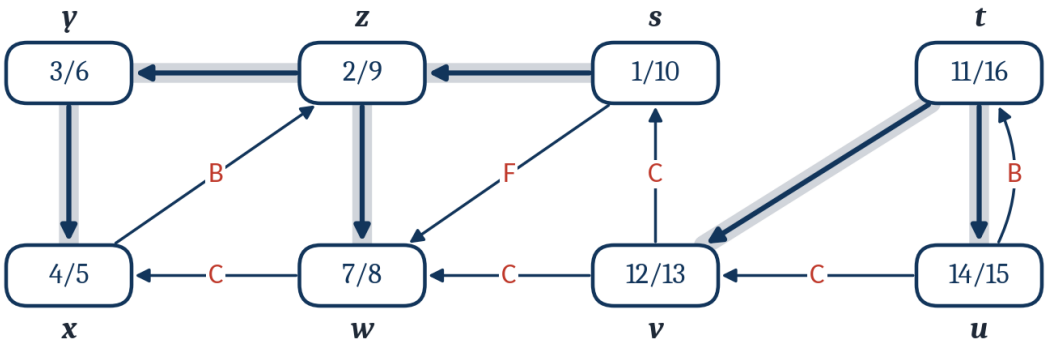
Setiap pasangan entah tersarang di dalam pasangan lain entah sepenuhnya lepas darinya; tak ada dua yang tumpang tindih sebagian. Itulah teorema kurung (§10.8.8), yang digambarkan pada **Gambar 12.14**.

Apa yang diberitahukannya tentang B dan C: selangnya adalah [2,5] dan [6,11], yang **saling lepas**. Jadi tak satu pun leluhur bagi yang lain di dalam hutannya — keduanya cabang terpisah di bawah A. Seandainya selangnya tersarang, yang satu akan menjadi leluhur bagi yang lain; seandainya keduanya tumpang tindih sebagian, algoritmanya rusak.



Gambar 12.14 Struktur kurung. Di atas, hutan mendalamnya dengan sisi pohon menunjuk ke bawah; di bawah, jamnya dari 1 sampai 16 dengan satu batang per simpul yang merentang $[d, f]$, dan pengurungannya ditulis di sepanjang dasarnya. Setiap pasangan batang tersarang atau lepas — tak pernah tumpang tindih sebagian.

Gambar 12.15 adalah contoh bercap waktu §10.8.4 selengkapnya, dan jawaban Anda sendiri layak direproduksi dalam bentuk itu: satu kotak per simpul yang memuat d/f, anak panah tebal untuk sisi pohon, dan **B**, **F**, **C** dituliskan pada yang lain.



Gambar 12.15 Pencarian mendalam bersimpul delapan, bercap waktu: s adalah 1/10, z adalah 2/9, y adalah 3/6, x adalah 4/5, w adalah 7/8, t adalah 11/16, v adalah 12/13, u adalah 14/15. Satu gambar membawa cap waktunya, hutannya, dan penggolongan sisinya selengkapnya sekaligus — bentuk yang layak ditiru ketika Anda menuliskan sebuah pencarian sendiri.

Jawaban. d/f seperti pada **Tabel 12.23**; dua pohon; lima busur pohon, dua mundur, satu maju, dan tiga silang; B dan C punya selang yang saling lepas dan tidak berkerabat di dalam hutannya.

> **UJI DIRI.** Biarkan segalanya sama kecuali bahwa daftar A disimpan sebagai $[D, C, B]$. Ulangi bagian (a) dan (c). Golongan mana yang berubah?
>
> *Jawaban.* d/f: A 1/12, D 2/3, C 4/9, E 5/8, F 6/7, B 10/11, G 13/14. Kini $A \to D$ adalah sisi **pohon** (ia diperiksa pertama, selagi D putih), $D \to A$ tetap mundur, $A \to C$, $C \to E$, $E \to F$, dan $A \to B$ adalah sisi pohon, $F \to E$ mundur, dan $C \to D$, $B \to D$, $G \to C$, $G \to F$ silang. Totalnya: lima pohon, dua mundur, **nol** maju, **empat** silang. Sisi majunya lenyap dan satu sisi silang muncul; vonis "tidak taksiklik" tidak berubah, sebab ia sifat grafnya dan bukan sifat pencariannya.

12.4.4 Persoalan Terkerjakan I — pencarian mendalam pada graf tak berarah

Persoalan. Jalankan pencarian mendalam pada graf *tak berarah* G_1, mulai dari A. (a) Berikan tabel peristiwa menurut urutan jam — setiap penemuan, setiap penyelesaian, dan setiap sisi yang mencapai simpul kelabu selain induknya. (b) Berikan d/f dan π untuk setiap simpul. (c) Golongkan setiap sisi, dan jelaskan mengapa hanya dua dari keempat golongan yang muncul. (d) Kaitkan banyaknya sisi mundur dengan banyaknya siklus bebas G_1.

Yang diuji. Pencarian mendalam pada graf tak berarah (§10.8.10), yang di dalamnya setiap sisi duduk di *dua* daftar ketetanggaan dan karena itu diperiksa dua kali, serta aturan yang menentukan arti penglihatan kedua itu.

Penyelesaian.

(a) Peristiwanya. Daftar ketetanggaannya, menurut abjad, adalah A: $[B, C]$, B: $[A, D, E]$, C: $[A, E, F]$, D: $[B, G]$, E: $[B, C, G, H]$, F: $[C, H]$, G: $[D, E, I]$, H: $[E, F, I]$, I: $[G, H]$. Ketika pencarian di u memeriksa tetangga v ada empat kasus:

- v **putih**: sebuah sisi pohon; berekursilah.
- v **kelabu** dan v adalah induk u: inilah sisi pohon $\{u, v\}$ yang terlihat lagi dari bawah. Lewati.
- v **kelabu** dan bukan induknya: v adalah leluhur sejati, jadi $\{u, v\}$ adalah **sisi mundur**.
- v **hitam**: v ditemukan sesudah u dan sudah selesai, jadi v keturunan u dan sisinya sudah digolongkan sebagai sisi mundur *dari pihak* v. Lewati.

Mengikuti daftar-daftarnya:

- Temukan A ($t = 1$). Tetangga pertamanya B putih: temukan B ($t = 2$).
- Di B: A adalah induknya. D putih: temukan D ($t = 3$).
- Di D: B adalah induknya. G putih: temukan G ($t = 4$).
- Di G: D adalah induknya. E putih: temukan E ($t = 5$).
- Di E: B **kelabu dan bukan induknya** (induknya G): **sisi mundur** EB. C putih: temukan C ($t = 6$).
- Di C: A kelabu dan bukan induknya: **sisi mundur** CA. E adalah induknya. F putih: temukan F ($t = 7$).
- Di F: C adalah induknya. H putih: temukan H ($t = 8$).
- Di H: E kelabu dan bukan induknya: **sisi mundur** HE. F adalah induknya. I putih: temukan I ($t = 9$).
- Di I: G kelabu dan bukan induknya: **sisi mundur** IG. H adalah induknya. Daftarnya habis: selesaikan I ($t = 10$).
- Menggulung balik: H tak punya sisa, selesai ($t = 11$); F selesai ($t = 12$); C selesai ($t = 13$).
- Kembali di E: G adalah induknya; H **hitam** — sisi HE, yang sudah digolongkan dari pihak H. Selesaikan E ($t = 14$).
- Kembali di G: I hitam (sisi IG, sudah digolongkan). Selesaikan G ($t = 15$). Lalu D ($t = 16$).
- Kembali di B: E hitam (sisi EB). Selesaikan B ($t = 17$). Kembali di A: C hitam (sisi CA). Selesaikan A ($t = 18$).

(b) Tabelnya. **Tabel 12.25** mengumpulkan cap waktu dan pendahulunya.

Tabel 12.25 Pencarian mendalam atas graf tak berarah G_1 dari A. Jamnya berakhir di $18 = 2|V|$. Selang-selangnya tersarang sempurna, sebab pohon mendalamnya di sini satu lintasan tunggal yang melewati kesembilan simpul.

v	A	B	C	D	E	F	G	H	I
d/f	1/18	2/17	6/13	3/16	5/14	7/12	4/15	8/11	9/10
π	–	A	E	B	G	C	D	F	H

(c) Penggolongannya. Sisi pohon: $AB, BD, DG, GE, EC, CF, FH, HI$ — delapan, yaitu $|V| - 1$, sebab grafnya terhubung dan hutannya satu pohon tunggal. Sisi mundur: EB, CA, HE, IG — empat. Bersama-sama keduanya mencakup seluruh $8 + 4 = 12$ sisi.

Sisi maju dan sisi silang **tidak dapat** muncul pada graf tak berarah. Sisi maju adalah sisi $\{u, v\}$ yang pertama kali diperiksa dari u ketika v sudah menjadi keturunan yang selesai — tetapi v, selagi dicari, memuat u di dalam daftarnya sendiri dan mendapati u kelabu, jadi ia sudah lebih dahulu menggolongkan sisi itu sebagai sisi mundur. Sisi silang akan menghubungkan dua simpul yang tak satu pun leluhur bagi yang lain — tetapi yang mana pun di antara keduanya yang ditemukan lebih dahulu akan mendapati yang lain masih putih dan menjadikan sisinya sisi pohon. Inilah teorema §10.8.10, dan penelusuran di atas memperlihatkannya bekerja: setiap penglihatan hitam adalah sisi yang sudah digolongkan dari ujung satunya.

(d) Sisi mundur dan siklus. G_1 terhubung, jadi banyaknya siklus bebasnya adalah $|E| - |V| + 1 = 12 - 9 + 1 = 4$ — peringkat sirkuit §12.4.5. Pencariannya menemukan persis **4** sisi mundur. Itu bukan kebetulan: setiap sisi yang tidak berada di pohon mendalam adalah sisi mundur, dan setiap sisi bukan-pohon menutup persis satu siklus bersama pohonnya. Keempat siklus itu adalah $E\,B\,D\,G\,E$, $C\,A\,B\,D\,G\,E\,C$, $H\,E\,C\,F\,H$, dan $I\,G\,E\,C\,F\,H\,I$ — masing-masing sebuah sisi mundur ditambah lintasan pohon di antara kedua ujungnya.

Kode 12.6 mencetak seluruh tabel peristiwa dan cap waktunya, lalu menjalankan pencarian sekali lagi dari E untuk uji diri di bawah.

Kode 12.6 Pencarian mendalam atas graf tak berarah, mencetak setiap penemuan, setiap penyelesaian, dan setiap sisi mundur menurut urutan jam, lalu tabel d/f, kedua golongan sisinya, dan struktur kurungnya; dijalankan dari A dan dari E. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o dfsu dfsu.cpp`.

```
// Persoalan Terkerjakan I: pencarian mendalam pada graf TAK BERARAH, dengan tabel
// waktu temu/selesai, setiap sisi digolongkan, dan struktur tanda kurungnya.
// Grafnya G1 dari Soal G: simpul A..I, daftar ketetanggaan alfabetis.
// Pada graf tak berarah hanya sisi pohon dan sisi mundur yang dapat muncul.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;

static const string NM = "ABCDEFGHI";
static const vector<pair<int,int>> EDGES = {
    {0,1},{0,2},{1,3},{1,4},{2,4},{2,5},{3,6},{4,6},{4,7},{5,7},{6,8},{7,8}
};

string stamp(int t) { string s = "t=" + to_string(t); return s + string(6 - s.size(), ' ');
}

void run(int start) {
    int n = (int)NM.size();
    vector<vector<int>> adj(n);
    for (auto [a, b] : EDGES) { adj[a].push_back(b); adj[b].push_back(a); }
    for (auto& l : adj) sort(l.begin(), l.end());
    vector<int> col(n, 0), d(n, 0), f(n, 0), pi(n, -1);      // 0 putih, 1 kelabu, 2 hitam
    vector<string> events;                                    // jejaknya, satu baris per
peristiwa
    vector<string> tree, back;
    int clk = 0;
    function<void(int)> visit = [&](int u) {
        col[u] = 1; d[u] = ++clk;
        events.push_back(stamp(clk) + "discover " + NM[u]);
        for (int v : adj[u]) {
            string e = string(1, NM[u]) + NM[v];
            if (col[v] == 0) {
                tree.push_back(e); pi[v] = u;
                visit(v);
            } else if (col[v] == 1 && v != pi[u]) {
                back.push_back(e);                            // kelabu dan bukan induknya
                events.push_back("         " + e + " finds " + NM[v] + " gray: back edge");
            }
            // induk kelabu: sisi pohon dilihat dari bawah; hitam: sisi mundur dilihat
            // dari ujung satunya, sudah digolongkan. Keduanya bukan sisi baru.
        }
        col[u] = 2; f[u] = ++clk;
        events.push_back(stamp(clk) + "finish   " + NM[u]);
    };
```

Kode 12.6 (Lanjutan) *Pencarian mendalam atas graf tak berarah, mencetak setiap penemuan, setiap penyelesaian, dan setiap sisi mundur menurut urutan jam, lalu tabel d/f, kedua golongan sisinya, dan struktur kurungnya; dijalankan dari A dan dari E. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o dfsu dfsu.cpp.*

```
    visit(start);
    for (int v = 0; v < n; ++v) if (col[v] == 0) visit(v);
    cout << "=== DFS of G1 starting at " << NM[start] << " ===\n";
    for (const string& s : events) cout << "  " << s << "\n";
    cout << "  v     "; for (int v = 0; v < n; ++v) cout << setw(6) << NM[v]; cout << "\n";
    cout << "  d/f   "; for (int v = 0; v < n; ++v) cout << setw(6) << (to_string(d[v]) +
"/" + to_string(f[v])); cout << "\n";
    cout << "  pi    "; for (int v = 0; v < n; ++v) cout << setw(6) << (pi[v] < 0 ?
string("-") : string(1, NM[pi[v]])); cout << "\n";
    cout << "  tree edges (" << tree.size() << "):"; for (auto& e : tree) cout << " " << e;
cout << "\n";
    cout << "  back edges (" << back.size() << "):"; for (auto& e : back) cout << " " << e;
cout << "\n";
    vector<pair<int,int>> ev;
    for (int v = 0; v < n; ++v) { ev.push_back({d[v], v}); ev.push_back({f[v], ~v}); }
    sort(ev.begin(), ev.end());
    cout << "  parentheses: ";
    for (auto [t, x] : ev) cout << (x >= 0 ? "(" + string(1, NM[x]) : string(1, NM[~x]) +
")") << " ";
    cout << "\n";
}

int main() { run(0); cout << "\n"; run(4); return 0; }
```

```
=== DFS of G1 starting at A ===
  t=1   discover A
  t=2   discover B
  t=3   discover D
  t=4   discover G
  t=5   discover E
           EB finds B gray: back edge
  t=6   discover C
           CA finds A gray: back edge
  t=7   discover F
  t=8   discover H
           HE finds E gray: back edge
  t=9   discover I
           IG finds G gray: back edge
  t=10  finish   I
  t=11  finish   H
  t=12  finish   F
  t=13  finish   C
  t=14  finish   E
  t=15  finish   G
  t=16  finish   D
  t=17  finish   B
  t=18  finish   A
  v          A     B     C     D     E     F     G     H     I
  d/f     1/18  2/17  6/13  3/16  5/14  7/12  4/15  8/11  9/10
  pi         -     A     E     B     G     C     D     F     H
  tree edges (8): AB BD DG GE EC CF FH HI
  back edges (4): EB CA HE IG
  parentheses: (A (B (D (G (E (C (F (H (I I) H) F) C) E) G) D) B) A)

=== DFS of G1 starting at E ===
  t=1   discover E
  t=2   discover B
  t=3   discover A
  t=4   discover C
           CE finds E gray: back edge
  t=5   discover F
  t=6   discover H
           HE finds E gray: back edge
  t=7   discover I
  t=8   discover G
  t=9   discover D
           DB finds B gray: back edge
  t=10  finish   D
           GE finds E gray: back edge
  t=11  finish   G
  t=12  finish   I
  t=13  finish   H
  t=14  finish   F
  t=15  finish   C
  t=16  finish   A
  t=17  finish   B
  t=18  finish   E
  v          A     B     C     D     E     F     G     H     I
  d/f     3/16  2/17  4/15  9/10  1/18  5/14  8/11  6/13  7/12
  pi         B     E     A     G     -     C     I     F     H
  tree edges (8): EB BA AC CF FH HI IG GD
  back edges (4): CE HE DB GE
  parentheses: (E (B (A (C (F (H (I (G (D D) G) I) H) F) C) A) B) E)
```

Jawaban. d/f dan π seperti pada **Tabel 12.25**; delapan sisi pohon dan empat sisi mundur, tanpa sisi maju atau silang; empat sisi mundur untuk peringkat sirkuit empat.

> **UJI DIRI.** Mulailah pencariannya di E alih-alih A. Berikan cap d/f dan sisi-sisi mundurnya.

Jawaban. E 1/18, B 2/17, A 3/16, C 4/15, F 5/14, H 6/13, I 7/12, G 8/11, D 9/10. Pohonnya lagi-lagi satu lintasan tunggal, $E\ B\ A\ C\ F\ H\ I\ G\ D$, dan sisi-sisi mundurnya adalah CE, HE, DB, dan GE — empat yang berbeda, tetapi tetap empat, sebagaimana dituntut peringkat sirkuitnya. Separuh kedua keluaran **Kode 12.6** membenarkan setiap nilainya.

12.4.5 Persoalan Terkerjakan J — komponen terhubung

Persoalan. G_3 adalah graf tak berarah atas dua belas simpul 1, … ,12, dengan kesebelas sisi

$$\{1,2\},\{2,3\},\{3,1\},\quad \{4,5\},\{4,6\},\{4,7\},\quad \{8,9\},\{9,10\},\{10,11\},\{11,12\},\{12,8\}$$

(a) Hitunglah komponen terhubungnya dan berikan label komponen setiap simpul. (b) Jawablah: apakah 1 dan 3 terhubung? 1 dan 4? 5 dan 7? 6 dan 11? 8 dan 12? (c) Sebuah hutan atas 12 simpul dengan k komponen punya berapa sisi? Apa yang diberitahukan perbandingannya dengan 11?

Yang diuji. Bahwa "apakah u terhubung ke v?" semestinya diubah dari sebuah pencarian menjadi sebuah pembacaan larik (§10.9.1).

Penyelesaian.

(a) Komponennya. Jalankan pencarian melebar dari setiap simpul yang belum berlabel, dan berikan label baru kepada tiap pencarian.

- Mulai di 1. Mencapai 2 dan 3 (segitiganya). Komponen **0** $=\{1,2,3\}$.
- 4 belum berlabel. Mulai di sana. Mencapai 5, 6, 7 (bintangnya). Komponen **1** $=\{4,5,6,7\}$.
- 8 belum berlabel. Mulai di sana. Mencapai 9, 10, 11, 12 mengelilingi siklus-lima. Komponen **2** $=\{8,9,10,11,12\}$.

Tabel 12.26 Label komponen setiap simpul G_3. Tiga komponen. Begitu larik ini ada, keterhubungan menjadi satu perbandingan bilangan bulat tunggal, dan itulah sebabnya lariknya — dan bukan pencariannya — yang menjadi jawabannya.

Simpul	1	2	3	4	5	6	7	8	9	10	11	12
komponen	0	0	0	1	1	1	1	2	2	2	2	2

Ongkos totalnya $\Theta(V+E)$ untuk *seluruh* lariknya, bukan per kueri, sebab pencarian-pencariannya mempartisi himpunan sisinya: tak ada sisi yang dipindai dua pencarian yang berbeda.

(b) Kuerinya. Bacalah langsung dari **Tabel 12.26**. 1 dan 3: keduanya berlabel 0, **ya**. 1 dan 4: 0 lawan 1, **tidak**. 5 dan 7: keduanya 1, **ya**. 6 dan 11: 1 lawan 2, **tidak**. 8 dan 12: keduanya 2, **ya**.

(c) Cacah sisinya. Sebuah hutan atas n simpul dengan k komponen punya persis $n-k$ sisi, sebab tiap komponennya sebuah pohon dengan (cacah simpulnya sendiri -1) sisi, dan cacah-cacahnya berjumlah. Di sini $12-3=9$. Grafnya punya **11** sisi, yaitu 2 lebih banyak daripada yang diperlukan sebuah hutan. Jadi grafnya memuat persis $11-9=2$ siklus bebas — dan memang kita dapat melihat keduanya: segitiga 1–2–3 dan siklus-lima 8–9–10–11–12. Bintangnya sebuah pohon dan tidak menyumbang satu pun.

WAWASAN KUNCI: RUMUS CACAH SIKLUS, DITURUNKAN DALAM SATU BARIS. Untuk graf tak berarah mana pun dengan k komponen, banyaknya siklus bebas — yaitu **peringkat sirkuit**-nya — adalah

$$r(G)=|E|-|V|+k.$$

(12.14)

Ambillah sebuah hutan merentang mana pun: ia punya $|V| - k$ sisi dan tanpa siklus. Setiap satu dari $|E| - (|V| - k)$ sisi yang tersisa menutup persis satu siklus baru ketika dikembalikan. Menurut Persamaan (12.14), $r(G_3) = 11 - 12 + 3 = 2$.

Jawaban. Komponennya {1,2,3}, {4,5,6,7}, {8, … ,12}; kuerinya menjawab ya, tidak, ya, tidak, ya; sebuah hutan akan punya 9 sisi, jadi ada 2 siklus bebas.

UJI DIRI. Tambahkan sisi {3,4} ke G_3. Ada berapa komponen sekarang, dan berapa siklus bebas?

Jawaban. Dua komponen, {1, … ,7} dan {8, … ,12}. Menurut Persamaan (12.14), $r = 12 - 12 + 2 = 2$ — **tidak berubah**. Sisi baru itu menyatukan dua komponen alih-alih menutup sebuah siklus: $|E|$ dan k bergerak bersama. Sisi yang penghapusannya memutus komponennya disebut *jembatan*, dan {3,4} adalah salah satunya.

12.4.6 Persoalan Terkerjakan K — kebipartitan, dan sertifikatnya

Persoalan. (a) Putuskan apakah G_1 bipartit. Jika ya, berikan kedua sisinya. (b) Sekarang tambahkan sisi tunggal EF ke G_1. Putuskan lagi, dan jika jawabannya berubah, perlihatkan sebuah siklus ganjil sebagai sertifikatnya — baik yang akan dilaporkan pewarnaan dua warna melebar maupun yang terpendek di dalam grafnya.

Yang diuji. Bahwa bipartit berarti *dapat diwarnai dua*, bahwa ujinya satu baris di dalam sebuah pencarian melebar, dan bahwa jawaban **negatif** harus disertai sertifikat (§10.9.3).

Penyelesaian.

(a) Ujinya pada G_1. Warnai A dengan 0 dan berikan kepada tiap simpul warna yang berlawanan dengan warna penemunya. Dari **Tabel 12.22** jaraknya adalah A: 0, B: 1, C: 1, D: 2, E: 2, F: 2, G: 3, H: 3, I: 4, jadi warnanya sekadar $d[v]$ mod 2:

- warna 0 (jarak genap): A, D, E, F, I;
- warna 1 (jarak ganjil): B, C, G, H.

Sekarang periksalah kedua belas sisinya, dan bagian inilah yang tidak boleh dilewati: $AB(0,1)$ ✓, $AC(0,1)$ ✓, $BD(1,0)$ ✓, $BE(1,0)$ ✓, $CE(1,0)$ ✓, $CF(1,0)$ ✓, $DG(0,1)$ ✓, $EG(0,1)$ ✓, $EH(0,1)$ ✓, $FH(0,1)$ ✓, $GI(1,0)$ ✓, $HI(1,0)$ ✓. Setiap sisi menghubungkan kedua pihaknya, jadi G_1 **bipartit**, dengan pihak-pihak $\{A, D, E, F, I\}$ dan $\{B, C, G, H\}$.

(b) Menambahkan EF. E dan F sama-sama berwarna 0 — keduanya duduk pada jarak 2 dari A. Sisi barunya menghubungkan dua simpul pada pihak yang sama, jadi tak ada pewarnaan dua warna yang selamat dan grafnya **tidak bipartit**.

Sertifikatnya adalah sebuah siklus ganjil, dan ada dua yang layak diberikan. Pewarnaan dua warna melebar melaporkan siklus yang dibangun dari kedua lintasan pohon menuju ujung-ujung sisi yang melanggar: dari E naik ke B ke A, turun ke C, turun ke F, lalu kembali lewat sisi barunya —

$$E - B - A - C - F - E, \qquad \text{panjang } 5.$$

Siklus ganjil *terpendek* di dalam grafnya lebih pendek: E dan F sama-sama bertetangga dengan C, jadi

$$E - C - F - E, \qquad \text{panjang } 3.$$

Keduanya jawaban yang lengkap, sebab teoremanya berbunyi *sebuah graf bipartit bila dan hanya bila ia tidak memuat siklus ganjil*; memperlihatkan satu siklus ganjil dengan panjang berapa pun sudah menyelesaikannya. Siklus-lima adalah sertifikat yang sepenuhnya sah dan itulah yang secara alami dihasilkan algoritmanya, sebab BFS mengenal pohonnya, bukan seluruh grafnya.

Mengapa siklus ganjil persis merupakan penghalangnya. Berjalanlah mengelilingi sebuah siklus sambil menyelang-nyeling warnanya. Sesudah langkah sebanyak bilangan genap Anda kembali ke warna yang Anda mulai; sesudah sebanyak bilangan ganjil, tidak. Siklus berpanjang ganjil karena itu memaksa simpul pertamanya menjadi kedua warna sekaligus. Sebaliknya, jika tak ada siklus ganjil, mewarnai menurut keparian jarak BFS berhasil: menurut Persamaan (12.4) setiap sisi menghubungkan simpul-simpul yang jaraknya berselisih paling banyak satu, dan ia tidak dapat menghubungkan dua simpul berjarak *sama*, sebab kalau begitu kedua lintasan pohonnya beserta sisi itu akan membentuk siklus ganjil.

Jawaban. G_1 bipartit dengan pihak-pihak $\{A, D, E, F, I\}$ dan $\{B, C, G, H\}$; dengan EF ditambahkan ia tidak bipartit, disertifikasi oleh $E\ B\ A\ C\ F\ E$ (panjang 5) atau $E\ C\ F\ E$ (panjang 3).

> **UJI DIRI.** Alih-alih EF, tambahkan (a) sisi AI; (b) sisi AH. Apakah G_1 masih bipartit pada masing-masing kasus?
>
> *Jawaban.* (a) **Tidak.** $d[A] = 0$ dan $d[I] = 4$ sama-sama genap, jadi A dan I berwarna sama; sebuah siklus ganjilnya adalah $A\ B\ D\ G\ I\ A$, sepanjang 5. (b) **Ya.** $d[H] = 3$ ganjil, jadi A dan H berada di pihak yang berlawanan dan kedua pihak yang sama tetap berlaku.

12.4.7 Persoalan Terkerjakan L — pendeteksian siklus, tak berarah dan berarah

Persoalan. (a) Nyatakan aturan pendeteksian siklus pada graf *tak berarah* lewat pencarian mendalam, dan jelaskan persis mengapa "saya sudah pernah melihat simpul ini" bukanlah aturannya. (b) Terapkan aturan Anda pada lintasan A–B–C–D, pada persegi A–B–C–D–A, dan pada G_1. (c) Nyatakan aturan untuk graf *berarah* lalu terapkan pada G_2 dan pada digraf $A \to B$, $A \to C$, $B \to C$.

Yang diuji. Kedua uji siklus §10.9.2 dan §11.2, dan beda di antara keduanya.

Penyelesaian.

(a) Aturan tak berarahnya, dan mengapa yang kentara itu keliru. Aturan yang kentara — *"selama pencarian mendalam, jika Anda mencapai simpul yang sudah pernah Anda lihat, ada siklus"* — **keliru untuk graf tak berarah**, dan contoh sanggahan terkecilnya punya dua simpul. Pada graf A–B, pencariannya mengunjungi A, berjalan lewat sisinya ke B, lalu memeriksa $\text{Adj}[B] = [A]$ dan melihat A yang sudah kelabu. Tetapi tak ada siklus: algoritmanya sekadar menoleh kembali ke sisi yang baru saja dilaluinya masuk. Pada graf tak berarah setiap sisi muncul di *kedua* daftar ketetanggaan, jadi hal ini terjadi di setiap simpul tanpa kecuali.

Aturan yang benar adalah:

> **UJI SIKLUS TAK BERARAH.** Selama pencarian mendalam dari u dengan induk p, periksalah tiap $v \in \text{Adj}[u]$. Lewatilah **persis satu** kemunculan p — yaitu sisi tempat Anda masuk. Jika ditemukan simpul bukan-putih yang lain, ada siklus.

Kata "persis satu" itu penting. Jika grafnya punya dua sisi sejajar antara A dan B, melewati *setiap* kemunculan induknya akan melewatkan siklus yang dibentuk kedua sisi itu. Melewati yang pertama lalu menguji sisanya menangkapnya.

(b) Diterapkan.

- **Lintasan** A–B–C–D**.** Kunjungi A, B (induk A), C (induk B), D (induk C). Pada tiap langkah satu-satunya tetangga bukan-putihnya adalah induknya, yang dilewati sekali. Tak ada siklus: **taksiklik**, dan dengan $|E| = 3 = |V| - 1$ serta satu komponen ia sebuah pohon.
- **Persegi** A–B–C–D–A**.** Kunjungi A, B, C, D; $\mathrm{Adj}[D] = [A, C]$; C adalah induknya, dilewati; A kelabu dan bukan induknya. **Siklus ditemukan.** Di sini $|E| = 4 > 3 = |V| - 1$, jadi Persamaan (12.14) sudah menjamin adanya satu.
- G_1**.** $|V| = 9$, $|E| = 12$, satu komponen, jadi peringkat sirkuitnya $12 - 9 + 1 = 4$: ada empat siklus bebas, dan Persoalan I menemukan keempat sisi mundurnya. **Punya siklus.**

(c) Aturan berarahnya. Pada graf berarah, warna ujung jauhnya adalah seluruh ujinya, dan **hanya kelabu yang berarti**:

> **UJI SIKLUS BERARAH.** Busur $u \to v$ yang diperiksa ketika v **kelabu** adalah sisi mundur, dan sebuah graf berarah punya siklus berarah bila dan hanya bila pencarian mendalam atasnya menghasilkan sekurang-kurangnya satu sisi mundur (§11.2.3). v yang **hitam** berarti seluruh pencarian v sudah selesai sebelum busur ini diperiksa, jadi v tidak dapat mencapai u, dan busurnya tidak menutup apa pun.

Diterapkan pada G_2: **Tabel 12.24** menemukan dua sisi mundur, $D \to A$ dan $F \to E$, jadi G_2 **punya siklus berarah** — yaitu $A \to B \to D \to A$ dan $E \to F \to E$. Diterapkan pada $A \to B$, $A \to C$, $B \to C$: mengunjungi A lalu B lalu C, busur $B \to C$ adalah sisi pohon dan $A \to C$ mendapati C hitam dengan $d[A] < d[C]$ — sebuah sisi *maju*, bukan sisi mundur. Nol sisi mundur, jadi yang ini **sebuah DAG**, sekalipun C "terlihat dua kali".

> **PERINGATAN: JEBAKANNYA.** Mengenakan uji *berarah* (tetangga bukan-putih mana pun berarti siklus) pada graf tak berarah akan melaporkan siklus pada setiap graf yang punya sebuah sisi. Mengenakan uji *tak berarah* (lewati induknya) pada graf berarah tak bermakna, sebab graf berarah tak punya "sisi tempat kita masuk" untuk dilewati. Kedua ujinya tidak dapat saling ditukar.

Jawaban. Tak berarah: lewati induknya persis sekali, lalu tetangga bukan-putih mana pun berarti siklus — lintasannya taksiklik, persegi dan G_1 tidak. Berarah: tetangga kelabu berarti siklus — G_2 punya dua, dan digraf tiga busur itu sebuah DAG.

> **UJI DIRI.** Putuskan masing-masingnya: (a) graf tak berarah dengan sisi AB, BC, CA, CD; (b) digraf $A \to B, B \to C, C \to D, D \to B$; (c) digraf $A \to B, A \to C, B \to D, C \to D$.
>
> *Jawaban.* (a) **Siklus**: di C (induk B), tetangga A kelabu dan bukan induknya. (b) **Siklus**: pencariannya berjalan A, B, C, D, dan di D busur $D \to B$ mendapati B kelabu — siklus $B \to C \to D \to B$. (c) **DAG**: D dicapai dari B sebagai sisi pohon, lalu dari C ia hitam dengan $d[C] > d[D]$, sebuah sisi silang; tak ada busur yang pernah bertemu simpul kelabu.

12.4.8 Persoalan G, H, J, K, dan L sebagai satu program

Kode 12.7 menjalankan Persoalan G, H, J, K, dan L. Cetakannya lengkap; keluaran di bawahnya adalah seluruh yang dicetaknya.

Kode 12.7 Persoalan Terkerjakan G, H, J, K, dan L, dihitung: penelusuran BFS, cap waktu dan penggolongan DFS, komponennya, pewarnaan dua warna beserta sertifikat siklus ganjilnya, dan kedua uji siklus. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o search search.cpp`.

```
// Persoalan Terkerjakan G, H, J, K, dan L Bab 12 -- menelusuri sebuah graf.
//   G  pencarian melebar: jejaknya, d, pi, dan pohonnya
//   H  pencarian mendalam: cap waktu, hutan, dan setiap busur digolongkan
//   J  komponen terhubung
//   K  kebipartitan, dan siklus ganjil yang meruntuhkannya
//   L  pendeteksian siklus, tak berarah dan berarah
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
#include <functional>
using namespace std;

static const string UN = "ABCDEFGHI";              // graf tak berarah G1 milik G dan K
static const vector<pair<int,int>> UE = {
    {0,1},{0,2},{1,3},{1,4},{2,4},{2,5},{3,6},{4,6},{4,7},{5,7},{6,8},{7,8}
};

// ---------------------------------------------------------------- G
void q1() {
    cout << "=== Problem G  breadth-first search from A ===\n";
    int n = (int)UN.size();
    vector<vector<int>> adj(n);
    for (auto [a,b] : UE) { adj[a].push_back(b); adj[b].push_back(a); }
    for (auto& l : adj) sort(l.begin(), l.end());          // alfabetis, sesuai bunyi soal
    vector<int> d(n, INT_MAX), pi(n, -1);
    d[0] = 0; queue<int> Q; Q.push(0);
    cout << "step  dequeue  discovered            queue after\n";
    int step = 0;
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        string disc;
        for (int v : adj[u]) if (d[v] == INT_MAX) {
            d[v] = d[u] + 1; pi[v] = u; Q.push(v);
            disc += string(1, UN[v]) + ":" + to_string(d[v]) + " ";
        }
        queue<int> tmp = Q; string qs;
        while (!tmp.empty()) { qs += string(1, UN[tmp.front()]) + ":" +
to_string(d[tmp.front()]) + " "; tmp.pop(); }
        cout << setw(4) << ++step << setw(9) << UN[u] << "  " << left << setw(22)
             << (disc.empty() ? "-" : disc) << right << (qs.empty() ? "(empty)" : qs) <<
"\n";
    }
    cout << "v      "; for (int v = 0; v < n; ++v) cout << setw(4) << UN[v]; cout << "\n";
    cout << "d[v]   "; for (int v = 0; v < n; ++v) cout << setw(4) << d[v];  cout << "\n";
```

Kode 12.7 (Lanjutan) *Persoalan Terkerjakan G, H, J, K, dan L, dihitung: penelusuran BFS, cap waktu dan penggolongan DFS, komponennya, pewarnaan dua warna beserta sertifikat siklus ganjilnya, dan kedua uji siklus. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o search search.cpp.*

```
    cout << "pi[v]  "; for (int v = 0; v < n; ++v) cout << setw(4) << (pi[v] < 0 ?
string("-") : string(1, UN[pi[v]])); cout << "\n";
    int tree = 0; for (int v = 0; v < n; ++v) if (pi[v] != -1) ++tree;
    cout << "tree edges: " << tree << " = |V| - 1 = " << n - 1 << "\n";
}

// ---------------------------------------------------------------- H
static const string DN = "ABCDEFG";                 // graf berarah G2 milik H dan L
static const vector<pair<int,int>> DE = {
    {0,1},{0,2},{0,3},{1,3},{2,3},{2,4},{3,0},{4,5},{5,4},{6,2},{6,5}
};
void q2() {
    cout << "\n=== Problem H  depth-first search, all seven vertices ===\n";
    int n = (int)DN.size();
    vector<vector<int>> adj(n);
    for (auto [u,v] : DE) adj[u].push_back(v);
    for (auto& l : adj) sort(l.begin(), l.end());
    vector<int> color(n, 0), d(n, 0), f(n, 0), pi(n, -1);   // 0 putih 1 kelabu 2 hitam
    vector<string> cls(DE.size());
    int clock_ = 0;
    function<void(int)> visit = [&](int u) {
        color[u] = 1; d[u] = ++clock_;
        for (int v : adj[u]) {
            int idx = -1;
            for (size_t i = 0; i < DE.size(); ++i) if (DE[i].first == u && DE[i].second ==
v) idx = (int)i;
            if (color[v] == 0)      { cls[idx] = "tree";    pi[v] = u; visit(v); }
            else if (color[v] == 1) { cls[idx] = "back"; }
            else                     { cls[idx] = (d[u] < d[v] ? "forward" : "cross"); }
        }
        color[u] = 2; f[u] = ++clock_;
    };
    for (int v = 0; v < n; ++v) if (color[v] == 0) visit(v);
    cout << "v       "; for (int v = 0; v < n; ++v) cout << setw(6) << DN[v]; cout << "\n";
    cout << "d/f     "; for (int v = 0; v < n; ++v) cout << setw(6) << (to_string(d[v]) +
"/" + to_string(f[v])); cout << "\n";
    cout << "pi      "; for (int v = 0; v < n; ++v) cout << setw(6) << (pi[v] < 0 ?
string("-") : string(1, DN[pi[v]])); cout << "\n";
    cout << "clock stopped at " << clock_ << " = 2|V| = " << 2 * n << "\n";
    int t = 0, b = 0, fw = 0, c = 0;
    cout << "arc     class\n";
    for (size_t i = 0; i < DE.size(); ++i) {
        cout << " " << DN[DE[i].first] << "->" << DN[DE[i].second] << "   " << cls[i] <<
"\n";
        if (cls[i] == "tree") ++t; else if (cls[i] == "back") ++b;
        else if (cls[i] == "forward") ++fw; else ++c;
    }
    cout << "tree " << t << ", back " << b << ", forward " << fw << ", cross " << c
         << "  (total " << DE.size() << ")\n";
    int roots = 0; for (int v = 0; v < n; ++v) if (pi[v] == -1) ++roots;
    cout << "the forest has " << roots << " trees, so " << n << " - " << roots
```

Kode 12.7 (Lanjutan) *Persoalan Terkerjakan G, H, J, K, dan L, dihitung: penelusuran BFS, cap waktu dan penggolongan DFS, komponennya, pewarnaan dua warna beserta sertifikat siklus ganjilnya, dan kedua uji siklus. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o search search.cpp.*

```
        << " = " << n - roots << " tree edges\n";
    vector<pair<int,int>> ev;                                   // pengurungannya
    for (int v = 0; v < n; ++v) { ev.push_back({d[v], v}); ev.push_back({f[v], ~v}); }
    sort(ev.begin(), ev.end());
    cout << "parenthesis structure: ";
    for (auto [time, x] : ev) cout << (x >= 0 ? "(" + string(1, DN[x]) : string(1, DN[~x])
+ ")") << " ";
    cout << "\n";
    cout << "the graph has " << b << " back edge(s), so it is "
         << (b ? "NOT acyclic" : "acyclic") << "\n";
}

// ---------------------------------------------------------------- J
void q3() {
    cout << "\n=== Problem J  connected components ===\n";
    int n = 12;
    vector<pair<int,int>> E = {
        {0,1},{1,2},{2,0},                                      // sebuah segitiga
        {3,4},{3,5},{3,6},                                      // sebuah bintang
        {7,8},{8,9},{9,10},{10,11},{11,7}                       // sebuah siklus-lima
    };
    vector<vector<int>> adj(n);
    for (auto [a,b] : E) { adj[a].push_back(b); adj[b].push_back(a); }
    vector<int> comp(n, -1); int nc = 0;
    for (int v = 0; v < n; ++v) if (comp[v] == -1) {
        queue<int> Q; Q.push(v); comp[v] = nc;
        while (!Q.empty()) { int u = Q.front(); Q.pop();
            for (int w : adj[u]) if (comp[w] == -1) { comp[w] = nc; Q.push(w); } }
        ++nc;
    }
    cout << "vertex  "; for (int v = 0; v < n; ++v) cout << setw(4) << v + 1; cout << "\n";
    cout << "comp    "; for (int v = 0; v < n; ++v) cout << setw(4) << comp[v]; cout <<
"\n";
    cout << "number of components: " << nc << "\n";
    auto same = [&](int a, int b) { return comp[a-1] == comp[b-1] ? "yes" : "no"; };
    cout << "  1 and 3  connected? " << same(1,3)  << "\n";
    cout << "  1 and 4  connected? " << same(1,4)  << "\n";
    cout << "  5 and 7  connected? " << same(5,7)  << "\n";
    cout << "  6 and 11 connected? " << same(6,11) << "\n";
    cout << "  8 and 12 connected? " << same(8,12) << "\n";
    cout << "edges " << E.size() << ", vertices " << n << ", components " << nc
         << "; a forest would need " << n - nc << " edges, this has " << E.size() << "\n";
}
```

Kode 12.7 (Lanjutan) *Persoalan Terkerjakan G, H, J, K, dan L, dihitung: penelusuran BFS, cap waktu dan penggolongan DFS, komponennya, pewarnaan dua warna beserta sertifikat siklus ganjilnya, dan kedua uji siklus. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o search search.cpp.*

```
// ---------------------------------------------------------------- K
void q4() {
    cout << "\n=== Problem K  bipartiteness ===\n";
    auto test = [](int n, vector<pair<int,int>> E, const string& name, const string& lbl) {
        vector<vector<int>> adj(n);
        for (auto [a,b] : E) { adj[a].push_back(b); adj[b].push_back(a); }
        for (auto& l : adj) sort(l.begin(), l.end());
        vector<int> col(n, -1), pi(n, -1);
        cout << name << ": ";
        for (int s = 0; s < n; ++s) if (col[s] == -1) {
            col[s] = 0; queue<int> Q; Q.push(s);
            while (!Q.empty()) { int u = Q.front(); Q.pop();
                for (int v : adj[u]) {
                    if (col[v] == -1) { col[v] = 1 - col[u]; pi[v] = u; Q.push(v); }
                    else if (col[v] == col[u]) {
                        // telusuri kedua ujung sampai titik temunya: siklus ganjilnya
                        vector<int> pu, pv; for (int x = u; x != -1; x = pi[x])
pu.push_back(x);
                        for (int x = v; x != -1; x = pi[x]) pv.push_back(x);
                        reverse(pu.begin(), pu.end()); reverse(pv.begin(), pv.end());
                        size_t k = 0; while (k < pu.size() && k < pv.size() && pu[k] ==
pv[k]) ++k;
                        string cyc; for (size_t i = pu.size(); i-- >= k && i < pu.size();)
cyc += lbl[pu[i]];
                        for (size_t i = k; i < pv.size(); ++i) cyc += lbl[pv[i]];
                        cyc += lbl[u];
                        cout << "NOT bipartite; the edge " << lbl[u] << "-" << lbl[v]
                             << " joins two vertices of the same color. Odd cycle: " << cyc
<< "\n";
                        return;
                    }
                }
            }
        }
        string s0, s1;
        for (int v = 0; v < n; ++v) (col[v] == 0 ? s0 : s1) += lbl[v];
        cout << "BIPARTITE; sides {" << s0 << "} and {" << s1 << "}\n";
    };
    test(9, UE, "G1", UN);
    vector<pair<int,int>> plus = UE; plus.push_back({4,5});     // tambahkan E-F
    test(9, plus, "G1 plus the edge E-F", UN);
}
```

Kode 12.7 (Lanjutan) *Persoalan Terkerjakan G, H, J, K, dan L, dihitung: penelusuran BFS, cap waktu dan penggolongan DFS, komponennya, pewarnaan dua warna beserta sertifikat siklus ganjilnya, dan kedua uji siklus. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o search search.cpp.*

```
// ---------------------------------------------------------------- L
void q5() {
    cout << "\n=== Problem L  cycle detection ===\n";
    // tak berarah: jebakan induknya
    auto undirected = [](int n, vector<pair<int,int>> E, const string& lbl) {
        vector<vector<int>> adj(n);
        for (auto [a,b] : E) { adj[a].push_back(b); adj[b].push_back(a); }
        vector<int> col(n, 0);
        function<bool(int,int)> go = [&](int u, int parent) {
            col[u] = 1;
            for (int v : adj[u]) {
                if (v == parent) { parent = -2; continue; }   // lewati sisi tempat kita
masuk, SEKALI
                if (col[v]) return true;
                if (go(v, u)) return true;
            }
            col[u] = 2; return false;
        };
        bool cyc = false;
        for (int v = 0; v < n; ++v) if (!col[v] && go(v, -1)) { cyc = true; break; }
        cout << "  " << lbl << ": " << (cyc ? "has a cycle" : "acyclic (a forest)") <<
"\n";
    };
    undirected(4, {{0,1},{1,2},{2,3}}, "path A-B-C-D          ");
    undirected(4, {{0,1},{1,2},{2,3},{3,0}}, "square A-B-C-D-A      ");
    undirected(9, UE, "G1                        ");
    undirected(2, {{0,1},{0,1}}, "two parallel edges A=B");
    // berarah: hanya KELABU yang berarti
    auto directed = [](int n, vector<pair<int,int>> E, const string& lbl) {
        vector<vector<int>> adj(n);
        for (auto [u,v] : E) adj[u].push_back(v);
        vector<int> col(n, 0);
        function<bool(int)> go = [&](int u) {
            col[u] = 1;
            for (int v : adj[u]) { if (col[v] == 1) return true;
                                   if (col[v] == 0 && go(v)) return true; }
            col[u] = 2; return false;
        };
        bool cyc = false;
        for (int v = 0; v < n; ++v) if (!col[v] && go(v)) { cyc = true; break; }
        cout << "  " << lbl << ": " << (cyc ? "has a directed cycle" : "is a DAG") << "\n";
    };
    directed(7, DE, "G2                      ");
    directed(3, {{0,1},{0,2},{1,2}}, "A->B, A->C, B->C       ");
    directed(3, {{0,1},{1,2},{2,0}}, "A->B, B->C, C->A       ");
}
```

Kode 12.7 (Lanjutan) *Persoalan Terkerjakan G, H, J, K, dan L, dihitung: penelusuran BFS, cap waktu dan penggolongan DFS, komponennya, pewarnaan dua warna beserta sertifikat siklus ganjilnya, dan kedua uji siklus. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o search search.cpp.*

```
int main() { q1(); q2(); q3(); q4(); q5(); return 0; }
```

```
=== Problem G  breadth-first search from A ===
step  dequeue  discovered            queue after
   1        A  B:1 C:1               B:1 C:1
   2        B  D:2 E:2               C:1 D:2 E:2
   3        C  F:2                   D:2 E:2 F:2
   4        D  G:3                   E:2 F:2 G:3
   5        E  H:3                   F:2 G:3 H:3
   6        F  -                     G:3 H:3
   7        G  I:4                   H:3 I:4
   8        H  -                     I:4
   9        I  -                     (empty)
v         A   B   C   D   E   F   G   H   I
d[v]      0   1   1   2   2   2   3   3   4
pi[v]     -   A   A   B   B   C   D   E   G
tree edges: 8 = |V| - 1 = 8

=== Problem H  depth-first search, all seven vertices ===
v            A     B     C     D     E     F     G
d/f       1/12   2/5  6/11   3/4  7/10   8/9 13/14
pi           -     A     A     B     C     E     -
clock stopped at 14 = 2|V| = 14
arc     class
 A->B   tree
 A->C   tree
 A->D   forward
 B->D   tree
 C->D   cross
 C->E   tree
 D->A   back
 E->F   tree
 F->E   back
 G->C   cross
 G->F   cross
tree 5, back 2, forward 1, cross 3  (total 11)
the forest has 2 trees, so 7 - 2 = 5 tree edges
parenthesis structure: (A (B (D D) B) (C (E (F F) E) C) A) (G G)
the graph has 2 back edge(s), so it is NOT acyclic

=== Problem J  connected components ===
vertex     1   2   3   4   5   6   7   8   9  10  11  12
comp       0   0   0   1   1   1   1   2   2   2   2   2
number of components: 3
  1 and 3  connected? yes
  1 and 4  connected? no
  5 and 7  connected? yes
  6 and 11 connected? no
  8 and 12 connected? yes
edges 11, vertices 12, components 3; a forest would need 9 edges, this has 11

=== Problem K  bipartiteness ===
G1: BIPARTITE; sides {ADEFI} and {BCGH}
G1 plus the edge E-F: NOT bipartite; the edge E-F joins two vertices of the same color. Odd
cycle: EBACFE

=== Problem L  cycle detection ===
  path A-B-C-D          : acyclic (a forest)
  square A-B-C-D-A      : has a cycle
  G1                    : has a cycle
  two parallel edges A=B: has a cycle
  G2                     : has a directed cycle
  A->B, A->C, B->C       : is a DAG
  A->B, B->C, C->A       : has a directed cycle
```

Setiap bilangan di Subbab 12.4.2 sampai 12.4.7 direproduksi di sana, termasuk siklus ganjil bersimpul lima `EBACFE` dan kasus sisi sejajar yang akan terlewat oleh aturan induk yang ceroboh.

12.5 Persoalan Terkerjakan Lanjutan III: DAG dan Pengurutan Topologis

12.5.1 DAG prasyarat subbab ini

Sebuah program studi punya delapan mata ajar. Anak panah $X \to Y$ berarti X *harus lulus sebelum* Y *boleh diambil*, dan tiap mata ajar memakan banyaknya pekan yang tertera pada **Tabel 12.27**.

Tabel 12.27 Kedelapan mata ajar digraf prasyarat yang dipakai dalam Persoalan M sampai Q, beserta durasi dan prasyaratnya. Sembilan busur seluruhnya: $A \to D, B \to D, B \to E, C \to G, D \to E, D \to F, E \to H, F \to H, G \to H$.

Kode	Mata ajar	Pekan	Prasyarat
A	Dasar Pemrograman	3	—
B	Matematika Diskret	2	—
C	Kalkulus	4	—
D	Struktur Data	5	A, B
E	Algoritma	6	B, D
F	Basis Data	3	D
G	Statistika	2	C
H	Pembelajaran Mesin	4	E, F, G

Daftar ketetanggaannya menurut abjad; gelung luar pencarian mendalam mana pun berjalan $A, B, \ldots, H$; antrean Kahn masuk-pertama-keluar-pertama dan diawali dengan simpul-simpul berderajat masuk nol menurut urutan abjad.

12.5.2 Persoalan Terkerjakan M — apakah ia DAG?

Persoalan. (a) Jalankan pencarian mendalam pada digraf **Tabel 12.27** dan berikan $d[v]/f[v]$ untuk kedelapan mata ajar. (b) Golongkan kesembilan busurnya. (c) Nyatakan, beserta teorema yang Anda pakai, apakah digraf itu DAG.

Yang diuji. Kriteria sisi mundur §11.2.3, dan kedisiplinan menuntaskan satu subpohon sepenuhnya sebelum memulai yang berikutnya.

Penyelesaian.

(a) Pencariannya. $\mathrm{Adj}[A] = [D]$, $\mathrm{Adj}[B] = [D, E]$, $\mathrm{Adj}[C] = [G]$, $\mathrm{Adj}[D] = [E, F]$, $\mathrm{Adj}[E] = [H]$, $\mathrm{Adj}[F] = [H]$, $\mathrm{Adj}[G] = [H]$, $\mathrm{Adj}[H] = [\,]$.

- A putih: kelabu, $d[A] = 1$. $\to D$ putih: kelabu, $d[D] = 2$.
- $\mathrm{Adj}[D]$, pertama E: putih, kelabu, $d[E] = 3$. $\to H$ putih: kelabu, $d[H] = 4$. H tak punya busur: hitam, $f[H] = 5$. Kembali di E: habis, hitam, $f[E] = 6$.
- $\mathrm{Adj}[D]$, berikutnya F: putih, kelabu, $d[F] = 7$. $\mathrm{Adj}[F] = [H]$: **hitam**, dan $d[F] = 7 > 4 = d[H]$, jadi **sisi silang**. F hitam, $f[F] = 8$.
- D habis: hitam, $f[D] = 9$. A habis: hitam, $f[A] = 10$.
- Gelung luar: B putih, kelabu, $d[B] = 11$. $\mathrm{Adj}[B] = [D, E]$: keduanya **hitam** dengan d lebih kecil, jadi keduanya **silang**. B hitam, $f[B] = 12$.
- C putih, kelabu, $d[C] = 13$. $\to G$ putih, kelabu, $d[G] = 14$. $\mathrm{Adj}[G] = [H]$: hitam, d lebih kecil, **silang**. G hitam, $f[G] = 15$. C hitam, $f[C] = 16$.
- D, E, F, G, H semuanya hitam; gelungnya berakhir. Jam $16 = 2|V|$. ✓

Tabel 12.28 Cap waktu pencarian mendalam pada DAG prasyarat. Tiga akar — A, B, dan C, ketiga mata ajar tanpa prasyarat — jadi hutannya punya tiga pohon dan $8 - 3 = 5$ sisi pohon.

v	A	B	C	D	E	F	G	H
d/f	1/10	11/12	13/16	2/9	3/6	7/8	14/15	4/5

(b) Penggolongannya. Dengan memakai cap waktu **Tabel 12.28**: pohon $A \to D, D \to E, D \to F, E \to H, C \to G$ (lima); silang $B \to D, B \to E, F \to H, G \to H$ (empat); maju tidak ada; **mundur tidak ada**.

(c) Vonisnya. Menurut teorema DAG (Teorema 11.1, §11.2.3), *sebuah graf berarah taksiklik bila dan hanya bila pencarian mendalam atasnya tidak menghasilkan sisi mundur.* Nol sisi mundur ditemukan, jadi digrafnya **sebuah DAG** — dan memang harus begitu, sebab kalau tidak program studinya takkan pernah dapat dirampungkan.

> **PERINGATAN: JEBAKANNYA.** Orang sering melabeli $B \to D$ sebagai sisi *maju* karena D "datang belakangan". Bukan: maju berarti ujung jauhnya adalah **keturunan simpul ini di dalam hutan mendalamnya**, dan D duduk di pohon A, bukan pohon B. Uji cap waktunya memutuskannya secara mekanis: $d[B] = 11 > 2 = d[D]$, dan keturunan selalu punya waktu temu yang *lebih besar*. Silang.

Jawaban. d/f seperti pada **Tabel 12.28**; lima busur pohon dan empat busur silang, tanpa busur mundur; digrafnya sebuah DAG.

> **UJI DIRI.** Apakah digrafnya masih DAG jika program studinya menambahkan (a) prasyarat $F \to E$? (b) prasyarat $H \to D$?
>
> *Jawaban.* (a) **Ya.** Siklus yang melewati busur baru itu memerlukan lintasan dari E kembali ke F, dan satu-satunya jalan E ke depan adalah $E \to H$, sebuah jalan buntu. (b) **Tidak.** $D \to E \to H \to D$ adalah siklus; pencarian mendalam akan memeriksa $H \to D$ selagi D kelabu (di dalam panggilan D sendiri, lewat E) dan melaporkan sebuah sisi mundur.

12.5.3 Persoalan Terkerjakan N — pengurutan topologis menurut waktu selesai

Persoalan. (a) Hasilkan sebuah urutan topologis DAG prasyarat dengan mendaftar simpul-simpulnya menurut urutan waktu selesai yang menurun, dengan memperlihatkan waktu selesainya. (b) Periksalah jawaban Anda dengan memastikan bahwa setiap busur menunjuk ke depan. (c) Mengapa algoritmanya memakai f dan bukan d? Berikan contoh dua simpul yang pada contoh itu mengurutkan menurut d akan gagal.

Yang diuji. Algoritma 1 §11.3.4 dan alasan mengapa ia benar (Teorema 11.2).

Penyelesaian.

(a) Urutannya. Urutkan menurut f menurun: $C(16)$, $G(15)$, $B(12)$, $A(10)$, $D(9)$, $F(8)$, $E(6)$, $H(5)$. Jadi

$$\mathrm{C} \prec \mathrm{G} \prec \mathrm{B} \prec \mathrm{A} \prec \mathrm{D} \prec \mathrm{F} \prec \mathrm{E} \prec \mathrm{H}.$$

Dalam praktik Anda sama sekali tidak mengurutkan: dorong setiap simpul ke depan sebuah daftar pada saat ia menghitam, dan daftar itu *adalah* jawabannya ketika pencariannya berakhir. Cara itu menjaga seluruh algoritmanya tetap $\Theta(V + E)$; pengurutan eksplisit akan menambah $\Theta(V \lg V)$ dengan percuma.

(b) Pemeriksaannya. Posisinya: C 1, G 2, B 3, A 4, D 5, F 6, E 7, H 8. Setiap busur:

$A \to D$: $4 < 5$ ✓. $B \to D$: $3 < 5$ ✓. $B \to E$: $3 < 7$ ✓. $C \to G$: $1 < 2$ ✓. $D \to E$: $5 < 7$ ✓. $D \to F$: $5 < 6$ ✓. $E \to H$: $7 < 8$ ✓. $F \to H$: $6 < 8$ ✓. $G \to H$: $2 < 8$ ✓.

Kesembilannya menunjuk ke depan, jadi urutannya sah. Pemeriksaan ini berongkos $\Theta(V + E)$ dan layak ada di setiap jawaban dan setiap program.

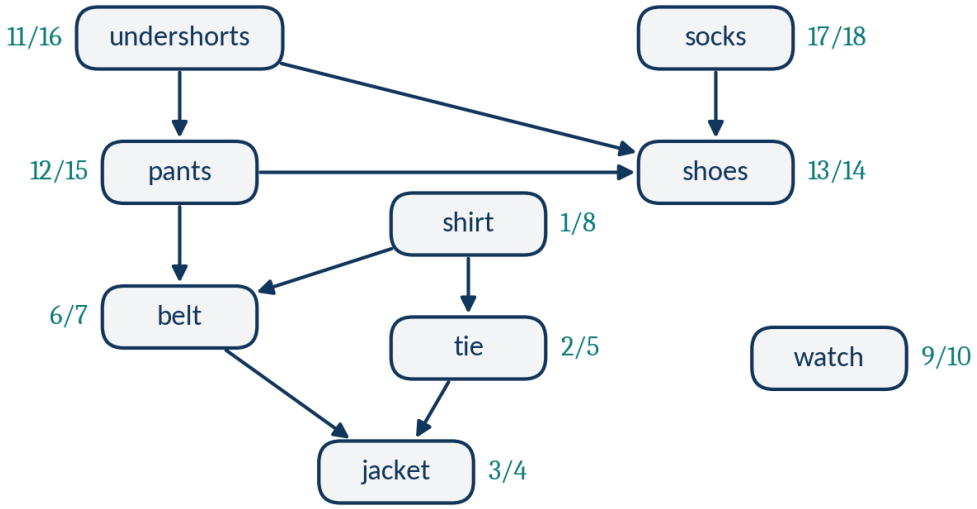
(c) Mengapa f dan bukan d. Sebab urutan *penemuan* tidak mengatakan apa pun tentang ketergantungan. Ambillah DAG dua simpul dengan busur tunggal $B \to A$, dan biarkan gelung luarnya menimbang A lebih dahulu. Maka $d[A] = 1$, $d[B] = 3$, dan mengurutkan menurut d menaik memberikan A, B — padahal busurnya menuntut B sebelum A, jadi urutannya keliru. Mengurutkan menurut f menurun memberikan $f[A] = 2$, $f[B] = 4$, jadi B, A — benar.

Alasannya, dinyatakan dengan semestinya: ketika sebuah busur $u \to v$ diperiksa, v entah putih (maka v selesai di dalam panggilan u), entah hitam (ia sudah selesai), entah kelabu — yang mustahil di sini, sebab kelabu berarti sisi mundur dan grafnya taksiklik. Dalam setiap kasus

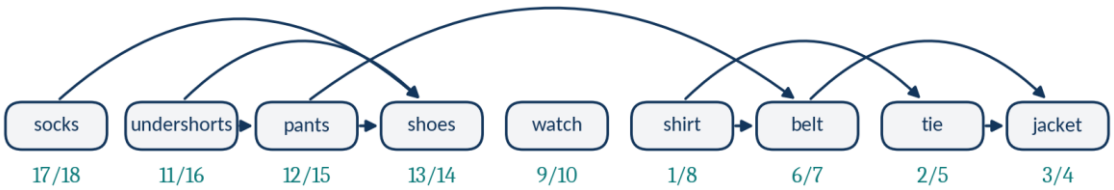
$$f[v] < f[u] \qquad \text{untuk setiap busur } u \to v \text{ pada sebuah DAG,} \tag{12.15}$$

dan karena itu, menurut Persamaan (12.15), f menurun selalu menempatkan u sebelum v.

Persoalan berpakaian §11.3.3 adalah contoh klasiknya, **Gambar 12.16**, dan jawabannya adalah **Gambar 12.17** — kesembilan simpul yang sama ditata menurut waktu selesai yang menurun, dengan setiap anak panah kini menunjuk ke kanan.



Gambar 12.16 DAG urutan untuk berpakaian, dengan cap waktu pencarian mendalam di samping tiap simpul: kemeja 1/8, dasi 2/5, jas 3/4, ikat pinggang 6/7, arloji 9/10, celana dalam 11/16, celana 12/15, sepatu 13/14, kaus kaki 17/18. Sembilan simpul, sembilan busur, delapan belas cap.



Gambar 12.17 Hasilnya: kesembilan busana yang sama ditata dari kiri ke kanan menurut waktu selesai yang menurun — kaus kaki 18, celana dalam 16, celana 15, sepatu 14, arloji 10, kemeja 8, ikat pinggang 7, dasi 5, jas 4 — dan kini setiap anak panah menunjuk ke kanan. Itulah *arti* pengurutan topologis.

Jawaban. C, G, B, A, D, F, E, H; kesembilan busurnya menunjuk ke depan; waktu selesai berhasil karena Persamaan (12.15), dan waktu temu gagal pada $B \to A$ yang dicari dari A.

> **UJI DIRI.** Jalankan metode mendalam sekali lagi dengan gelung luar berurutan H, G, F, E, D, C, B, A. Urutan apa yang dihasilkannya, dan apakah ia sah?

Jawaban. Tiap panggilan luar kini bermula di simpul yang semua penerusnya sudah selesai, jadi setiap panggilan seketika selesai: H 1/2, G 3/4, F 5/6, E 7/8, D 9/10, C 11/12, B 13/14, A 15/16. f menurun memberikan A, B, C, D, E, F, G, H — menurut abjad, dan **sah**: setiap satu dari kesembilan busurnya berjalan dari huruf yang lebih awal ke huruf yang lebih akhir. Sebuah urutan yang berbeda, sama benarnya.

12.5.4 Persoalan Terkerjakan O — pengurutan topologis dengan algoritma Kahn

Persoalan. (a) Berikan derajat masuk awal setiap mata ajar. (b) Jalankan algoritma Kahn, **Pseudokode 12.6**, dengan memberikan tabel satu baris per putaran: mata ajar yang dikeluarkan, derajat masuk sesudah penghapusannya, dan antrean sesudahnya. (c) Bandingkan jawaban Anda dengan Persoalan N. Pada berapa posisi kedua urutan itu sepakat, dan mengapa hal itu tidak menjadi masalah?

Pseudokode 12.6 Algoritma Kahn: berulang-ulang keluarkan simpul yang tak lagi punya prasyarat tersisa, lalu buang busur-busur keluarnya.

```
KAHN(G)
 1  for each vertex v:  indeg[v] = 0
 2  for each arc (u, v):  indeg[v] = indeg[v] + 1
 3  Q = an empty FIFO queue;  L = an empty list
 4  for each vertex v with indeg[v] == 0:  ENQUEUE(Q, v)
 5  while Q is not empty
 6      u = DEQUEUE(Q);  append u to L
 7      for each v in Adj[u]
 8          indeg[v] = indeg[v] - 1
 9          if indeg[v] == 0:  ENQUEUE(Q, v)
10  if |L| < |V| then report that G has a cycle
11  return L
```

Yang diuji. Algoritma 2 §11.3.9, dijalankan persis, dan pemahaman bahwa sebuah DAG biasanya punya banyak urutan topologis.

Penyelesaian.

(a) Derajat masuknya. Cacahlah anak panah yang tiba: A 0, B 0, C 0, D 2 (dari A, B), E 2 (dari B, D), F 1 (dari D), G 1 (dari C), H 3 (dari E, F, G). Jumlahnya $0 + 0 + 0 + 2 + 2 + 1 + 1 + 3 = 9 = |E|$, dan itulah pemeriksaan yang dijalankan sebelum mulai.

(b) Jalannya. Antrean bermula dengan simpul-simpul berderajat masuk nol menurut urutan abjad, $[A, B, C]$, dan **Tabel 12.29** mencatat setiap putarannya.

Tabel 12.29 Algoritma Kahn pada DAG prasyarat, satu baris per penghapusan. Entri yang berubah tercetak tebal. Delapan mata ajar dikeluarkan dalam delapan putaran, jadi grafnya taksiklik dan urutannya sah.

Putaran	Keluarkan	Derajat masuk sesudahnya (A…H)	Antrean sesudahnya
1	A	0 0 0 **1** 2 1 1 3	B C
2	B	0 0 0 **0** **1** 1 1 3	C D
3	C	0 0 0 0 1 1 **0** 3	D G
4	D	0 0 0 0 **0** **0** 0 3	G E F
5	G	0 0 0 0 0 0 0 **2**	E F
6	E	0 0 0 0 0 0 0 **1**	F
7	F	0 0 0 0 0 0 0 **0**	H
8	H	semua nol	(kosong)

Urutannya: **A, B, C, D, G, E, F, H.**

(c) Perbandingannya. Urutan mendalamnya adalah C, G, B, A, D, F, E, H; urutan Kahn adalah A, B, C, D, G, E, F, H. Keduanya sepakat pada **satu** posisi dari delapan — yang terakhir, H — dan berselisih di semua posisi lain.

Itu bukan masalah. Urutan topologis adalah urutan linear mana pun yang di dalamnya setiap busur menunjuk ke depan, dan DAG-nya hanya mengikat *sebagian* pasangan. Tak ada apa pun di dalam datanya yang mengaitkan A dengan C, jadi yang mana pun boleh lebih dahulu; tak ada yang mengaitkan G dengan D; dan seterusnya. Kedua urutan diperiksa busur demi busur, dan keduanya lolos. Satu-satunya simpul yang dipaksa berada di tempat terakhir adalah H, sebab setiap mata ajar lain menjangkaunya.

> **WAWASAN KUNCI: ALGORITMA MANA YANG DIPAKAI, DALAM SATU BARIS MASING-MASING.** Pakailah **Kahn** ketika Anda menginginkan pendeteksian siklus secara gratis, ketika Anda ingin memproses simpul-simpulnya dalam gelombang (segala yang berderajat masuk nol dapat berjalan sejajar), atau ketika kedalaman rekursi mengkhawatirkan. Pakailah metode **mendalam** ketika program Anda sudah punya DFS, atau ketika Anda juga memerlukan waktu selesainya untuk keperluan lain. Keduanya $\Theta(V + E)$; tak ada lagi yang membedakan keduanya.

Jawaban. Derajat masuknya 0,0,0,2,2,1,1,3; urutan Kahn A, B, C, D, G, E, F, H; satu posisi yang sama dengan Persoalan N, dan itu wajar sebab keduanya sah.

> **UJI DIRI.** Ganti antrean FIFO Kahn dengan antrean prioritas-min yang selalu melepaskan mata ajar siap yang abjadnya terkecil. Urutan apa yang dihasilkan?
>
> *Jawaban.* Siap $\{A, B, C\}$: keluarkan A (D turun menjadi 1). Siap $\{B, C\}$: keluarkan B (D menjadi 0, E menjadi 1). Siap $\{C, D\}$: keluarkan C (G siap). Siap $\{D, G\}$: keluarkan D (E, F siap). Siap $\{E, F, G\}$: keluarkan E, lalu F, lalu G, dan akhirnya H. Urutannya A, B, C, D, E, F, G, H — yang abjadnya paling awal di antara semua urutan yang sah, dan itulah yang selalu dihasilkan antrean prioritas atas nama.

12.5.5 Persoalan Terkerjakan P — mencacah urutan topologis

Persoalan. (a) Berapa banyak urutan topologis yang berbeda dimiliki DAG prasyarat? Jelaskan cara mencacahnya tanpa mendaftar seluruh 8! permutasi. (b) Berapa prasyarat tambahan yang harus ditambahkan supaya urutannya tunggal?

Yang diuji. Arti urutan topologis sebagai *perluasan linear* sebuah urutan parsial, dan kriteria ketunggalannya.

Penyelesaian.

(a) Cacahnya. Ada $8! = 40\,320$ permutasi atas delapan mata ajar; yang merupakan urutan topologis sah berjumlah **84**. Angka itu diperoleh dengan menguji setiap permutasi terhadap kesembilan busurnya, yaitu $40\,320 \times 9$ perbandingan — cukup cepat untuk delapan simpul dan tanpa harapan untuk dua puluh.

Metode yang tidak mendaftar bersifat rekursif. Misalkan $N(S)$ adalah banyaknya urutan sah atas himpunan mata ajar S; pilihlah mata ajar yang datang pertama — yang harus salah satu yang tak punya prasyarat tersisa di dalam S — lalu cacahlah urutan sisanya:

$$N(\varnothing) = 1, \qquad N(S) = \sum_{\substack{v \in S \\ v \text{ tak punya prasyarat di } S}} N(S \setminus \{v\}). \tag{12.16}$$

Memoisasi Persamaan (12.16) atas *himpunan* S memberikan kerja $O(2^n \cdot n)$ — untuk $n = 8$ itu $256 \times 8 = 2048$ langkah alih-alih 40.320, dan selisihnya melebar dengan cepat. Mencacah perluasan linear pada umumnya #P-lengkap [10], jadi tak ada metode polinomial yang dikenal; intinya adalah gagasan eksponensial-tetapi-jauh-lebih-baik itu, bukan sebuah mukjizat.

Pemeriksaan kewajaran atas 84: ketiga akar A, B, C dapat diselang-seling cukup bebas, H dipaksa terakhir, dan E serta F sama-sama sesudah D tetapi tidak saling terkait. Seandainya setiap mata ajar bebas, jawabannya $8! = 40\,320$; seandainya DAG-nya satu rantai tunggal, jawabannya 1. Delapan puluh empat terletak di antara keduanya dan jauh lebih dekat ke rantai, dan memang begitulah yang semestinya dihasilkan sembilan kendala atas delapan simpul.

(b) Memaksakan ketunggalan. Sebuah urutan topologis tunggal **bila dan hanya bila DAG-nya memuat lintasan Hamilton** — lintasan berarah yang mengunjungi setiap simpul — sebab lintasan semacam itu memaksa setiap pasangan ke dalam urutan relatif yang tetap, dan tanpa lintasan itu ada pasangan bersebelahan dalam urutannya yang tidak terkait dan dapat ditukar. Dalam istilah Kahn: urutannya tunggal persis ketika antreannya tak pernah memuat lebih dari satu simpul.

Jadi kita perlu menambahkan busur sampai suatu urutan sah punya *ketujuh* pasangan bersebelahannya disambungkan busur. Lihatlah lebih dahulu mengapa satu busur tidak mungkin cukup: A, B, dan C semuanya sumber, dan satu busur baru hanya dapat mencabut status sumber salah satunya, jadi dua sumber tersisa dan yang mana pun boleh lebih dahulu. Minimum yang persis diperoleh dengan memeriksa setiap satu dari ke-84 urutan sah dan mencacah berapa dari ketujuh pasangan bersebelahannya yang *sudah* berupa busur. Yang terbaik adalah 4, dicapai oleh

$$A,\ B,\ D,\ E,\ C,\ G,\ F,\ H,$$

yang pasangan bersebelahannya $B \to D$, $D \to E$, $C \to G$, dan $F \to H$ adalah prasyarat yang sudah ada. Ketiga lainnya — $A \to B$, $E \to C$, dan $G \to F$ — harus ditambahkan, dan menambahkannya tidak menciptakan siklus. Jadi **tiga prasyarat tambahan perlu dan cukup**: tak ada urutan yang punya lebih dari empat mata rantainya sudah hadir, jadi tak ada dua tambahan yang dapat menuntaskan sebuah lintasan Hamilton. **Kode 12.8** mengerjakan persis pemeriksaan ini.

Jawaban. 84 urutan sah, dapat dicacah dengan Persamaan (12.16); tiga prasyarat tambahan (misalnya $A \to B$, $E \to C$, $G \to F$) diperlukan supaya urutannya tunggal.

> **UJI DIRI.** Tambahkan satu prasyarat $F \to E$. Berapa urutan sah yang tersisa?
>
> *Jawaban.* **42**, persis separuhnya. Sebelum perubahan itu, E dan F dapat saling ditukar: keduanya memerlukan D (dan E juga memerlukan B, yang sudah mendahului D), dan keduanya hanya diperlukan oleh H. Jadi menukar E dan F mengubah urutan sah mana pun menjadi urutan sah yang lain, memasangkan ke-84 urutan itu. Busur baru itu mempertahankan persis separuh yang di dalamnya F mendahului E.

12.5.6 Persoalan Terkerjakan Q — lintasan kritis

Persoalan. (a) Hitunglah pekan paling awal setiap mata ajar dapat dirampungkan, dengan mengandaikan kapasitas pengajaran tak terbatas, dan berikan lintasan kritisnya. (b) Berapa lama program studinya jika mata ajar diambil satu per satu secara ketat? (c) Tambahkan prasyarat $H \to A$. Apa yang kini terjadi pada masing-masing dari kedua algoritma pengurutan topologis?

Yang diuji. Lintasan terpanjang dalam sebuah DAG lewat satu pelonggaran per busur dalam urutan topologis — Persamaan (12.2) §12.2.5 sekali lagi, dengan pekan alih-alih menit.

Penyelesaian.

(a) Pekan selesai terawal. Proseslah mata ajarnya dalam urutan topologis *mana pun* — urutan Kahn pun jadi — dan terapkan Persamaan (12.2): sebuah sumber selesai pada durasinya sendiri; selainnya $ef[v] = \max_{u \to v} ef[u] + weeks[v]$.

- $ef[A] = 3$. $ef[B] = 2$. $ef[C] = 4$.
- D: pendahulu $A(3)$, $B(2)$; max $= 3$; $ef[D] = 3 + 5 = 8$.
- G: pendahulu $C(4)$; $ef[G] = 4 + 2 = 6$.
- E: pendahulu $B(2)$, $D(8)$; max $= 8$; $ef[E] = 8 + 6 = 14$.
- F: pendahulu $D(8)$; $ef[F] = 8 + 3 = 11$.
- H: pendahulu $E(14)$, $F(11)$, $G(6)$; max $= 14$; $ef[H] = 14 + 4 = \mathbf{18}$.

Menelusuri maksimumnya mundur, H mengambil waktunya dari E, E dari D, D dari A:

$$\mathrm{A} \to \mathrm{D} \to \mathrm{E} \to \mathrm{H}, \qquad 3 + 5 + 6 + 4 = 18 \text{ pekan.}$$

Itulah **lintasan kritisnya**: perpanjanglah mata ajar mana pun di atasnya dan seluruh program studinya memanjang; perpendeklah mata ajar mana pun di luarnya dan tak ada yang berubah. G, misalnya, selesai pada pekan 6 tetapi baru diperlukan pada pekan 14, jadi ia membawa kelonggaran $14 - 6 = 8$ pekan.

(b) Satu per satu. $3 + 2 + 4 + 5 + 6 + 3 + 2 + 4 = 29$ pekan. DAG-nya menghemat $29 - 18 = 11$ pekan, dan selisih itu persis banyaknya studi sejajar yang diizinkan kendala urutannya.

(c) Menambahkan $H \to A$. Grafnya kini memuat siklus berarah $A \to D \to E \to H \to A$, dan kedua algoritmanya melaporkannya, dengan ungkapannya masing-masing.

- **Kahn** hanya mengeluarkan **3** dari 8 mata ajar. Derajat masuk kelima simpul yang berada pada atau sesudah siklus itu tak pernah mencapai nol, jadi antreannya kosong terlalu dini. Kelima simpul yang gagal dikeluarkannya menyebut masalahnya.
- **Metode mendalam** menemukan **1** sisi mundur, $H \to A$, yang diperiksa selagi A masih kelabu. Menurut teorema DAG, satu sisi mundur itu adalah sertifikatnya.

Tak satu pun menghasilkan urutan yang "hampir benar", dan itulah sifat yang penting: spesifikasi yang rusak harus gagal dengan lantang. Pengurutan topologis yang diam-diam mengembalikan daftar sebagian akan lebih buruk daripada tak berguna di dalam sebuah penjadwal.

Jawaban. Selesai terawal A 3, B 2, C 4, D 8, G 6, E 14, F 11, H 18; lintasan kritis $A \to D \to E \to H$, 18 pekan terhadap 29 bila satu per satu; dengan $H \to A$ kedua algoritma melaporkan sebuah siklus.

UJI DIRI. Algoritma (E) dipendekkan dari 6 pekan menjadi 3. Berapa pekan selesai yang baru, dan apa lintasan kritisnya?

Jawaban. $ef[E] = 8 + 3 = 11$, dan $ef[F] = 11$ seperti sebelumnya, jadi $ef[H] = \max(11,11,6) + 4 = 15$. Program studinya menyusut menjadi **15 pekan**, dan kini ada **dua** lintasan kritis, $A \to D \to E \to H$ dan $A \to D \to F \to H$, keduanya 15 pekan. Memendekkan E sebanyak tiga pekan hanya menghemat tiga pekan, dan memendekkan E sendirian lebih jauh lagi takkan menghemat apa pun, sebab F kini sama kritisnya.

12.5.7 Persoalan M sampai Q sebagai satu program

Kode 12.8 menjawab Persoalan M sampai Q secara mekanis, termasuk cacah urutan sahnya, prasyarat tambahan tersedikit yang memaksa urutan tunggal, dan siklus yang disuntikkan pada Persoalan Q.

Kode 12.8 Persoalan Terkerjakan M sampai Q: cap waktu dan penggolongan pencarian mendalam, kedua pengurutan topologis, cacah urutan sah beserta uji ketunggalannya, lintasan kritisnya, dan siklus yang disuntikkan. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o dag dag.cpp`.

```
// Persoalan Terkerjakan M sampai Q Bab 12 -- DAG dan pengurutan topologis.
//   M  apakah digraf prasyarat itu DAG?  (DFS, dan uji sisi mundur)
//   N  pengurutan topologis menurut waktu selesai pencarian mendalam
//   O  pengurutan topologis dengan algoritma derajat-masuk Kahn
//   P  ada berapa urutan topologis berbeda, dan berapa prasyarat tambahan
//      yang memaksa urutannya menjadi tunggal?
//   Q  lintasan kritis: secepat apa seluruh kurikulum dapat dirampungkan
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <numeric>
#include <functional>
using namespace std;

static const vector<string> NAME = {
    "A ProgBasics", "B DiscreteMath", "C Calculus", "D DataStruct",
    "E Algorithms", "F Databases",   "G Statistics", "H MachineLearn"
};
static const vector<int> WEEKS = { 3, 2, 4, 5, 6, 3, 2, 4 };
static const vector<pair<int,int>> PRE = {
    {0,3},{1,3},{1,4},{2,6},{3,4},{3,5},{4,7},{5,7},{6,7}
};
static const int N = 8;
static string tag(int v) { return NAME[v].substr(0, 1); }

vector<vector<int>> build(const vector<pair<int,int>>& E) {
    vector<vector<int>> adj(N);
    for (auto [u,v] : E) adj[u].push_back(v);
    for (auto& l : adj) sort(l.begin(), l.end());
    return adj;
}

int main() {
    auto adj = build(PRE);
```

Kode 12.8 (Lanjutan) *Persoalan Terkerjakan M sampai Q: cap waktu dan penggolongan pencarian mendalam, kedua pengurutan topologis, cacah urutan sah beserta uji ketunggalannya, lintasan kritisnya, dan siklus yang disuntikkan. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o dag dag.cpp.*

```
    cout << "=== Problem M  depth-first search, and the back-edge test ===\n";
    vector<int> col(N,0), d(N,0), f(N,0), pi(N,-1);
    vector<string> cls(PRE.size());
    int clk = 0;
    function<void(int)> visit = [&](int u) {
        col[u] = 1; d[u] = ++clk;
        for (int v : adj[u]) {
            int idx = -1;
            for (size_t i = 0; i < PRE.size(); ++i)
                if (PRE[i].first == u && PRE[i].second == v) idx = (int)i;
            if (col[v] == 0)      { cls[idx] = "tree";    pi[v] = u; visit(v); }
            else if (col[v] == 1) { cls[idx] = "BACK"; }
            else                  { cls[idx] = (d[u] < d[v] ? "forward" : "cross"); }
        }
        col[u] = 2; f[u] = ++clk;
    };
    for (int v = 0; v < N; ++v) if (col[v] == 0) visit(v);
    cout << "v      "; for (int v = 0; v < N; ++v) cout << setw(7) << tag(v); cout << "\n";
    cout << "d/f    "; for (int v = 0; v < N; ++v) cout << setw(7) << (to_string(d[v]) +
"/" + to_string(f[v])); cout << "\n";
    cout << "clock stopped at " << clk << " = 2|V| = " << 2*N << "\n";
    int back = 0;
    for (size_t i = 0; i < PRE.size(); ++i) {
        cout << "  " << tag(PRE[i].first) << "->" << tag(PRE[i].second) << "  " << cls[i]
<< "\n";
        if (cls[i] == "BACK") ++back;
    }
    cout << "back edges: " << back << "  -> the digraph IS "
         << (back ? "NOT a DAG" : "a DAG") << "\n";

    cout << "\n=== Problem N  topological sort by decreasing finishing time ===\n";
    vector<int> byF(N); iota(byF.begin(), byF.end(), 0);
    sort(byF.begin(), byF.end(), [&](int a, int b){ return f[a] > f[b]; });
    cout << "  order:";
    for (int v : byF) cout << " " << tag(v) << "(" << f[v] << ")";
    cout << "\n";
    bool ok = true; vector<int> pos(N);
    for (int i = 0; i < N; ++i) pos[byF[i]] = i;
    for (auto [u,v] : PRE) if (pos[u] >= pos[v]) ok = false;
    cout << "  every arc points forward? " << (ok ? "yes" : "NO") << "\n";
```

Kode 12.8 (Lanjutan) *Persoalan Terkerjakan M sampai Q: cap waktu dan penggolongan pencarian mendalam, kedua pengurutan topologis, cacah urutan sah beserta uji ketunggalannya, lintasan kritisnya, dan siklus yang disuntikkan. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o dag dag.cpp.*

```
    cout << "\n=== Problem O  topological sort by Kahn's in-degree algorithm ===\n";
    vector<int> indeg(N, 0);
    for (auto [u,v] : PRE) ++indeg[v];
    cout << "  initial in-degrees: ";
    for (int v = 0; v < N; ++v) cout << tag(v) << "=" << indeg[v] << " ";
    cout << "\n";
    queue<int> Q;
    for (int v = 0; v < N; ++v) if (indeg[v] == 0) Q.push(v);
    vector<int> kahn;
    cout << "  round  emit  in-degrees after         queue after\n";
    int round = 0;
    while (!Q.empty()) {
        int u = Q.front(); Q.pop(); kahn.push_back(u);
        for (int v : adj[u]) if (--indeg[v] == 0) Q.push(v);
        string s; for (int v = 0; v < N; ++v) s += to_string(indeg[v]) + " ";
        queue<int> t = Q; string qs;
        while (!t.empty()) { qs += tag(t.front()) + " "; t.pop(); }
        cout << setw(7) << ++round << setw(6) << tag(u) << "  " << left << setw(23) << s
             << right << (qs.empty() ? "(empty)" : qs) << "\n";
    }
    cout << "  emitted " << kahn.size() << " of " << N << " -> "
         << (kahn.size() == (size_t)N ? "acyclic" : "a cycle exists") << "\n";
    cout << "  order:";
    for (int v : kahn) cout << " " << tag(v);
    cout << "\n  DFS order:";
    for (int v : byF) cout << " " << tag(v);
    cout << "\n";
    int agree = 0; for (int i = 0; i < N; ++i) if (byF[i] == kahn[i]) ++agree;
    cout << "  positions on which the two orders agree: " << agree << " of " << N << "\n";

    cout << "\n=== Problem P  how many distinct topological orders? ===\n";
    vector<int> perm(N); iota(perm.begin(), perm.end(), 0);
    long long total = 0, legal = 0;
    int bestChain = -1; vector<int> bestOrder;
    do {
        ++total;
        vector<int> p(N); for (int i = 0; i < N; ++i) p[perm[i]] = i;
        bool good = true;
        for (auto [u,v] : PRE) if (p[u] >= p[v]) { good = false; break; }
        if (!good) continue;
        ++legal;
        // berapa pasangan bertetangga dalam urutan ini yang sudah berupa prasyarat?
        int chain = 0;
        for (int i = 0; i + 1 < N; ++i)
            for (auto [u,v] : PRE) if (u == perm[i] && v == perm[i+1]) ++chain;
        if (chain > bestChain) { bestChain = chain; bestOrder = perm; }
```

Kode 12.8 (Lanjutan) *Persoalan Terkerjakan M sampai Q: cap waktu dan penggolongan pencarian mendalam, kedua pengurutan topologis, cacah urutan sah beserta uji ketunggalannya, lintasan kritisnya, dan siklus yang disuntikkan. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o dag dag.cpp.*

```
    } while (next_permutation(perm.begin(), perm.end()));
    cout << "  permutations checked: " << total << "\n";
    cout << "  legal topological orders: " << legal << "\n";
    // urutan tunggal menuntut lintasan Hamilton; setiap mata rantai yang hilang harus
ditambahkan
    cout << "  best legal order for a Hamiltonian path:";
    for (int v : bestOrder) cout << " " << tag(v);
    cout << "\n  it already contains " << bestChain << " of its " << N - 1
         << " links, so at least " << N - 1 - bestChain
         << " extra prerequisites are needed to force a unique order\n";

    cout << "\n=== Problem Q  the critical path ===\n";
    vector<int> ef(N, 0), pred(N, -1);
    for (int u : kahn) {
        if (ef[u] == 0) ef[u] = WEEKS[u];
        for (int v : adj[u]) if (ef[u] + WEEKS[v] > ef[v]) { ef[v] = ef[u] + WEEKS[v];
pred[v] = u; }
    }
    cout << "  subject          weeks  earliest finish\n";
    for (int v : kahn)
        cout << "  " << left << setw(16) << NAME[v] << right << setw(5) << WEEKS[v]
             << setw(16) << ef[v] << "\n";
    int last = (int)(max_element(ef.begin(), ef.end()) - ef.begin());
    vector<int> path;
    for (int x = last; x != -1; x = pred[x]) path.push_back(x);
    reverse(path.begin(), path.end());
    cout << "  critical path (" << ef[last] << " weeks):";
    for (size_t i = 0; i < path.size(); ++i) cout << (i ? " -> " : " ") << tag(path[i]);
    cout << "\n  total of all eight subjects taken one at a time: "
         << accumulate(WEEKS.begin(), WEEKS.end(), 0) << " weeks\n";

    cout << "\n=== the trap: one extra prerequisite H -> A ===\n";
    vector<pair<int,int>> bad = PRE; bad.push_back({7,0});
    auto adj2 = build(bad);
    vector<int> id2(N, 0);
    for (auto [u,v] : bad) ++id2[v];
    queue<int> Q2; for (int v = 0; v < N; ++v) if (id2[v] == 0) Q2.push(v);
    int emitted = 0;
    while (!Q2.empty()) { int u = Q2.front(); Q2.pop(); ++emitted;
        for (int v : adj2[u]) if (--id2[v] == 0) Q2.push(v); }
    cout << "  Kahn emits " << emitted << " of " << N
         << " -> the remaining " << N - emitted << " vertices lie on or after a cycle\n";
    vector<int> c2(N, 0); int nb = 0;
    function<void(int)> v2 = [&](int u) {
        c2[u] = 1;
        for (int v : adj2[u]) { if (c2[v] == 1) ++nb; else if (c2[v] == 0) v2(v); }
        c2[u] = 2;
    };
```

Kode 12.8 (Lanjutan) *Persoalan Terkerjakan M sampai Q: cap waktu dan penggolongan pencarian mendalam, kedua pengurutan topologis, cacah urutan sah beserta uji ketunggalannya, lintasan kritisnya, dan siklus yang disuntikkan. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o dag dag.cpp.*

```
    for (int v = 0; v < N; ++v) if (!c2[v]) v2(v);
    cout << "  DFS finds " << nb << " back edge(s) -> the same verdict, by a different
route\n";
    return 0;
}
```

Keluaran lengkapnya membenarkan setiap nilai yang dihitung dengan tangan:

```
=== Problem M  depth-first search, and the back-edge test ===
v            A      B      C      D      E      F      G      H
d/f       1/10  11/12  13/16    2/9    3/6    7/8  14/15    4/5
clock stopped at 16 = 2|V| = 16
  A->D  tree
  B->D  cross
  B->E  cross
  C->G  tree
  D->E  tree
  D->F  tree
  E->H  tree
  F->H  cross
  G->H  cross
back edges: 0  -> the digraph IS a DAG

=== Problem N  topological sort by decreasing finishing time ===
  order: C(16) G(15) B(12) A(10) D(9) F(8) E(6) H(5)
  every arc points forward? yes

=== Problem O  topological sort by Kahn's in-degree algorithm ===
  initial in-degrees: A=0 B=0 C=0 D=2 E=2 F=1 G=1 H=3
  round  emit  in-degrees after         queue after
      1     A  0 0 0 1 2 1 1 3        B C
      2     B  0 0 0 0 1 1 1 3        C D
      3     C  0 0 0 0 1 1 0 3        D G
      4     D  0 0 0 0 0 0 0 3        G E F
      5     G  0 0 0 0 0 0 0 2        E F
      6     E  0 0 0 0 0 0 0 1        F
      7     F  0 0 0 0 0 0 0 0        H
      8     H  0 0 0 0 0 0 0 0        (empty)
  emitted 8 of 8 -> acyclic
  order: A B C D G E F H
  DFS order: C G B A D F E H
  positions on which the two orders agree: 1 of 8

=== Problem P  how many distinct topological orders? ===
  permutations checked: 40320
  legal topological orders: 84
  best legal order for a Hamiltonian path: A B D E C G F H
  it already contains 4 of its 7 links, so at least 3 extra prerequisites are needed to
force a unique order

=== Problem Q  the critical path ===
  subject          weeks  earliest finish
  A ProgBasics        3               3
  B DiscreteMath      2               2
  C Calculus          4               4
  D DataStruct        5               8
  G Statistics        2               6
  E Algorithms        6              14
  F Databases         3              11
  H MachineLearn      4              18
  critical path (18 weeks): A -> D -> E -> H
  total of all eight subjects taken one at a time: 29 weeks

=== the trap: one extra prerequisite H -> A ===
  Kahn emits 3 of 8 -> the remaining 5 vertices lie on or after a cycle
  DFS finds 1 back edge(s) -> the same verdict, by a different route
```

12.6 Persoalan Terkerjakan Lanjutan IV: Pohon Merentang Minimum

12.6.1 Graf berbobot subbab ini

Persoalan R, S, U, V, dan W memakai satu graf. G tak berarah, terhubung, dan berbobot, atas $V = \{A, \ldots, H\}$ dengan ketiga belas sisi pada **Tabel 12.30**.

Tabel 12.30 Graf berbobot G pada Persoalan R, S, U, V, dan W: delapan simpul dan tiga belas sisi. Dua sisi, AB dan FG, sama-sama berbobot 3.

Sisi	AB	AC	BC	BD	CD	CE	DE	DF	EF	EG	FG	FH	GH
w	3	5	1	6	4	9	2	8	7	10	3	12	11

Putuskan setiap seri menurut urutan abjad kedua ujungnya. Bila digambar, G adalah rantai segitiga — ABC, BCD (lewat BD), CDE, DEF, EFG, FGH — dan itulah sebabnya begitu banyak siklusnya pendek. Persoalan T memakai graf bersimpul sembilan pada **Gambar 12.1** sebagai gantinya.

12.6.2 Persoalan Terkerjakan R — Kruskal, dengan penelusuran himpunan saling lepas

Persoalan. Jalankan **algoritma Kruskal** pada G. Berikan tabel satu baris per sisi yang diperiksa, menurut urutan pemeriksaannya, yang memperlihatkan wakil yang dikembalikan `Find` untuk tiap ujung, keputusan terima atau tolak, dan hutan himpunan saling lepasnya (larik `parent` dan `rank`) sesudah langkah itu. Nyatakan pohon dan bobotnya, dan katakan bagaimana Anda tahu kapan harus berhenti.

Yang diuji. Bahwa Kruskal sesungguhnya *dua* algoritma yang berjalin — sebuah pengurutan, dan sebuah struktur himpunan saling lepas (§11.10) — dan bahwa Anda dapat menuliskan apa yang sedang dikerjakan yang kedua.

Penyelesaian.

Daftar sisi yang terurut. Bobot lebih dahulu, lalu abjad:

$$BC\ 1,\ DE\ 2,\ AB\ 3,\ FG\ 3,\ CD\ 4,\ AC\ 5,\ BD\ 6,\ EF\ 7,\ DF\ 8,\ CE\ 9,\ EG\ 10,\ GH\ 1$$

Jalannya. `parent` ditulis sebagai delapan huruf, satu per simpul A sampai H; `rank` sebagai delapan angka. Implementasinya adalah yang ada di **Kode 12.1**: `Union` menempatkan akar berperingkat lebih rendah di bawah akar berperingkat lebih tinggi, dan bila seri menempatkan akar kedua di bawah akar pertama serta menaikkan peringkat akar pertama; `Find` memampatkan lintasan yang dilaluinya.

Tabel 12.31 Kruskal pada G, dengan hutan himpunan saling lepasnya sesudah setiap keputusan. Tujuh sisi diterima, lima ditolak, satu tak pernah diperiksa. Baris 11 layak dipelajari: sisinya *ditolak*, namun `parent[G]` berubah dari `F` menjadi `B`, sebab kedua panggilan `Find` memampatkan lintasannya dalam perjalanan menuju jawabannya.

#	Sisi	w	Find(a)	Find(b)	Vonis	parent	rank
1	BC	1	B	C	ACCEPT	ABBDEFGH	01000000
2	DE	2	D	E	ACCEPT	ABBDDFGH	01010000
3	AB	3	A	B	ACCEPT	BBBDDFGH	01010000
4	FG	3	F	G	ACCEPT	BBBDDFFH	01010100
5	CD	4	B	D	ACCEPT	BBBBDFFH	02010100
6	AC	5	B	B	reject	BBBBDFFH	02010100
7	BD	6	B	B	reject	BBBBDFFH	02010100
8	EF	7	B	F	ACCEPT	BBBBBBFH	02010100
9	DF	8	B	B	reject	BBBBBBFH	02010100
10	CE	9	B	B	reject	BBBBBBFH	02010100
11	EG	10	B	B	reject	BBBBBBBH	02010100
12	GH	11	B	H	ACCEPT	BBBBBBBB	02010100

Ikutilah tiga baris dengan rinci, sebab ketiganya memperlihatkan setiap perilaku strukturnya. **Baris 3**, AB: `Find(A)` $=A$ (peringkat 0) dan `Find(B)` $=B$ (peringkat 1). Peringkatnya berbeda, jadi akar berperingkat lebih rendah, A, ditempatkan di bawah B, dan tak ada peringkat yang berubah. **Baris 5**, CD: `Find(C)` $=B$ (peringkat 1) dan `Find(D)` $=D$ (peringkat 1). Seri: D ditempatkan di bawah B dan `rank[B]` naik menjadi 2. **Baris 8**, EF: `Find(E)` berjalan $E \to D \to B$ dan memampatkan, menetapkan `parent[E]` menjadi B; `Find(F)` $=F$ (peringkat 1). B berperingkat 2, jadi F ditempatkan di bawah B.

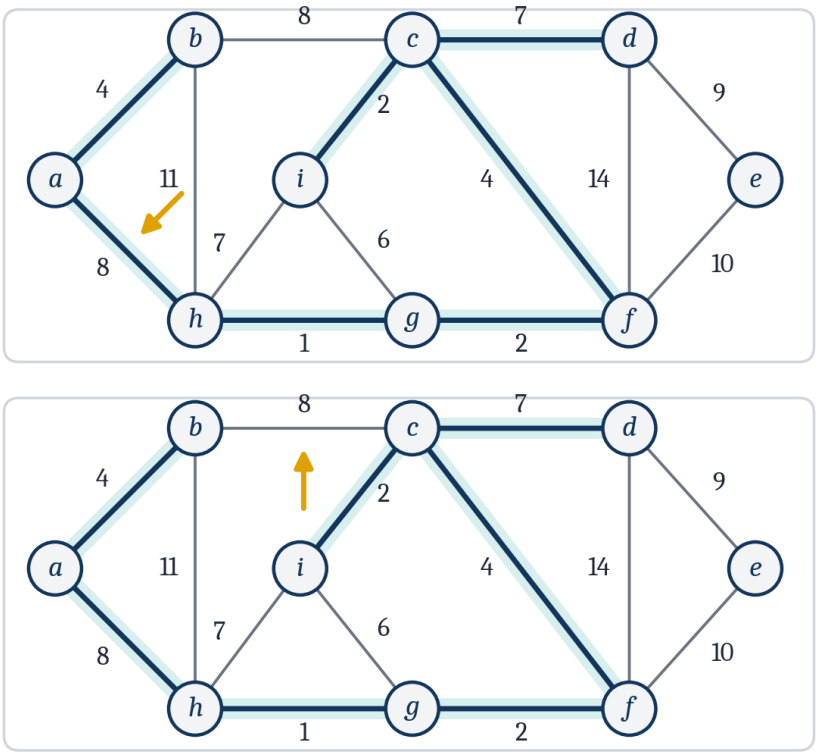
Aturan berhentinya. Sesudah baris 12 pohonnya punya $7 = |V| - 1$ sisi, dan pohon merentang atas delapan simpul tidak dapat punya lebih. Sisi ketiga belas, FH 12, tak perlu diperiksa sama sekali: apa pun ia, kedua ujungnya sudah tersambung. Berhenti di $|V| - 1$ menghemat satu perbandingan di sini dan dapat menghemat sangat banyak pada graf rapat.

Jawabannya.

$$T = \{BC\,1,\ DE\,2,\ AB\,3,\ FG\,3,\ CD\,4,\ EF\,7,\ GH\,11\}, \qquad w(T) = 1 + 2 + 3 + 3 + 4 + 7 + 11 = \mathbf{31}.$$

> **PERINGATAN: JEBAKAN PADA PERINGKATNYA.** `rank` adalah *batas atas ketinggian*, bukan ketinggiannya. Ia naik hanya ketika dua akar berperingkat **sama** disatukan (baris 1 sampai 5 memperlihatkan kedua perilakunya), dan pemampatan lintasan tak pernah menurunkannya. Sesudah jalannya, `rank[B]` bernilai 2 padahal pohon B hanya satu tingkat dalamnya. Itu tidak berbahaya dan memang disengaja: menghitung ulang ketinggian yang persis akan lebih mahal daripada yang dihematnya.

Gambar 12.18 memperlihatkan langkah, pada graf bersimpul sembilan Bab 11, tempat jawaban Kruskal berpisah jalan dengan jawaban Prim — dua sisi berbobot sama, nasib yang berlawanan, diputuskan sepenuhnya oleh partisinya dan bukan oleh bobotnya.



Gambar 12.18 Kruskal pada graf bersimpul sembilan Bab 11 di bobot 8: (a, h) diterima, (b, c) ditolak. Prim, pada graf yang sama, mengerjakan sebaliknya — ia mengambil (b, c) dan tak pernah menimbang (a, h). Kedua pohonnya berbobot 37. Bobot yang sama adalah tempat kedua algoritma itu bebas berselisih.

Jawaban. $T = \{BC, DE, AB, FG, CD, EF, GH\}$, berbobot 31, ditemukan sesudah memeriksa dua belas dari ketiga belas sisi.

> **UJI DIRI.** Naikkan $w(FG)$ dari 3 menjadi 13 lalu jalankan ulang Kruskal. Sisi mana yang kini dimuat pohonnya, dan berapa bobotnya?
>
> *Jawaban.* BC 1, DE 2, AB 3, CD 4 diterima seperti sebelumnya; AC 5 dan BD 6 ditolak; EF 7 diterima; DF 8 dan CE 9 ditolak; EG 10 kini **diterima**, sebab G belum tersambung ke apa pun; GH 11 diterima. Pohonnya $\{BC, DE, AB, CD, EF, EG, GH\}$, berbobot $1 + 2 + 3 + 4 + 7 + 10 + 11 = 38$.

12.6.3 Persoalan Terkerjakan S — Prim, dengan larik kunci

Persoalan. Jalankan **Prim–Jarník** pada G dari A. Berikan tabel satu baris per pengambilan yang memperlihatkan simpul yang diambil, sisi yang ditetapkan, total berjalan, dan simpul-simpul yang tersisa sesudahnya sebagai `v:key/π`. Bandingkan pohon yang dihasilkan dengan pohon Persoalan R.

Yang diuji. Kedua larik §11.9.2. Untuk simpul v yang belum berada di pohon T,

$$key[v] = \min\{\, w(u, v) : u \in T,\ \{u, v\} \in E \,\}, \tag{12.17}$$

yaitu bobot sisi termurah yang sejauh ini ditemukan dari pohon yang sedang tumbuh ke v (∞ bila tak ada), dan $\pi[v]$ adalah ujung pohon sisi itu. Selebihnya pembukuan belaka: ambillah simpul berkunci terkecil, tetapkan sisinya, lalu pulihkan Persamaan (12.17) dengan menurunkan kunci tetangga-tetangganya.

Penyelesaian. Mulailah dengan $key[A] = 0$, yang lain ∞, kedelapan simpul menunggu.

Tabel 12.32 Prim–Jarník dari A, satu baris per pengambilan. Delapan pengambilan, tujuh sisi ditetapkan, total 31. Setiap kunci yang turun adalah sebuah `DecreaseKey`: C dari $5/A$ ke $1/B$ pada langkah 2, D dari $6/B$ ke $4/C$ pada langkah 3, E dari $9/C$ ke $2/D$ pada langkah 4, F dari $8/D$ ke $7/E$ pada langkah 5, G dari $10/E$ ke $3/F$ pada langkah 6, dan H dari $12/F$ ke $11/G$ pada langkah 7.

Langkah	Ambil	Sisi yang ditetapkan	Total	Yang menunggu sesudahnya, sebagai v:key/π
1	A	(akar)	0	$B{:}3/A$ $C{:}5/A$ $D{:}\infty$ $E{:}\infty$ $F{:}\infty$ $G{:}\infty$ $H{:}\infty$
2	B	$AB(3)$	3	$C{:}1/B$ $D{:}6/B$ $E{:}\infty$ $F{:}\infty$ $G{:}\infty$ $H{:}\infty$
3	C	$BC(1)$	4	$D{:}4/C$ $E{:}9/C$ $F{:}\infty$ $G{:}\infty$ $H{:}\infty$
4	D	$CD(4)$	8	$E{:}2/D$ $F{:}8/D$ $G{:}\infty$ $H{:}\infty$
5	E	$DE(2)$	10	$F{:}7/E$ $G{:}10/E$ $H{:}\infty$
6	F	$EF(7)$	17	$G{:}3/F$ $H{:}12/F$
7	G	$FG(3)$	20	$H{:}11/G$
8	H	$GH(11)$	31	(kosong)

Perbandingannya. Membaca **Tabel 12.32**, pohon Prim adalah $\{AB, BC, CD, DE, EF, FG, GH\}$, berbobot 31 — *himpunan sisi yang sama* dengan milik Kruskal, hanya ditetapkan dalam urutan yang berbeda. Kruskal mengambilnya menurut urutan bobot 1,2,3,3,4,7,11; Prim mengambilnya menurut urutan pohon yang tumbuh mencapainya, 3,1,4,2,7,3,11. Kesepakatan atas himpunan sisinya tidak dijamin pada umumnya (lihat **Gambar 12.18**); kesepakatan atas *bobotnya* selalu dijamin.

Jalannya Prim di sini punya bentuk yang tidak lazim: pohonnya tumbuh sebagai lintasan $A \to B \to C \to \cdots \to H$, jadi setiap pengambilan menetapkan sisi menuju simpul yang baru saja ditambahkan. Itu kebetulan graf rantai-segitiga ini, bukan sifat Prim. Pada graf bersimpul sembilan Bab 11 perilakunya sangat berbeda, seperti diperlihatkan Persoalan T — sisi termurahnya, berbobot 1, baru diambil pada langkah ketujuh.

Jawaban. Prim dari A menetapkan $AB, BC, CD, DE, EF, FG, GH$ dalam urutan itu, berbobot 31 — pohon yang sama dengan milik Kruskal.

> **UJI DIRI.** Jalankan Prim pada G dari H. Dalam urutan apa simpul-simpulnya diambil, dan sisi mana yang ditetapkan masing-masingnya?
>
> *Jawaban.* H (akar), G lewat $GH(11)$, F lewat $FG(3)$, E lewat $EF(7)$, D lewat $DE(2)$, C lewat $CD(4)$, B lewat $BC(1)$, A lewat $AB(3)$. Total 31, ketujuh sisi yang sama, ditumbuhkan dari ujung rantai yang lain.

12.6.4 Persoalan Terkerjakan T — Prim, dengan tumpukan biner

Persoalan. Larik Persoalan S menemukan kunci minimum dengan memindai setiap simpul yang menunggu, yang berongkos $\Theta(V)$ per pengambilan dan $\Theta(V^2)$ seluruhnya. Implementasi yang sungguhan memakai tumpukan biner. `std::priority_queue` C++ tidak punya `DecreaseKey`, jadi siasat bakunya (§11.9.10) bersifat **malas**: ketika sebuah kunci turun, dorong entri *baru* dan biarkan entri lamanya tetap di dalam tumpukan; ketika sebuah entri mencapai puncak, buang ia jika simpulnya sudah berada di pohon atau kuncinya sudah basi. (a) Jalankan versi ini, **Pseudokode 12.7**, pada graf bersimpul sembilan **Gambar 12.1** dari simpul a, dengan tetangga dipindai menurut abjad, dan berikan tumpukannya sesudah setiap langkah. (b) Cacahlah dorongan, pengeluaran, dan pengeluaran basinya. (c) Berikan waktu jalannya. (d) Bandingkan pohonnya dengan pohon Bab 11.

Pseudokode 12.7 Algoritma Prim dengan tumpukan biner dan `DecreaseKey` malas: kunci yang membaik didorong sebagai entri baru, dan entri yang basi dibuang ketika sampai di puncak.

```
LAZY-PRIM(G, w, r)
 1  for each vertex v:  key[v] = infinity;  pi[v] = NIL;  intree[v] = FALSE
 2  key[r] = 0;  H = a min-heap containing (0, r)
 3  while H is not empty
 4      (k, u) = EXTRACT-MIN(H)
 5      if intree[u] or k != key[u]
 6          continue                          // entri basi: buang
 7      intree[u] = TRUE                      // tetapkan sisi (pi[u], u)
 8      for each v in Adj[u]
 9          if not intree[v] and w(u, v) < key[v]
10              key[v] = w(u, v);  pi[v] = u
11              INSERT(H, (key[v], v))        // entri lama tetap tertinggal
```

Yang diuji. Implementasi Prim dengan antrean prioritas (§11.9.4 dan §11.9.7), dan persisnya apa yang dimuat sebuah tumpukan pada setiap saat.

Penyelesaian.

(a) Jalannya. Keempat belas sisi grafnya adalah ab 4, ah 8, bc 8, bh 11, cd 7, cf 4, ci 2, de 9, df 14, ef 10, fg 2, gh 1, gi 6, hi 7. Tumpukannya diperlihatkan sebagai entri-entrinya dalam urutan menaik, masing-masing ditulis `simpul:kunci`; bila dua entri berkunci sama, entri dengan simpul yang abjadnya lebih awal keluar lebih dahulu. **Tabel 12.33** adalah jalannya.

Tabel 12.33 Prim dengan tumpukan biner dan `DecreaseKey` malas pada graf bersimpul sembilan, dari a. Sembilan pengambilan sungguhan menetapkan delapan sisi dengan bobot total 37; empat pengeluaran lainnya mendapati entri basi — salinan lama g, h (dua kali), dan e yang kuncinya sudah turun sejak itu — dan membuangnya.

Langkah	Keluarkan	Tetapkan	Total	Kunci yang turun	Tumpukan sesudahnya
1	a: 0	(akar)	0	$b: \infty \to 4$, $h: \infty \to 8$	b: 4 h: 8
2	b: 4	ab(4)	4	$c: \infty \to 8$	c: 8 h: 8
3	c: 8	bc(8)	12	$d \to 7$, $f \to 4$, $i \to 2$	i: 2 f: 4 d: 7 h: 8
4	i: 2	ci(2)	14	$g \to 6$, $h: 8 \to 7$	f: 4 g: 6 d: 7 h: 7 h: 8
5	f: 4	cf(4)	18	$e \to 10$, $g: 6 \to 2$	g: 2 g: 6 d: 7 h: 7 h: 8 e: 10
6	g: 2	fg(2)	20	$h: 7 \to 1$	h: 1 g: 6 d: 7 h: 7 h: 8 e: 10
7	h: 1	gh(1)	21	—	g: 6 d: 7 h: 7 h: 8 e: 10
—	g: 6	basi, dibuang	21	—	d: 7 h: 7 h: 8 e: 10
8	d: 7	cd(7)	28	$e: 10 \to 9$	h: 7 h: 8 e: 9 e: 10
—	h: 7, h: 8	keduanya basi	28	—	e: 9 e: 10
9	e: 9	de(9)	37	—	e: 10
—	e: 10	basi, dibuang	37	—	(kosong)

Tiga saat layak dicermati. **Langkah 2 ke langkah 3:** c dan h sama-sama berkunci 8, dan c keluar lebih dahulu menurut aturan serinya — persis pilihan yang diambil di §11.9.5. **Langkah 4:** kunci h turun dari 8 menjadi 7, dan tumpukannya kini memuat *dua* entri untuk h, yaitu h: 7 yang hidup dan h: 8 yang basi. **Langkah 6:** kunci h turun lagi, menjadi 1, dan entri ketiga didorong. Ketika salinan basi h: 7 dan h: 8 akhirnya mencapai puncak sesudah langkah 8, uji pada baris 5 **Pseudokode 12.7** melihat bahwa h sudah berada di pohon dan membuang keduanya.

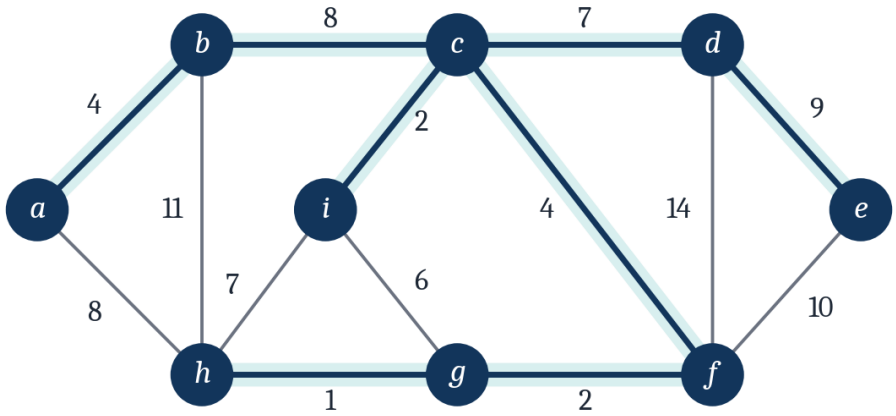
(b) Cacahnya. Setiap dorongan bersesuaian dengan satu kunci yang diturunkan, ditambah dorongan awal akarnya: $1 + 12 = 13$ dorongan. Setiap dorongan akhirnya dikeluarkan: 13 pengeluaran, yang 9 di antaranya pengambilan sungguhan (satu per simpul) dan 4 basi. **Kode 12.9** mencetak persis cacah-cacah ini.

(c) Waktu jalannya. Setiap sisi dapat menurunkan sebuah kunci paling banyak sekali: ia diperiksa dari kedua ujungnya, tetapi ketika ujung keduanya diambil, ujung pertamanya sudah berada di pohon, dan baris 9 mengabaikannya. Jadi seluruhnya ada paling banyak $|E| + 1$ dorongan (di sini $12 + 1 = 13$ dari kemungkinan $14 + 1$) dan tumpukannya tak pernah memuat lebih dari $|E| + 1$ entri. Setiap dorongan dan setiap pengeluaran berongkos $O(\lg E)$, dan $\lg E \leq \lg V^2 = 2\lg V$, jadi seluruh jalannya berongkos

$$O(E \lg E) = O(E \lg V), \tag{12.18}$$

yaitu batas yang sama dengan Prim bertumpukan `DecreaseKey`, menurut Persamaan (12.18). Harga kemalasannya adalah memori — tumpukannya dapat memuat sampai $|E|$ entri alih-alih $|V|$ — dan harga itu biasanya layak dibayar demi kode yang sesederhana ini.

(d) Pohonnya. Sisi-sisi yang ditetapkan adalah ab 4, bc 8, ci 2, cf 4, fg 2, gh 1, cd 7, de 9, total 37 — pohon pada **Gambar 12.19**, sisi demi sisi.



Gambar 12.19 Pohon merentang minimum yang sudah jadi pada graf bersimpul sembilan: (a, b), (b, c), (c, i), (c, f), (f, g), (g, h), (c, d), (d, e), berbobot total 37. Delapan sisi untuk sembilan simpul. Kruskal mencapai himpunan delapan sisi yang berbeda — juga berbobot 37 — sebab dua sisi berbobot 8 seri.

Kode 12.9 menjalankan versi malas itu dua kali, dari a dan dari i, dengan mencetak tumpukannya sesudah setiap langkah.

Kode 12.9 Algoritma Prim dengan `std::priority_queue` dan `DecreaseKey` malas, mencetak tumpukannya sesudah setiap langkah dan mencacah entri basinya; dijalankan dari a dan dari i. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o primheap primheap.cpp`.

```
// Persoalan Terkerjakan T: algoritma Prim dengan antrean prioritas tumpukan biner,
// dijejak pada graf sembilan simpul Bab 11 (graf pada gambar peta
// pengulangan). Tumpukannya std::priority_queue dengan DecreaseKey "malas":
// kunci yang membaik didorong sebagai entri baru, dan entri yang kuncinya tidak lagi
// sama dengan key[v] (atau simpulnya sudah di dalam pohon) dibuang ketika
// sampai di puncak. Isi tumpukan dicetak setelah setiap langkah.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
using namespace std;

static const string NM = "abcdefghi";
struct Edge { int a, b, w; };
static const vector<Edge> G = {
    {0,1,4}, {0,7,8}, {1,2,8}, {1,7,11}, {2,3,7}, {2,5,4}, {2,8,2},
    {3,4,9}, {3,5,14}, {4,5,10}, {5,6,2}, {6,7,1}, {6,8,6}, {7,8,7}
};
using Entry = pair<int,int>;                                    // (kunci, simpul)

string heapContents(priority_queue<Entry, vector<Entry>, greater<>> h) {
    string s;                                                   // mengeluarkan salinannya,
yang terkecil dahulu
    while (!h.empty()) { s += string(1, NM[h.top().second]) + ":" +
to_string(h.top().first) + " "; h.pop(); }
    return s.empty() ? "(empty)" : s;
}

void prim(int root) {
    int n = (int)NM.size();
    vector<vector<pair<int,int>>> adj(n);
    for (const Edge& e : G) { adj[e.a].push_back({e.b, e.w}); adj[e.b].push_back({e.a,
e.w}); }
    for (auto& l : adj) sort(l.begin(), l.end());               // tetangga secara alfabetis
    vector<int> key(n, INT_MAX), pi(n, -1);
    vector<char> inTree(n, 0);
    priority_queue<Entry, vector<Entry>, greater<>> Q;
    key[root] = 0; Q.push({0, root});
    int total = 0, pops = 0, stale = 0, pushes = 1, step = 0;
    cout << "=== Prim from " << NM[root] << " with a binary heap ===\n";
    while (!Q.empty()) {
        auto [k, u] = Q.top(); Q.pop(); ++pops;
        if (inTree[u] || k != key[u]) {                         // entri yang sudah basi
            ++stale;
            cout << "      pop " << NM[u] << ":" << k << "  stale, discarded\n";
            continue;
```

Kode 12.9 (Lanjutan) *Algoritma Prim dengan std::priority_queue dan DecreaseKey malas, mencetak tumpukannya sesudah setiap langkah dan mencacah entri basinya; dijalankan dari a dan dari i. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o primheap primheap.cpp.*

```
        }
        inTree[u] = 1; total += k;
        string edge = pi[u] < 0 ? "(root)" : string(1, NM[pi[u]]) + NM[u] + "(" +
to_string(k) + ")";
        string upd;
        for (auto [v, w] : adj[u])
            if (!inTree[v] && w < key[v]) {
                upd += string(1, NM[v]) + ":" + (key[v] == INT_MAX ? "inf" :
to_string(key[v]))
                        + "->" + to_string(w) + " ";
                key[v] = w; pi[v] = u; Q.push({w, v}); ++pushes;
            }
        cout << "step " << ++step << "  pop " << NM[u] << ":" << k << "  add " << left <<
setw(7) << edge
             << right << " total " << setw(2) << total << "  updates " << (upd.empty() ? "-
" : upd) << "\n"
             << "        heap: " << heapContents(Q) << "\n";
    }
    cout << "tree:";
    for (int v = 0; v < n; ++v) if (pi[v] >= 0) cout << " " << NM[pi[v]] << NM[v] << "(" <<
key[v] << ")";
    cout << "\ntotal weight " << total << "; heap pushes " << pushes << ", pops " << pops
         << ", stale pops " << stale << "\n";
}

int main() { prim(0); cout << "\n"; prim(8); return 0; }
```

```
=== Prim from a with a binary heap ===
step 1  pop a:0  add (root)  total  0  updates b:inf->4 h:inf->8
        heap: b:4 h:8
step 2  pop b:4  add ab(4)   total  4  updates c:inf->8
        heap: c:8 h:8
step 3  pop c:8  add bc(8)   total 12  updates d:inf->7 f:inf->4 i:inf->2
        heap: i:2 f:4 d:7 h:8
step 4  pop i:2  add ci(2)   total 14  updates g:inf->6 h:8->7
        heap: f:4 g:6 d:7 h:7 h:8
step 5  pop f:4  add cf(4)   total 18  updates e:inf->10 g:6->2
        heap: g:2 g:6 d:7 h:7 h:8 e:10
step 6  pop g:2  add fg(2)   total 20  updates h:7->1
        heap: h:1 g:6 d:7 h:7 h:8 e:10
step 7  pop h:1  add gh(1)   total 21  updates -
        heap: g:6 d:7 h:7 h:8 e:10
      pop g:6  stale, discarded
step 8  pop d:7  add cd(7)   total 28  updates e:10->9
        heap: h:7 h:8 e:9 e:10
      pop h:7  stale, discarded
      pop h:8  stale, discarded
step 9  pop e:9  add de(9)   total 37  updates -
        heap: e:10
      pop e:10  stale, discarded
tree: ab(4) bc(8) cd(7) de(9) cf(4) fg(2) gh(1) ci(2)
total weight 37; heap pushes 13, pops 13, stale pops 4

=== Prim from i with a binary heap ===
step 1  pop i:0  add (root)  total  0  updates c:inf->2 g:inf->6 h:inf->7
        heap: c:2 g:6 h:7
step 2  pop c:2  add ic(2)   total  2  updates b:inf->8 d:inf->7 f:inf->4
        heap: f:4 g:6 d:7 h:7 b:8
step 3  pop f:4  add cf(4)   total  6  updates e:inf->10 g:6->2
        heap: g:2 g:6 d:7 h:7 b:8 e:10
step 4  pop g:2  add fg(2)   total  8  updates h:7->1
        heap: h:1 g:6 d:7 h:7 b:8 e:10
step 5  pop h:1  add gh(1)   total  9  updates a:inf->8
        heap: g:6 d:7 h:7 a:8 b:8 e:10
      pop g:6  stale, discarded
step 6  pop d:7  add cd(7)   total 16  updates e:10->9
        heap: h:7 a:8 b:8 e:9 e:10
      pop h:7  stale, discarded
step 7  pop a:8  add ha(8)   total 24  updates b:8->4
        heap: b:4 b:8 e:9 e:10
step 8  pop b:4  add ab(4)   total 28  updates -
        heap: b:8 e:9 e:10
      pop b:8  stale, discarded
step 9  pop e:9  add de(9)   total 37  updates -
        heap: e:10
      pop e:10  stale, discarded
tree: ha(8) ab(4) ic(2) cd(7) de(9) cf(4) fg(2) gh(1)
total weight 37; heap pushes 13, pops 13, stale pops 4
```

Jawaban. Tiga belas dorongan, tiga belas pengeluaran, empat basi; pohon $\{ab, bc, ci, cf, fg, gh, cd, de\}$ berbobot 37; waktu jalan $O(E \lg V)$.

> **UJI DIRI.** Jalankan versi malas itu dari i alih-alih a (separuh kedua keluaran di atas). Apakah pohonnya sama?
>
> *Jawaban.* Bobotnya lagi-lagi **37**, dengan empat pengeluaran basi lagi, tetapi pohonnya **berbeda**: ia memuat ha 8 alih-alih bc 8. Dari i pohon yang tumbuh mencapai h (lewat gh 1) sebelum mencapai b, jadi a pertama kali tersambung lewat h, dengan kunci 8, dan b lalu masuk dengan murah lewat ab 4, sehingga bc tak terpakai. Inilah kedua pohon merentang minimum §11.12: seri antara ah dan bc, keduanya berbobot 8, diputuskan secara berbeda tergantung dari mana pohonnya bermula.

12.6.5 Persoalan Terkerjakan U — sifat potongan dan sifat siklus

Persoalan. (a) Nyatakan sifat potongan dengan tepat, dengan mendefinisikan *potongan*, *sisi yang melintas*, *menghormati*, dan *sisi ringan*, lalu buktikan. (b) Untuk potongan $S = \{A, B, C\}$ pada G, daftarlah setiap sisi yang melintas, sebutkan sisi ringannya, dan pastikan ia berada di pohon Persoalan R. (c) Nyatakan dan benarkan **sifat siklus**, dan untuk masing-masing dari keenam sisi bukan-pohon G sebutkan siklus yang ditutupnya dan sisi pohon terberat pada siklus itu.

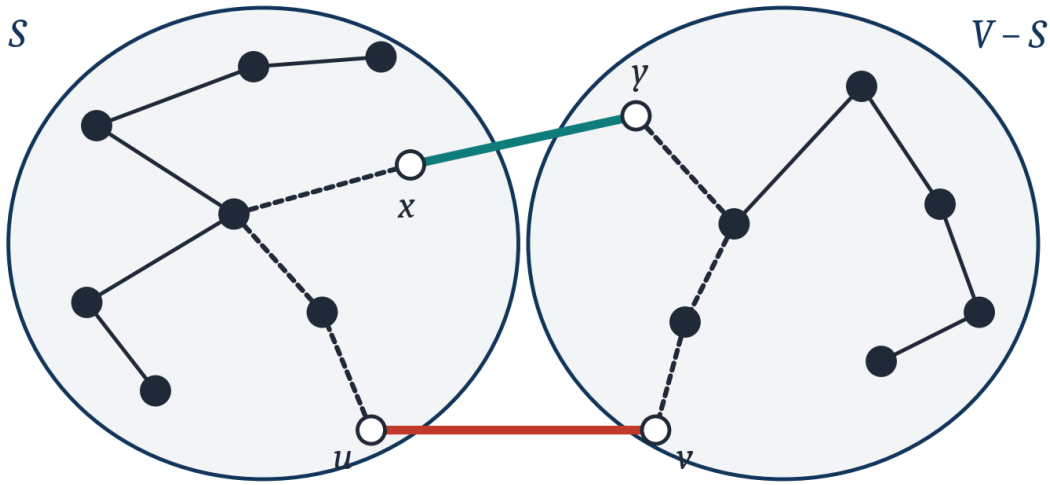
Yang diuji. Teorema yang membuat setiap algoritma pohon merentang minimum benar (§11.7), dan bayangan cerminnya.

Penyelesaian.

(a) Pernyataannya.

> **DEFINISI.** Sebuah **potongan** $(S, V - S)$ adalah partisi mana pun atas simpul-simpulnya menjadi dua bagian yang tidak kosong. Sebuah sisi $\{u, v\}$ **melintasi** potongan itu jika persis satu dari u, v berada di S. Sebuah himpunan sisi A **menghormati** potongan itu jika tak ada sisi A yang melintasinya. Sisi melintas berbobot minimum adalah **sisi ringan** bagi potongan itu (bisa ada beberapa).
>
> **SIFAT POTONGAN (Teorema 11.5, dalam bentuk §11.7.5).** Misalkan A adalah upahimpunan sisi-sisi suatu pohon merentang minimum G, misalkan $(S, V - S)$ adalah potongan mana pun yang dihormati A, dan misalkan (u, v) adalah sisi ringan bagi potongan itu. Maka (u, v) **aman** bagi A: artinya, $A \cup \{(u, v)\}$ juga merupakan upahimpunan suatu pohon merentang minimum.

Buktinya adalah argumen pertukaran, yang digambarkan pada **Gambar 12.20**. Ambillah pohon merentang minimum T yang memuat A dan andaikan sisi ringan (u, v) tidak berada di dalamnya. Menambahkan (u, v) ke T menciptakan persis satu siklus; siklus itu harus melintasi potongannya untuk kedua kalinya, pada suatu sisi (x, y). Karena A menghormati potongannya, $(x, y) \notin A$. Karena (u, v) ringan, $w(u, v) \leq w(x, y)$. Jadi $T' = T - \{(x, y)\} + \{(u, v)\}$ adalah pohon merentang dengan $w(T') \leq w(T)$; karena T minimum, $w(T') = w(T)$ dan T' juga minimum. T' memuat A dan (u, v). ■



Gambar 12.20 Pertukaran yang membuktikan sifat potongan. S dan $V - S$ adalah kedua gumpalannya; sisi merah (u, v) di bawah adalah sisi ringannya, yang diandaikan tidak berada di pohon; garis bertitik adalah lintasan pohon yang tunggal dari u ke v; dan sisi toska (x, y) adalah sisi pertama lintasan itu yang melintasi potongannya. Menukar (x, y) keluar demi (u, v) adalah seluruh buktinya.

(b) Potongan $S = \{A, B, C\}$. Telusurilah ketiga belas sisinya dan pertahankan yang persis satu ujungnya berada di S:

- AB — keduanya di S. Tidak. AC — keduanya di dalam. Tidak. BC — keduanya di dalam. Tidak.
- BD — B di dalam, D di luar. **Melintas**, bobot 6.
- CD — **melintas**, bobot 4.
- CE — **melintas**, bobot 9.
- $DE, DF, EF, EG, FG, FH, GH$ — kedua ujungnya di luar S. Tidak.

Sisi yang melintas: $\{BD\ 6,\ CD\ 4,\ CE\ 9\}$. Sisi ringannya adalah **CD** berbobot 4, dan **Tabel 12.31** memperlihatkannya diterima pada baris 5. Sifat potongan menjanjikan persis hal itu.

(c) Sifat siklusnya.

> **SIFAT SIKLUS.** Jika sebuah sisi e adalah sisi terberat yang **tunggal** pada suatu siklus G, maka e **tidak** termasuk dalam pohon merentang minimum mana pun. Jika ia sekadar seri sebagai yang terberat, maka suatu pohon merentang minimum mungkin memuatnya.

Mengapa: misalkan T adalah pohon merentang minimum yang memuat e semacam itu. Membuang e membelah T menjadi dua komponen; siklus yang melewati e harus memuat sisi lain e' yang menghubungkan kedua komponen itu, dan $w(e') < w(e)$ menurut hipotesisnya. Maka $T - e + e'$ adalah pohon merentang yang bobotnya tegas lebih kecil — sebuah kontradiksi.

Setiap sisi bukan-pohon menutup persis satu siklus bersama pohonnya, yaitu lintasan pohon di antara kedua ujungnya ditambah sisi itu sendiri; **Tabel 12.34** mendaftar keenamnya.

Tabel 12.34 Setiap sisi bukan-pohon G, siklus yang ditutupnya bersama pohon merentang minimumnya, dan sisi pohon terberat pada siklus itu. Keenamnya tegas terberat, jadi tak satu pun dapat muncul di pohon merentang minimum mana pun — dan itulah bukti lengkap bahwa pohon Persoalan R satu-satunya.

Sisi bukan-pohon	w	Siklus yang ditutupnya	Sisi pohon terberat di atasnya	Vonis
AC	5	A–B–C–A	3 (AB)	tegas terberat ⇒ tak di MST mana pun
BD	6	B–C–D–B	4 (CD)	tak di MST mana pun
CE	9	C–D–E–C	4 (CD)	tak di MST mana pun
DF	8	D–E–F–D	7 (EF)	tak di MST mana pun
EG	10	E–F–G–E	7 (EF)	tak di MST mana pun
FH	12	F–G–H–F	11 (GH)	tak di MST mana pun

Jawaban. Sisi ringan potongan $\{A, B, C\}$ adalah CD (4), dan ia berada di pohonnya; setiap sisi bukan-pohon tegas terberat pada siklusnya, jadi tak ada pohon merentang minimum yang memuat satu pun di antaranya.

> **UJI DIRI.** Ambillah potongan $S = \{A, B, C, D\}$. Sisi mana yang melintasinya, dan mana yang ringan?
>
> *Jawaban.* CE 9, DE 2, dan DF 8 melintasinya. Sisi ringannya adalah DE, berbobot 2, dan ia berada di pohonnya — diterima pada baris 2 **Tabel 12.31**.

12.6.6 Persoalan Terkerjakan V — penyatuan menurut peringkat dan pemampatan lintasan

Persoalan. Bertolak dari delapan himpunan tunggal $A, \ldots, H$, kerjakan `Union(A,B)`, `Union(C,D)`, `Union(E,F)`, `Union(G,H)`, `Union(A,C)`, `Union(E,G)`, `Union(A,E)`, dengan **penyatuan menurut peringkat beserta pemampatan lintasan**. (a) Berikan larik `parent` dan `rank` sesudah setiap operasi. (b) Lalu kerjakan `Find` pada setiap unsur dan berikan lariknya lagi. (c) Berapa ketinggian

terbesar yang mungkin bagi pohon atas 8 unsur dengan penyatuan menurut peringkat, dan berapa tanpanya?

Yang diuji. Kedua siasat §11.10.5 dan §11.10.6, dan batas yang membuat keduanya bekerja.

Penyelesaian.

(a) Ketujuh penyatuannya. `MakeSet` membiarkan setiap unsur menjadi induknya sendiri dengan peringkat 0: `parent = ABCDEFGH`, `rank = 00000000`. Pada penyatuan menurut peringkat, akar berperingkat *lebih rendah* ditempatkan di bawah akar berperingkat lebih tinggi; bila seri, yang mana pun boleh dipilih dan peringkat yang bertahan naik satu. Kesepakatannya di sini menempatkan akar argumen kedua di bawah akar argumen pertama bila peringkatnya seri.

Tabel 12.35 Tujuh penyatuan menurut peringkat. Peringkat sebuah akar naik hanya ketika dua peringkat yang sama bertemu — ketujuh penyatuan di sini kebetulan semuanya pertemuan semacam itu, dan itulah sebabnya strukturnya tetap seimbang sempurna.

Operasi	Akibat	parent	rank
`Union(A,B)`	peringkat seri di 0; B di bawah A, `rank[A]`$$1	`AACDEFGH`	`10000000`
`Union(C,D)`	peringkat seri di 0; D di bawah C	`AACCEFGH`	`10100000`
`Union(E,F)`	peringkat seri di 0; F di bawah E	`AACCEEGH`	`10101000`
`Union(G,H)`	peringkat seri di 0; H di bawah G	`AACCEEGG`	`10101010`
`Union(A,C)`	peringkat seri di 1; C di bawah A, `rank[A]`$$2	`AAACEEGG`	`20101010`
`Union(E,G)`	peringkat seri di 1; G di bawah E, `rank[E]`$$2	`AAACEEEG`	`20102010`
`Union(A,E)`	peringkat seri di 2; E di bawah A, `rank[A]`$$3	`AAACAEEG`	`30102010`

Sesudah penyatuan ketujuh, hutannya adalah satu pohon seimbang setinggi 3: A di akar, dengan B, C, dan E di bawahnya; D di bawah C; F dan G di bawah E; H di bawah G:

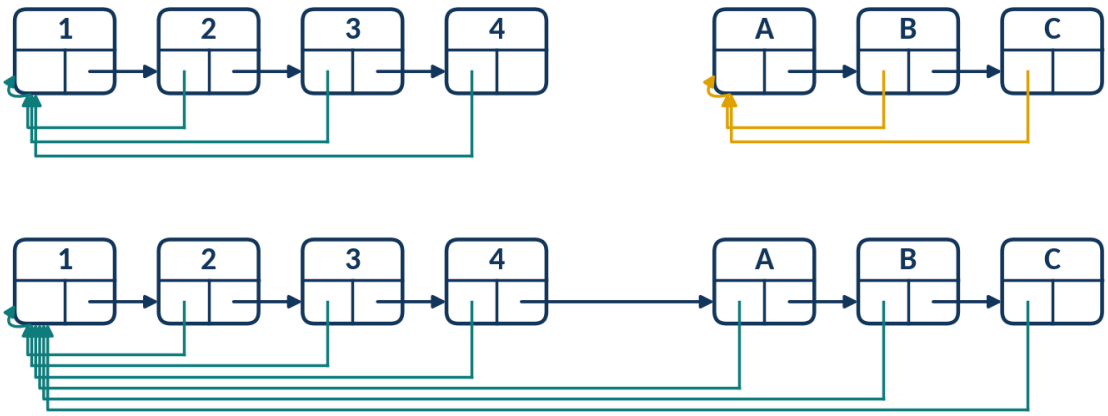
```
      A
    /  |  \
   B   C   E
       |   | \
       D   F  G
              |
              H
```

(b) Sesudah `Find` pada setiap unsur. Setiap `Find` mengarahkan ulang setiap simpul pada lintasannya langsung ke akar. Dikerjakan dalam urutan A sampai H: `Find(D)` berjalan $D \to C \to A$ dan mengarahkan ulang D; `Find(F)` berjalan $F \to E \to A$ dan mengarahkan ulang F; `Find(G)` berjalan $G \to E \to A$ dan mengarahkan ulang G; dan ketika `Find(H)` berjalan, G sudah menunjuk ke A, jadi ia berjalan $H \to G \to A$ dan mengarahkan ulang H. Hasilnya

`parent = AAAAAAAA`, `rank = 30102010`.

Setiap unsur kini menunjuk langsung ke A, dan setiap `Find` berikutnya berongkos satu langkah. Peringkatnya dibiarkan **basi** — `rank[C]` masih berbunyi 1 walaupun C kini sebuah daun — dan itu perilaku yang benar, bukan kutu: peringkat adalah batas atas ketinggian, dan pemampatan hanya dapat membuat ketinggian sesungguhnya lebih kecil.

Gambar 12.21 memperlihatkan penyajian lebih awal berbasis *daftar* atas struktur abstrak yang sama (§11.10.3), dan layak dibandingkan: menyatukan dua himpunan di sana berongkos waktu sebanding dengan ukuran daftar yang lebih kecil, sebab setiap unsurnya harus diarahkan ulang. Hutan pada persoalan ini mengerjakan tugas yang sama dalam waktu nyaris tetap dengan mengarahkan ulang hanya satu akar.



Gambar 12.21 Himpunan saling lepas sebagai daftar. **Atas:** dua himpunan, tiap unsur membawa penunjuk balik ke wakilnya. **Bawah:** hasil menyatukan keduanya — daftar yang lebih pendek disambungkan dan setiap unsurnya diarahkan ulang. Bandingkan ongkosnya dengan **Tabel 12.35**, yang di dalamnya sebuah penyatuan mengubah persis satu penunjuk.

(c) Ketinggiannya. Pada penyatuan menurut peringkat, sebuah pohon yang akarnya berperingkat r memuat sekurang-kurangnya 2^r unsur:

$$size(x) \geq 2^{rank(x)} \qquad \text{untuk setiap akar } x. \tag{12.19}$$

Bukti Persamaan (12.19) adalah induksi atas banyaknya penyatuan. Ia berlaku bagi himpunan tunggal ($1 \geq 2^0$). Penyatuan akar-akar berperingkat berbeda membiarkan peringkat yang lebih besar tak berubah dan hanya menambah unsur, jadi ketaksamaannya bertahan. Penyatuan dua akar berperingkat sama r menghasilkan akar berperingkat $r+1$ yang pohonnya memuat sekurang-kurangnya $2^r + 2^r = 2^{r+1}$ unsur. Karena ketinggian tak pernah melebihi peringkat, pohon atas n unsur tingginya paling banyak $\lfloor \lg n \rfloor$, dan untuk $n = 8$ itu $\lg 8 = \mathbf{3}$ — yang dicapai persis oleh jalannya di atas.

Tanpa penyatuan menurut peringkat, menempel secara membabi buta dapat membangun rantai: `Union(A,B)`, `Union(B,C)`, …, `Union(G,H)` yang dikerjakan secara naif memberikan lintasan delapan simpul, setinggi **7**, dan setiap `Find` lalu berongkos $\Theta(n)$. Selisih terukurnya mencolok. Pada 100.000 unsur yang dirantai dengan cara yang sama, jalannya tanpa kedua siasat itu mengikuti 4 999 950 000 penunjuk induk untuk mengerjakan satu `Find` per unsur; dengan penyatuan menurut peringkat dan pemampatan lintasan pekerjaan yang sama berongkos 99 999 — faktor lima puluh ribu (Soal Latihan 12.11). Batas teramortisasi di baliknya, $O(m\,\alpha(n))$ untuk m operasi, berasal dari Tarjan [11].

Jawaban. Larik-larik pada **Tabel 12.35**; sesudah pencarian-pencariannya, setiap induk adalah A dan peringkatnya tak berubah; ketinggiannya paling banyak 3 dengan penyatuan menurut peringkat dan dapat mencapai 7 tanpanya.

> **UJI DIRI.** Bertolak lagi dari delapan himpunan tunggal, kerjakan `Union(A,B)`, `Union(B,C)`, `Union(C,D)`, …, `Union(G,H)` — barisan perantai itu — tetapi *dengan* penyatuan menurut peringkat. Bagaimana lariknya di akhir, dan berapa ketinggiannya?
>
> *Jawaban.* Penyatuan pertama seri di peringkat 0, jadi B ditempatkan di bawah A dan `rank[A]` menjadi 1. Setiap penyatuan berikutnya `Union(X,Y)` mendapati akar A (peringkat 1) untuk X dan sebuah himpunan tunggal (peringkat 0) untuk Y, jadi Y ditempatkan langsung di bawah A. Akhirnya `parent = AAAAAAAA`, `rank = 10000000`, ketinggian **1**. Penyatuan menurut peringkat sendirian sudah mengalahkan rantainya, sebelum pemampatan lintasan mengerjakan apa pun.

12.6.7 Persoalan Terkerjakan W — apakah pohon merentang minimumnya tunggal?

Persoalan. (a) Apakah pohon merentang minimum G tunggal? Benarkan dengan menghapus setiap sisi pohon bergiliran dan menghitung ulang. (b) Dua sisi G berbobot sama. Apakah hal itu membuat pohonnya tidak tunggal? Jelaskan. (c) Turunkan $w(AC)$ dari 5 menjadi 3 lalu jawab (a) lagi.

Yang diuji. §11.12: beda antara "ada seri" dan "pohonnya tidak tunggal".

Penyelesaian.

(a) Ujinya. Uji penghapusan sisi adalah uji yang langsung: untuk setiap sisi pohon e, hapuslah ia dari G lalu hitung ulang pohon merentang minimumnya. Jika pohon yang baru seberat yang lama, ada pohon merentang minimum alternatif yang menghindari e; jika setiap penghapusan membuat keadaannya tegas lebih buruk, pohonnya tunggal.

Tabel 12.36 Uji ketunggalan pada G: setiap sisi pohon dihapus bergiliran lalu Kruskal dijalankan ulang. Setiap baris tegas lebih berat daripada 31, jadi pohon merentang minimum G **tunggal**.

Sisi dihapus	w	Pohon merentang terbaik yang baru	Vonis
BC	1	35	lebih berat 4
DE	2	37	lebih berat 6
AB	3	33	lebih berat 2
FG	3	38	lebih berat 7
CD	4	33	lebih berat 2
EF	7	32	lebih berat 1
GH	11	32	lebih berat 1

Tujuh baris **Tabel 12.36**, tujuh kenaikan tegas, jadi pohonnya tunggal. Perhatikan baris terakhirnya: menghapus GH memaksakan satu-satunya jalan lain ke H, yaitu FH pada 12, yang menaikkan totalnya persis $12 - 11 = 1$. Tabel sifat siklus **Tabel 12.34** membuktikan hal yang sama dengan lebih murah, dalam satu lintasan.

(b) Bobot yang sama. AB dan FG sama-sama berbobot 3, namun pohonnya tetap tunggal. Jadi **bobot yang berulang tidak berarti pohonnya tidak tunggal.** Pernyataan yang benar adalah:

- Jika semua bobot sisi **berbeda**, pohon merentang minimumnya tunggal (§11.12.2). Keberbedaan itu *cukup*.
- Ia tidak *perlu*. Yang perlu sekaligus cukup adalah syarat sifat siklus **Tabel 12.34**: pohonnya tunggal persis ketika setiap sisi bukan-pohon adalah sisi **tegas** terberat pada siklus yang ditutupnya.

Di sini AB dan FG seri, tetapi keduanya terletak di bagian graf yang berbeda dan tak ada siklus yang memuat keduanya, jadi serinya tak pernah memaksakan pilihan.

(c) Menurunkan $w(AC)$ menjadi 3. Kini segitiga A–B–C membawa bobot AB 3, BC 1, AC 3. Pohon merentang minimumnya masih berbobot **31** — Kruskal kini menjumpai AB dan AC bersama-sama pada bobot 3, mengambil salah satunya, dan menolak yang lain — tetapi pilihannya sungguh-sungguh bebas. Menghapus AB lalu menghitung ulang masih memberikan 31, dengan AC menggantikannya. Jadi kini ada **dua** pohon merentang minimum:

$$\{BC, DE, AB, FG, CD, EF, GH\} \quad \text{dan} \quad \{BC, DE, AC, FG, CD, EF, GH\},$$

keduanya berbobot 31. Uji sifat siklusnya sepakat: AC tak lagi *tegas* terberat pada A–B–C–A; ia seri dengan AB.

Jawaban. Tunggal, sebab setiap penghapusan membuat pohonnya lebih berat; bobot berulang tidak dengan sendirinya meruntuhkan ketunggalan; dengan $w(AC) = 3$ ada persis dua pohon merentang minimum.

> **UJI DIRI.** Kembalikan $w(AC) = 5$, lalu turunkan $w(FH)$ dari 12 menjadi 11. Apakah pohonnya masih tunggal?
>
> *Jawaban.* **Tidak.** FH menutup siklus F–G–H–F, yang sisi pohon terberatnya GH pada 11. Kini FH juga berbobot 11: ia seri alih-alih tegas terberat, jadi $\{BC, DE, AB, FG, CD, EF, FH\}$ adalah pohon merentang minimum kedua berbobot 31. Mencacah setiap pohon merentang dengan gaya kasar mendapati persis dua yang berbobot 31.

12.6.8 Persoalan R, S, U, V, dan W sebagai satu program

Kode 12.10 menjalankan Persoalan R, S, U, V, dan W, dengan mencetak hutan himpunan saling lepas sesudah setiap keputusan Kruskal dan simpul-simpul yang menunggu sesudah setiap pengambilan Prim.

Kode 12.10 Persoalan Terkerjakan R, S, U, V, dan W: penelusuran Kruskal beserta hutan himpunan saling lepasnya, penelusuran Prim beserta larik kuncinya, sifat potongan dan sifat siklus, barisan penyatuan menurut peringkat, dan uji ketunggalan yang dijalankan dua kali. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o mst mst.cpp`.

```
// Persoalan Terkerjakan R, S, U, V, dan W Bab 12 -- pohon merentang minimum.
//   R  Kruskal, dengan hutan himpunan-lepas dicetak setelah setiap keputusan
//   S  Prim dari A, dengan larik key dicetak setelah setiap pengambilan
//   U  sifat potongan dan sifat siklus, diperiksa sisi demi sisi
//   V  penyatuan menurut peringkat dan pemampatan lintasan, dengan tangan dan dengan mesin
//   W  apakah pohon merentang minimumnya tunggal?
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <numeric>
#include <climits>
using namespace std;

static const string NM = "ABCDEFGH";
struct E { int a, b, w; };
static const vector<E> G = {
    {0,1,3},{0,2,5},{1,2,1},{1,3,6},{2,3,4},{2,4,9},{3,4,2},
    {3,5,8},{4,5,7},{4,6,10},{5,6,3},{5,7,12},{6,7,11}
};
static const int N = 8;
static string nm(const E& e) { return string(1, NM[e.a]) + "-" + string(1, NM[e.b]); }

struct DSU {
    vector<int> p, r;
    explicit DSU(int n) : p(n), r(n, 0) { iota(p.begin(), p.end(), 0); }
    int find(int x) { return p[x] == x ? x : p[x] = find(p[x]); }
    bool unite(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return false;
        if (r[x] < r[y]) swap(x, y);
        p[y] = x; if (r[x] == r[y]) ++r[x];
        return true;
    }
    string show() {
        string s = "p:"; for (int i = 0; i < (int)p.size(); ++i) s += string(1, NM[p[i]]);
        s += "  r:"; for (int x : r) s += to_string(x);
        return s;
    }
};
```

Kode 12.10 (Lanjutan) *Persoalan Terkerjakan R, S, U, V, dan W: penelusuran Kruskal beserta hutan himpunan saling lepasnya, penelusuran Prim beserta larik kuncinya, sifat potongan dan sifat siklus, barisan penyatuan menurut peringkat, dan uji ketunggalan yang dijalankan dua kali. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o mst mst.cpp.*

```
vector<E> sorted_edges(vector<E> e) {
    sort(e.begin(), e.end(), [](const E& x, const E& y) {
        if (x.w != y.w) return x.w < y.w;
        if (x.a != y.a) return x.a < y.a;
        return x.b < y.b; });
    return e;
}

int kruskalWeight(const vector<E>& edges, int& taken) {
    DSU d(N); int t = 0; taken = 0;
    for (const E& e : sorted_edges(edges)) if (d.unite(e.a, e.b)) { t += e.w; ++taken; }
    return t;
}

int main() {
    cout << "=== Problem R  Kruskal ===\n";
    vector<E> S = sorted_edges(G);
    cout << "  sorted edge list:";
    for (const E& e : S) cout << " " << nm(e) << "(" << e.w << ")";
    cout << "\n";
    DSU d(N);
    int total = 0, taken = 0; vector<E> mst;
    cout << "  step  edge   w  find(a) find(b)  verdict   disjoint-set forest\n";
    int step = 0;
    for (const E& e : S) {
        int fa = d.find(e.a), fb = d.find(e.b);
        bool ok = d.unite(e.a, e.b);
        if (ok) { total += e.w; ++taken; mst.push_back(e); }
        cout << setw(6) << ++step << "  " << setw(4) << nm(e) << setw(4) << e.w
             << setw(8) << NM[fa] << setw(8) << NM[fb] << "  "
             << left << setw(10) << (ok ? "ACCEPT" : "reject") << right
             << d.show() << "\n";
        if (taken == N - 1) { cout << "  |V|-1 = " << N-1 << " edges taken; the remaining "
                                << S.size() - step << " edges cannot be needed\n";
break; }
    }
    cout << "  MST:"; for (const E& e : mst) cout << " " << nm(e) << "(" << e.w << ")";
    cout << "\n  total weight " << total << "\n";
```

Kode 12.10 (Lanjutan) *Persoalan Terkerjakan R, S, U, V, dan W: penelusuran Kruskal beserta hutan himpunan saling lepasnya, penelusuran Prim beserta larik kuncinya, sifat potongan dan sifat siklus, barisan penyatuan menurut peringkat, dan uji ketunggalan yang dijalankan dua kali. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o mst mst.cpp.*

```
    cout << "\n=== Problem S  Prim from A ===\n";
    vector<vector<pair<int,int>>> adj(N);
    for (const E& e : G) { adj[e.a].push_back({e.b, e.w}); adj[e.b].push_back({e.a, e.w});
}
    for (auto& l : adj) sort(l.begin(), l.end());
    vector<int> key(N, INT_MAX), pi(N, -1); vector<char> inQ(N, 1);
    key[0] = 0;
    int ptotal = 0; vector<string> ptree;
    cout << "  step  extract  edge committed  running total  queue (v:key/pi)\n";
    for (int s = 1; s <= N; ++s) {
        int u = -1;
        for (int v = 0; v < N; ++v) if (inQ[v] && (u == -1 || key[v] < key[u])) u = v;
        inQ[u] = 0; ptotal += key[u];
        string ed = (pi[u] == -1) ? "(root)"
                  : string(1, NM[pi[u]]) + "-" + string(1, NM[u]) + "(" + to_string(key[u])
+ ")";
        if (pi[u] != -1) ptree.push_back(string(1, NM[pi[u]]) + "-" + string(1, NM[u]));
        for (auto [v, w] : adj[u]) if (inQ[v] && w < key[v]) { key[v] = w; pi[v] = u; }
        string q;
        vector<int> rest;
        for (int v = 0; v < N; ++v) if (inQ[v]) rest.push_back(v);
        sort(rest.begin(), rest.end(), [&](int x, int y){ return key[x] < key[y]; });
        for (int v : rest) q += string(1, NM[v]) + ":" +
            (key[v] == INT_MAX ? "inf" : to_string(key[v])) +
            (pi[v] == -1 ? "" : "/" + string(1, NM[pi[v]])) + " ";
        cout << setw(6) << s << setw(9) << NM[u] << setw(16) << ed
             << setw(15) << ptotal << "  " << (q.empty() ? "(empty)" : q) << "\n";
    }
    cout << "  Prim's tree:"; for (const string& e : ptree) cout << " " << e;
    cout << "\n  total weight " << ptotal
         << (ptotal == total ? "  -- the same as Kruskal's\n" : "  -- DISAGREES\n");

    cout << "\n=== Problem U  the cut property and the cycle property ===\n";
    vector<int> Sset = {0, 1, 2};                           // potongan S = {A,B,C}
    cout << "  cut S = {A,B,C}, V-S = {D,E,F,G,H}\n  crossing edges:";
    int light = INT_MAX; string le;
    for (const E& e : G) {
        bool ina = find(Sset.begin(), Sset.end(), e.a) != Sset.end();
        bool inb = find(Sset.begin(), Sset.end(), e.b) != Sset.end();
        if (ina != inb) { cout << " " << nm(e) << "(" << e.w << ")";
                          if (e.w < light) { light = e.w; le = nm(e); } }
    }
    cout << "\n  the light edge is " << le << " of weight " << light << "\n";
    bool inMst = false; for (const E& e : mst) if (nm(e) == le) inMst = true;
    cout << "  is it in the MST found in Problem R? " << (inMst ? "yes" : "no")
         << "  -- exactly as Theorem 11.5 promises\n";
    cout << "  the cycle property, checked on every non-tree edge:\n";
    for (const E& e : G) {
```

Kode 12.10 (Lanjutan) *Persoalan Terkerjakan R, S, U, V, dan W: penelusuran Kruskal beserta hutan himpunan saling lepasnya, penelusuran Prim beserta larik kuncinya, sifat potongan dan sifat siklus, barisan penyatuan menurut peringkat, dan uji ketunggalan yang dijalankan dua kali. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o mst mst.cpp.*

```
        bool tree = false; for (const E& t : mst) if (t.a == e.a && t.b == e.b) tree =
true;
        if (tree) continue;
        vector<vector<int>> t(N);
        for (const E& x : mst) { t[x.a].push_back(x.b); t[x.b].push_back(x.a); }
        vector<int> par(N, -2); queue<int> Q; Q.push(e.a); par[e.a] = -1;
        while (!Q.empty()) { int u = Q.front(); Q.pop();
            for (int v : t[u]) if (par[v] == -2) { par[v] = u; Q.push(v); } }
        vector<int> path; for (int x = e.b; x != -1; x = par[x]) path.push_back(x);
        int heaviest = 0; string ps;
        for (size_t i = 0; i + 1 < path.size(); ++i) {
            for (const E& x : mst)
                if ((x.a == path[i] && x.b == path[i+1]) || (x.b == path[i] && x.a ==
path[i+1]))
                    heaviest = max(heaviest, x.w);
            ps += string(1, NM[path[i]]) + "-";
        }
        ps += string(1, NM[path.back()]);
        cout << "    " << nm(e) << "(" << e.w << "): closes the cycle " << ps
             << ", whose heaviest tree edge weighs " << heaviest
             << "  -> " << (e.w > heaviest ? "strictly heaviest, so in NO MST"
                                          : "ties, so a second MST exists") << "\n";
    }

    cout << "\n=== Problem V  union by rank and path compression ===\n";
    DSU u1(8);
    vector<pair<int,int>> ops = {{0,1},{2,3},{4,5},{6,7},{0,2},{4,6},{0,4}};
    cout << "  operation   result   forest\n";
    for (auto [x, y] : ops) {
        bool ch = u1.unite(x, y);
        cout << "  Union(" << NM[x] << "," << NM[y] << ")   "
             << left << setw(9) << (ch ? "merged" : "no-op") << right << u1.show() << "\n";
    }
    cout << "  Find on every element, to show path compression:\n    ";
    for (int i = 0; i < 8; ++i) cout << NM[i] << "->" << NM[u1.find(i)] << " ";
    cout << "\n    " << u1.show() << "\n";
    cout << "  the tallest possible tree on 8 elements with union by rank has height "
         << 3 << " = lg 8; without it, a chain of 8 would have height 7\n";
```

Kode 12.10 (Lanjutan) *Persoalan Terkerjakan R, S, U, V, dan W: penelusuran Kruskal beserta hutan himpunan saling lepasnya, penelusuran Prim beserta larik kuncinya, sifat potongan dan sifat siklus, barisan penyatuan menurut peringkat, dan uji ketunggalan yang dijalankan dua kali. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o mst mst.cpp.*

```
    cout << "\n=== Problem W  is the minimum spanning tree unique? ===\n";
    int t0; int w0 = kruskalWeight(G, t0);
    cout << "  the MST weighs " << w0 << "\n";
    int alt = 0;
    for (const E& e : mst) {
        vector<E> without;
        for (const E& x : G) if (!(x.a == e.a && x.b == e.b)) without.push_back(x);
        int tk; int w = kruskalWeight(without, tk);
        cout << "    delete " << nm(e) << "(" << e.w << "): best spanning tree now weighs
";
        if (tk != N - 1) cout << "-- disconnected --\n";
        else { cout << w << (w == w0 ? "   <= SAME, so a second MST exists\n" : "
heavier\n");
               if (w == w0) ++alt; }
    }
    cout << "  rows with the same weight: " << alt
         << (alt == 0 ? "  -> the MST is UNIQUE\n" : "  -> the MST is NOT unique\n");
    cout << "  note: the weights are NOT all distinct (A-B and F-G both weigh 3),\n"
            "  and yet the tree is unique. Distinct weights are sufficient, not
necessary.\n";
    vector<E> G2 = G; for (E& e : G2) if (e.a == 0 && e.b == 2) e.w = 3;   // A-C: 5 -> 3
    int t2; int w2 = kruskalWeight(G2, t2);
    cout << "  now lower A-C from 5 to 3: the MST weighs " << w2 << "; deleting each tree
edge:\n";
    DSU dd(N); vector<E> mst2;
    for (const E& e : sorted_edges(G2)) if (dd.unite(e.a, e.b)) mst2.push_back(e);
    int alt2 = 0;
    for (const E& e : mst2) {
        vector<E> without;
        for (const E& x : G2) if (!(x.a == e.a && x.b == e.b)) without.push_back(x);
        int tk; int w = kruskalWeight(without, tk);
        if (tk == N - 1 && w == w2) { ++alt2;
            cout << "    dropping " << nm(e) << " still gives " << w << " -- a second
MST\n"; }
    }
    cout << "  alternatives found: " << alt2 << "\n";
    return 0;
}
```

Keluaran lengkapnya mereproduksi setiap tabel di atas, termasuk kehalusan baris 11 **Tabel 12.31**:

```
=== Problem R  Kruskal ===
  sorted edge list: B-C(1) D-E(2) A-B(3) F-G(3) C-D(4) A-C(5) B-D(6) E-F(7) D-F(8) C-E(9)
E-G(10) G-H(11) F-H(12)
  step  edge   w  find(a) find(b)  verdict   disjoint-set forest
     1   B-C   1       B       C  ACCEPT    p:ABBDEFGH  r:01000000
     2   D-E   2       D       E  ACCEPT    p:ABBDDFGH  r:01010000
     3   A-B   3       A       B  ACCEPT    p:BBBDDFGH  r:01010000
     4   F-G   3       F       G  ACCEPT    p:BBBDDFFH  r:01010100
     5   C-D   4       B       D  ACCEPT    p:BBBBDFFH  r:02010100
     6   A-C   5       B       B  reject    p:BBBBDFFH  r:02010100
     7   B-D   6       B       B  reject    p:BBBBDFFH  r:02010100
     8   E-F   7       B       F  ACCEPT    p:BBBBBBFH  r:02010100
     9   D-F   8       B       B  reject    p:BBBBBBFH  r:02010100
    10   C-E   9       B       B  reject    p:BBBBBBFH  r:02010100
    11   E-G  10       B       B  reject    p:BBBBBBBH  r:02010100
    12   G-H  11       B       H  ACCEPT    p:BBBBBBBB  r:02010100
  |V|-1 = 7 edges taken; the remaining 1 edges cannot be needed
  MST: B-C(1) D-E(2) A-B(3) F-G(3) C-D(4) E-F(7) G-H(11)
  total weight 31

=== Problem S  Prim from A ===
  step  extract  edge committed  running total  queue (v:key/pi)
     1        A          (root)              0  B:3/A C:5/A D:inf E:inf F:inf G:inf H:inf
     2        B          A-B(3)              3  C:1/B D:6/B E:inf F:inf G:inf H:inf
     3        C          B-C(1)              4  D:4/C E:9/C F:inf G:inf H:inf
     4        D          C-D(4)              8  E:2/D F:8/D G:inf H:inf
     5        E          D-E(2)             10  F:7/E G:10/E H:inf
     6        F          E-F(7)             17  G:3/F H:12/F
     7        G          F-G(3)             20  H:11/G
     8        H         G-H(11)             31  (empty)
  Prim's tree: A-B B-C C-D D-E E-F F-G G-H
  total weight 31  -- the same as Kruskal's

=== Problem U  the cut property and the cycle property ===
  cut S = {A,B,C}, V-S = {D,E,F,G,H}
  crossing edges: B-D(6) C-D(4) C-E(9)
  the light edge is C-D of weight 4
  is it in the MST found in Problem R? yes  -- exactly as Theorem 11.5 promises
  the cycle property, checked on every non-tree edge:
    A-C(5): closes the cycle C-B-A, whose heaviest tree edge weighs 3  -> strictly
heaviest, so in NO MST
    B-D(6): closes the cycle D-C-B, whose heaviest tree edge weighs 4  -> strictly
heaviest, so in NO MST
    C-E(9): closes the cycle E-D-C, whose heaviest tree edge weighs 4  -> strictly
heaviest, so in NO MST
    D-F(8): closes the cycle F-E-D, whose heaviest tree edge weighs 7  -> strictly
heaviest, so in NO MST
    E-G(10): closes the cycle G-F-E, whose heaviest tree edge weighs 7  -> strictly
heaviest, so in NO MST
    F-H(12): closes the cycle H-G-F, whose heaviest tree edge weighs 11  -> strictly
heaviest, so in NO MST
```

```
=== Problem V  union by rank and path compression ===
  operation   result   forest
  Union(A,B)   merged   p:AACDEFGH  r:10000000
  Union(C,D)   merged   p:AACCEFGH  r:10100000
  Union(E,F)   merged   p:AACCEEGH  r:10101000
  Union(G,H)   merged   p:AACCEEGG  r:10101010
  Union(A,C)   merged   p:AAACEEGG  r:20101010
  Union(E,G)   merged   p:AAACEEEG  r:20102010
  Union(A,E)   merged   p:AAACAEEG  r:30102010
  Find on every element, to show path compression:
    A->A B->A C->A D->A E->A F->A G->A H->A
    p:AAAAAAAA  r:30102010
  the tallest possible tree on 8 elements with union by rank has height 3 = lg 8; without
it, a chain of 8 would have height 7

=== Problem W  is the minimum spanning tree unique? ===
  the MST weighs 31
    delete B-C(1): best spanning tree now weighs 35   heavier
    delete D-E(2): best spanning tree now weighs 37   heavier
    delete A-B(3): best spanning tree now weighs 33   heavier
    delete F-G(3): best spanning tree now weighs 38   heavier
    delete C-D(4): best spanning tree now weighs 33   heavier
    delete E-F(7): best spanning tree now weighs 32   heavier
    delete G-H(11): best spanning tree now weighs 32   heavier
  rows with the same weight: 0  -> the MST is UNIQUE
  note: the weights are NOT all distinct (A-B and F-G both weigh 3),
  and yet the tree is unique. Distinct weights are sufficient, not necessary.
  now lower A-C from 5 to 3: the MST weighs 31; deleting each tree edge:
    dropping A-B still gives 31 -- a second MST
  alternatives found: 1
```

12.7 Latihan Uji Diri Singkat

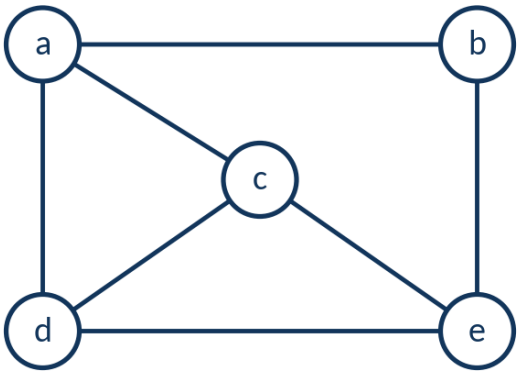
Tiga puluh pertanyaan dengan jawaban satu baris, meliputi setiap gagasan Bab 9, 10, dan 11. Tutuplah jawabannya, kerjakan menurun sepanjang daftarnya, dan jujurlah kepada diri sendiri. Apa pun yang terlewat menyebut subbab yang perlu dibaca ulang; rujukannya diberikan di akhir setiap jawaban, dan **Tabel 12.1** menunjuk persoalan terkerjakan yang sejenis.

1. **Dua bahan apa yang harus dimiliki sebuah persoalan sebelum algoritma *greedy* dapat dibuktikan benar?** Sifat pilihan rakus dan *optimal substructure*. (§9.5)
2. **Nyatakan sifat pilihan rakus dalam satu kalimat.** Penyelesaian optimal menyeluruh dapat dicapai dengan mengambil pilihan yang optimal setempat lebih dahulu — artinya, *suatu* penyelesaian optimal memuat pilihan rakusnya. (§9.5.2)
3. **Maksimal atau maksimum — apa bedanya?** Maksimal berarti tak ada lagi yang dapat ditambahkan; maksimum berarti tak ada yang lebih besar. Kerakusan memberikan maksimal secara gratis; seluruh beban pembuktiannya ada pada memperoleh maksimum. (§9.4.2)
4. **Aturan pemilihan aktivitas mana yang benar, dan berapa biaya algoritmanya?** Waktu selesai terawal; $\Theta(n\lg n)$ untuk mengurutkan, lalu $\Theta(n)$ untuk satu sapuan — $\Theta(n)$ saja jika masukannya sudah terurut. (§9.4.12)
5. **Berilah sistem koin dan jumlah yang di situ aturan kembalian rakus gagal.** Koin $\{1,3,4\}$ dan jumlah 6: kerakusan mengambil $4 + 1 + 1$ (tiga koin) padahal $3 + 3$ (dua) yang optimal. (§9.3.3)
6. **Bahan yang mana yang tidak dimiliki *knapsack* 0-1?** Sifat pilihan rakus. Ia punya *optimal substructure*, dan itulah sebabnya pemrograman dinamis berhasil padanya. (§9.6.4)

7. **Berapa biaya algoritma Huffman, dan mengapa?** $O(n\lg n)$: satu pembangunan *heap* seharga $O(n)$, lalu $n-1$ iterasi yang masing-masing mengerjakan tiga operasi *heap* seharga $O(\lg n)$. (§9.7.11)
8. **Dua cara yang saling bebas untuk menghitung** $B(T)$**?** Jumlahkan $f(c) \times$ (kedalaman c) atas daunnya; atau jumlahkan label seluruh simpul dalamnya. Keduanya harus sepakat. (§9.7.5)
9. **Ketiga syarat dalam definisi matroid?** S berhingga; $\mathcal{I}$ bersifat turunan (upahimpunan himpunan bebas juga bebas); sifat pertukaran (jika $|A| < |B|$ dengan keduanya bebas, ada $x \in B \setminus A$ yang membuat $A \cup \{x\}$ bebas). (§9.8.2)
10. **Pada penjadwalan satuan waktu bertenggat dan berpenalti, himpunan tugas mana yang bebas?** Yang semuanya dapat dijadwalkan awal — setara dengan yang memenuhi $N_t(A) \leq t$ untuk tiap t, dengan $N_t(A)$ mencacah tugas A yang bertenggat paling lambat t. (§9.8.6)
11. **Nyatakan lema jabat tangan.** $\sum_v \deg(v) = 2|E|$, sebab setiap sisi menyumbang satu pada derajat masing-masing kedua ujungnya. (§10.4.3)
12. **Ruang untuk daftar ketetanggaan lawan matriks ketetanggaan?** $\Theta(V+E)$ lawan $\Theta(V^2)$. Pada sejuta simpul, matriksnya 10^{12} entri sedangkan daftarnya boleh jadi 10^6. (§10.5.5)
13. **Waktu jalan BFS dan DFS?** Keduanya $\Theta(V+E)$ di atas daftar ketetanggaan, dan $\Theta(V^2)$ di atas matriks ketetanggaan. (§10.7.7, §10.8.7)
14. **Tiga hal apa yang dihitung BFS?** Jarak $d[v]$ dari sumbernya, pendahulu $\pi[v]$, dan — lewat simpul mana yang punya d berhingga — himpunan yang terjangkau dari sumbernya. (§10.7.2)
15. **Mengapa jarak BFS merupakan jarak lintasan terpendek?** Simpulnya meninggalkan antrean menurut d yang tidak menurun dan setiap simpul ditemukan dari tetangga yang satu tingkat lebih dekat, jadi $d[v] = \delta(s, v)$ lewat induksi atas δ. (§10.7.9)
16. **Ketiga warna DFS, dalam satu baris.** Putih: belum ditemukan. Kelabu: sudah ditemukan, masih di tumpukan rekursi. Hitam: sudah selesai, seluruh upapohonnya sudah dijelajahi. (§10.8.2)
17. **Nyatakan teorema kurung.** Untuk dua simpul mana pun, selang $[d, f]$-nya entah tersarang seluruhnya entah lepas seluruhnya; keduanya tak pernah tumpang tindih sebagian. Tersarang berarti keleluhuran. (§10.8.8)
18. **Nyatakan teorema lintasan putih.** v menjadi keturunan u di dalam hutan mendalamnya bila dan hanya bila, pada saat u ditemukan, v terjangkau dari u lewat lintasan yang seluruh simpulnya putih. (§10.8.9)
19. **Keempat golongan sisi, dan mana yang dapat muncul pada graf tak berarah?** Pohon, mundur, maju, silang. Pada graf tak berarah hanya sisi pohon dan sisi mundur yang muncul. (§10.8.10)
20. **Berapa biaya menjawab "apakah** u **dan** v **terhubung?" untuk banyak pasangan?** $\Theta(V+E)$ sekali untuk melabeli tiap simpul dengan komponennya, lalu $O(1)$ per kueri. (§10.9.1)
21. **Nyatakan kriteria kebipartitan.** Sebuah graf bipartit bila dan hanya bila ia tidak memuat siklus ganjil; setara dengan itu, bila dan hanya bila pewarnaan dua warna menurut keparian jarak BFS berhasil. (§10.9.3)
22. **Nyatakan teorema DAG.** Sebuah graf berarah taksiklik bila dan hanya bila pencarian mendalam atasnya tidak menghasilkan sisi mundur. (§11.2.3)
23. **Kedua algoritma pengurutan topologis dan ongkosnya?** Pencarian mendalam dengan simpulnya didaftar menurut waktu selesai menurun, dan pembuangan berulang simpul berderajat masuk nol ala Kahn. Keduanya $\Theta(V+E)$. (§11.3)
24. **Mengapa** f **menurun dan bukan** d **menaik?** Sebab untuk setiap busur $u \to v$ pada sebuah DAG, $f[v] < f[u]$ selalu, sedangkan $d[v] < d[u]$ dapat gagal. (§11.3.6)

25. **Berapa sisi yang dimiliki sebuah pohon merentang, dan berapa pohon merentang yang dimiliki K_n?** $|V| - 1$ sisi; n^{n-2} pohon merentang, menurut rumus Cayley. (§11.4)
26. **Nyatakan sifat potongan.** Jika A terletak di dalam suatu pohon merentang minimum dan potongan $(S, V - S)$ menghormati A, maka sisi ringan mana pun yang melintasi potongan itu aman bagi A. (§11.7.2)
27. **Apa itu sisi aman?** Sisi yang dapat ditambahkan pada himpunan yang sekarang sambil menjaganya tetap menjadi upahimpunan suatu pohon merentang minimum. (§11.8.1)
28. **Biaya Prim lawan biaya Kruskal?** Prim: $O(E\lg V)$ dengan *heap* biner, $O(E + V\lg V)$ dengan *heap* Fibonacci. Kruskal: $O(E\lg E) = O(E\lg V)$, didominasi pengurutannya. (§11.9.7, §11.10.12)
29. **Apa yang dibeli penyatuan menurut peringkat dan pemampatan lintasan?** Barisan m operasi apa pun atas n elemen berongkos $O(m\,\alpha(n))$, dan $\alpha(n) \leq 4$ untuk setiap n yang akan pernah dijalankan. (§11.10.7)
30. **Kapan pohon merentang minimumnya tunggal?** Persis ketika setiap sisi bukan-pohon adalah sisi yang *tegas* terberat pada siklus yang ditutupnya. Bobot yang berbeda-beda cukup bagi hal ini tetapi tidak perlu. (§11.12, §12.6.7)

Satu gambar layak ada di depan Anda selagi mengerjakan daftar itu: **Gambar 12.22** memuat setiap kata kosakata graf di balik latihan 11 sampai 21.



$$V = \{a, b, c, d, e\}$$

$$E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$$

Gambar 12.22 Definisi graf, dengan contoh bersimpul lima yang dipakai di sepanjang Bab 10: $V = \{a, b, c, d, e\}$ dan $E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$. Gambar dan kedua daftar himpunannya membawa informasi yang identik; gambarnya hanya lebih enak dipandang. Derajat 3,2,3,3,3 berjumlah $14 = 2 \times 7$ — latihan 11 dalam satu gambar.

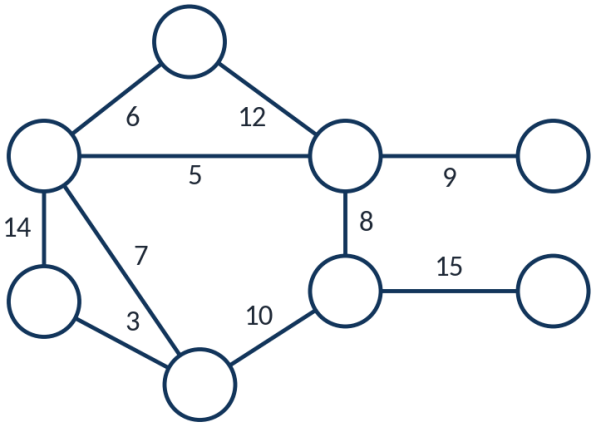
12.8 Profitina dalam Praktik

Profitina adalah platform visibilitas AI — GEO, *Generative Engine Optimization* — yang berjalan di lingkungan produksi pada `app.profitina.com`. Ia memindai situs web sebuah merek, menilai seberapa siap situs itu *disitir oleh mesin jawaban AI* berdasarkan sinyal-sinyal nyata seperti keterambilan isi, data terstruktur, dan kredibilitas teknis situsnya, lalu membantu memperbaiki apa yang ditemukannya. Proyek perancangan §12.2 meminta Anda menemukan persoalan nyata dan mengenali algoritma yang sudah diajarkan di dalamnya. Itulah pekerjaan sehari-hari yang biasa dalam membangun perangkat lunak, dan pemindaian sebuah situs web ternyata dibangun persis dari algoritma-algoritma yang dikerjakan bab ini. Uraian berikut menggambarkan bentuk setiap persoalan dan golongan penyelesaiannya, bukan penilaian milik produk itu.

Perayapannya adalah pencarian melebar. Sebuah pemindaian bermula di halaman muka lalu mengikuti tautan ke luar. Banyaknya klik dari halaman muka ke sebuah halaman — **kedalaman klik**-nya — adalah sinyal yang nyata, sebab halaman yang terkubur enam klik ke dalam adalah halaman yang boleh jadi tak pernah dianggarkan waktu oleh perayapnya untuk dicapai.

Kedalaman klik persis $d[v]$ dari pencarian melebar atas graf tautannya, dan ia harus dihitung dengan BFS alih-alih DFS: perayapan mendalam akan mengembara sampai kedalaman empat puluh menyusuri rantai penomoran halaman lalu melaporkan kedalaman yang mustahil bagi halaman yang hanya satu klik dari pintu depan. Lintasan yang sama menjawab pertanyaan kedua secara gratis: halaman mana pun dengan $d[v] = \infty$ adalah **halaman yatim**, tak ditautkan dari mana pun, dan halaman yatim tak kasatmata bagi perayap maupun mesin jawaban.

Pendeteksian nyaris-kembar adalah pohon merentang minimum dengan sisi beratnya dipotong. Situs menimbun halaman yang mengatakan hal yang nyaris sama — `/harga` dan `/harga-2024`, tiga halaman cabang yang hanya berbeda pada alamatnya. Mesin jawaban yang berhadapan dengan beberapa halaman yang nyaris identik boleh jadi tidak menyitir satu pun, atau menyitir yang keliru. Ukurlah jarak isi antara setiap pasangan halaman, bangunlah pohon merentang minimum atas jarak itu, lalu **hapuslah setiap sisi pohon yang lebih berat daripada sebuah ambang**: yang tersisa adalah satu komponen per gugus nyaris-kembar. Ini adalah penggugusan pautan-tunggal, dan alasan mengapa ia persoalan pohon merentang adalah sifat potongan Persoalan U — sisi teringan yang meninggalkan sekelompok halaman mana pun persis sisi yang menentukan kelompok mana yang sepatutnya ia gabungi lebih dahulu. **Gambar 12.23** adalah bentuk masukannya: sebagian sisi mahal tak terhindarkan, sebagian yang tampak murah justru mubazir, dan membedakan keduanya adalah seluruh persoalannya.



Gambar 12.23 Sebuah graf berbobot dari Bab 11. Sebagian sisi yang mahal tak terhindarkan — yang berbobot 15 adalah satu-satunya jalan masuk ke simpulnya — dan sebagian yang tampak murah justru mubazir. Graf kemiripan halaman berwatak persis seperti ini, dan algoritma yang sama memisahkan kedua jenis sisinya.

Rencana perbaikannya adalah sebuah DAG, dan ia diurutkan secara topologis. Begitu sebuah pemindaian menghasilkan daftar perbaikan, perbaikan-perbaikan itu punya urutan prasyarat: tanda kanonik harus ditambahkan sebelum halaman kembar digabungkan, dan peta situs sebaiknya tidak dikirim ulang sampai `robots.txt` benar. Jalankan dalam urutan yang melanggar kendala itu dan kerjanya terbuang. Urutan menjalankannya adalah pengurutan topologis, penyelesaian sedini mungkinnya adalah lintasan kritis Persoalan Q, dan ketergantungan melingkar di dalam rencananya harus dilaporkan alih-alih diam-diam dijalankan separuh — yang merupakan argumen §12.2.5 untuk algoritma Kahn, kini dibuat di lingkungan produksi.

Pemindaian ulang adalah persoalan penjadwalan. Platform itu memindai ulang setiap merek secara otomatis dengan irama yang ditetapkan tingkat langganannya, dan memberi tahu pemiliknya ketika skornya bergerak. Pada setiap saat banyak pemindaian ulang jatuh tempo, masing-masing dengan saat paling lambat ia semestinya sudah berjalan; memutuskan mana yang

dijalankan berikutnya adalah persoalan antrean prioritas, dan tenggat-terawal-dahulu — aturan yang dibuktikan optimal oleh Persoalan C untuk keterlambatan terburuk — adalah aturan pertama yang wajar untuk ditimbang. Persoalan C juga membawa peringatannya: aturan yang tepat bergantung pada tujuannya, dan aturan yang meminimumkan keterlambatan *terburuk* bukanlah aturan yang meminimumkan keterlambatan *rata-rata*.

Dua aturan mengatur semua ini, dan keduanya berasal dari prinsip produk itu sendiri. Pertama, ia menghormati undang-undang pelindungan data pribadi Indonesia, UU PDP 27/2022 [12]: graf tautan dan matriks kemiripan halaman diturunkan dari halaman publik dan tidak menyentuh data pribadi. Kedua, **aplikasinya menasihati dan penggunanya memutuskan** — penggugusannya mengusulkan sebuah URL kanonik, ia tidak diam-diam menghapus halaman, dan sebuah skor adalah taksiran dari sinyal terukur, tak pernah jaminan untuk disitir.

12.8.1 Studi kasus terkerjakan: Profitina Laundry

Studi kasus yang dapat dijalankan selalu **Profitina Laundry**, usaha pengajaran buku ini, tak pernah produk yang sesungguhnya. Binatu itu kini punya situs web kecil, dan ia ingin ditemukan dan disitir ketika seseorang bertanya kepada mesin jawaban *"di mana saya bisa mencucikan pakaian dekat ITS?"*. **Kode 12.11** mengaudit situs itu dengan tiga algoritma di atas.

Kode 12.11 Audit situs web Profitina Laundry: kedalaman klik dan halaman yatim lewat pencarian melebar, penggugusan nyaris-kembar lewat pohon merentang minimum dengan sisi beratnya dipotong, dan rencana perbaikan yang diurutkan secara topologis beserta lintasan kritisnya. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o laundry laundry.cpp`.

```
// Profitina dalam Praktik: mengaudit situs Profitina Laundry sebagaimana
// mesin visibilitas AI mengaudit situs kliennya. Tiga soal graf:
//   Bagian 1  kedalaman klik dari halaman muka, lewat pencarian melebar,
//             dan halaman yatim yang tak ditautkan apa pun;
//   Bagian 2  gugus halaman nyaris-kembar, dengan membangun pohon merentang
//             minimum atas jarak isi lalu memotong tiap sisi di atas ambang;
//   Bagian 3  rencana perbaikan, sebagai DAG, dengan urutan topologis.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <numeric>
#include <algorithm>
#include <climits>
using namespace std;

// ------------------------------------------------- Bagian 1: perayapannya
static const vector<string> URL = {
    "/", "/layanan", "/harga", "/cabang", "/cabang/keputih", "/cabang/gebang",
    "/tentang", "/kontak", "/blog", "/blog/merawat-kain", "/blog/tips-setrika",
    "/promo-lama"
};
static const vector<pair<int,int>> LINK = {
    {0,1},{0,2},{0,3},{0,6},{0,7},{0,8},{1,2},{3,4},{3,5},
    {8,9},{8,10},{9,10},{10,9},{4,7},{5,7},{11,0}
};

void part1() {
    int n = (int)URL.size();
    vector<vector<int>> out(n), in(n);
    for (auto [u,v] : LINK) { out[u].push_back(v); in[v].push_back(u); }
    vector<int> depth(n, INT_MAX);
    depth[0] = 0; queue<int> Q; Q.push(0);
    while (!Q.empty()) { int u = Q.front(); Q.pop();
        for (int v : out[u]) if (depth[v] == INT_MAX) { depth[v] = depth[u] + 1; Q.push(v);
} }
    cout << "=== Part 1: click depth from the home page ===\n";
    cout << left << setw(24) << "URL" << right << setw(8) << "depth"
         << setw(10) << "inlinks" << "  verdict\n";
    int deep = 0, orphan = 0;
    for (int v = 0; v < n; ++v) {
        string verdict = "ok";
        if (depth[v] == INT_MAX) { verdict = "ORPHAN - unreachable from /"; ++orphan; }
        else if (depth[v] > 2)   { verdict = "too deep for a crawler budget"; ++deep; }
        cout << left << setw(24) << URL[v] << right << setw(8);
        if (depth[v] == INT_MAX) cout << "inf"; else cout << depth[v];
```

Kode 12.11 (Lanjutan) *Audit situs web Profitina Laundry: kedalaman klik dan halaman yatim lewat pencarian melebar, penggugusan nyaris-kembar lewat pohon merentang minimum dengan sisi beratnya dipotong, dan rencana perbaikan yang diurutkan secara topologis beserta lintasan kritisnya. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o laundry laundry.cpp.*

```
        cout << setw(10) << in[v].size() << "  " << verdict << "\n";
    }
    cout << "pages: " << n << ", orphans: " << orphan << ", deeper than 2 clicks: " << deep
<< "\n";
}

// ---------------------------------------------- Bagian 2: gugus nyaris-kembar
struct D { int a, b, dist; };
static const vector<string> PAGE = {
    "/cabang/keputih", "/cabang/gebang", "/cabang/rungkut",
    "/harga", "/harga-2024", "/layanan"
};
static const vector<D> DIST = {                        // 0 = identik, 100 = tak terkait
    {0,1,6},{0,2,9},{1,2,7},{0,3,64},{1,3,70},{2,3,66},
    {3,4,4},{0,4,61},{1,4,72},{2,4,68},{3,5,38},{4,5,41},
    {0,5,55},{1,5,58},{2,5,52}
};
struct DSU {
    vector<int> p, r;
    explicit DSU(int n) : p(n), r(n,0) { iota(p.begin(), p.end(), 0); }
    int find(int x) { return p[x]==x ? x : p[x]=find(p[x]); }
    bool unite(int x, int y) { x=find(x); y=find(y);
        if (x==y) return false;
        if (r[x]<r[y]) swap(x,y);
        p[y]=x; if (r[x]==r[y]) ++r[x];
        return true; }
};
void part2(int threshold) {
    cout << "\n=== Part 2: near-duplicate clusters (threshold " << threshold << ") ===\n";
    int n = (int)PAGE.size();
    vector<D> e = DIST;
    sort(e.begin(), e.end(), [](const D& x, const D& y){ return x.dist < y.dist; });
    DSU d(n); vector<D> mst; int total = 0;
    for (const D& x : e) if (d.unite(x.a, x.b)) { mst.push_back(x); total += x.dist; }
    cout << "  minimum spanning tree over " << n << " pages, " << DIST.size() << " measured
pairs:\n";
    for (const D& x : mst)
        cout << "    " << left << setw(18) << PAGE[x.a] << setw(18) << PAGE[x.b]
             << right << setw(5) << x.dist
             << (x.dist <= threshold ? "  keep (same cluster)" : "  CUT") << "\n";
    cout << "  tree weight " << total << "; edges " << mst.size() << " = " << n << " -
1\n";
    DSU c(n);
    for (const D& x : mst) if (x.dist <= threshold) c.unite(x.a, x.b);
    vector<vector<int>> cl(n);
    for (int v = 0; v < n; ++v) cl[c.find(v)].push_back(v);
    int k = 0;
    for (int v = 0; v < n; ++v) if (!cl[v].empty()) {
        cout << "  cluster " << ++k << ":";
```

Kode 12.11 (Lanjutan) *Audit situs web Profitina Laundry: kedalaman klik dan halaman yatim lewat pencarian melebar, penggugusan nyaris-kembar lewat pohon merentang minimum dengan sisi beratnya dipotong, dan rencana perbaikan yang diurutkan secara topologis beserta lintasan kritisnya. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o laundry laundry.cpp.*

```
        for (int p : cl[v]) cout << " " << PAGE[p];
        cout << (cl[v].size() > 1 ? "   <- pick one canonical URL\n" : "\n");
    }
    cout << "  clusters: " << k << " (cutting the " << mst.size() - count_if(mst.begin(),
mst.end(),
            [&](const D& x){ return x.dist <= threshold; }) << " heaviest tree edges)\n";
}

// ---------------------------------------- Bagian 3: DAG rencana perbaikannya
static const vector<string> FIX = {
    "audit-inventory", "fix-robots.txt", "add-canonicals", "merge-duplicates",
    "add-schema.org", "rewrite-thin-pages", "submit-sitemap", "re-scan"
};
static const vector<int> HOURS = { 2, 1, 3, 4, 5, 8, 1, 1 };
static const vector<pair<int,int>> DEP = {
    {0,1},{0,2},{2,3},{0,5},{3,4},{5,4},{1,6},{3,6},{4,7},{6,7}
};
void part3() {
    cout << "\n=== Part 3: the remediation plan, topologically sorted ===\n";
    int n = (int)FIX.size();
    vector<vector<int>> adj(n); vector<int> indeg(n, 0);
    for (auto [u,v] : DEP) { adj[u].push_back(v); ++indeg[v]; }
    queue<int> Q; for (int v = 0; v < n; ++v) if (!indeg[v]) Q.push(v);
    vector<int> order, ef(n, 0), pred(n, -1);
    while (!Q.empty()) { int u = Q.front(); Q.pop(); order.push_back(u);
        if (!ef[u]) ef[u] = HOURS[u];
        for (int v : adj[u]) { if (ef[u] + HOURS[v] > ef[v]) { ef[v] = ef[u] + HOURS[v];
pred[v] = u; }
                              if (--indeg[v] == 0) Q.push(v); } }
    if ((int)order.size() != n) { cout << "  the plan contains a circular dependency\n";
return; }
    cout << "  step  task                 hours  earliest finish\n";
    for (size_t i = 0; i < order.size(); ++i)
        cout << setw(6) << i + 1 << "  " << left << setw(20) << FIX[order[i]]
             << right << setw(5) << HOURS[order[i]] << setw(16) << ef[order[i]] << "\n";
    int last = (int)(max_element(ef.begin(), ef.end()) - ef.begin());
    vector<int> path; for (int x = last; x != -1; x = pred[x]) path.push_back(x);
    reverse(path.begin(), path.end());
    cout << "  one worker, tasks end to end: " << accumulate(HOURS.begin(), HOURS.end(), 0)
         << " h\n  unlimited workers, critical path: " << ef[last] << " h  (";
    for (size_t i = 0; i < path.size(); ++i) cout << (i ? " -> " : "") << FIX[path[i]];
    cout << ")\n";
}

int main() { part1(); part2(30); part3(); return 0; }
```

Keluarannya:

```
=== Part 1: click depth from the home page ===
URL                          depth   inlinks  verdict
/                                0         1  ok
/layanan                         1         1  ok
/harga                           1         2  ok
/cabang                          1         1  ok
/cabang/keputih                  2         1  ok
/cabang/gebang                   2         1  ok
/tentang                         1         1  ok
/kontak                          1         3  ok
/blog                            1         1  ok
/blog/merawat-kain               2         2  ok
/blog/tips-setrika               2         2  ok
/promo-lama                    inf         0  ORPHAN - unreachable from /
pages: 12, orphans: 1, deeper than 2 clicks: 0

=== Part 2: near-duplicate clusters (threshold 30) ===
  minimum spanning tree over 6 pages, 15 measured pairs:
    /harga            /harga-2024           4  keep (same cluster)
    /cabang/keputih   /cabang/gebang        6  keep (same cluster)
    /cabang/gebang    /cabang/rungkut       7  keep (same cluster)
    /harga            /layanan             38  CUT
    /cabang/rungkut   /layanan             52  CUT
  tree weight 107; edges 5 = 6 - 1
  cluster 1: /cabang/keputih /cabang/gebang /cabang/rungkut   <- pick one canonical URL
  cluster 2: /harga /harga-2024   <- pick one canonical URL
  cluster 3: /layanan
  clusters: 3 (cutting the 2 heaviest tree edges)

=== Part 3: the remediation plan, topologically sorted ===
  step  task                 hours  earliest finish
     1  audit-inventory         2               2
     2  fix-robots.txt          1               3
     3  add-canonicals          3               5
     4  rewrite-thin-pages      8              10
     5  merge-duplicates        4               9
     6  add-schema.org          5              15
     7  submit-sitemap          1              10
     8  re-scan                 1              16
  one worker, tasks end to end: 25 h
  unlimited workers, critical path: 16 h  (audit-inventory -> rewrite-thin-pages -> add-
schema.org -> re-scan)
```

Tiga temuan, dalam bahasa yang dipahami pemilik binatunya.

`/promo-lama` tak kasatmata. Ia menaut *keluar* ke halaman muka tetapi tak ada apa pun yang menaut *masuk* kepadanya, jadi derajat masuknya nol dan kedalaman kliknya tak hingga. Mesin jawaban takkan menemukannya, dan begitu pula pelanggan. Tautilah ia dari suatu tempat atau pensiunkanlah ia.

Lima dari enam halaman yang diaudit adalah nyaris-kembar dari sesuatu. Memotong kedua sisi pohon yang di atas ambang 30 menyisakan tiga gugus: ketiga halaman cabangnya (berjarak 6 dan 7), kedua halaman harganya (berjarak 4), dan `/layanan` seorang diri. Rekomendasinya adalah satu URL kanonik per gugus — bukan penghapusan, sebab aplikasinya menasihati dan pemiliknya memutuskan.

Perbaikannya memakan 16 jam, bukan 25. Dikerjakan tegas satu tugas sesudah tugas lain, rencananya 25 jam; lintasan kritis melewati DAG-nya 16, sebab `fix-robots.txt`, `add-canonicals`, dan `rewrite-thin-pages` semuanya dapat mulai begitu inventarisnya diaudit. Kesembilan jam yang dihemat itu seluruhnya berasal dari penjadwalan, diperoleh dari satu pengurutan topologis dan satu pelonggaran per busur.

12.9 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Memilih persoalan yang algoritmanya sekadar hiasan. BFS pada papan yang begitu kecil sehingga metode apa pun dapat menyelesaikannya. *Perbaikannya:* kenakan uji pemikul beban §12.2.2 sebelum Anda terikat pada gagasannya — apakah metode naifnya selamat pada masukan seribu kali lebih besar?

Menulis rancangan sesudah programnya. Ia lalu terbaca sebagai uraian, bukan argumen, dan tidak memuat alternatif yang ditolak. *Perbaikannya:* kerjakan tahap 1 sampai 6 **Tabel 12.2** lebih dahulu, dan simpanlah apa yang Anda tulis.

Melewatkan evaluasi. Program yang hanya dijalankan pada contoh masukannya belum terbukti berhasil. *Perbaikannya:* orakel gaya kasar atau pascakondisi, beberapa ribu pemeriksaan silang acak, dan pengukuran waktu pada lima ukuran yang dibagi ongkos ramalannya, seperti yang dikerjakan **Kode 12.2**.

Memercayai aturan rakus karena ia berhasil sekali. Tiga dari keempat aturan aktivitas memberikan optimumnya pada pekerjaan pengering Persoalan B, dan dua di antaranya keliru. *Perbaikannya:* buktikan aturannya dengan argumen pertukaran atau patahkan ia dengan contoh sanggahan, seperti yang dikerjakan Persoalan A untuk koin; kesepakatan pada satu contoh tidak membuktikan apa pun.

Menulis "penyelesaian optimalnya" di dalam argumen pertukaran. Optimumnya jarang tunggal, dan sifat pilihan rakus berkata *suatu* penyelesaian optimal memuat pilihan rakusnya. *Perbaikannya:* tulislah "misalkan A adalah *sebuah* … berukuran maksimum".

Membaca sifat pertukaran matroid sebagai "setiap unsur berhasil". Ia menjanjikan bahwa *suatu* $x \in B \setminus A$ dapat ditambahkan ke A, bukan semuanya; di Persoalan F, a_4 berhasil dan a_5 tidak.

Menuliskan kode Huffman tanpa memeriksanya. Satu kekeliruan pada satu penggabungan diam-diam mengubah beberapa kata kode. *Perbaikannya:* hitunglah $B(T)$ dengan kedua cara, menurut Persamaan (12.9) dan menurut Persamaan (12.10); jika keduanya tidak sepakat, ada kekeliruan.

Hanya memperlihatkan hasil akhir sebuah penelusuran. Penelusurannya *adalah* jawabannya ketika pertanyaannya meminta penelusuran; pohon akhir tanpa pengerjaan tidak memperlihatkan apa pun tentang apakah `Find` dan `Union` dipahami. *Perbaikannya:* satu baris per langkah, setiap nilai antara dituliskan, seperti pada **Tabel 12.31**.

Mengenakan uji siklus berarah pada graf tak berarah. Setiap sisi lalu tampak seperti siklus, sebab tiap sisi duduk di kedua daftar ketetanggaan. *Perbaikannya:* lewatilah persis satu kemunculan induknya, dan tidak lebih (Persoalan L).

Mencari sisi maju atau sisi silang pada pencarian mendalam tak berarah. Tidak ada; penglihatan kedua atas sebuah sisi dari ujung hitamnya adalah sisi mundur yang sama yang terlihat lagi (Persoalan I).

Menyebut busur ke simpul hitam "maju" karena sasarannya datang belakangan. Maju berarti *keturunan di dalam hutan mendalamnya*; uji cap waktu $d[u] < d[v]$ yang memutuskannya, dan simpul di pohon lain selalu gagal dalam uji itu (Persoalan M).

Mengurutkan menurut waktu temu untuk memperoleh urutan topologis. Hanya waktu selesai yang berhasil, menurut Persamaan (12.15); Persoalan N punya contoh sanggahan dua simpulnya.

Melupakan entri basi di dalam tumpukan malas. Tanpa uji pada baris 5 **Pseudokode 12.7**, Prim akan memproses h tiga kali dalam jalannya Persoalan T dan menambahkan bobotnya ke total tiga kali. *Perbaikannya:* pada setiap pengeluaran, buanglah entrinya jika simpulnya sudah berada di pohon atau kuncinya sudah basi (Persoalan T).

Mengira bobot yang sama membuat pohon merentang minimum tidak tunggal. Keberbedaan itu cukup, tidak perlu; kriteria yang sesungguhnya adalah sifat siklus (Persoalan W).

Mengira Prim dan Kruskal harus mengembalikan sisi yang sama. Keduanya harus mengembalikan *bobot* yang sama. Pada graf dengan bobot seri keduanya bisa saja berselisih tentang sisi yang mana (**Gambar 12.18**, dan jalan kedua Persoalan T).

Melaporkan cacah sisi mundur seolah-olah tak bergantung pada urutan. *Vonis* "tidak taksiklik" tak bergantung pada urutan; cacah dan penggolongannya bergantung (Soal Latihan 12.5).

Menganggap `rank` sebagai ketinggian. Ia batas atas yang sengaja dibiarkan basi oleh pemampatan lintasan (Persoalan R).

12.10 Ringkasan Bab

Bab ini mengerjakan Bab 9 sampai 11 sebagai persoalan, bukan sebagai teori.

§12.2 membawa satu persoalan nyata dari halaman kosong sampai program yang terukur dalam sembilan tahap. Perencana Rute Laundry untuk Profitina Laundry memodelkan sebuah jaringan jalan dan sebuah alur pemrosesan sebagai graf, memilih pencarian melebar, Kruskal di atas hutan himpunan saling lepas, dan algoritma Kahn dengan menimbang alternatif yang disebut namanya, meramalkan ongkos $\Theta(m \lg m)$ lebih dahulu, dan menelusuri ketiga algoritmanya dengan tangan sebelum menulis sebaris kode pun. Programnya lalu sepakat dengan penelusuran tangannya baris demi baris: enam dari empat belas zona dalam dua lompatan dan dua tak terjangkau, tulang punggung sebelas jalan sepanjang 6,9 km terhadap 12,3 km jalan calon (penghematan 43,9%), dan lintasan kritis 121 menit dengan kelonggaran 106 menit pada penerbitan faktur. Evaluasinya memeriksa silang Kruskal terhadap Prim pada 2000 graf acak tanpa ketidakcocokan, memeriksa pascakondisi tingkat BFS pada 2000 graf lagi tanpa pelanggaran, dan mengukur kedua algoritma pada lima ukuran terhadap pertumbuhan ramalannya — serta menjelaskan mengapa yang linear berhenti tampak linear begitu datanya keluar dari *cache*.

§12.3 mengerjakan enam persoalan rakus: sistem koin yang di atasnya kerakusan gagal, beserta bukti bahwa ia berhasil untuk {1,5,10}; pemilihan aktivitas yang dijalankan dengan dua aturan dan dibuktikan dengan argumen pertukaran; meminimumkan keterlambatan maksimum, dibuktikan dengan menukar pembalikan bersebelahan; kode Huffman yang dibangun penggabungan demi penggabungan, digambar sebagai pohon dan dihitung ongkosnya dengan dua cara; kedua *knapsack*, yang di antaranya satu bahan memisahkan keberhasilan dari kehilangan 27,8%; dan algoritma rakus matroid yang menjadwalkan pesanan berwaktu satuan, dibenarkan terhadap seluruh 128 upahimpunan. §12.4 menelusuri pencarian melebar, pencarian mendalam pada graf berarah dan tak berarah dengan tabel cap waktu yang lengkap, komponen beserta peringkat sirkuitnya, kebipartitan beserta sertifikat siklus ganjilnya, dan kedua uji siklus. §12.5 mengerjakan DAG prasyarat: uji sisi mundur, pengurutan topologis menurut waktu selesai dan

menurut Kahn, ke-84 urutan sah beserta tiga prasyarat tambahan yang diperlukan untuk memaksa satu urutan, dan lintasan kritisnya. §12.6 mengerjakan pohon merentang minimum: Kruskal dengan setiap `Find` dan `Union`, Prim dengan larik kunci dan dengan tumpukan biner malas, sifat potongan dan sifat siklus, penyatuan menurut peringkat beserta batas 2^{rank}, dan uji ketunggalannya. Setiap persoalan ditutup dengan varian untuk menguji diri, dan dua belas program lengkap membenarkan setiap bilangannya.

12.10.1 Menatap ke depan

Setiap algoritma graf dalam bab ini memperlakukan sisinya entah sebagai tak berbobot, entah sebagai ongkos yang *dijumlahkan atas sebuah pohon*. Bab 13 mengubah pertanyaannya: dengan bobot yang berupa waktu tempuh, berapa **lintasan** termurah dari satu simpul ke simpul lain? Pencarian melebar menjawabnya ketika setiap sisi berongkos sama; algoritma Dijkstra menjawabnya ketika ongkosnya tak negatif; Bellman–Ford menjawabnya ketika sebagian ongkos negatif; dan pada sebuah DAG satu sapuan topologis — sapuan yang sama yang menghasilkan lintasan kritis §12.2.8 dan Persoalan Q — menjawabnya dalam waktu linear. Kesimpulan proyek perancangannya sudah menyebut hal itu sebagai langkah berikutnya.

12.11 Soal Latihan

12.1 Untuk sistem koin {1,4,6}, temukan jumlah terkecil tempat aturan penukaran uang rakus **Pseudokode 12.2** gagal, berikan kedua pembayarannya, dan cacahlah jumlah dari 1 sampai 30 yang pada jumlah itu kerakusan kalah. Apakah kegagalan terkecilnya sesuai dengan batas Kozen dan Zaks yang dikutip di §12.3.2?

12.2 Kenakan uji pemikul beban §12.2.2 pada ketiga usulan ini, dan untuk masing-masingnya katakan apa yang dapat diargumentasikan tahap rancangannya. (a) Penyelesai Sudoku yang memakai penjejakan mundur mendalam. (b) Aplikasi peta kampus yang menghitung rute dengan belokan tersedikit antara dua gedung. (c) Aplikasi kuis yang mengocok sepuluh pertanyaan.

12.3 Jalankan pencarian melebar pada graf G_1 §12.4.1 dari sumber C alih-alih A, berikan d dan π. Simpul mana yang berubah jaraknya, dan berapa jarak terbesar dari A dan dari C?

12.4 Pada contoh Rute Laundry §12.2.4, tim operasional menemukan sebuah jalan dari **Menur ke Klampis**. Hitunglah ulang jarak melebarnya dari Sukolilo. Berapa zona yang kini terletak di dalam janji dua lompatan, dan zona mana yang bergabung?

12.5 Jalankan pencarian mendalam pada digraf G_2 §12.4.1 dengan gelung luarnya memindai G, F, E, D, C, B, A alih-alih A sampai G. Berikan cap waktunya dan golongkan setiap busurnya. Mana di antara berikut ini yang tidak berubah: banyaknya sisi pohon, banyaknya sisi mundur, vonis "tidak taksiklik"?

12.6 Pencarian mendalam atas DAG prasyarat pada Persoalan M menghasilkan lima sisi pohon. Terangkan mengapa bilangan itu harus sama dengan 8 dikurangi banyaknya pohon di dalam hutannya, dan katakan apa yang akan berubah jika gelung luarnya bermula di H.

12.7 Susunlah sebuah contoh pemilihan aktivitas yang di situ aturan *durasi terpendek dahulu* mengembalikan himpunan yang tegas lebih kecil daripada aturan waktu-selesai-terawal. Pakailah aktivitas sesedikit mungkin, dan nyatakan jawaban masing-masing aturannya.

12.8 Andaikan seluruh n simbol pada sebuah contoh Huffman berfrekuensi **sama**, yaitu f. (a) Bagaimana bentuk pohonnya, dan berapa panjang kata kodenya, ketika $n = 8$? Ketika $n = 6$? (b) Berapa $B(T)$-nya dibandingkan kode panjang-tetap pada masing-masing kasusnya? (c) Benar

atau salah: setiap pohon Huffman atas tiga simbol atau lebih punya kata kode berpanjang 1. Buktikan atau berikan contoh sanggahan, lalu nyatakan sesuatu yang *memang* benar bagi setiap pohon Huffman.

12.9 Ambillah graf G §12.6.1. Hitunglah pohon merentang minimumnya sesudah masing-masing transformasi ini lalu katakan apakah *pohonnya* berubah, bukan sekadar bobotnya: (a) tambahkan 10 pada tiap bobot; (b) kuadratkan tiap bobot; (c) gantilah tiap bobot w dengan $20 - w$.

12.10 Berapa banyak siklus bebas yang dimuat graf G §12.6.1? Turunkan bilangannya dengan dua cara: dari Persamaan (12.14), dan dengan mencacah sisi bukan-pohon **Tabel 12.34**.

12.11 Rantaikan 100.000 unsur dengan `Union(0,1)`, `Union(1,2)`, …, lalu jalankan satu `Find` pada tiap unsur. Cacahlah seluruh banyaknya penunjuk induk yang diikuti, (a) tanpa penyatuan menurut peringkat dan tanpa pemampatan lintasan, dan (b) dengan keduanya. Terangkan nisbahnya.

12.12 Pada DAG prasyarat **Tabel 12.27**, Basis Data (F) diperpanjang dari 3 pekan menjadi 7. Hitunglah ulang setiap waktu selesai terawalnya dan lintasan kritisnya. Berapa lama program studinya memanjang, dan mata ajar mana yang meninggalkan lintasan kritisnya?

12.13 Gantilah **antrean** Kahn dengan sebuah **tumpukan** (dorong simpul berderajat masuk nol menurut urutan abjad, lalu keluarkan berulang-ulang). Jalankan ia pada DAG prasyarat **Tabel 12.27** lalu berikan urutannya. Bandingkan dengan urutan Persoalan N dan O.

12.14 Untuk masing-masing ketiga komponen graf G_3 Persoalan J, katakan apakah ia bipartit lalu berikan entah kedua pihaknya entah sebuah siklus ganjil.

12.15 Ambillah gagasan Anda sendiri untuk sebuah proyek perancangan yang lolos uji pemikul beban. Dalam tidak lebih dari 400 kata, tuliskan tahap 1 sampai 5 **Tabel 12.2**: persoalannya, modelnya, algoritma yang dipilih beserta satu alternatif yang ditolak dan alasannya, garis besar pseudokodenya, dan kompleksitas yang diramalkan. Sesudah itu periksalah tulisan Anda terhadap kolom ketiga **Tabel 12.2**.

12.12 Penyelesaian Soal Latihan

12.1 Dari jalannya **Kode 12.12** yang sudah dipastikan: kegagalan terkecilnya **8**, tempat kerakusan membayar $6 + 1 + 1$ (tiga koin) dan optimumnya $4 + 4$ (dua). Kerakusan kalah pada **8** dari jumlah-jumlah 1 sampai 30; keempat yang pertama adalah 8, 9, 14, dan 15 (pada 9, kerakusan membayar $6 + 1 + 1 + 1$ terhadap $4 + 4 + 1$). Batas Kozen–Zaks berkata bahwa kegagalan pertama, jika ada, terletak di bawah jumlah kedua koin terbesar, $6 + 4 = 10$; dan memang $8 < 10$.

12.2 (a) **Sudoku dengan penjejakan mundur.** "Coba setiap pengisian sel yang kosong" secara naif bisa mencapai 9^{81} dan tanpa harapan — tetapi begitu pula penjejakan mundur tanpa pemangkasan. Ini memikul beban *hanya jika* rancangannya mengargumentasikan pemangkasannya (perambatan kendala, pengurutan sel paling terkendala). Argumentasikan hal itu, dan ia menjadi proyek yang kuat. (b) **Belokan tersedikit antara dua gedung.** Modelkan belokannya, bukan jalannya: sebuah simpul adalah pasangan (persimpangan, arah hadap), dan sisi yang mengubah arah hadap berongkos satu. Maka ia pencarian melebar pada graf bersimpul $2|E|$, dan metode naifnya — mencacah rutenya — mati seketika. **Pemikul beban**, dan cukup tak biasa sehingga langkah pemodelannya sendiri menjadi bagian yang menarik dari rancangannya.

(c) **Mengocok sepuluh pertanyaan.** Fisher–Yates adalah $\Theta(n)$ dan sepuluh tetaplah sepuluh. **Hiasan**; tak ada apa pun yang dapat diargumentasikan rancangannya.

12.3 Dari jalannya **Kode 12.12** yang sudah dipastikan, dikumpulkan di **Tabel 12.37**:

Tabel 12.37 Pencarian melebar atas G_1 dari C, berdampingan dengan pencarian dari A. Setiap simpul berubah jaraknya. Jarak terbesar ke simpul mana pun — **eksentrisitas** sumbernya — adalah 4 dari A dan 3 dari C, jadi C lebih di tengah di antara keduanya dan pohonnya satu tingkat lebih dangkal.

v	A	B	C	D	E	F	G	H	I
d dari A	0	1	1	2	2	2	3	3	4
d dari C	1	2	0	3	1	1	2	2	3
π dari C	C	A	–	B	C	C	E	E	G

12.4 Menambahkan Menur–Klampis menempatkan Klampis satu lompatan dari Menur, sehingga **dua** lompatan dari Sukolilo alih-alih tiga. Jarak barunya adalah Sukolilo 0; Menur 1, Keputih 1; Gebang 2, Semolowaru 2, **Klampis 2**, Kejawan 2; Nginden 3, Manyar 3, Mulyosari 3; Kalijudan 4, DukuhSetro 4; Bulak dan Kenjeran masih tak terjangkau. Cacah dua lompatannya naik dari 6 menjadi **7**, dan zona yang bergabung adalah **Klampis**. Perhatikan akibat berantainya: Manyar turun dari 4 menjadi 3 dan DukuhSetro dari 5 menjadi 4, jadi satu jalan membelikan perbaikan bagi tiga zona — tetapi ia sama sekali tidak membelikan apa pun bagi Bulak dan Kenjeran, sebab menambahkan sisi di dalam satu komponen tak pernah dapat menciptakan lintasan menuju komponen lain.

12.5 Dengan gelung luar dibalik, cap waktunya menjadi G 1/14, C 2/13, D 3/8, A 4/7, B 5/6, E 9/12, F 10/11, dan penggolongannya menjadi enam sisi pohon ($G \to C$, $C \to D$, $D \to A$, $A \to B$, $C \to E$, $E \to F$), empat mundur ($A \to C$, $A \to D$, $B \to D$, $F \to E$), satu maju ($G \to F$), dan **nol** sisi silang.

Jadi banyaknya sisi pohon **berubah**, dari 5 menjadi 6, sebab hutannya kini punya satu pohon alih-alih dua ($7 - 1 = 6$). Banyaknya sisi mundur **berubah**, dari 2 menjadi 4. Vonis "tidak taksiklik" **tidak berubah**, dan hanya itulah satu-satunya dari ketiganya yang dijamin teorema DAG: ketaksiklikan adalah sifat grafnya, sedangkan penggolongannya adalah sifat pencariannya.

12.6 Setiap simpul kecuali akar memperoleh persis satu pendahulu, dan setiap sisi pohon adalah pasangan $(\pi[v], v)$ untuk persis satu simpul semacam itu. Jadi cacahnya adalah $|V| -$ (banyaknya akar) $= 8 - 3 = 5$. Akarnya adalah simpul yang dicapai gelung luarnya selagi ia masih putih, yang di sini adalah A, B, dan C. Jika gelung luarnya bermula di H, maka H — yang tak punya busur keluar — akan selesai seketika sebagai pohon bersimpul satu, dan hutannya bertambah satu akar: banyaknya akar naik menjadi empat dan banyaknya sisi pohon turun menjadi $8 - 4 = 4$, sebab busur menuju H yang dahulu sisi pohon ($E \to H$) kini mendapati H hitam dan menjadi sisi silang.

12.7 Tiga aktivitas mencukupi: $a_1 = [0,5)$, $a_2 = [4,6)$, $a_3 = [5,10)$. Durasinya 5, 2, dan 5, jadi *durasi terpendek dahulu* mengambil a_2, yang tumpang tindih dengan kedua yang lain, lalu berhenti dengan $\{a_2\}$, berukuran **1**. *Waktu selesai terawal* mengambil a_1 (selesai pada 5), lalu a_3 (mulai pada 5), memberikan $\{a_1, a_3\}$, berukuran **2**. Dua itulah yang optimal. Contohnya minimal: dengan dua aktivitas kedua aturannya tak dapat berbeda pada ukuran jawabannya.

12.8 (a) Dengan $n = 8$ frekuensi yang sama, setiap penggabungan menyatukan dua pohon berbobot sama, jadi pohonnya adalah **pohon biner sempurna** sedalam 3 dan setiap kata kodenya berpanjang $\lg 8 = 3$. Dengan $n = 6$ cacahnya tidak tinggal sama: menggabungkan dua daun berbobot f menghasilkan simpul berbobot $2f$, jadi sesudah tiga penggabungan antreannya memuat tiga simpul berbobot $2f$; dua di antaranya bergabung menjadi $4f$, dan penggabungan

terakhir menyatukan $2f$ dengan $4f$. Pohonnya karena itu berakhir dengan **empat** simbol pada kedalaman 3 dan **dua** pada kedalaman 2 — panjang $\lceil \lg 6 \rceil = 3$ dan $\lfloor \lg 6 \rfloor = 2$. (b) Untuk $n = 8$, $B(T) = 8f \times 3 = 24f$, persis ongkos panjang-tetapnya: **sama sekali tanpa penghematan**. Untuk $n = 6$, $B(T) = 4f \times 3 + 2f \times 2 = 16f$ terhadap panjang-tetap $6f \times 3 = 18f$, yaitu penghematan $2f$, atau 11,1%. (c) **Salah.** Bagian (a) adalah contoh sanggahannya: dengan delapan frekuensi yang sama setiap kata kodenya berpanjang 3, dan tak ada yang berpanjang 1. Kata kode satu bit hanya muncul ketika salah satu dari kedua pohon yang disatukan pada penggabungan terakhir berupa satu daun tunggal, yang menuntut simbol itu cukup sering dibandingkan sisanya, seperti R pada Persoalan D. Yang *memang* benar bagi setiap pohon Huffman adalah bahwa ia **penuh** — setiap simpul dalam punya tepat dua anak, jadi tak ada bit yang terbuang — dan bahwa kedua simbol paling jarang bersaudara pada kedalaman maksimum (§9.7.8).

12.9 Dari jalannya yang sudah dipastikan: pohon aslinya berbobot 31 dan berupa $\{BC, DE, AB, FG, CD, EF, GH\}$. (a) **Tambah 10:** bobot $31 + 7 \times 10 = 101$, **pohon yang sama**. Menambahkan tetapan k menambahkan persis $k(|V| - 1)$ pada *setiap* pohon merentang, jadi peringkatnya tak tersentuh. (b) **Kuadratkan:** bobot 209, **pohon yang sama**. Mengkuadratkan bersifat naik tegas pada bobot tak negatif, jadi ia melestarikan *urutan* bobotnya, dan baik Kruskal maupun Prim hanya bergantung pada urutan itu. (Ia takkan melestarikan urutannya seandainya ada bobot yang negatif.) (c) $20 - w$**:** bobot 79, dan **pohon yang berbeda** — $\{FH, GH, EG, CE, DF, BD, AC\}$, yang merupakan pohon merentang *maksimum* graf aslinya. Membalikkan urutan bobotnya mengubah peminimuman menjadi pemaksimuman.

Pelajarannya: transformasi yang melestarikan urutan bobot membiarkan pohonnya tak berubah; transformasi yang membalik urutannya membalikkan pohon itu dari dalam ke luar.

12.10 Menurut Persamaan (12.14), peringkat sirkuitnya adalah $|E| - |V| + k = 13 - 8 + 1 = \mathbf{6}$. Dihitung dengan cara yang lain, pohon merentangnya memakai $|V| - 1 = 7$ sisi, jadi $13 - 7 = 6$ sisi tersisa, dan masing-masingnya menutup persis satu siklus baru ketika dikembalikan — itulah persis keenam baris **Tabel 12.34**. Kedua cacahnya sepakat, dan memang harus, sebab $|E| - (|V| - k)$ dan $|E| - |V| + k$ adalah ungkapan yang sama.

12.11 Diukur oleh **Kode 12.12**: **(a)** tanpa penyatuan menurut peringkat dan tanpa pemampatan lintasan, rantainya membangun lintasan berisi 100.000 simpul dan satu `Find` per unsur mengikuti $0 + 1 + 2 + \cdots + 99\,999 = \frac{99\,999 \times 100\,000}{2} = 4\,999\,950\,000$ penunjuk induk. **(b)** Dengan penyatuan menurut peringkat dan pemampatan lintasan, totalnya 99 999 — satu lompatan per unsur, sebab sesudah penyatuannya setiap unsur sudah menunjuk langsung ke akar. Nisbahnya kira-kira **50.000 banding 1**, dan ia tumbuh linear terhadap n: struktur naifnya $\Theta(n^2)$ atas n kali `Find`, yang baik $O(n\,\alpha(n))$.

12.12 Dengan Basis Data 7 pekan, $ef[F] = ef[D] + 7 = 8 + 7 = 15$, lalu

$$ef[H] = \max\{ef[E], ef[F], ef[G]\} + 4 = \max\{14,15,6\} + 4 = \mathbf{19}.$$

Program studinya memanjang dari 18 pekan menjadi 19, dan lintasan kritisnya berpindah dari $A \to D \to E \to H$ menjadi $A \to D \to F \to H$. **Algoritma (E) meninggalkan lintasan kritisnya**, memperoleh kelonggaran satu pekan; Basis Data bergabung ke dalamnya. Inilah perilaku khas lintasan kritis: memperpanjang tugas yang punya kelonggaran tidak berakibat apa-apa sampai kelonggarannya terpakai habis, dan sesudah itu lintasannya melompat.

12.13 Mendorong A, B, C lalu mengeluarkan berulang-ulang memberikan: keluarkan C (kirim, dorong G); keluarkan G (kirim); keluarkan B (kirim); keluarkan A (kirim, dorong D); keluarkan D

(kirim, dorong E lalu F); keluarkan F (kirim); keluarkan E (kirim, dorong H); keluarkan H. Urutannya:

$$C,\ G,\ B,\ A,\ D,\ F,\ E,\ H.$$

Itu **identik dengan urutan mendalam** Persoalan N dan sama sekali berbeda dari urutan berbasis antrean Persoalan O. Kemiripannya bukan kebetulan contoh ini semata: tumpukan membuat algoritma Kahn menyelam sedalam yang diizinkan derajat masuknya sebelum menjejak mundur, dan itulah yang dilakukan pencarian mendalam. Namun ia bukan teorema — keduanya dapat berbeda ketika sebuah simpul menjadi siap jauh sebelum ia dicapai — jadi pada DAG mana pun kesepakatannya harus diperiksa, bukan diandaikan.

12.14 Segitiga {1,2,3} **tidak bipartit**: 1–2–3–1 adalah siklus ganjil berpanjang 3. **Bintang** {4,5,6,7} **bipartit**, dengan pihak-pihak {4} dan {5,6,7}; setiap bintang bipartit, sebab ia sebuah pohon, dan setiap pohon bipartit (pohon tak punya siklus sama sekali, jadi tak punya siklus ganjil). **Siklus-lima** {8,9,10,11,12} **tidak bipartit**: 8–9–10–11–12–8 adalah siklus ganjil berpanjang 5. Jadi G_3 secara keseluruhan tidak bipartit, dan pewarnaan dua warnanya gagal pada dua dari ketiga komponennya secara saling bebas.

12.15 Tak ada jawaban tunggal; periksalah tulisan Anda terhadap kolom ketiga **Tabel 12.2**. Tulisan itu lengkap jika pembaca yang belum pernah melihat program Anda dapat, dari 400 kata Anda saja: menyebut persoalannya, menyebut masukannya beserta ukurannya, menyebut algoritmanya, mengatakan mengapa alternatif yang kentara ditolak, dan meramalkan waktu jalannya. Jika salah satu dari kelimanya hilang, tambahkanlah — masing-masingnya hanya memerlukan satu kalimat. Sesudah itu kenakan uji pemikul beban sekali lagi pada apa yang Anda tulis: jika metode naif yang Anda tolak ternyata selamat pada masukan seribu kali lebih besar, argumen untuk algoritma Anda memerlukan masukan yang lebih besar, atau persoalan yang lain.

12.12.1 Program-programnya

Kode 12.12 menghitung setiap bilangan yang dikutip dalam penyelesaian Soal Latihan 12.1, 12.3, 12.5, 12.7, 12.9, dan 12.11.

Kode 12.12 Program untuk Soal Latihan 12.1, 12.3, 12.5, 12.7, 12.9, dan 12.11. Kompilasilah dengan `g++ -std=c++17 -O2 -Wall -o exercises exercises.cpp` lalu jalankan `./exercises`.

```
// Program-program untuk soal latihan Bab 12.
//   Ex 12.1   sistem koin {1, 4, 6}: di mana penukaran uang secara greedy gagal?
//   Latihan 12.3   BFS dari sumber lain pada graf yang sama
//   Latihan 12.5   DFS dengan gelung luar dibalik: hutan lain, graf sama
//   Latihan 12.7   contoh aktivitas tempat "durasi terpendek dahulu" gagal
//   Latihan 12.9   menambah tetapan pada tiap bobot tak mengubah MST-nya;
//                  mengkuadratkannya belum tentu demikian
//   Latihan 12.11  union-find dengan dan tanpa penyatuan menurut peringkat
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <numeric>
#include <functional>
#include <climits>
using namespace std;

// ---------------------------------------------------------------- Ex 12.1
void ex1() {
    cout << "=== Ex 12.1  greedy change-making with the coins {1, 4, 6} ===\n";
    const vector<int> coins = {6, 4, 1};                    // yang terbesar lebih dahulu
    const int maxX = 30;
    vector<int> best(maxX + 1, INT_MAX); best[0] = 0;      // koin tersedikit, dengan
pemrograman dinamis
    for (int x = 1; x <= maxX; ++x)
        for (int c : coins) if (c <= x && best[x - c] + 1 < best[x]) best[x] = best[x - c]
+ 1;
    int bad = 0;
    for (int x = 1; x <= maxX; ++x) {
        int g = 0, r = x;
        for (int c : coins) { g += r / c; r %= c; }
        if (g > best[x]) {
            if (++bad <= 4) cout << "  amount " << setw(2) << x << ": greedy uses " << g
                                 << " coins, the optimum uses " << best[x] << "\n";
        }
    }
    cout << "  amounts 1.." << maxX << " on which greed loses: " << bad << "\n";
}
```

Kode 12.12 (Lanjutan) *Program untuk Soal Latihan 12.1, 12.3, 12.5, 12.7, 12.9, dan 12.11. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o exercises exercises.cpp lalu jalankan ./exercises.*

```
// ------------------------------------------------------------ Latihan 12.3
static const string UN = "ABCDEFGHI";
static const vector<pair<int,int>> UE = {
    {0,1},{0,2},{1,3},{1,4},{2,4},{2,5},{3,6},{4,6},{4,7},{5,7},{6,8},{7,8}
};
void ex3() {
    cout << "\n=== Ex 12.3  BFS from C instead of A ===\n";
    int n = (int)UN.size();
    vector<vector<int>> adj(n);
    for (auto [a,b] : UE) { adj[a].push_back(b); adj[b].push_back(a); }
    for (auto& l : adj) sort(l.begin(), l.end());
    for (int s : {0, 2}) {
        vector<int> d(n, INT_MAX), pi(n, -1);
        d[s] = 0; queue<int> Q; Q.push(s);
        while (!Q.empty()) { int u = Q.front(); Q.pop();
            for (int v : adj[u]) if (d[v] == INT_MAX) { d[v] = d[u]+1; pi[v] = u;
Q.push(v); } }
        cout << "  source " << UN[s] << "   d: ";
        for (int v = 0; v < n; ++v) cout << UN[v] << "=" << d[v] << " ";
        cout << "\n            pi: ";
        for (int v = 0; v < n; ++v) cout << UN[v] << "<-" << (pi[v] < 0 ? '-' : UN[pi[v]])
<< " ";
        cout << "\n";
    }
    cout << "  the eccentricity of A is 4 and of C is 3, so the two trees differ in
shape\n";
}

// ------------------------------------------------------------ Latihan 12.5
static const string DN = "ABCDEFG";
static const vector<pair<int,int>> DE = {
    {0,1},{0,2},{0,3},{1,3},{2,3},{2,4},{3,0},{4,5},{5,4},{6,2},{6,5}
};
void dfsWithOrder(const vector<int>& order, const string& label) {
    int n = (int)DN.size();
    vector<vector<int>> adj(n);
    for (auto [u,v] : DE) adj[u].push_back(v);
    for (auto& l : adj) sort(l.begin(), l.end());
    vector<int> col(n,0), d(n,0), f(n,0), pi(n,-1); vector<string> cls(DE.size());
    int clk = 0;
    function<void(int)> visit = [&](int u) {
        col[u] = 1; d[u] = ++clk;
        for (int v : adj[u]) {
            int idx = -1;
            for (size_t i = 0; i < DE.size(); ++i)
                if (DE[i].first == u && DE[i].second == v) idx = (int)i;
            if (col[v] == 0) { cls[idx] = "T"; pi[v] = u; visit(v); }
            else if (col[v] == 1) cls[idx] = "B";
            else cls[idx] = (d[u] < d[v] ? "F" : "C");
```

Kode 12.12 (Lanjutan) *Program untuk Soal Latihan 12.1, 12.3, 12.5, 12.7, 12.9, dan 12.11. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o exercises exercises.cpp lalu jalankan ./exercises.*

```
        }
        col[u] = 2; f[u] = ++clk;
    };
    for (int v : order) if (col[v] == 0) visit(v);
    cout << "  " << label << "\n    d/f: ";
    for (int v = 0; v < n; ++v) cout << DN[v] << "=" << d[v] << "/" << f[v] << " ";
    int t=0,b=0,fw=0,c=0;
    for (const string& s : cls) { if (s=="T") ++t; else if (s=="B") ++b; else if (s=="F")
++fw; else ++c; }
    cout << "\n    tree " << t << ", back " << b << ", forward " << fw << ", cross " << c
<< "\n    ";
    for (size_t i = 0; i < DE.size(); ++i)
        cout << DN[DE[i].first] << "->" << DN[DE[i].second] << ":" << cls[i] << " ";
    cout << "\n";
}
void ex5() {
    cout << "\n=== Ex 12.5  the same digraph, two outer-loop orders ===\n";
    vector<int> fwd(7); iota(fwd.begin(), fwd.end(), 0);
    vector<int> rev = fwd; reverse(rev.begin(), rev.end());
    dfsWithOrder(fwd, "outer loop A,B,C,D,E,F,G");
    dfsWithOrder(rev, "outer loop G,F,E,D,C,B,A");
    cout << "  the COUNT of back edges changed (2 then 4) but neither is zero:\n"
            "  the verdict 'not acyclic' is order-independent, the classification is
not\n";
}

// ------------------------------------------------------------ Latihan 12.7
struct Act { int i, s, f; };
bool compat(const Act& a, const Act& b) { return a.f <= b.s || b.f <= a.s; }
void ex7() {
    cout << "\n=== Ex 12.7  an instance where 'shortest duration first' fails ===\n";
    vector<Act> A = { {1,0,5}, {2,4,6}, {3,5,10} };
    cout << "  a1=[0,5), a2=[4,6), a3=[5,10)\n";
    vector<Act> left = A; vector<int> pick;
    while (!left.empty()) {
        const Act* best = &left[0];
        for (const Act& a : left) if (a.f - a.s < best->f - best->s) best = &a;
        Act p = *best; pick.push_back(p.i);
        vector<Act> keep; for (const Act& a : left) if (a.i != p.i && compat(a, p))
keep.push_back(a);
        left = keep;
    }
    sort(pick.begin(), pick.end());
    cout << "  shortest duration picks a2 (2 units) and then nothing:";
    for (int x : pick) cout << " a" << x;
    cout << "  -> size " << pick.size() << "\n";
    left = A; vector<int> pick2;
    while (!left.empty()) {
        const Act* best = &left[0];
        for (const Act& a : left) if (a.f < best->f) best = &a;
```

Kode 12.12 (Lanjutan) *Program untuk Soal Latihan 12.1, 12.3, 12.5, 12.7, 12.9, dan 12.11. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o exercises exercises.cpp lalu jalankan ./exercises.*

```
        Act p = *best; pick2.push_back(p.i);
        vector<Act> keep; for (const Act& a : left) if (a.i != p.i && compat(a, p))
keep.push_back(a);
        left = keep;
    }
    sort(pick2.begin(), pick2.end());
    cout << "  earliest finish picks:";
    for (int x : pick2) cout << " a" << x;
    cout << "  -> size " << pick2.size() << "\n";
}

// ------------------------------------------------------------ Latihan 12.9
struct E { int a, b; long long w; };
static const string NM = "ABCDEFGH";
struct DSU {
    vector<int> p, r;
    explicit DSU(int n) : p(n), r(n,0) { iota(p.begin(), p.end(), 0); }
    int find(int x) { return p[x]==x ? x : p[x]=find(p[x]); }
    bool unite(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return false;
        if (r[x] < r[y]) swap(x, y);
        p[y] = x;
        if (r[x] == r[y]) ++r[x];
        return true;
    }
};
pair<long long,string> mstOf(vector<E> e, int n) {
    sort(e.begin(), e.end(), [](const E& x, const E& y){
        if (x.w != y.w) return x.w < y.w;
        if (x.a != y.a) return x.a < y.a;
        return x.b < y.b; });
    DSU d(n); long long t = 0; string s;
    for (const E& x : e) if (d.unite(x.a, x.b)) { t += x.w; s += string(1,NM[x.a]) +
string(1,NM[x.b]) + " "; }
    return {t, s};
}
void ex9() {
    cout << "\n=== Ex 12.9  transforming the weights ===\n";
    vector<E> G = { {0,1,3},{0,2,5},{1,2,1},{1,3,6},{2,3,4},{2,4,9},{3,4,2},
                    {3,5,8},{4,5,7},{4,6,10},{5,6,3},{5,7,12},{6,7,11} };
    auto base = mstOf(G, 8);
    cout << "  original       weight " << setw(5) << base.first << "   tree " <<
base.second << "\n";
    vector<E> plus = G; for (E& e : plus) e.w += 10;
    auto p = mstOf(plus, 8);
    cout << "  every w + 10   weight " << setw(5) << p.first << "   tree " << p.second
         << (p.second == base.second ? " (same tree)" : " (DIFFERENT)") << "\n";
    cout << "                 check: " << base.first << " + 7*10 = " << base.first + 70 <<
"\n";
```

Kode 12.12 (Lanjutan) *Program untuk Soal Latihan 12.1, 12.3, 12.5, 12.7, 12.9, dan 12.11. Kompilasilah dengan g++ -std=c++17 -O2 -Wall -o exercises exercises.cpp lalu jalankan ./exercises.*

```
    vector<E> sq = G; for (E& e : sq) e.w = e.w * e.w;
    auto q = mstOf(sq, 8);
    cout << "  every w square weight " << setw(5) << q.first << "   tree " << q.second
         << (q.second == base.second ? " (same tree)" : " (DIFFERENT)") << "\n";
    vector<E> neg = G; for (E& e : neg) e.w = 20 - e.w;
    auto r = mstOf(neg, 8);
    cout << "  every 20 - w   weight " << setw(5) << r.first << "   tree " << r.second
         << (r.second == base.second ? " (same tree)" : " (DIFFERENT)") << "\n";
}

// ------------------------------------------------------------ Latihan 12.11
void ex11() {
    cout << "\n=== Ex 12.11  union by rank, measured ===\n";
    int n = 100000;
    {   // naif: pasang akar pertama di bawah yang kedua, tanpa pemampatan
        vector<int> p(n); iota(p.begin(), p.end(), 0);
        function<int(int)> find = [&](int x){ while (p[x] != x) x = p[x]; return x; };
        for (int i = 1; i < n; ++i) { int a = find(i - 1), b = find(i); if (a != b) p[a] =
b; }
        long long steps = 0;
        for (int i = 0; i < n; ++i) { int x = i; while (p[x] != x) { x = p[x]; ++steps; } }
        cout << "  no rank, no compression:  total parent hops for " << n
             << " finds = " << steps << "\n";
    }
    {
        DSU d(n);
        for (int i = 1; i < n; ++i) d.unite(i - 1, i);
        long long steps = 0;
        for (int i = 0; i < n; ++i) { int x = i; while (d.p[x] != x) { x = d.p[x]; ++steps;
} }
        cout << "  union by rank + compression: total parent hops = " << steps << "\n";
    }
}

int main() { ex1(); ex3(); ex5(); ex7(); ex9(); ex11(); return 0; }
```

Keluarannya, yang menjadi asal setiap bilangan yang dikutip dalam penyelesaian di atas:

```
=== Ex 12.1  greedy change-making with the coins {1, 4, 6} ===
  amount  8: greedy uses 3 coins, the optimum uses 2
  amount  9: greedy uses 4 coins, the optimum uses 3
  amount 14: greedy uses 4 coins, the optimum uses 3
  amount 15: greedy uses 5 coins, the optimum uses 4
  amounts 1..30 on which greed loses: 8

=== Ex 12.3  BFS from C instead of A ===
  source A   d: A=0 B=1 C=1 D=2 E=2 F=2 G=3 H=3 I=4
            pi: A<-- B<-A C<-A D<-B E<-B F<-C G<-D H<-E I<-G
  source C   d: A=1 B=2 C=0 D=3 E=1 F=1 G=2 H=2 I=3
            pi: A<-C B<-A C<-- D<-B E<-C F<-C G<-E H<-E I<-G
  the eccentricity of A is 4 and of C is 3, so the two trees differ in shape

=== Ex 12.5  the same digraph, two outer-loop orders ===
  outer loop A,B,C,D,E,F,G
    d/f: A=1/12 B=2/5 C=6/11 D=3/4 E=7/10 F=8/9 G=13/14
    tree 5, back 2, forward 1, cross 3
    A->B:T A->C:T A->D:F B->D:T C->D:C C->E:T D->A:B E->F:T F->E:B G->C:C G->F:C
  outer loop G,F,E,D,C,B,A
    d/f: A=4/7 B=5/6 C=2/13 D=3/8 E=9/12 F=10/11 G=1/14
    tree 6, back 4, forward 1, cross 0
    A->B:T A->C:B A->D:B B->D:B C->D:T C->E:T D->A:T E->F:T F->E:B G->C:T G->F:F
  the COUNT of back edges changed (2 then 4) but neither is zero:
  the verdict 'not acyclic' is order-independent, the classification is not

=== Ex 12.7  an instance where 'shortest duration first' fails ===
  a1=[0,5), a2=[4,6), a3=[5,10)
  shortest duration picks a2 (2 units) and then nothing: a2  -> size 1
  earliest finish picks: a1 a3  -> size 2

=== Ex 12.9  transforming the weights ===
  original       weight    31   tree BC DE AB FG CD EF GH
  every w + 10   weight   101   tree BC DE AB FG CD EF GH  (same tree)
                 check: 31 + 7*10 = 101
  every w square weight   209   tree BC DE AB FG CD EF GH  (same tree)
  every 20 - w   weight    79   tree FH GH EG CE DF BD AC  (DIFFERENT)

=== Ex 12.11  union by rank, measured ===
  no rank, no compression:  total parent hops for 100000 finds = 4999950000
  union by rank + compression: total parent hops = 99999
```

12.12.2 Bacaan lanjutan

Penyajian yang menjadi acuan untuk bahan Bab 9 sampai 11, dan karena itu juga untuk bab ini, adalah Cormen, Leiserson, Rivest, dan Stein [1], yang Bab 15, 19, 20, dan 21-nya membahas berturut-turut algoritma *greedy*, himpunan saling lepas, algoritma graf dasar, dan pohon merentang minimum. Persoalan meminimumkan keterlambatan maksimum pada Persoalan C dibahas, dengan pembuktian pertukaran bersebelahan yang sama, oleh Kleinberg dan Tardos [7]. Teori yang menghubungkan matroid dengan algoritma *greedy*, di balik Persoalan F, berasal dari Edmonds [9]. Fakta tentang sistem koin kanonis yang dikutip pada Persoalan A berasal dari Kozen dan Zaks [5] serta Pearson [6]. Makalah asli Huffman tahun 1952 [8] panjangnya empat halaman dan sepenuhnya mudah dibaca. Makalah pengurutan topologis Kahn [4] berasal dari tahun 1962 dan dipicu persis oleh persoalan penjadwalan §12.2. Kruskal [2] dan Prim [3] adalah makalah aslinya dari tahun 1956 dan 1957; Tarjan [11] adalah asal batas $\alpha(n)$ pada Persoalan V dan Soal Latihan 12.11. Persoalan mencacah pada Persoalan P bersifat #P-lengkap, sebagaimana dibuktikan Brightwell dan Winkler [10]. Untuk matematika diskret di balik rumus Cayley dan penjumlahan yang dipakai di sepanjang bab, lihat Rosen [13].

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, Feb. 1956.

[3] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, Nov. 1957.

[4] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962.

[5] D. Kozen and S. Zaks, "Optimal bounds for the change-making problem," *Theoretical Computer Science*, vol. 123, no. 2, pp. 377–388, 1994.

[6] D. Pearson, "A polynomial-time algorithm for the change-making problem," *Operations Research Letters*, vol. 33, no. 3, pp. 231–234, 2005.

[7] J. Kleinberg and É. Tardos, *Algorithm Design*. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[8] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[9] J. Edmonds, "Matroids and the greedy algorithm," *Mathematical Programming*, vol. 1, no. 1, pp. 127–136, 1971.

[10] G. Brightwell and P. Winkler, "Counting linear extensions," *Order*, vol. 8, no. 3, pp. 225–242, Sep. 1991.

[11] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, Apr. 1975.

[12] Republic of Indonesia, *Undang-Undang Republik Indonesia Nomor 27 Tahun 2022 tentang Pelindungan Data Pribadi* (Personal Data Protection Law), Jakarta, Indonesia, 2022.

[13] K. H. Rosen, *Discrete Mathematics and Its Applications*, 8th ed. New York, NY, USA: McGraw-Hill Education, 2019.

Bab 13
Lintasan Terpendek Sumber Tunggal

Di akhir bab ini Anda akan sanggup menyatakan persoalan lintasan terpendek dengan cukup cermat untuk diperdebatkan, dan menyebutkan apa jawabannya bahkan pada kasus-kasus yang jawabannya tidak ada. Anda akan sanggup membuktikan bahwa setiap sublintasan sebuah lintasan terpendek adalah lintasan terpendek pula, dan membuktikan ketaksamaan segitiga yang diam-diam menjadi sandaran setiap algoritma lintasan terpendek. Anda akan tahu persis apa yang diperbuat sebuah sisi berbobot negatif terhadap persoalan ini (tidak apa-apa) dan apa yang diperbuat sebuah siklus negatif terhadapnya (menghancurkannya), dan Anda akan sanggup membedakan keduanya di dalam kode. Anda akan sanggup menuliskan satu-satunya primitif — **relaksasi** — yang darinya setiap algoritma dalam bab ini dibangun, lalu membuktikan kelima sifat yang membuatnya aman. Anda akan sanggup menjalankan algoritma Dijkstra dengan tangan pada graf contoh bab ini, sambil mencetak antrean prioritasnya pada setiap langkah, membuktikannya benar, menghasilkan sebuah graf yang di atasnya algoritma itu diam-diam memberikan jawaban yang salah, dan menyebutkan waktu jalannya untuk tiga implementasi antrean yang berbeda. Anda akan sanggup melakukan hal yang sama untuk Bellman–Ford: menelusurinya sapuan demi sapuan, membuktikan bahwa $|V| - 1$ sapuan sudah cukup, dan menerangkan mengapa satu sapuan tambahan mendeteksi siklus negatif. Anda akan sanggup menyelesaikan persoalannya dalam $\Theta(V + E)$ pada graf berarah asiklik tanpa antrean prioritas sama sekali. Dan Anda akan sanggup memulihkan rutenya yang sesungguhnya, bukan sekadar panjangnya, dari larik pendahulu — yang, bagaimanapun juga, itulah yang sebenarnya diinginkan siapa pun yang mengajukan pertanyaan itu.

13.1 Dari Mana Kita Melanjutkan

Rujukan baku untuk bahan ini adalah Bab 22 Cormen, Leiserson, Rivest, dan Stein [1], dan gambar-gambar dalam bab ini mengikuti kebiasaan menggambar buku itu dengan dekat, sehingga pembaca yang kelak membuka buku tersebut akan langsung merasa akrab. Ada dua bab terdahulu yang mengalir langsung ke dalam bab ini, dan ada baiknya disebutkan dengan tepat apa yang disumbangkan masing-masing.

13.1.1 Apa yang terbawa dari Bab 10 dan 11

Dari **Bab 10** kita mengambil grafnya beserta kosakatanya. Sebuah **graf berarah** (atau *digraph*) $G = (V, E)$ adalah sebuah pasangan: sehimpunan berhingga V berisi **simpul** (*vertices*) dan sehimpunan $E \subseteq V \times V$ berisi **sisi** (*edges*), yang tiap sisinya merupakan pasangan *terurut*, sehingga (u, v) dan (v, u) adalah dua hal yang berbeda. Kita menuliskan $|V|$ untuk banyaknya simpul dan $|E|$ untuk banyaknya sisi, dan kita memelihara kebiasaan buku ini untuk melepas garis tegaknya di dalam notasi asimtotik: $O(V + E)$ selalu berarti $O(|V| + |E|)$. Sebuah **lintasan** dari v_1 ke v_k adalah barisan simpul $p = \langle v_1, v_2, \dots, v_k \rangle$ yang setiap pasangan berurutannya merupakan sebuah sisi, yaitu $(v_i, v_{i+1}) \in E$ untuk $i = 1,2, \dots, k - 1$. **Panjang**-nya adalah banyaknya sisi, yaitu $k - 1$. Sebuah lintasan disebut **siklus** bila $v_1 = v_k$ dan $k > 1$. Kita menyimpan graf di dalam **daftar ketetanggaan**: $Adj[u]$ adalah daftar simpul yang ditunjuk oleh u.

Bab 10 juga memberi kita *breadth-first search* (penelusuran melebar), yang menghitung, dalam waktu $O(V + E)$, *banyaknya sisi* yang paling sedikit dari sebuah sumber s ke setiap simpul lain. Itulah persoalan lintasan terpendek untuk kasus khusus yang setiap sisinya bernilai sama. Bab ini adalah apa yang terjadi ketika sisi-sisinya berhenti bernilai sama.

Dari **Bab 11** kita mengambil bobotnya. Sebuah **graf berbobot** adalah graf beserta sebuah **fungsi bobot** $w: E \to \mathbb{R}$ yang memberikan sebuah bilangan real kepada tiap sisi. Bab 11 memakai bobot untuk bertanya *berapa himpunan sisi termurah yang menjaga semuanya tetap terhubung?* — pertanyaan pohon merentang minimum — dan menjawabnya dengan Prim–Jarník dan Kruskal. Bab 11 juga memberi kita **antrean prioritas**, dengan ketiga operasinya `Insert`, `ExtractMin`, dan `DecreaseKey`, beserta ketiga implementasinya yang ongkosnya akan kita perlukan lagi di sini: larik tak terurut, *binary heap*, dan *Fibonacci heap*. Dan Bab 11 memberi kita **pengurutan topologis**, yang mendaftarkan simpul-simpul sebuah graf berarah asiklik dalam urutan yang menghormati setiap anak panah; §13.9 memakainya secara langsung.

> **DEFINISI (diulang dari Bab 11).** Sebuah **antrean prioritas** atas sehimpunan Q berisi elemen yang masing-masing memegang sebuah *kunci* berupa bilangan mendukung: `Insert(Q, v)`, yang menambahkan v; `ExtractMin(Q)`, yang membuang lalu mengembalikan elemen berkunci terkecil; dan `DecreaseKey(Q, v, k)`, yang menurunkan kunci v menjadi k. Ketiga implementasi yang kita pakai berongkos, per operasi: larik tak terurut — $\Theta(V)$, $\Theta(1)$; *binary heap* — $O(\lg V)$, $O(\lg V)$; *Fibonacci heap* — $O(\lg V)$ teramortisasi, $\Theta(1)$ teramortisasi.

13.1.2 Apa yang baru

Bab 11 mengukur *seluruh jaringan*: ia mencari cara termurah untuk menjaga semuanya tetap tersambung, dan jawabannya berupa sehimpunan sisi tanpa titik awal tertentu. Bab ini mengukur *sebuah perjalanan*. Ada tempat Anda memulai, ada tempat-tempat yang ingin Anda capai, dan pertanyaannya adalah: berapa ongkosnya untuk sampai ke sana, dan lewat rute yang mana?

Itu bukan pertanyaan yang sama, dan ada baiknya perbedaannya diperjelas sejak awal, sebab mahasiswa mengacaukan keduanya setiap tahun. Pohon merentang minimum adalah *subgraf terhubung* yang termurah. Pohon lintasan terpendek adalah pohon berisi *rute termurah dari satu simpul tertentu*. Keduanya biasanya pohon yang berbeda, dan tidak satu pun di antaranya merupakan jawaban bagi yang lain. Soal Latihan 13.4 meminta Anda mencari graf berisi empat simpul yang membuat keduanya berbeda, dan Soal Latihan 13.5 meminta Anda membuktikan bahwa sebuah pohon lintasan terpendek dapat benar-benar lebih mahal, dalam bobot sisi total, daripada pohon merentang minimum graf yang sama.

13.2 Persoalan Lintasan Terpendek, Dinyatakan dengan Cermat

13.2.1 Dari “banyaknya sisi” ke “bobot total”

Penelusuran melebar menjawab “seberapa sedikit *sisi* jaraknya?”. Itu pertanyaan yang tepat ketika setiap sisi berukuran sama — ketika grafnya adalah jejaring sosial dan sebuah sisi berarti *mengenal*, atau sebuah labirin dan sebuah sisi berarti *satu langkah*. Ia menjadi pertanyaan yang salah begitu sisi-sisinya berbeda ukuran. Jalan antara dua kelurahan di Surabaya bisa dua kilometer atau dua belas; sebuah tautan jaringan bisa berongkos satu milidetik atau empat puluh; sebuah ruas perjalanan kurir bisa menghabiskan setengah liter bensin atau tiga. Ketika kita menempelkan bilangan semacam itu pada tiap sisi, pertanyaan alaminya berubah dari *berapa banyak sisi* menjadi *berapa banyak seluruhnya*.

Perumumannya cukup dinyatakan dalam satu baris: *perumumkan jarak ke keadaan berbobot*. Secara formal, kita diserahi sebuah graf berarah $G = (V, E)$ beserta sebuah fungsi bobot $w: E \to \mathbb{R}$ yang memberikan sebuah nilai real kepada tiap sisi. Perhatikan $\mathbb{R}$, bukan $\mathbb{R}^{\geq 0}$: definisinya mengizinkan bilangan negatif, dan §13.5 membahas berapa harga yang kita bayar untuk izin itu.

13.2.2 Bobot sebuah lintasan

DEFINISI. Misalkan $p = \langle v_1, v_2, \ldots, v_k \rangle$ sebuah lintasan di dalam graf berarah berbobot. **Bobot lintasan** p, yang ditulis $w(p)$, adalah jumlah bobot sisi-sisinya:

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}). \tag{13.1}$$

Bacalah Persamaan (13.1) dengan kata-kata sebelum Anda memakainya. Indeks i berjalan dari 1 sampai $k - 1$ — satu nilai untuk tiap *sisi* lintasannya, bukan untuk tiap simpul, dan itulah sebabnya batas atasnya $k - 1$ dan bukan k. Suku $w(v_i, v_{i+1})$ adalah bobot sisi yang menghantar dari simpul ke-i lintasan itu ke simpul ke-$(i + 1)$. Menjumlahkan semuanya memberikan biaya total menempuh seluruh lintasan dari ujung ke ujung. Lintasan berisi satu simpul, $p = \langle v_1 \rangle$, sama sekali tidak punya sisi, sehingga jumlahnya kosong dan $w(p) = 0$ — sebuah kesepakatan yang akan terus-menerus kita pakai, sebab kesepakatan itulah yang membuat $\delta(s, s) = 0$ keluar dengan benar.

CONTOH: Pada graf contoh Dijkstra di **Gambar 13.6**, ambillah lintasan $p = \langle s, x, u, v \rangle$. Ia punya tiga sisi. Bobotnya $w(s, x) = 5$, $w(x, u) = 3$, dan $w(u, v) = 1$. Jadi

$$w(p) = 5 + 3 + 1 = 9.$$

Ambil sebagai gantinya $q = \langle s, u, v \rangle$, dengan dua sisi berbobot $w(s, u) = 10$ dan $w(u, v) = 1$, sehingga $w(q) = 11$. Kedua lintasan itu berjalan dari s ke v; yang pertama lebih murah meskipun memakai lebih banyak sisi. Itulah seluruh inti pembobotan: *lebih sedikit sisi* dan *lebih sedikit biaya* adalah dua hal yang berbeda.

13.2.3 Bobot lintasan terpendek

DEFINISI. Bobot lintasan terpendek dari u ke v, yang ditulis $\delta(u, v)$ dan dibaca "delta u, v", adalah

$$\delta(u, v) = \begin{cases} \min\{\, w(p) : p \text{ adalah lintasan dari } u \text{ ke } v \,\} & \text{bila lintasan sem} \\ \infty & \text{bila tidak.} \end{cases}$$

(13.2)

Sebuah **lintasan terpendek** dari u ke v adalah sembarang lintasan p dari u ke v dengan $w(p) = \delta(u, v)$.

Ada tiga hal tentang definisi Persamaan (13.2), dan masing-masing pernah menjerat seseorang.

Pertama, δ adalah sebuah *bilangan*, sedangkan lintasan terpendek adalah sebuah *barisan simpul*. Keduanya jenis benda yang berbeda dan layak kita sediakan kata yang berbeda pula. Ketika kita berkata "hitunglah lintasan terpendeknya", yang biasanya kita maksud adalah "hitunglah semua bilangan $\delta(s, v)$, beserta pembukuan tambahan secukupnya untuk memulihkan satu lintasan bagi masing-masingnya".

Kedua, kata sandangnya adalah **sebuah** lintasan terpendek, bukan **satu-satunya**. Bisa ada banyak, semuanya berbobot sama. Pada graf contoh Dijkstra, $\delta(s, y) = 7$ lewat rute tunggal $\langle s, x, y \rangle$; tetapi seandainya kita tambahkan sisi (s, y) berbobot 7, akan ada dua lintasan terpendek dan tidak ada alasan untuk lebih menyukai salah satunya. Algoritma mengembalikan salah satu di antaranya dan tidak berkewajiban menyebutkan yang mana.

Ketiga, $\delta(u, v) = \infty$ bukan kegagalan, melainkan sebuah jawaban: artinya *sama sekali tidak ada rute*. Di dalam kode kita mewakilinya dengan sebuah nilai penjaga yang besar, dan setiap pembandingan harus ditulis sedemikian rupa sehingga menambahkan sebuah bobot kepada nilai penjaga itu tidak melimpah. Semua program dalam bab ini memakai deklarasi `const long long INF = LLONG_MAX / 4;` — seperempat dari nilai terbesar yang dapat ditampung sebuah `long long`, sehingga menambahkan bobot sisi yang wajar kepadanya masih muat — dan mengawal setiap relaksasi dengan `d[u] < INF`, yang persis itulah sebabnya program-program itu tidak pernah melingkar balik.

13.2.4 Sebuah contoh yang dikerjakan dengan tangan

Marilah kita hitung $\delta(s, v)$ untuk graf di **Gambar 13.6** dengan cara yang lambat dan jujur: dengan mendaftarkan setiap lintasan sederhana dari s ke v lalu menjumlahkannya.

Sisi-sisinya adalah $w(s, u) = 10$, $w(s, x) = 5$, $w(u, v) = 1$, $w(u, x) = 2$, $w(x, u) = 3$, $w(x, v) = 9$, $w(x, y) = 2$, $w(v, y) = 4$, $w(y, v) = 6$, $w(y, s) = 7$.

Tabel 13.1 Setiap lintasan sederhana dari s ke v di dalam graf contoh Dijkstra, ditimbang satu sisi setiap kalinya; minimumnya, 9, adalah $\delta(s, v)$, yang dicapai oleh $\langle s, x, u, v \rangle$. Mencacah lintasan memang benar tetapi mustahil dipakai secara umum: graf dengan n simpul dapat memiliki lebih dari $(n - 2)!$ lintasan sederhana antara sepasang simpul.

Lintasan dari s ke v	Bobot sisinya	Total
$\langle s, u, v \rangle$	$10 + 1$	11
$\langle s, x, v \rangle$	$5 + 9$	14
$\langle s, x, u, v \rangle$	$5 + 3 + 1$	9
$\langle s, x, y, v \rangle$	$5 + 2 + 6$	13
$\langle s, u, x, v \rangle$	$10 + 2 + 9$	21
$\langle s, u, x, y, v \rangle$	$10 + 2 + 2 + 6$	20

Jadi $\delta(s, v) = 9$. Mengapa barisnya tepat enam? Lintasan sederhana tidak boleh mengunjungi simpul yang sama dua kali, dan dari s hanya ada dua jalan keluar, ke u dan ke x; bila setiap cabang diikuti sampai tiba di v atau kehabisan simpul yang belum dikunjungi, hasilnya tepat keenam rute ini. (Jalan seperti $s \to x \to u \to v \to y \to v$ memang lebih panjang lagi, tetapi ia mengunjungi v dua kali, sehingga bukan lintasan sederhana dan tidak mungkin lebih pendek daripada lintasan sederhana yang dikandungnya.) Lihatlah berapa ongkosnya bagi kita: enam lintasan untuk graf berisi lima simpul. Pada tiga puluh simpul, cara yang sama akan mendaftarkan lebih banyak lintasan daripada banyaknya atom di dalam sebutir pasir. Algoritma-algoritma dalam bab ini memperoleh jawaban yang sama dalam waktu polinom terhadap $|V|$ dan $|E|$ — Dijkstra dalam beberapa ratus operasi mesin untuk graf sebesar ini — dan alasan mereka sanggup melakukannya adalah teorema struktural §13.3.

13.2.5 Keempat ragam persoalannya

Persoalan ini hadir dalam empat ragam, dan keempatnya berhubungan dengan cara yang tidak simetris.

DEFINISI. Diberi sebuah graf berarah berbobot $G = (V, E, w)$: persoalan **sepasang tunggal** menanyakan $\delta(u, v)$ beserta sebuah lintasan terpendek, untuk satu pasangan u, v yang diberikan; persoalan **sumber tunggal** menanyakan $\delta(s, v)$ beserta sebuah lintasan terpendek, untuk satu sumber s yang diberikan dan *setiap* $v \in V$; persoalan **tujuan tunggal** menanyakan $\delta(u, t)$ untuk satu tujuan t yang diberikan dan setiap $u \in V$; persoalan **semua pasangan** menanyakan $\delta(u, v)$ untuk setiap pasangan terurut $u, v \in V$.

Tabel 13.2 menjajarkan keempatnya, beserta reduksi yang dipakai masing-masing.

Tabel 13.2 Keempat persoalan lintasan terpendek. Tiga dari empatnya tereduksi ke yang kedua, dan itulah sebabnya yang kedua itulah yang kita selesaikan.

Ragam	Yang ditetapkan	Yang ditanyakan	Cara kita menyelesaikannya
Sepasang tunggal	sumber s, sasaran t	$\delta(s, t)$ dan satu lintasan	selesaikan sumber tunggal dari s, berhenti lebih awal bila Anda mau
Sumber tunggal	sumber s	$\delta(s, v)$ untuk semua v	bab ini: Dijkstra, Bellman–Ford, relaksasi DAG
Tujuan tunggal	sasaran t	$\delta(u, t)$ untuk semua u	balikkan setiap sisi, lalu selesaikan sumber tunggal dari t
Semua pasangan	tidak ada	$\delta(u, v)$ untuk semua u, v	Bab 15: Floyd–Warshall, Johnson; atau $\lvert V \rvert$ kali jalankan bab ini

13.2.6 Mengapa sumber tunggal yang kita selesaikan

Persoalan sepasang tunggal tampak lebih mudah daripada persoalan sumber tunggal — tentunya menghitung satu bilangan lebih ringan daripada menghitung $|V|$ bilangan? — dan kejutannya adalah tidak seorang pun tahu bagaimana memanfaatkan keringanan itu. Setiap algoritma yang kita miliki untuk persoalan sepasang tunggal adalah algoritma sumber tunggal yang kebetulan diizinkan berhenti begitu sasarannya keluar dari antrean prioritas. Pada kasus terburuk, sasarannya adalah simpul terakhir yang keluar, dan Anda toh sudah menyelesaikan seluruh persoalan sumber tunggalnya. Ringkasnya: *penyelesaian persoalan sumber tunggal sekaligus menyelesaikan persoalan sepasang tunggal secara efisien*. Tidak ada metode yang lebih baik secara asimtotik yang diketahui untuk kasus umumnya, dan ini salah satu pertanyaan terbuka tertua dalam bidang ini.

Persoalan tujuan tunggal adalah persoalan sumber tunggal di dalam cermin. Bangunlah **graf transpos** $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$ dengan $E^{\mathrm{T}} = \{(v, u) : (u, v) \in E\}$, sambil mempertahankan bobot tiap sisinya, lalu jalankan sebuah algoritma sumber tunggal dari t di dalam G^{T}. Sebuah lintasan $u \rightsquigarrow t$ di dalam G persis sama dengan lintasan $t \rightsquigarrow u$ di dalam G^{T} dengan bobot yang sama, sehingga bilangan-bilangannya keluar identik. Membalikkan sisi-sisinya berongkos $\Theta(V + E)$, yang cuma-cuma dibandingkan dengan apa pun yang lain yang kita kerjakan.

Persoalan semua pasangan dapat diselesaikan dengan menjalankan sebuah algoritma sumber tunggal sekali dari tiap simpul — $|V|$ kali jalan. Untuk graf jarang dengan bobot tak negatif, cara itu sebenarnya bersaing: $|V|$ kali menjalankan Dijkstra dengan *Fibonacci heap* berongkos $O(V^2 \lg V + VE)$, yang mengalahkan $\Theta(V^3)$ milik Floyd–Warshall ketika E kecil. Bab 15 mengangkat hal ini dengan semestinya, termasuk algoritma Johnson, yang seluruh gagasannya adalah *membobot ulang* sebuah graf bersisi negatif supaya Dijkstra boleh dijalankan di atasnya secara sah.

Jadi: tiga dari empat ragamnya entah tereduksi ke sumber tunggal atau dibangun darinya. Itulah sebabnya bab ini menyelesaikan sumber tunggal dan tidak yang lain.

13.2.7 Di mana persoalan ini sungguh-sungguh muncul

Tiga penerapan klasik layak diberi satu kalimat masing-masing, sebab masing-masing menekan persoalannya dengan cara yang berbeda.

Teori umum di balik semuanya — lintasan terpendek sebagai kasus paling sederhana dari aliran jaringan berongkos minimum — adalah pokok bahasan Ahuja, Magnanti, dan Orlin [2], dan pembahasan algoritmis yang lebih lembut atas lahan yang sama adalah Kleinberg dan Tardos [3].

Perutean jaringan statis dan dinamis. Setiap paket yang menyeberangi internet dirutekan oleh sebuah algoritma keturunan dari kedua algoritma dalam bab ini. Protokol *link-state* OSPF membuat tiap penghala membangun peta seluruh jaringannya lalu menjalankan algoritma Dijkstra di atasnya. Protokol *distance-vector* RIP membuat tiap penghala hanya menyimpan tabel jarak lalu menukarkannya dengan tetangganya — itulah Bellman–Ford, dijalankan secara tersebar, satu sapuan per ronde pertukaran. Kedua keluarga protokol itu, harfiah sekali, adalah kedua algoritma bab ini dengan tumpukan jaringan terpasang di sekelilingnya.

Perencanaan gerak robot. Diskretkan lantainya menjadi sebuah kisi atau sebuah graf penanda tempat, bobotilah tiap sisi dengan waktu atau energi yang diperlukan robotnya untuk melintasinya, dan rute aman yang termurah adalah sebuah lintasan terpendek. Bobot di sini secara alami tak negatif, sehingga Dijkstra berlaku.

Pembuatan peta dan rute di dalam lalu lintas. Inilah penerapan yang ada di saku semua orang. Bobotnya adalah waktu tempuh, yang berubah sepanjang hari, sehingga grafnya dibobot ulang terus-menerus dan pertanyaannya berbentuk sepasang tunggal. Sistem produksi memakai varian yang direkayasa habis-habisan — A* dengan heuristik geometris, *contraction hierarchies* — tetapi kebenaran semuanya diperdebatkan persis dari sifat-sifat yang dibuktikan di §13.6.

Pada daftar itu tambahkan dua lagi yang akan terus dijumpai dalam buku ini. **Penjadwalan dengan pendahuluan:** bila tugas-tugasnya membentuk DAG dan bobot sisinya adalah durasi, maka lintasan *terpanjang* adalah lintasan kritis proyeknya — §13.9.7 menunjukkan bahwa itu algoritma yang sama dengan tandanya dibalik. Dan **arbitrase mata uang:** ambillah kurs, bobotilah tiap sisi dengan $-\ln(\text{kurs})$, dan sebuah siklus negatif persis merupakan serangkaian transaksi yang mengembalikan uang lebih banyak daripada yang dimulainya. Pendeteksi siklus negatif Bellman–Ford adalah pendeteksi arbitrase.

13.3 Substruktur Optimal Lintasan Terpendek

Segala sesuatu dalam bab ini bertumpu pada satu fakta struktural, dan ia layak dinyatakan sebelum algoritma mana pun, sebab setiap algoritma adalah pemanfaatan atasnya.

13.3.1 Teoremanya

> **TEOREMA 13.1 (substruktur optimal lintasan terpendek).** Sublintasan dari lintasan terpendek adalah lintasan terpendek. Tepatnya: misalkan $p = \langle v_1, v_2, \dots, v_k \rangle$ sebuah lintasan terpendek dari v_1 ke v_k, dan untuk sembarang $1 \le i \le j \le k$ misalkan $p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ adalah sublintasan p dari v_i ke v_j. Maka p_{ij} adalah lintasan terpendek dari v_i ke v_j.

Sebelum buktinya, perjelaslah apa yang sedang diklaim, sebab pernyataan ini mudah diangguki dan sukar dipakai. Ia berkata: bila cara termurah dari Depot ke Keputih kebetulan melewati ITS, maka bagian perjalanan itu dari Depot ke ITS dengan sendirinya adalah cara termurah dari Depot ke ITS, dan bagian dari ITS ke Keputih dengan sendirinya adalah cara termurah dari ITS ke

Keputih. Bukan *sebuah* cara yang murah — melainkan yang *paling* murah. Anda tidak dapat membangun keseluruhan yang optimal dari bagian yang tidak optimal.

13.3.2 Buktinya, dengan menggunting dan menempel

Seluruh buktinya muat dalam satu baris: *seandainya suatu sublintasan bukan lintasan terpendek, kita dapat menggantinya dengan sublintasan yang lebih pendek dan memperoleh lintasan total yang lebih pendek*. Inilah bukti itu selengkapnya, selangkah demi selangkah.

Bukti. Uraikan lintasan p menjadi tiga potong berurutan pada kedua simpul yang dipilih:

$$v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k.$$

Setiap sisi p terletak tepat pada salah satu dari ketiga potongan itu, sehingga bobotnya menjumlah:

$$w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk}). \tag{13.3}$$

Sekarang andaikan, demi memperoleh kontradiksi, bahwa p_{ij} *bukan* lintasan terpendek dari v_i ke v_j. Maka menurut definisi δ ada suatu lintasan p'_{ij} dari v_i ke v_j dengan

$$w(p'_{ij}) < w(p_{ij}). \tag{13.4}$$

Bangunlah lintasan baru p' dari v_1 ke v_k dengan *menggunting keluar* p_{ij} lalu *menempelkan masuk* p'_{ij}:

$$v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k.$$

Ini lintasan yang sah: p_{1i} berakhir di v_i dan p'_{ij} bermula di v_i, sehingga keduanya bersambung; p'_{ij} berakhir di v_j dan p_{jk} bermula di v_j, sehingga keduanya bersambung pula. Bobotnya adalah

$$\begin{aligned} w(p') = w(p_{1i}) + w(p'_{ij}) + w(p_{jk}) \;&<\; w(p_{1i}) + w(p_{ij}) + w(p_{jk}) \\ &= \; w(p), \end{aligned}$$

dengan ketaksamaan tegasnya berasal dari Persamaan (13.4) dan kesamaan terakhirnya adalah Persamaan (13.3). Jadi $w(p') < w(p)$: kita telah menghasilkan lintasan dari v_1 ke v_k yang benar-benar lebih murah daripada p. Tetapi p diandaikan sebagai lintasan *terpendek* dari v_1 ke v_k, sehingga lintasan semacam itu tidak ada. Kontradiksi. Karena itu tidak ada p'_{ij} yang lebih murah, dan p_{ij} adalah lintasan terpendek dari v_i ke v_j. ■

13.3.3 Membaca buktinya sekali lagi, pelan-pelan

Argumennya disebut **gunting-dan-tempel** dan Anda sudah berjumpa dengannya di Bab 9 untuk algoritma *greedy* dan di Bab 7 untuk pemrograman dinamis. Ia selalu punya empat gerakan yang sama, dan keempatnya layak disebut namanya, sebab Anda perlu sanggup mengulanginya sendiri, dan Bab 16 membahas soal-soal yang menuntut persis argumen ini.

1. **Andaikan subpenyelesaiannya tidak optimal.** Di sini: andaikan p_{ij} dapat dikalahkan.

2. **Ambillah subpenyelesaian yang lebih baik yang diserahkan oleh pengandaian itu.** Di sini: p'_{ij} dengan $w(p'_{ij}) < w(p_{ij})$.
3. **Sambungkan ia ke dalam keseluruhannya.** Di sini: ganti bagian tengahnya lalu periksalah bahwa hasilnya masih lintasan yang sah — pada langkah inilah argumennya dapat gagal untuk persoalan lain, jadi jangan pernah lewatkan pemeriksaannya.
4. **Amati bahwa keseluruhannya membaik, yang bertentangan dengan keoptimalannya.**

Langkah 3 itulah yang layak dicurigai. Ia berhasil di sini sebab sebuah lintasan tidak lain adalah barisan sisi dan dua lintasan mana pun yang berbagi satu titik ujung dapat dirangkaikan. Ia *tidak* berhasil untuk, misalnya, lintasan **sederhana** terpanjang: menyambungkan sublintasan yang lebih murah dapat menengok kembali sebuah simpul dan membuat hasilnya tidak sederhana lagi. Itulah persis sebabnya persoalan lintasan sederhana terpanjang bersifat NP-sukar sedangkan persoalan lintasan terpendek mudah, dan perbedaan itu layak Anda bawa terus.

13.3.4 Apa yang kita peroleh dari substruktur optimal

Substruktur optimal adalah lisensi untuk membangun lintasan terpendek *sedikit demi sedikit*. Bila saya tahu sebuah lintasan terpendek dari s ke u, dan (u, v) adalah sebuah sisi, maka yang terbaik yang dapat saya perbuat bagi v *melalui* u adalah $\delta(s, u) + w(u, v)$ — saya tidak perlu meninjau ulang rute ke u, sebab teoremanya menjamin bahwa awalan sebuah rute optimal ke v melalui u adalah rute optimal ke u. Setiap algoritma dalam bab ini adalah jawaban atas pertanyaan *dalam urutan apa sebaiknya saya menimbang simpul-simpulnya, supaya ketika saya memakai* $\delta(s, u)$ *nilainya sudah diketahui?* Dijkstra menjawab *dalam urutan jarak yang menaik*; Bellman–Ford menjawab *saya tidak tahu, jadi coba semuanya* $|V| - 1$ *kali*; algoritma DAG menjawab *dalam urutan topologis*. Itulah seluruh bab ini dalam tiga kalimat.

13.4 Ketaksamaan Segitiga

13.4.1 Pernyataannya

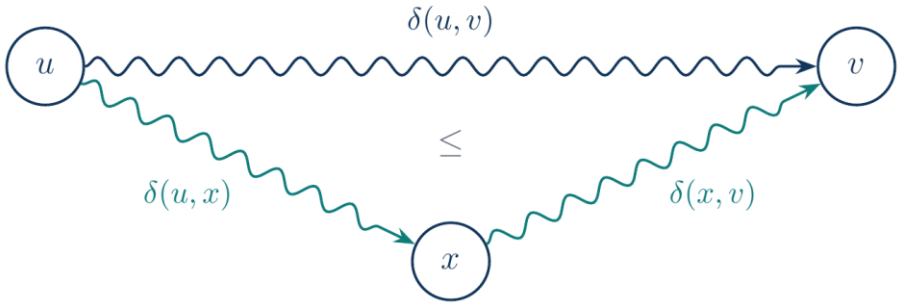
TEOREMA 13.2 (ketaksamaan segitiga, bentuk simpul). Untuk sembarang tiga simpul u, x, v sebuah graf berarah berbobot,

$$\delta(u, v) \leq \delta(u, x) + \delta(x, v). \tag{13.5}$$

Nama Persamaan (13.5) berasal dari geometri: sisi lurus sebuah segitiga tidak pernah lebih panjang daripada kedua sisi lainnya digabung. Versi grafnya mengatakan hal yang sama tentang rute. Pergi dari u ke v secara langsung (artinya, lewat rute terbaik yang tersedia) tidak pernah lebih buruk daripada dipaksa memutar melalui x.

13.4.2 Buktinya

Gambar 13.1 menggambarkan keadaannya: sebuah anak panah berombak lurus dari u ke v, dan rute kedua yang menyelam turun melalui x. Garis berombak adalah notasi baku untuk "suatu lintasan, kita tidak peduli yang mana".



Gambar 13.1 Ketaksamaan segitiga: anak panah berombak di atas adalah lintasan terpendek dari u ke v, pasangan di bawah adalah lintasan terpendek dari u ke x disusul lintasan terpendek dari x ke v. Rute bawah itu adalah *sebuah* lintasan dari u ke v, sehingga ia tidak mungkin lebih murah daripada yang terbaik.

Bukti. Ada tiga kasus, dan kasus-kasus yang merosot pun penting.

Kasus 1: entah $\delta(u,x) = \infty$ atau $\delta(x,v) = \infty$. Maka ruas kanannya ∞ dan ketaksamaannya berlaku secara sepele, berapa pun $\delta(u,v)$. (Inilah sebabnya kita mendefinisikan $\infty + c = \infty$.)

Kasus 2: keduanya berhingga, dan $\delta(u,v)$ berhingga. Misalkan p_1 lintasan terpendek dari u ke x, sehingga $w(p_1) = \delta(u,x)$, dan misalkan p_2 lintasan terpendek dari x ke v, sehingga $w(p_2) = \delta(x,v)$. Merangkaikan keduanya memberikan lintasan $q = p_1 p_2$ dari u ke v — sah, sebab p_1 berakhir di tempat p_2 bermula — dengan bobot

$$w(q) = w(p_1) + w(p_2) = \delta(u,x) + \delta(x,v).$$

Nah, $\delta(u,v)$ menurut definisinya adalah bobot *minimum* atas semua lintasan dari u ke v, dan q adalah salah satu lintasan semacam itu. Minimum atas sebuah himpunan paling banyak sama dengan sembarang anggota tertentu himpunan itu, sehingga

$$\delta(u,v) \;\leq\; w(q) \;=\; \delta(u,x) + \delta(x,v).$$

Kasus 3: kedua suku di ruas kanan berhingga tetapi $\delta(u,v) = \infty$. Ini tidak mungkin terjadi: Kasus 2 sudah memperlihatkan lintasan q yang sungguh-sungguh ada dari u ke v, sehingga himpunan lintasan dari u ke v tidak kosong dan minimumnya berhingga. ■

Versi satu barisnya — *lintasan terpendek $u \rightsquigarrow v$ tidak lebih panjang daripada lintasan $u \rightsquigarrow v$ mana pun yang lain, khususnya lintasan yang merangkaikan $u \rightsquigarrow x$ terpendek dengan $x \rightsquigarrow v$ terpendek* — persis adalah Kasus 2, dan itulah kalimat yang perlu diingat.

13.4.3 Bentuk sisinya, yang itulah yang kita pakai

Bagi algoritma, kasus khusus yang berguna adalah kasus yang memutarnya cuma satu sisi.

AKIBAT 13.3 (ketaksamaan segitiga, bentuk sisi). Untuk sembarang sisi $(u,v) \in E$ dan sembarang sumber s,

$$\delta(s,v) \;\leq\; \delta(s,u) + w(u,v).$$

(13.6)

Bukti. Terapkan Teorema 13.2 dengan tripel (s,u,v): $\delta(s,v) \leq \delta(s,u) + \delta(u,v)$. Lalu perhatikan bahwa sisi tunggal $\langle u,v \rangle$ adalah lintasan dari u ke v berbobot $w(u,v)$, sehingga $\delta(u,v) \leq w(u,v)$ menurut definisi δ sebagai sebuah minimum. Menyulihkannya memberikan hasilnya. ■

Tataplah Persamaan (13.6) sampai ia terasa jelas dengan sendirinya, sebab seluruh §13.6 dibangun di atasnya. Ia berkata: *jarak sebenarnya ke v tidak pernah dapat dikalahkan dengan pergi ke u secara optimal lalu mengambil satu sisi lagi.* Segala sesuatu yang dikerjakan algoritma-algoritma itu adalah berusaha membuat estimasi $d[v]$ mereka memenuhi ketaksamaan ini dengan kesamaan.

13.4.4 Sebuah pemeriksaan yang dikerjakan

Ambillah lagi graf contoh Dijkstra, yang jarak sebenarnya dari s-nya telah kita hitung sebagai $\delta(s,s) = 0$, $\delta(s,x) = 5$, $\delta(s,y) = 7$, $\delta(s,u) = 8$, $\delta(s,v) = 9$. Periksalah bentuk sisinya pada kesepuluh sisinya, seperti yang dikerjakan **Tabel 13.3**:

Tabel 13.3 Bentuk sisi ketaksamaan segitiga diperiksa pada setiap sisi graf contoh Dijkstra. Baris-baris yang menunjukkan *kesamaan* — (s,x), (x,u), (u,v), (x,y) — persis keempat sisi pohon lintasan terpendeknya, sebagaimana diterangkan §13.6.9.

Sisi (u,v)	$w(u,v)$	$\delta(s,u)+w(u,v)$	$\delta(s,v)$	Berlaku?
(s,u)	10	$0+10=10$	8	$8 \le 10$ ya
(s,x)	5	$0+5=5$	5	kesamaan
(u,v)	1	$8+1=9$	9	kesamaan
(u,x)	2	$8+2=10$	5	ya
(x,u)	3	$5+3=8$	8	kesamaan
(x,v)	9	$5+9=14$	9	ya
(x,y)	2	$5+2=7$	7	kesamaan
(v,y)	4	$9+4=13$	7	ya
(y,v)	6	$7+6=13$	9	ya
(y,s)	7	$7+7=14$	0	ya

13.5 Bobot Negatif dan Siklus Negatif

Definisi §13.2 mengizinkan w mengambil nilai negatif. Bagian ini membahas berapa harga izin itu. Perbedaan yang ditariknya — antara *sisi* negatif, yang tidak berbahaya, dan *siklus* negatif, yang mematikan — adalah satu hal yang paling sering disalahpahami dalam seluruh topik ini.

13.5.1 Sisi negatif bukan masalah

Bobot negatif bukan hal yang asing. Ruas perjalanan kurir yang berakhir di titik antar mitra yang membayar rabat berongkos bersih negatif. Reaksi kimia yang melepaskan energi punya perubahan energi negatif. Langkah dagang yang menghasilkan lebih banyak daripada yang dibelanjakannya berongkos negatif. Pada tiap kasus itu, aritmetika §13.2 berjalan tanpa tergores: $w(p)$ tetap sebuah jumlah, $\delta(u,v)$ tetap sebuah minimum, dan Teorema 13.1 serta 13.2 dibuktikan tanpa pernah mengandaikan $w \ge 0$. Tengoklah kembali dan periksalah: bukti gunting-dan-tempel itu hanya pernah *menjumlahkan* dan *membandingkan* bobot. Tidak ada di sana yang menuntut bobotnya positif.

Graf contoh Bellman–Ford di §13.8 punya tiga sisi negatif — $w(x,t) = -2$, $w(y,x) = -3$, dan $w(t,z) = -4$ — beserta jawaban yang terdefinisi sempurna, yang dihitung §13.8.

13.5.2 Siklus negatif itu mematikan

Sekarang andaikan grafnya memuat sebuah siklus c yang bobot totalnya negatif, $w(c) < 0$, dan andaikan c terjangkau dari sumber s — artinya, ada suatu lintasan dari s ke sebuah simpul pada c.

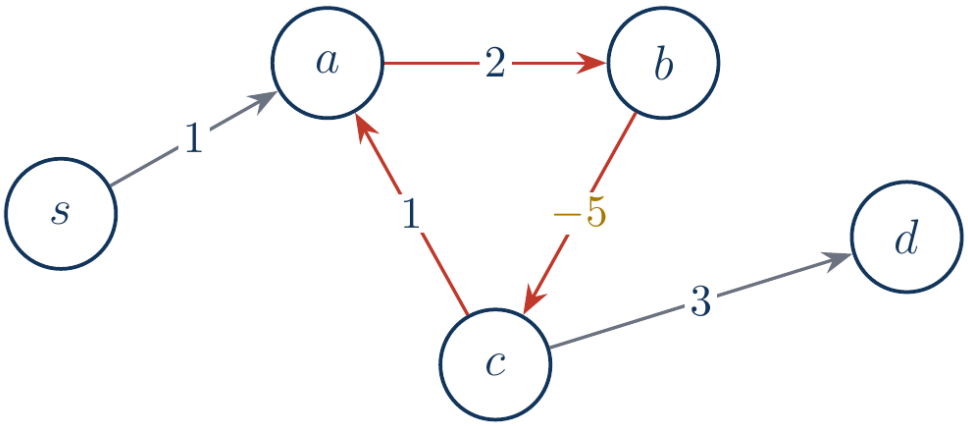
> **PERINGATAN:** Bila sebuah siklus berbobot negatif terjangkau dari s dan v terjangkau dari siklus itu, maka $\delta(s, v)$ *tidak ada* sebagai bilangan berhingga. Menurut kesepakatan kita menuliskan $\delta(s, v) = -\infty$.

Mengapa. Misalkan c siklusnya, misalkan x sembarang simpul padanya, misalkan p sebuah lintasan $s \rightsquigarrow x$, dan misalkan q sebuah lintasan $x \rightsquigarrow v$. Maka untuk setiap bilangan bulat $k \geq 0$ jalan

$$s \overset{p}{\rightsquigarrow} x \underbrace{\overset{c}{\rightsquigarrow} x \overset{c}{\rightsquigarrow} x \cdots \overset{c}{\rightsquigarrow} x}_{k \text{ putaran}} \overset{q}{\rightsquigarrow} v$$

berbobot $w(p) + k\,w(c) + w(q)$. Karena $w(c) < 0$, nilai itu mengecil tanpa batas seiring bertambahnya k: pilihlah k yang cukup besar dan "panjangnya" berada di bawah bilangan mana pun yang Anda sebutkan. Tidak ada minimumnya, sehingga $\delta(s, v)$ tidak terdefinisi, dan $-\infty$ adalah cara yang lazim untuk mencatat hal itu.

Gambar 13.2 memperlihatkan contoh menarik yang terkecil. Siklus $a \to b \to c \to a$ berbobot $2 + (-5) + 1 = -2$. Ia terjangkau dari s lewat sisi (s, a), dan d terjangkau darinya lewat sisi (c, d). Jadi $\delta(s, a) = \delta(s, b) = \delta(s, c) = \delta(s, d) = -\infty$, sedangkan $\delta(s, s) = 0$ — sumbernya sendiri aman hanya karena di sini tidak ada sisi yang menuntun kembali ke dalam s.



Gambar 13.2 Sebuah graf dengan siklus negatif yang terjangkau. Berkeliling siklus merah $a \to b \to c \to a$ berongkos $2 + (-5) + 1 = -2$, sehingga tiap putaran tambahan membuat "lintasannya" lebih murah dua. Setiap simpul yang terjangkau dari siklus itu — di sini a, b, c, dan d — memiliki $\delta(s, \cdot) = -\infty$. Persoalannya tidak sukar pada graf ini; ia *salah rumus*.

Perhatikan baik-baik apa yang *tidak* diklaim. Siklus negatif yang **tidak terjangkau** dari s sama sekali tidak berbahaya: Anda tidak dapat naik ke atasnya, sehingga ia tidak dapat merusak jarak mana pun dari s. Dan simpul yang tidak terjangkau *dari* siklus itu mempertahankan jarak berhingganya yang biasa. Bellman–Ford sebagaimana ditulis pada **Gambar 13.13** mengembalikan `false` begitu ia melihat siklus negatif terjangkau mana pun, dan itulah perilaku yang aman dan lazim; Soal Latihan 13.13 meminta Anda memodifikasinya supaya ia melabeli tiap simpul dengan $+\infty$, sebuah bilangan berhingga, atau $-\infty$ secara satu per satu, dan program penyelesaiannya mengerjakan persis itu.

13.5.3 Lintasan terpendek tidak punya siklus

Pengamatan kedua tentang bobot negatif adalah sebuah teorema yang sedang menyamar.

> **TEOREMA 13.4.** Bila grafnya tidak memuat siklus berbobot negatif yang terjangkau dari s, maka untuk setiap simpul v yang terjangkau dari s ada sebuah lintasan terpendek dari s ke v yang **sederhana** — tidak mengulang simpul mana pun — dan karena itu punya paling banyak $|V| - 1$ sisi.

Bukti. Misalkan p sembarang lintasan terpendek dari s ke v; menurut hipotesisnya $\delta(s, v)$ berhingga, sehingga lintasan semacam itu ada. Andaikan p mengulang sebuah simpul x. Maka p berbentuk

$$s \rightsquigarrow x \overset{c}{\rightsquigarrow} x \rightsquigarrow v$$

dengan c adalah siklus di antara kedua kunjungan ke x. Membuang c meninggalkan lintasan p' dari s ke v berbobot $w(p') = w(p) - w(c)$.

Nah, c terjangkau dari s (kita mencapainya menyusuri p), sehingga menurut hipotesisnya $w(c) \geq 0$. Karena itu $w(p') = w(p) - w(c) \leq w(p)$. Tetapi p adalah lintasan terpendek, sehingga $w(p') \geq \delta(s, v) = w(p)$. Kedua ketaksamaan itu memaksa $w(p') = w(p)$: lintasan yang lebih pendek itu *juga* lintasan terpendek. Ulangilah pemotongannya untuk setiap simpul yang terulang; tiap langkah membuang sekurang-kurangnya satu sisi, sehingga prosesnya berakhir, dan yang tersisa adalah lintasan terpendek yang sederhana.

Lintasan sederhana mengunjungi tiap simpul paling banyak sekali, sehingga ia punya paling banyak $|V|$ simpul dan karenanya paling banyak $|V| - 1$ sisi. ■

Diringkas menjadi semboyan: *lintasan terpendek mana pun di dalam graf G tidak mungkin lebih panjang daripada $n - 1$ sisi, dengan n adalah banyaknya simpul.* Hafalkan kalimat itu di luar kepala. Itulah sebabnya Bellman–Ford menjalankan tepat $|V| - 1$ sapuan — bukan $|V|$, bukan $|V| + 1$ — dan itu pula sebabnya sapuan ke-$|V|$ merupakan pendeteksi siklus negatif dan bukan kerja yang berguna.

> **CONTOH:** Graf contoh Dijkstra punya lima simpul, sehingga setiap lintasan terpendek dari s punya paling banyak empat sisi. Periksalah: yang terpanjang yang kita temukan adalah $\langle s, x, u, v \rangle$, tiga sisi. Graf contoh Bellman–Ford juga punya lima simpul, dan lintasan terpendeknya yang terpanjang adalah $\langle s, y, x, t, z \rangle$ — tepat empat sisi, sebanyak yang diizinkan. Itulah sebabnya contoh Bellman–Ford di §13.8.4 memerlukan keempat sapuannya dan akan salah dengan tiga sapuan.

13.5.4 Algoritma mana bertahan terhadap apa

Kita belum berjumpa dengan algoritmanya, tetapi **Tabel 13.4** layak ada di depan mata sejak awal, sebab baris yang penting adalah baris kedua.

Tabel 13.4 Apa yang dikerjakan masing-masing dari ketiga algoritma itu pada tiap rezim bobot. Kata yang perlu diperhatikan adalah *diam-diam*: Dijkstra tidak macet dan tidak mengeluh pada sisi negatif, ia mengembalikan bilangan salah yang tampak masuk akal. §13.7.3 membangun contoh terkecilnya.

Keadaan	Dijkstra	Bellman–Ford	Relaksasi DAG
Semua bobot ≥ 0	benar	benar	benar (bila asiklik)
Sebagian bobot < 0, tanpa siklus negatif	**salah, diam-diam**	benar	benar (bila asiklik)
Siklus negatif terjangkau dari s	salah, diam-diam	mendeteksinya, mengembalikan `false`	tidak mungkin muncul: DAG sama sekali tidak punya siklus
Siklus negatif tak terjangkau dari s	salah bila ada sisi negatif yang terjangkau	benar untuk bagian yang terjangkau	tidak mungkin muncul

13.6 Relaksasi: Satu-satunya Primitif

Setiap algoritma dalam bab ini — dan setiap algoritma dalam Bab 15 juga — dibangun dari satu operasi. Pelajarilah sekali di sini dan sisa bab ini tinggal soal pembukuan tentang *kapan* menerapkannya.

13.6.1 Kedua lariknya: d dan π

> **DEFINISI.** Untuk tiap simpul v kita memelihara dua nilai. $d[v]$, **estimasi lintasan terpendek**, adalah sebuah bilangan yang selalu merupakan *batas atas* bagi $\delta(s, v)$: bobot rute terbaik dari s ke v yang kebetulan sudah kita temukan sejauh ini. Ia bermula pada ∞ untuk setiap simpul kecuali s, yang bermula pada 0. $\pi[v]$, **pendahulu** (*predecessor*), adalah simpul tempat kita tiba di v menyusuri rute terbaik sejauh ini itu, atau `NIL` bila kita belum menemukan rute apa pun ke v.

Pasangan (d, π) itulah yang sedang dihitung oleh tiap algoritma kita; bilangannya adalah jawaban atas *seberapa jauh*, pendahulunya adalah jawaban atas *lewat mana*, dan §13.10 menunjukkan cara mengubah yang kedua menjadi daftar simpul yang sesungguhnya.

> **PERINGATAN:** $d[v]$ adalah sebuah *estimasi*, dan selama pengerjaannya ia bisa terlalu besar. Ia tidak pernah terlalu kecil — itulah sifat batas atas §13.6.5, dan itu hal pertama yang kita buktikan. Pemula menulis kode yang memperlakukan $d[v]$ sebagai jawaban di tengah-tengah gelungnya; itulah kutu yang paling sering terjadi dalam seluruh pokok bahasan ini.

13.6.2 Inisialisasi

Baris-baris pertama ketiga algoritmanya identik, dan buku teks memberi mereka sebuah nama, **Pseudokode 13.1**:

Pseudokode 13.1 Inisialisasi yang dipakai bersama oleh setiap algoritma sumber tunggal: semua estimasi tak hingga, tanpa pendahulu, dan sumbernya berjarak nol.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  untuk tiap simpul v di V[G]
2      d[v] <- tak hingga
3      pi[v] <- NIL
4  d[s] <- 0
```

Inilah baris 1–3 pseudokode Dijkstra di **Gambar 13.4**, baris 01–04 Bellman–Ford di **Gambar 13.13**, dan baris 01–03 algoritma DAG di **Gambar 13.19**. Ongkosnya $\Theta(V)$ dan ia berkata: *saya tahu satu rute, yaitu rute kosong dari s ke dirinya sendiri, dan rute itu tidak berongkos apa pun; tentang segala sesuatu yang lain saya sama sekali tidak tahu apa-apa.*

13.6.3 Langkah relaksasinya

> **DEFINISI. Merelaksasi** sebuah sisi (u, v) berarti menguji apakah rute terbaik ke v yang ditemukan sejauh ini dapat diperbaiki dengan pergi ke u lalu mengambil sisi (u, v), dan, bila ya, mencatat perbaikan itu. **Pseudokode 13.2** menyatakannya dengan tepat.

Pseudokode 13.2 Merelaksasi sisi (u, v): satu pembandingan, dan dua pemberian nilai bila rute lewat u benar-benar lebih baik.

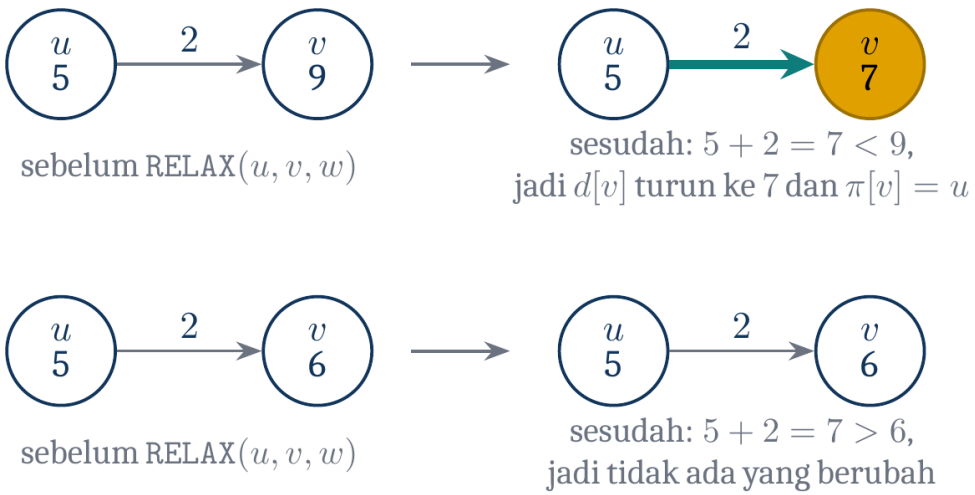
```
RELAX(u, v, w)
1  bila d[v] > d[u] + w(u,v)
2      d[v]  <- d[u] + w(u,v)
3      pi[v] <- u
```

Hanya itu seluruhnya: satu pembandingan, dan dua pemberian nilai bila pembandingannya berhasil. Ketiga baris yang sama muncul, disorot, di dalam pseudokode Dijkstra pada **Gambar 13.4** dengan label *relaxing edges* (merelaksasi sisi), sebab hanya baris-baris itulah dalam Dijkstra yang berbuat sesuatu terhadap jawabannya.

Kata "relaksasi" dipinjam dari kiasan fisis tentang pegas teregang yang dibiarkan mengendur. Estimasi $d[v]$ bermula teregang kencang pada ∞ dan tiap relaksasi yang berhasil membiarkannya menyusut sedikit lebih dekat ke nilai istirahatnya $\delta(s, v)$. Ia hanya pernah turun, tidak pernah naik — lihatlah baris 1: pemberian nilainya terjadi hanya ketika nilai barunya *lebih kecil*.

13.6.4 Sebuah relaksasi yang dikerjakan

Gambar 13.3 memperlihatkan kedua hasilnya pada gambar mungil yang sama. Tiap simpul dilabeli namanya di atas dan nilai d-nya yang sekarang di bawah, dan sisi (u, v) berbobot 2.



Gambar 13.3 Kedua hasil `RELAX(u,v,w)` dengan $w(u, v) = 2$. **Atas:** $d[u] + w = 5 + 2 = 7$, yang kurang daripada $d[v] = 9$ yang sekarang, sehingga $d[v]$ turun menjadi 7 dan $\pi[v]$ menjadi u. **Bawah:** $d[u] + w = 7$, yang *tidak* kurang daripada $d[v] = 6$ yang sekarang, sehingga sama sekali tidak terjadi apa-apa. Relaksasi yang tidak mengubah apa pun bukanlah langkah yang terbuang; ia adalah penegasan bahwa estimasi yang sekarang sudah mengalahkan rute ini.

Pada kasus atas: kita tahu ada rute ke v berongkos 9. Kita tahu ada rute ke u berongkos 5. Menambahkan satu sisi berbobot 2 memberikan rute ke v berongkos 7, yang lebih baik, sehingga kita mengambilnya dan mengingat bahwa kita tiba lewat u. Pada kasus bawah: rute melalui u akan berongkos 7, sedangkan kita sudah punya yang berongkos 6, sehingga kita pertahankan yang kita punya dan membiarkan $\pi[v]$ apa adanya.

WAWASAN KUNCI: Relaksasi adalah *satu-satunya* cara d dan π pernah berubah sesudah inisialisasi. Dijkstra, Bellman–Ford, dan algoritma DAG berbeda hanya dalam **urutan mereka merelaksasi sisi, dan berapa kali**. Karena itu segala sesuatu yang kita buktikan tentang d dan π di bagian ini berlaku bagi ketiganya sekaligus.

13.6.5 Sifat batas atas

LEMA 13.5 (sifat batas atas). Sesudah `INITIALIZE-SINGLE-SOURCE`, dan sesudah sembarang barisan langkah relaksasi apa pun, $d[v] \geq \delta(s, v)$ berlaku untuk setiap $v \in V$. Lebih jauh, begitu $d[v]$ mencapai nilai $\delta(s, v)$ ia tidak pernah berubah lagi.

Bukti, dengan induksi atas banyaknya langkah relaksasi yang dikerjakan.

Kasus basis: nol langkah. Tepat sesudah inisialisasi, $d[s] = 0$ dan $d[v] = \infty$ untuk $v \neq s$. Untuk $v \neq s$, $\infty \geq \delta(s, v)$ secara sepele. Untuk s sendiri kita perlu $0 \geq \delta(s, s)$. Nah, $\delta(s, s) \leq 0$, sebab lintasan satu simpul $\langle s \rangle$ berbobot 0; dan karena kita sudah mengandaikan tidak ada siklus negatif yang terjangkau dari s, tidak ada siklus melalui s yang berbobot negatif, sehingga tidak ada rute dari s kembali ke s yang berongkos kurang daripada 0, yang memberikan $\delta(s, s) = 0$. Karena itu $d[s] = 0 = \delta(s, s)$. Kasus basisnya berlaku.

Langkah induksi. Andaikan $d[x] \geq \delta(s, x)$ untuk setiap x, dan misalkan langkah berikutnya adalah `RELAX(u,v,w)`. Hanya $d[v]$ yang dapat berubah, dan hanya menjadi nilai $d[u] + w(u, v)$. Jadi cukuplah ditunjukkan bahwa

$$d[u] + w(u, v) \;\geq\; \delta(s, v).$$

Menurut hipotesis induksinya $d[u] \geq \delta(s, u)$, sehingga

$$d[u] + w(u, v) \;\geq\; \delta(s, u) + w(u, v) \;\geq\; \delta(s, v),$$

dengan ketaksamaan keduanya persis adalah Akibat 13.3, bentuk sisi ketaksamaan segitiga. Karena itu $d[v]$ yang baru, berapa pun nilainya, tetap sekurang-kurangnya $\delta(s, v)$. ■

Kalimat kedua lemanya. Andaikan $d[v] = \delta(s, v)$ pada suatu saat. Perubahan apa pun pada $d[v]$ sesudah itu akan datang dari relaksasi yang berhasil, yang menurut baris 1 `RELAX` *menurunkan* $d[v]$ secara tegas, sehingga memberikan $d[v] < \delta(s, v)$ — mustahil menurut bagian yang baru saja dibuktikan. Jadi $d[v]$ membeku. ■

Dengan bahasa sehari-hari: **estimasinya tidak pernah terlalu optimistis, dan begitu ia benar, ia tetap benar.** Kalimat itulah lisensi bagi setiap algoritma di sini untuk berhenti begitu tidak ada relaksasi yang mengubah apa pun.

13.6.6 Sifat tanpa-lintasan

LEMA 13.6 (sifat tanpa-lintasan). Bila tidak ada lintasan dari s ke v, maka $d[v] = \delta(s, v) = \infty$ setiap saat.

Bukti. Bahwa $\delta(s, v) = \infty$ adalah definisinya. Menurut Lema 13.5, $d[v] \geq \delta(s, v) = \infty$, dan karena ∞ adalah nilai terbesar yang dibawa aritmetika kita, $d[v] = \infty$. ■

Layak dilihat *mengapa* $d[v]$ tidak pernah dapat turun. Supaya $d[v]$ menjadi berhingga, suatu relaksasi sisi (u, v) pasti sudah menemukan $d[u]$ berhingga. Kejarlah balik: $d[u]$ menjadi berhingga karena relaksasi suatu (t, u) dengan $d[t]$ berhingga, dan seterusnya. Rantai itu harus berakhir di s, satu-satunya simpul yang d-nya berhingga pada awalnya — dan rantai itu adalah sebuah lintasan dari s ke v, yang bertentangan dengan pengandaian bahwa lintasan semacam itu tidak ada.

Sifat inilah yang membolehkan kita menulis `if (d[u] < INF && d[u] + e.w < d[e.v])` di dalam setiap program bab ini dan yakin bahwa kita tidak sedang diam-diam mengerjakan aritmetika di atas sebuah nilai penjaga.

13.6.7 Sifat kekonvergenan

> **LEMA 13.7 (sifat kekonvergenan).** Andaikan $s \rightsquigarrow u \to v$ adalah lintasan terpendek dari s ke v untuk suatu simpul u, v — artinya, $(u, v) \in E$ dan $\delta(s, v) = \delta(s, u) + w(u, v)$. Bila pada suatu saat $d[u] = \delta(s, u)$, maka pada sembarang waktu *sesudah* sisi (u, v) direlaksasi, $d[v] = \delta(s, v)$.

Bukti. Tinjaulah saat tepat sesudah relaksasi (u, v). Ada dua hal yang mungkin sudah terjadi sebelumnya.

Bila $d[v]$ sudah sama dengan $\delta(s, v)$, maka menurut separuh pembekuan Lema 13.5 ia masih sama dengan $\delta(s, v)$, dan selesailah.

Bila tidak, $d[v] > \delta(s, v)$ tepat sebelum relaksasinya. Pada saat relaksasi, $d[u] = \delta(s, u)$ — ia mencapai nilai itu lebih awal dan, menurut Lema 13.5 lagi, tidak mungkin bergerak sejak itu. Jadi uji pada baris 1 `RELAX` membandingkan

$$d[v] \;>\; \delta(s, v) \;=\; \delta(s, u) + w(u, v) \;=\; d[u] + w(u, v),$$

yang bernilai benar, sehingga baris 2 menyala dan menetapkan $d[v] \leftarrow d[u] + w(u, v) = \delta(s, v)$. Sesudah itu, Lema 13.5 membekukannya. ■

Dengan bahasa sehari-hari: **bila Anda punya jawaban yang benar untuk u, dan u adalah perhentian terakhir sebelum v pada sebuah rute terbaik, maka merelaksasi satu sisi itu memberi Anda jawaban yang benar untuk v, secara permanen.** Inilah lema kuda beban. Setiap bukti kebenaran dalam bab ini adalah argumen bahwa algoritmanya merelaksasi sisi-sisi suatu lintasan terpendek dalam urutan yang membolehkan Lema 13.7 menyala berulang kali.

13.6.8 Sifat relaksasi lintasan

> **LEMA 13.8 (sifat relaksasi lintasan).** Misalkan $p = \langle v_0, v_1, \ldots, v_k \rangle$ sebuah lintasan terpendek dari $s = v_0$ ke v_k. Andaikan sisi-sisi p direlaksasi **berurutan** — (v_0, v_1), lalu (v_1, v_2), ..., lalu (v_{k-1}, v_k) — mungkin dengan sejumlah relaksasi lain diselang-selingkan di mana saja. Maka sesudah yang terakhir dari k relaksasi itu, $d[v_k] = \delta(s, v_k)$, dan tetap demikian selamanya.

Bukti, dengan induksi atas i, dengan klaimnya berbunyi: sesudah sisi (v_{i-1}, v_i) direlaksasi, $d[v_i] = \delta(s, v_i)$.

Kasus basis $i = 0$. Sebelum relaksasi apa pun, $d[v_0] = d[s] = 0 = \delta(s, s)$ sebagaimana ditunjukkan pada Lema 13.5. Jadi klaimnya berlaku secara hampa untuk awal lintasannya.

Langkah induksi. Andaikan $d[v_{i-1}] = \delta(s, v_{i-1})$ berlaku pada saat sisi (v_{i-1}, v_i) direlaksasi. (Menurut Lema 13.5 ia berlaku sejak suatu saat sebelumnya dan tidak mungkin terganggu.) Karena p adalah lintasan terpendek dari s ke v_k, Teorema 13.1 berkata bahwa awalannya $\langle v_0, \dots, v_i \rangle$ adalah lintasan terpendek dari s ke v_i, dan karena itu $\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i)$. Itu persis hipotesis Lema 13.7 dengan $u = v_{i-1}$, $v = v_i$. Menerapkannya: sesudah relaksasi itu $d[v_i] = \delta(s, v_i)$, secara permanen.

Mengambil $i = k$ memberikan lemanya. ■

Ada dua hal yang perlu diperhatikan, sebab keduanya dipakai nanti dan keduanya mengejutkan orang.

Penyelang-nyelingannya cuma-cuma. Relaksasi lain boleh terjadi di antara relaksasi sisi-sisi p, sebanyak apa pun dan dalam urutan apa pun. Mereka tidak dapat mencederai: menurut Lema 13.5 mereka hanya dapat menurunkan estimasi, tidak pernah sampai di bawah δ, sehingga hipotesis $d[v_{i-1}] = \delta(s, v_{i-1})$ tiap langkahnya selamat. Inilah yang membuat Bellman–Ford berhasil: ia merelaksasi *setiap* sisi pada setiap sapuan, sehingga ia khususnya merelaksasi sisi-sisi p dalam urutan yang benar, dengan banyak sekali kerja yang tak relevan tercampur di dalamnya.

Hanya urutan relatifnya yang penting. Kita perlu (v_0, v_1) direlaksasi sebelum (v_1, v_2) sebelum (v_2, v_3) dan seterusnya. Kita tidak perlu mereka berurutan langsung, dan kita tidak perlu tahu p itu apa. Satu sapuan atas semua sisi menjamin sekurang-kurangnya satu langkah maju lagi pada rantainya, dan itu persis argumen pencacahan §13.8.3.

13.6.9 Sifat subgraf pendahulu

Kita sudah memelihara π dan belum berkata apa-apa tentangnya. Inilah nilainya.

DEFINISI. Subgraf pendahulu $G_\pi = (V_\pi, E_\pi)$ didefinisikan oleh

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}, \qquad E_\pi = \{(\pi[v], v) : v \in V_\pi \setminus \{s\}\}.$$

(13.7)

Dengan kata-kata, Persamaan (13.7) berkata: pertahankan setiap simpul yang punya pendahulu (ditambah s sendiri), lalu gambarkan satu sisi masuk ke tiap simpul semacam itu, dari pendahulunya.

DEFINISI. Sebuah **pohon lintasan terpendek** berakar di s adalah subgraf berarah $G' = (V', E')$ dengan $V' \subseteq V$ dan $E' \subseteq E$ sedemikian sehingga (i) V' persis adalah himpunan simpul yang terjangkau dari s; (ii) G' adalah pohon berakar dengan akar s; dan (iii) untuk setiap $v \in V'$, lintasan sederhana tunggal dari s ke v di dalam G' adalah lintasan terpendek dari s ke v di dalam G.

LEMA 13.9 (sifat subgraf pendahulu). Bila grafnya tidak memuat siklus berbobot negatif yang terjangkau dari s, dan bila pada suatu saat $d[v] = \delta(s, v)$ untuk setiap $v \in V$, maka G_π adalah pohon lintasan terpendek berakar di s.

Sketsa bukti, dalam tiga langkah. *Ia asiklik.* Andaikan G_π memuat sebuah siklus $\langle c_0, c_1, \dots, c_k = c_0 \rangle$, dengan $\pi[c_i] = c_{i-1}$. Tiap $\pi[c_i]$ ditetapkan oleh relaksasi (c_{i-1}, c_i) pada saat $d[c_i]$ menjadi $d[c_{i-1}] + w(c_{i-1}, c_i)$; sesudah itu $d[c_{i-1}]$ hanya mungkin menurun, sehingga pada saat ini $d[c_i] \geq d[c_{i-1}] + w(c_{i-1}, c_i)$. Menjumlahkannya sekeliling siklus, setiap suku d muncul sekali di tiap ruas dan saling meniadakan, sehingga tersisa $0 \geq w(\text{siklus})$: bobot siklusnya tidak positif. Bila ia negatif kita bertentangan dengan hipotesisnya; kasus sisanya, bobot tepat nol, disingkirkan di Soal Latihan 13.10 dengan meninjau saat siklus itu tertutup. Karena $\pi[s]$ tidak pernah ditetapkan,

s berderajat masuk 0, dan setiap simpul V_π yang lain berderajat masuk tepat 1 (masing-masing punya tepat satu nilai π). Sebuah graf berarah asiklik yang satu simpulnya berderajat masuk 0 dan semua yang lain berderajat masuk 1 adalah pohon berakar.

Ia merentang himpunan yang benar. $\pi[v] \neq \text{NIL}$ persis ketika $d[v]$ menjadi berhingga, yang menurut Lema 13.6 terjadi persis ketika v terjangkau dari s.

Lintasan-lintasannya adalah lintasan terpendek. Berjalanlah mundur dari v menyusuri π: $v, \pi[v], \pi[\pi[v]], \ldots$ hingga mencapai s. Pada saat $\pi[v]$ terakhir kali ditetapkan kita punya $d[v] = d[\pi[v]] + w(\pi[v], v)$, dan karena tidak ada d yang berubah sejak itu (semuanya berada pada nilai akhirnya δ), $\delta(s, v) = \delta(s, \pi[v]) + w(\pi[v], v)$. Menerapkan hal itu pada tiap langkah lalu menjumlahkannya meneleskop menjadi: bobot lintasan-π dari s ke v sama dengan $\delta(s, v)$. ■

Lema inilah lisensi bagi §13.10: ia berkata bahwa Anda boleh mencetak sebuah lintasan terpendek dengan mengikuti π mundur, dan yang Anda peroleh sungguh-sungguh adalah lintasan terpendek.

13.6.10 Kelima sifatnya dalam satu halaman

Tabel 13.5 mengumpulkan semuanya, beserta kegunaan masing-masing.

Tabel 13.5 Kelima sifat relaksasi. Semuanya dibuktikan sekali, di sini, lalu dipakai oleh ketiga algoritmanya; tidak ada algoritma dalam bab ini yang memerlukan argumen kebenaran yang tidak tereduksi ke tabel ini.

Sifat	Pernyataan	Yang dilisensikannya
Batas atas (13.5)	$d[v] \geq \delta(s, v)$ selalu; begitu sama, membeku	tidak pernah memercayai d terlalu awal; berhenti ketika tidak ada yang berubah
Tanpa-lintasan (13.6)	tidak ada rute $\Rightarrow d[v] = \delta(s, v) = \infty$	pengawal `d[u] < INF` di dalam setiap program
Kekonvergenan (13.7)	$d[u] = \delta(s, u)$ dan (u, v) pada lintasan terpendek, relaksasikan $(u, v) \Rightarrow d[v] = \delta(s, v)$	induksi di dalam setiap bukti kebenaran
Relaksasi lintasan (13.8)	relaksasikan sisi-sisi lintasan terpendek berurutan $\Rightarrow$ ujungnya selesai	$\lvert V \rvert - 1$ sapuan Bellman–Ford; sapuan tunggal algoritma DAG
Subgraf pendahulu (13.9)	ketika semua $d = \delta$, G_π adalah pohon lintasan terpendek	memulihkan rutenya dari π

13.7 Algoritma Dijkstra

13.7.1 Gagasannya

Algoritma Edsger Dijkstra [4], diterbitkan pada 1959 dan konon dirancang dalam sekitar dua puluh menit di sebuah kafe di Amsterdam, menyelesaikan persoalan sumber tunggal untuk graf yang bobot sisinya seluruhnya **tak negatif**. Wataknya dapat diringkas dalam lima pernyataan pendek, dan tiap pernyataan layak dibuka satu per satu.

Bobot sisi tak negatif. Pembatasannya sungguh-sungguh ada dan §13.7.3 menunjukkan dengan tepat di mana ia menggigit.

Bersifat greedy, mirip algoritma Prim untuk MST. Prim dari Bab 11 menumbuhkan satu pohon keluar dari sebuah akar, selalu menambahkan sisi termurah yang meninggalkan pohonnya. Dijkstra menumbuhkan satu pohon keluar dari sumbernya, selalu menambahkan simpul yang *jarak totalnya dari sumber* paling kecil. Kedua algoritma itu sebentuk; yang berbeda hanya

kuncinya. Kunci Prim bagi v adalah $\min\{w(u,v) : u \in \text{pohon}\}$, yaitu biaya *satu sisi*; kunci Dijkstra bagi v adalah $\min\{d[u] + w(u,v) : u \in S\}$, yaitu biaya *seluruh perjalanan*. Luruskan perbedaan itu dan Anda tidak akan pernah lagi mengacaukan keduanya.

Mirip penelusuran melebar — bila semua bobot sama dengan 1*, orang boleh langsung memakai BFS saja.* Dengan bobot satuan, "simpul terdekat yang belum selesai" sama dengan "yang jaraknya paling sedikit sisi", dan antrean prioritasnya merosot menjadi antrean FIFO Bab 10.

Pakailah Q, sebuah antrean prioritas yang berkunci $d[v]$. BFS memakai antrean masuk-pertama-keluar-pertama sebab semua simpul garis depannya sama baiknya. Di sini tidak demikian, sehingga kita memerlukan struktur yang dapat menyerahkan kembali yang terkecil, dan yang dapat ditata ulang ketika sebuah nilai d turun.

Gagasan dasarnya: peliharalah sehimpunan S berisi simpul yang sudah selesai; pada tiap langkah pilihlah simpul "terdekat" u, tambahkan ia ke S, lalu relaksasikan semua sisi yang keluar dari u.

> **WAWASAN KUNCI:** Dijkstra memproses simpul dalam urutan menaik menurut jarak sebenarnya dari s. Begitu sebuah simpul meninggalkan antrean prioritas, nilai d-nya *final* dan tidak pernah ditinjau ulang. Seluruh bukti kebenarannya adalah klaim bahwa hal itu aman — dan hal itu aman hanya karena tidak ada bobot sisi yang negatif, sehingga meninggalkan S tidak pernah dapat dibuat lebih murah dengan berkelana lebih jauh ke luar lebih dahulu.

13.7.2 Pseudokodenya

Gambar 13.4 memberikan pseudokodenya, dengan langkah relaksasinya disorot.

```
DIJKSTRA(G, w, s)
1  untuk tiap v ∈ V
2      lakukan d[v] ← ∞
3  d[s] ← 0
4  S ← ∅        ▷ himpunan simpul yang sudah tuntas
5  Q ← V
6  selama Q ≠ ∅
7      lakukan u ← EXTRACT-MIN(Q)
8          S ← S ∪ {u}
9          untuk tiap v ∈ Adj[u]
10             lakukan jika d[v] > d[u] + w(u, v)
11                 maka d[v] ← d[u] + w(u, v)
```

relaksasi sisi

Gambar 13.4 Algoritma Dijkstra: baris 1–3 menginisialisasi, baris 4–5 membuat himpunan selesai S yang kosong dan antrean Q, dan baris 6–8 berulang kali menyatakan simpul terdekat yang belum selesai sebagai selesai. Baris 9–11 yang disorot adalah `RELAX`, yang diterapkan pada setiap sisi yang keluar dari simpul itu.

Baris demi baris, dengan kata-kata:

Baris 1–2. Untuk setiap simpul v, tetapkan estimasinya $d[v]$ menjadi ∞. Kita belum tahu apa-apa. **Baris 3.** Tetapkan $d[s] \leftarrow 0$. Sumbernya berjarak nol dari dirinya sendiri. **Baris 4.** $S \leftarrow \emptyset$. S adalah

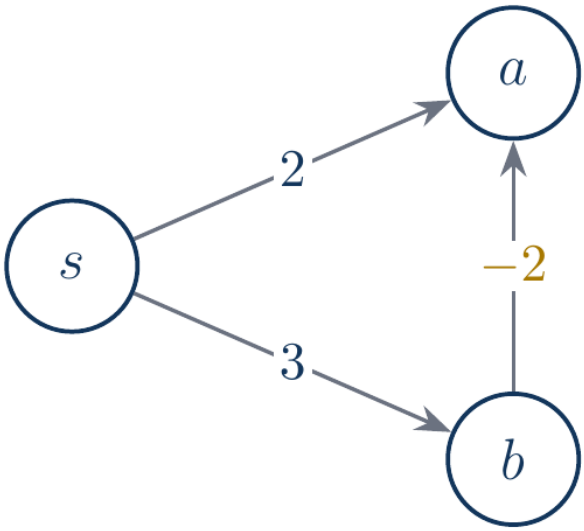
himpunan simpul yang jawabannya final. Komentar di dalam pseudokodenya menyebut mereka simpul yang *sudah tuntas*, kata yang paling jelas untuknya. **Baris 5.** $Q \leftarrow V$. Masukkan setiap simpul ke dalam antrean prioritas, berkunci d-nya yang sekarang. Pada saat ini s berkunci 0 dan semua yang lain berkunci ∞. **Baris 6.** Selama antreannya belum kosong — sehingga badan gelungnya berjalan tepat $|V|$ kali. **Baris 7.** $u \leftarrow \text{Extract} - \text{Min}(Q)$. Keluarkan simpul yang estimasinya terkecil. Pada putaran pertama ini pastilah s, sebab semua yang lain berada pada ∞. **Baris 8.** $S \leftarrow S \cup \{u\}$. Nyatakan u selesai. Sejak sekarang $d[u]$ tidak akan pernah berubah. **Baris 9.** Untuk setiap simpul v yang ditunjuk u… **Baris 10–11.** …relaksasikan sisi (u, v): bila rute melalui u mengalahkan rute terbaik ke v yang diketahui sejauh ini, ambillah ia. Ada satu catatan implementasi yang penting di sini: *menetapkan $d[v]$ berarti memperbarui Q (memerlukan `Decrease-Key`)*. Antrean prioritasnya berkunci d, sehingga menurunkan $d[v]$ harus menurunkan kunci v di dalam antreannya, atau antreannya akan menyerahkan kembali simpul yang salah pada giliran berikutnya.

Perhatikan apa yang *tidak* dikerjakan pseudokodenya: ia tidak pernah menetapkan π. Gambarnya membiarkan π tersirat; ia muncul secara eksplisit di **Pseudokode 13.2** dan di pseudokode Bellman–Ford pada §13.8.2. Setiap implementasi dalam bab ini menetapkan $\pi[v] \leftarrow u$ berdampingan dengan baris 11, sebab tanpanya Anda memperoleh jarak dan tidak memperoleh rute; lihat §13.10.

13.7.3 Mengapa bobotnya harus tak negatif

Kerakusan Dijkstra bertumpu pada satu pengandaian: **mengambil satu sisi tambahan tidak pernah dapat membuat sebuah perjalanan lebih murah**. Bila setiap $w \geq 0$, pengandaian itu benar, dan karena itu simpul yang estimasinya terkecil tidak mungkin dapat diperbaiki oleh rute melalui simpul mana pun yang masih ada di dalam antrean — setiap rute semacam itu harus membayar sekurang-kurangnya sama banyaknya untuk sampai ke sana lalu membayar lebih, atau sekurang-kurangnya tidak membayar apa-apa, untuk kembali. Bila ada $w < 0$, pengandaiannya runtuh: putaran panjang dapat berakhir dengan rabat besar.

Gambar 13.5 adalah contoh sanggahan terkecilnya. Tiga simpul, tiga sisi, dan — ini yang penting — **sama sekali tidak ada siklus**, sehingga pastilah tidak ada siklus negatif dan δ terdefinisi sempurna: $\delta(s, a) = 1$ lewat rute $\langle s, b, a \rangle$ yang berongkos $3 + (-2)$, dan $\delta(s, b) = 3$.



Gambar 13.5 Sebuah graf yang di atasnya Dijkstra diam-diam mengembalikan jawaban yang salah. Tidak ada siklus, sehingga tidak ada siklus negatif, dan jarak sebenarnya adalah $\delta(s, a) = 1$ dan $\delta(s, b) = 3$. Dijkstra memfinalkan a pada 2 — sebelum ia pernah menengok sisi (b, a) — dan tidak pernah meninjaunya kembali.

Telusurilah. Mula-mula $d[s] = 0$, $d[a] = d[b] = \infty$.

1. `ExtractMin` mengembalikan s dengan $d = 0$. Relaksasikan (s, a): $d[a] \leftarrow 2$. Relaksasikan (s, b): $d[b] \leftarrow 3$.
2. `ExtractMin` mengembalikan a dengan $d = 2$ (sebab $2 < 3$). a **bergabung ke S dan kini final pada** 2. Ia tidak punya sisi keluar.
3. `ExtractMin` mengembalikan b dengan $d = 3$. Relaksasikan (b, a): nilai calonnya adalah $3 + (-2) = 1 < 2$. Tetapi a sudah berada di dalam S. Dijkstra versi buku teks entah melewatkan pembaruannya sebab a sudah difinalkan, atau — lebih buruk lagi — menuliskan $d[a] = 1$ ke dalam sebuah larik yang tidak akan pernah dibaca siapa pun lagi.

Bagaimanapun juga jawaban yang dilaporkan adalah $d[a] = 2$ sedangkan kebenarannya adalah 1. Program **Kode 13.1** menjalankan persis itu lalu mencetak putusan kedua algoritmanya berdampingan:

Kode 13.1 Dijkstra dan Bellman–Ford pada graf tiga simpul dengan satu sisi negatif dan tanpa siklus. Kompilasilah dengan `g++ -std=c++17 -O2 -o dijfail dijfail.cpp`.

```
#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>

using namespace std;

const long long INF = LLONG_MAX / 4;
struct Edge { int u, v; long long w; };

// Dijkstra, ditulis sedemikian rupa sehingga sebuah simpul FINAL begitu ia keluar dari
antrean.
vector<long long> dijkstra(int n, const vector<Edge>& edges, int s, bool talk,
                           const vector<string>& name) {
    vector<vector<pair<int,long long>>> adj(n);
    for (const Edge& e : edges) adj[e.u].push_back({e.v, e.w});
    vector<long long> d(n, INF);
    vector<bool> done(n, false);
    d[s] = 0;
    priority_queue<pair<long long,int>, vector<pair<long long,int>>,
                   greater<pair<long long,int>>> Q;
    Q.push({0, s});
    while (!Q.empty()) {
        auto [key, u] = Q.top(); Q.pop();
        if (done[u]) continue;
        done[u] = true;                              // u bergabung ke S dan tidak pernah
ditinjau ulang
        if (talk) cout << "  finalize " << name[u] << " at d=" << d[u] << "\n";
        for (auto [v, w] : adj[u])
            if (!done[v] && d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
    }
    return d;
}

vector<long long> bellmanFord(int n, const vector<Edge>& edges, int s) {
    vector<long long> d(n, INF);
    d[s] = 0;
    for (int i = 1; i <= n - 1; ++i)
        for (const Edge& e : edges)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) d[e.v] = d[e.u] + e.w;
    return d;
}
```

Kode 13.1 (Lanjutan) *Dijkstra dan Bellman–Ford pada graf tiga simpul dengan satu sisi negatif dan tanpa siklus. Kompilasilah dengan g++ -std=c++17 -O2 -o dijfail dijfail.cpp.*

```
int main() {
    vector<string> name = {"s", "a", "b"};
    // s->a = 2, s->b = 3, b->a = -2. Sama sekali tanpa siklus, jadi tanpa siklus negatif.
    vector<Edge> edges = {{0,1,2}, {0,2,3}, {2,1,-2}};

    cout << "Dijkstra on a graph with one negative edge:\n";
    vector<long long> dd = dijkstra(3, edges, 0, true, name);
    vector<long long> db = bellmanFord(3, edges, 0);

    cout << "\nvertex | Dijkstra | Bellman-Ford | truth\n";
    cout <<   "-------+----------+--------------+-------\n";
    const char* truth[] = {"0", "1", "3"};
    for (int v = 0; v < 3; ++v)
        cout << "   " << name[v] << "   |    " << dd[v]
             << "     |      " << db[v] << "       |   " << truth[v]
             << (dd[v] == db[v] ? "" : "   <-- Dijkstra is WRONG") << "\n";
    return 0;
}
```

```
Dijkstra on a graph with one negative edge:
  finalize s at d=0
  finalize a at d=2
  finalize b at d=3

vertex | Dijkstra | Bellman-Ford | truth
-------+----------+--------------+-------
   s   |    0     |      0       |   0
   a   |    2     |      1       |   1   <-- Dijkstra is WRONG
   b   |    3     |      3       |   3
```

> **PERINGATAN:** Dijkstra tidak mendeteksi hal ini. Ia tidak macet, ia tidak memperingatkan, ia tidak mengembalikan kode galat. Ia mengembalikan bilangan yang tampak sepenuhnya masuk akal dan salah. Bila Anda tidak yakin bahwa setiap bobotnya tak negatif, jangan pakai Dijkstra — pakailah Bellman–Ford, yang lebih lambat dan benar, atau bobotilah ulang grafnya dengan metode Johnson di Bab 15.

Sebuah "perbaikan" yang populer adalah menambahkan konstanta besar c kepada tiap bobot sisi supaya semuanya menjadi tak negatif. Cara itu tidak berhasil, dan layak dilihat mengapa: lintasan berisi k sisi bertambah kc, sehingga lintasan bersisi banyak dihukum lebih berat daripada lintasan bersisi sedikit, dan algoritmanya mulai lebih menyukai rute berlompatan pendek yang sebenarnya bukan yang termurah. Soal Latihan 13.6 meminta Anda menyusun graf yang membuat cara ini gagal.

13.7.4 Contohnya, selangkah demi selangkah

Sekarang kita menelusuri Dijkstra pada graf lima simpul, yang digambar dalam enam panel. Kita mengikuti setiap panelnya, dan pada tiap langkah kita juga menuliskan antrean prioritasnya, yang tersirat pada gambar-gambar itu tetapi tidak digambarkan di sana.

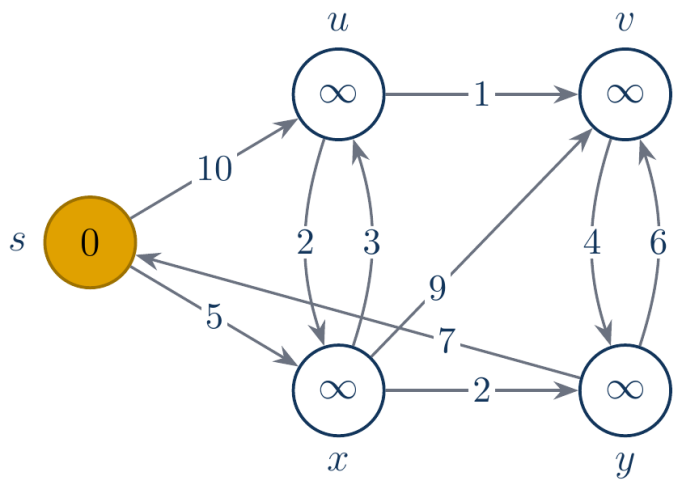
Grafnya punya simpul s, u, v, x, y dan sepuluh sisi berarah:

$$w(s,u) = 10,\ w(s,x) = 5,\ w(u,v) = 1,\ w(u,x) = 2,\ w(x,u) = 3,$$

$$w(x,v) = 9,\ w(x,y) = 2,\ w(v,y) = 4,\ w(y,v) = 6,\ w(y,s) = 7.$$

Dalam kebiasaan menggambar panel-panel ini, simpul **putih** masih berada di dalam Q, simpul **kuning** adalah simpul yang baru saja dikeluarkan, dan simpul **biru tua** berada di dalam S (sudah selesai). Anak panah tebal adalah sisi pendahulu yang sekarang, yaitu sisi-sisi G_π.

Panel (a) — sesudah inisialisasi. Setiap d bernilai ∞ kecuali $d[s] = 0$, dan Q memuat kelima simpulnya.

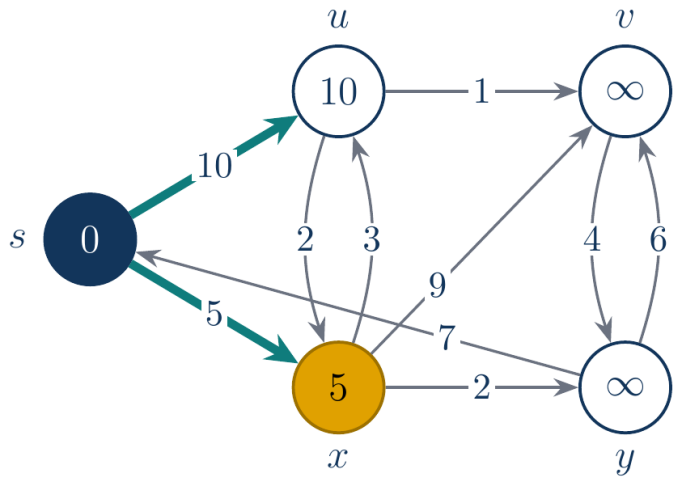


Gambar 13.6 Contoh Dijkstra, panel (a): grafnya tepat sesudah `INITIALIZE-SINGLE-SOURCE`. $d[s] = 0$ (kuning, hendak dikeluarkan) dan setiap estimasi lainnya adalah ∞. $Q = \{s = 0,\ u = \infty,\ v = \infty,\ x = \infty,\ y = \infty\}$.

Langkah 1. `ExtractMin` mengembalikan s, satu-satunya simpul berkunci berhingga. $S = \{s\}$. Relaksasikan kedua sisi yang keluar dari s:

- $(s, u), w = 10$: $d[u] = \infty > 0 + 10 = 10$, sehingga $d[u] \leftarrow 10, \pi[u] \leftarrow s$.
- $(s, x), w = 5$: $d[x] = \infty > 0 + 5 = 5$, sehingga $d[x] \leftarrow 5, \pi[x] \leftarrow s$.

Kini $Q = \{x = 5,\ u = 10,\ v = \infty,\ y = \infty\}$, yaitu keadaan yang digambarkan pada **Gambar 13.7**.

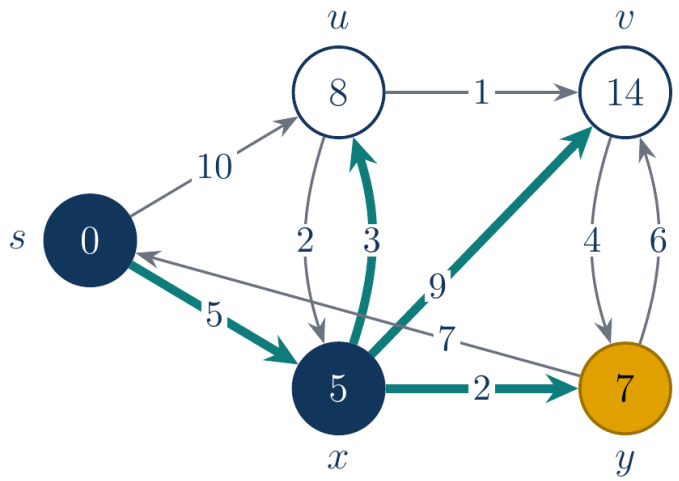


Gambar 13.7 Panel (b): s biru tua (sudah selesai), kedua sisi yang keluar darinya sudah direlaksasi, dan x — kuning — adalah simpul berikutnya yang akan dikeluarkan sebab $5 < 10$. Perhatikan bahwa u sudah punya estimasi 10; estimasi itu akan diperbaiki sebelum u pernah dikeluarkan.

Langkah 2. `ExtractMin` mengembalikan x dengan $d[x] = 5$. $S = \{s, x\}$. Relaksasikan ketiga sisi yang keluar dari x:

- (x, u), $w = 3$: $d[u] = 10 > 5 + 3 = 8$, sehingga $d[u] \leftarrow 8$, $\pi[u] \leftarrow x$. **Inilah `Decrease-Key` yang diperingatkan oleh catatan implementasi di §13.7.2**: u tadinya duduk di dalam Q dengan kunci 10 dan kini harus duduk di sana dengan kunci 8.
- (x, v), $w = 9$: $d[v] = \infty > 5 + 9 = 14$, sehingga $d[v] \leftarrow 14$, $\pi[v] \leftarrow x$.
- (x, y), $w = 2$: $d[y] = \infty > 5 + 2 = 7$, sehingga $d[y] \leftarrow 7$, $\pi[y] \leftarrow x$.

Kini $Q = \{y = 7,\ u = 8,\ v = 14\}$ — lihat **Gambar 13.8**.

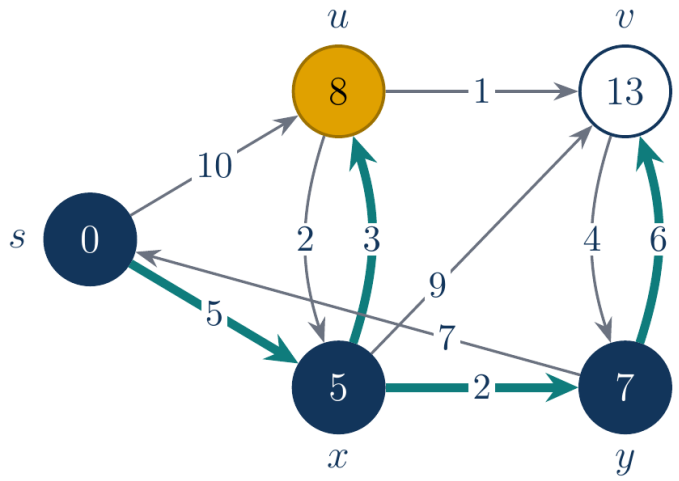


Gambar 13.8 Panel (c): s dan x biru tua. Estimasi bagi u turun dari 10 menjadi 8 dan sisi pendahulunya berayun dari (s, u) ke (x, u) — anak panah tebalnya kini datang dari arah x. y kuning: pada 7 ia kunci terkecil yang tersisa.

Langkah 3. `ExtractMin` mengembalikan y dengan $d[y] = 7$. $S = \{s, x, y\}$. Relaksasikan kedua sisi yang keluar dari y:

- (y, v), $w = 6$: $d[v] = 14 > 7 + 6 = 13$, sehingga $d[v] \leftarrow 13$, $\pi[v] \leftarrow y$. Sebuah `Decrease-Key` lagi.
- (y, s), $w = 7$: $d[s] = 0$, dan $7 + 7 = 14 > 0$, sehingga tidak terjadi apa-apa. Sebuah relaksasi yang tidak mengubah apa pun.

Kini $Q = \{u = 8,\ v = 13\}$, sebagaimana diperlihatkan **Gambar 13.9**.

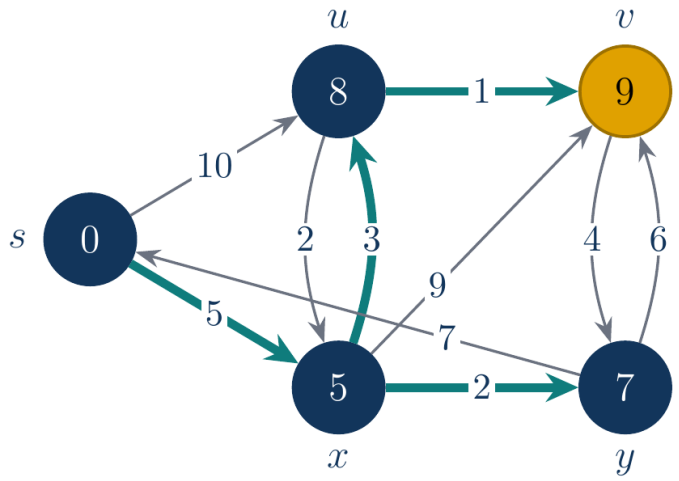


Gambar 13.9 Panel (d): s, x, dan y biru tua. Estimasi v membaik dari 14 menjadi 13 dan anak panah pendahulunya yang tebal berayun dari x ke y. u, kuning pada 8, adalah yang berikutnya.

Langkah 4. `ExtractMin` mengembalikan u dengan $d[u] = 8$. $S = \{s, x, y, u\}$. Relaksasikan kedua sisi yang keluar dari u:

- (u, v), $w = 1$: $d[v] = 13 > 8 + 1 = 9$, sehingga $d[v] \leftarrow 9$, $\pi[v] \leftarrow u$. `Decrease-Key` yang ketiga; estimasi v kini sudah diturunkan tiga kali, dari ∞ menjadi 14 menjadi 13 menjadi 9.
- (u, x), $w = 2$: $d[x] = 5$, dan $8 + 2 = 10 > 5$, sehingga tidak terjadi apa-apa.

Kini $Q = \{v = 9\}$; gambarnya adalah **Gambar 13.10**.

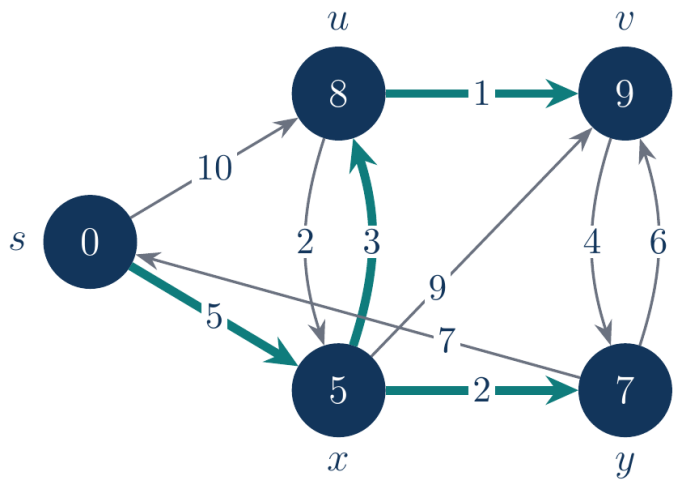


Gambar 13.10 Panel (e): empat simpul biru tua, v kuning pada 9, dan anak panah pendahulunya kini datang dari u. Setiap sisi di dalam grafnya sudah direlaksasi sekurang-kurangnya sekali kecuali kedua sisi yang meninggalkan v.

Langkah 5. `ExtractMin` mengembalikan v dengan $d[v] = 9$. S kini adalah kelima simpulnya. Relaksasikan satu sisi yang keluar dari v:

- (v, y), $w = 4$: $d[y] = 7$, dan $9 + 4 = 13 > 7$, sehingga tidak terjadi apa-apa.

Q kosong; algoritmanya berhenti. **Gambar 13.11** adalah gambar yang sudah selesai.



Gambar 13.11 Panel (f): setiap simpul biru tua, algoritmanya selesai. Keempat sisi tebal (s, x), (x, u), (u, v), dan (x, y) adalah subgraf pendahulu G_π — pohon lintasan terpendek yang berakar di s. Membaca sebuah rute darinya tinggal soal menyusuri anak panah tebalnya mundur dari simpul mana pun ke s.

13.7.5 Penelusuran yang sama dalam bentuk tabel

Tabel 13.6 Jalannya Dijkstra selengkapnya pada graf contoh, satu baris per putaran gelung `while`. Kunci di dalam Q hanya pernah turun, sedangkan kunci yang dikeluarkan, 0,5,7,8,9, tidak pernah turun; §13.7.8 membuktikan fakta yang kedua.

Langkah	u Yang dikeluarkan	$d[u]$	Sisi yang direlaksasi (hasilnya)	Q Sesudah langkah itu	S Sesudahnya
0	—	—	(inisialisasi)	$s = 0, u = \infty, v = \infty, x = \infty, y = \infty$	$\emptyset$
1	s	0	$(s,u) \to 10$; $(s,x) \to 5$	$x = 5, u = 10, v = \infty, y = \infty$	$\{s\}$
2	x	5	$(x,u) \to 8$; $(x,v) \to 14$; $(x,y) \to 7$	$y = 7, u = 8, v = 14$	$\{s,x\}$
3	y	7	$(y,v) \to 13$; (y,s) tidak berubah	$u = 8, v = 13$	$\{s,x,y\}$
4	u	8	$(u,v) \to 9$; (u,x) tidak berubah	$v = 9$	$\{s,x,y,u\}$
5	v	9	(v,y) tidak berubah	kosong	semuanya

Jawaban akhirnya dikumpulkan pada **Tabel 13.7**.

Tabel 13.7 Keluaran Dijkstra pada graf contoh: jarak, pendahulu, dan rute yang diperoleh dengan menyusuri π mundur. Baris d-nya cocok dengan **Tabel 13.1**, yang menemukan $\delta(s,v) = 9$ dengan mendaftarkan enam lintasan.

v	s	u	v	x	y
$d[v] = \delta(s,v)$	0	8	9	5	7
$\pi[v]$	NIL	x	u	s	x
lintasan terpendek	s	s,x,u	s,x,u,v	s,x	s,x,y

13.7.6 Programnya

Kode 13.2 adalah program C++ yang lengkap. Ia membangun graf contoh, menjalankan Dijkstra dengan *binary heap* (yaitu `priority_queue` pustaka standar), mencetak penelusurannya dalam bentuk yang sama dengan **Tabel 13.6**, lalu menutupnya dengan jarak, pendahulu, dan rute yang disusun ulang.

Ada satu keputusan implementasi yang layak diterangkan sebelum Anda membacanya. `priority_queue` pustaka standar C++ tidak punya `DecreaseKey`. Alih-alih menulis *heap* kita sendiri, kita memakai akal-akalan **penghapusan malas** (*lazy deletion*), persis seperti yang dikerjakan Bab 11 untuk Prim: ketika $d[v]$ turun, kita cukup mendorong salinan *kedua* v ke dalam antreannya dengan kunci baru yang lebih kecil, dan membiarkan salinan basinya di tempatnya. Salinan dikeluarkan dari yang terkecil lebih dahulu, sehingga salinan yang baik selalu keluar sebelum salinan-salinan basinya, dan salinan basi dikenali pada saat pengeluaran sebab kuncinya melebihi $d[v]$ yang sekarang (atau sebab simpulnya sudah berada di dalam S) lalu dibuang. Antreannya boleh memuat sampai $|E|$ entri alih-alih $|V|$, sehingga ongkosnya menjadi $O(E\lg E) = O(E\lg V)$ — secara asimtotik sama dengan Dijkstra *binary heap* yang semestinya, sebab $\lg E < \lg V^2 = 2\lg V$.

Kode 13.2 Algoritma Dijkstra dengan antrean prioritas *binary heap* pada graf contoh, yang mencetak antreannya pada setiap langkah. Kompilasilah dengan `g++ -std=c++17 -O2 -o dijkstra dijkstra.cpp`.

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>

using namespace std;

const long long INF = LLONG_MAX / 4;          // "tak hingga" yang masih aman kita tambahi

struct Edge { int to; long long w; };

// Graf contoh Dijkstra: simpul s,u,v,x,y.
const vector<string> NAME = {"s", "u", "v", "x", "y"};
enum { S = 0, U = 1, V = 2, X = 3, Y = 4 };

// Cetak antrean prioritasnya tanpa merusaknya: kita menyalinnya lalu menguras salinannya.
// Entri basi (simpul yang sudah di dalam S, atau kunci yang dikalahkan DecreaseKey
// berikutnya) dilewati, sehingga yang tercetak adalah isi logis Q.
string queueContents(priority_queue<pair<long long,int>,
                                    vector<pair<long long,int>>,
                                    greater<pair<long long,int>>> q,
                     const vector<bool>& done, const vector<long long>& d) {
    vector<pair<long long,int>> live;
    vector<bool> seen(done.size(), false);
    while (!q.empty()) {
        auto [key, v] = q.top(); q.pop();
        if (done[v] || seen[v] || key > d[v]) continue;
        seen[v] = true;
        live.push_back({key, v});
    }
    sort(live.begin(), live.end());
    string out = "{";
    for (size_t i = 0; i < live.size(); ++i) {
        if (i) out += ", ";
        out += NAME[live[i].second] + "=" +
               (live[i].first >= INF ? string("INF") : to_string(live[i].first));
    }
    return out + "}";
}
```

Kode 13.2 (Lanjutan) *Algoritma Dijkstra dengan antrean prioritas binary heap pada graf contoh, yang mencetak antreannya pada setiap langkah. Kompilasilah dengan g++ -std=c++17 -O2 -o dijkstra dijkstra.cpp.*

```
int main() {
    int n = 5;
    vector<vector<Edge>> adj(n);
    auto add = [&](int a, int b, long long w) { adj[a].push_back({b, w}); };
    add(S, U, 10); add(S, X, 5);
    add(U, V, 1);  add(U, X, 2);
    add(X, U, 3);  add(X, V, 9); add(X, Y, 2);
    add(V, Y, 4);
    add(Y, V, 6);  add(Y, S, 7);

    vector<long long> d(n, INF);
    vector<int> pi(n, -1);
    vector<bool> done(n, false);                  // done[v] == "v berada di dalam S"
    d[S] = 0;

    priority_queue<pair<long long,int>, vector<pair<long long,int>>,
                   greater<pair<long long,int>>> Q;
    for (int v = 0; v < n; ++v) Q.push({d[v], v});

    cout << "initial Q = " << queueContents(Q, done, d) << "\n\n";

    int step = 0;
    while (!Q.empty()) {
        auto [key, u] = Q.top(); Q.pop();
        if (done[u]) continue;                    // salinan basi: abaikan saja
        if (key > d[u]) continue;                 // digantikan oleh kunci yang lebih kecil
        done[u] = true;
        cout << "step " << ++step << ": EXTRACT-MIN gives " << NAME[u]
             << " with d=" << (d[u] >= INF ? string("INF") : to_string(d[u])) << "\n";
        if (d[u] >= INF) { cout << "    unreachable; no edges relaxed\n\n"; continue; }
        for (const Edge& e : adj[u]) {
            long long cand = d[u] + e.w;
            cout << "    relax (" << NAME[u] << "," << NAME[e.to] << ") w=" << e.w
                 << ": d[" << NAME[e.to] << "]="
                 << (d[e.to] >= INF ? string("INF") : to_string(d[e.to]))
                 << " vs " << d[u] << "+" << e.w << "=" << cand << " -> ";
            if (cand < d[e.to]) {
                d[e.to] = cand; pi[e.to] = u;
                Q.push({cand, e.to});
                cout << "improved, d[" << NAME[e.to] << "]=" << cand
                     << ", pi[" << NAME[e.to] << "]=" << NAME[u] << "\n";
            } else {
                cout << "no change\n";
            }
        }
        cout << "    Q = " << queueContents(Q, done, d) << "\n\n";
```

Kode 13.2 (Lanjutan) *Algoritma Dijkstra dengan antrean prioritas binary heap pada graf contoh, yang mencetak antreannya pada setiap langkah. Kompilasilah dengan g++ -std=c++17 -O2 -o dijkstra dijkstra.cpp.*

```
    }

    cout << "vertex |  d  | pi | shortest path from s\n";
    cout << "-------+-----+----+----------------------\n";
    for (int v = 0; v < n; ++v) {
        vector<string> path;
        for (int t = v; t != -1; t = pi[t]) path.push_back(NAME[t]);
        reverse(path.begin(), path.end());
        string p;
        for (size_t i = 0; i < path.size(); ++i) { if (i) p += " -> "; p += path[i]; }
        cout << "   " << NAME[v] << "   |" << setw(4) << d[v] << " |  "
             << (pi[v] == -1 ? string("-") : NAME[pi[v]]) << " | " << p << "\n";
    }
    return 0;
}
```

13.7.7 Apa yang dicetaknya

```
initial Q = {s=0, u=INF, v=INF, x=INF, y=INF}

step 1: EXTRACT-MIN gives s with d=0
    relax (s,u) w=10: d[u]=INF vs 0+10=10 -> improved, d[u]=10, pi[u]=s
    relax (s,x) w=5: d[x]=INF vs 0+5=5 -> improved, d[x]=5, pi[x]=s
    Q = {x=5, u=10, v=INF, y=INF}

step 2: EXTRACT-MIN gives x with d=5
    relax (x,u) w=3: d[u]=10 vs 5+3=8 -> improved, d[u]=8, pi[u]=x
    relax (x,v) w=9: d[v]=INF vs 5+9=14 -> improved, d[v]=14, pi[v]=x
    relax (x,y) w=2: d[y]=INF vs 5+2=7 -> improved, d[y]=7, pi[y]=x
    Q = {y=7, u=8, v=14}

step 3: EXTRACT-MIN gives y with d=7
    relax (y,v) w=6: d[v]=14 vs 7+6=13 -> improved, d[v]=13, pi[v]=y
    relax (y,s) w=7: d[s]=0 vs 7+7=14 -> no change
    Q = {u=8, v=13}

step 4: EXTRACT-MIN gives u with d=8
    relax (u,v) w=1: d[v]=13 vs 8+1=9 -> improved, d[v]=9, pi[v]=u
    relax (u,x) w=2: d[x]=5 vs 8+2=10 -> no change
    Q = {v=9}

step 5: EXTRACT-MIN gives v with d=9
    relax (v,y) w=4: d[y]=7 vs 9+4=13 -> no change
    Q = {}

vertex |  d  | pi | shortest path from s
-------+-----+----+----------------------
   s   |   0 |  - | s
   u   |   8 |  x | s -> x -> u
   v   |   9 |  u | s -> x -> u -> v
   x   |   5 |  s | s -> x
   y   |   7 |  x | s -> x -> y
```

Bandingkan itu, baris demi baris, dengan keenam panel §13.7.4 dan dengan **Tabel 13.6**. Programnya mengerjakan persis apa yang kita kerjakan dengan tangan, termasuk ketiga saat `Decrease-Key` dan ketiga relaksasi yang tidak mengubah apa pun.

13.7.8 Kebenarannya

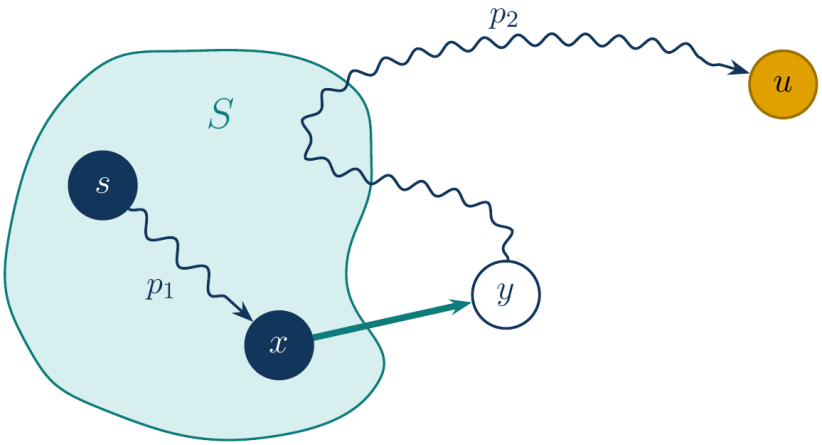
Bukti ini layak ditempuh pelan-pelan, sebab inilah argumen terdalam dalam bab ini dan di sinilah pengandaian ketaknegatifannya akhirnya dibelanjakan.

Apa yang hendak dibuktikan. *Kapan pun u ditambahkan ke S, $d[u] = \delta(s,u)$, dan kesamaan itu dipelihara sesudahnya.* Dengan kata lain, pada saat sebuah simpul meninggalkan antrean prioritas, estimasinya sudah merupakan jarak yang sebenarnya, dan tidak ada yang terjadi kemudian yang mengganggunya.

Itu adalah sebuah *invarian gelung*, dalam pengertian Bab 2. Marilah kita nyatakan dengan cara itu.

> **TEOREMA 13.10 (kebenaran Dijkstra).** Jalankan `Dijkstra(G, w, s)` pada graf berarah berbobot dengan $w(u,v) \geq 0$ untuk setiap sisi. Pada awal tiap putaran gelung `while` di baris 6, $d[v] = \delta(s,v)$ untuk setiap $v \in S$. Akibatnya, pada saat berakhir $d[v] = \delta(s,v)$ untuk setiap $v \in V$, dan G_π adalah pohon lintasan terpendek yang berakar di s.

Bukti. Buktinya dengan kontradiksi, dan ia berjalan di dalam gambar **Gambar 13.12**, yang akan kita rujuk tiga kali.



Gambar 13.12 Gambar di balik bukti kebenaran Dijkstra: wilayah hijau-biru muda adalah himpunan selesai S, dan u adalah simpul pertama yang konon dikeluarkan dengan estimasi yang salah. Lintasan terpendek yang sebenarnya meninggalkan S lewat sisi (x,y), sehingga berbentuk $s \overset{p_1}{\rightsquigarrow} x \to y \overset{p_2}{\rightsquigarrow} u$.

Menyiapkan kontradiksinya. Menurut Lema 13.5 kita sudah tahu $d[v] \geq \delta(s,v)$ untuk setiap v pada setiap saat — dalam lambang, $\forall v,\ d[v] \geq \delta(s,v)$. Andaikan teoremanya salah. Maka pada suatu pengeluaran invariannya rusak. Misalkan u adalah simpul **pertama** yang pernah dikeluarkan dan $d[u] \neq \delta(s,u)$; menurut sifat batas atas hal ini berarti

$$d[u] > \delta(s,u). \tag{13.8}$$

Kata *pertama* itu memikul beban yang sangat besar: ia berarti setiap simpul yang sudah berada di dalam S memiliki nilai yang benar, dan fakta itulah yang akan kita pakai dua kali.

Langkah 1: $u \neq s$. Ketika s dikeluarkan (yang tentu saja paling dahulu, sebab $d[s] = 0$ dan semua yang lain bernilai $\infty \geq 0$), $d[s] = 0 = \delta(s,s)$, sehingga invariannya berlaku untuk s dan u tidak mungkin s.

Langkah 2: u terjangkau dari s. Seandainya tidak, $\delta(s,u) = \infty$, dan menurut Lema 13.6 $d[u] = \infty$ juga, yang bertentangan dengan Persamaan (13.8), sebab ketaksamaan itu tegas. Jadi ada lintasan terpendek p dari s ke u.

Langkah 3: uraikan p pada batas S. Tinjaulah saat tepat sebelum u dikeluarkan. Pada saat itu $s \in S$ dan $u \notin S$. Berjalanlah menyusuri p dari s menuju u dan misalkan y adalah simpul **pertama** pada p yang tidak berada di dalam S, dan misalkan x adalah pendahulunya pada p, sehingga $x \in S$. (Mungkin saja $x = s$; mungkin saja $y = u$. Tidak satu pun dari kedua kasus khusus itu mencederai argumennya.) Inilah persis yang digambarkan **Gambar 13.12**: p terbelah menjadi

$$s \overset{p_1}{\rightsquigarrow} x \;\rightarrow\; y \overset{p_2}{\rightsquigarrow} u.$$

Langkah 4: $d[y] = \delta(s, y)$. Inilah jantung buktinya, dan ia memerlukan dua pengamatan.

Pertama, $x \in S$ dan x dikeluarkan *sebelum* u; karena u adalah simpul pertama yang dikeluarkan dengan estimasi yang salah, estimasi x benar: $d[x] = \delta(s, x)$.

Kedua, ketika algoritmanya menyisipkan x ke dalam S pada baris 8, ia langsung merelaksasi setiap sisi yang keluar dari x pada baris 9–11 — termasuk sisi (x, y). Pada saat itu $d[x]$ sudah sama dengan $\delta(s, x)$. Dan p adalah lintasan terpendek yang melalui sisi (x, y), sehingga menurut substruktur optimal (Teorema 13.1) awalannya $s \rightsquigarrow x \rightarrow y$ adalah lintasan terpendek ke y, yaitu $\delta(s, y) = \delta(s, x) + w(x, y)$. Itu persis hipotesis sifat kekonvergenan, Lema 13.7. Menerapkannya: sesudah relaksasi itu, $d[y] = \delta(s, y)$, dan menurut Lema 13.5 ia tidak bergerak sejak itu.

Langkah 5: $\delta(s, y) \leq \delta(s, u)$ *— dan di sinilah ketaknegatifannya dibelanjakan.* Sublintasan p_2 berjalan dari y ke u dan, menurut substruktur optimal lagi, merupakan lintasan terpendek, sehingga $w(p_2) = \delta(y, u)$. Menguraikan p,

$$\delta(s, u) \;=\; \delta(s, y) + w(p_2) \;=\; \delta(s, y) + \delta(y, u).$$

Setiap sisi p_2 berbobot tak negatif, sehingga $w(p_2) \geq 0$, dan karena itu

$$\delta(s, y) \;\leq\; \delta(s, u). \tag{13.9}$$

Inilah satu-satunya langkah yang memakai $w \geq 0$, dan segala sesuatunya bergantung padanya. Seandainya p_2 dapat berbobot total negatif, maka $\delta(s, y)$ dapat *melampaui* $\delta(s, u)$: rute sebenarnya ke u akan melewati sebuah simpul yang *lebih jauh* daripada u itu sendiri, dan algoritma *greedy* yang selalu mengambil simpul terdekat tidak akan pernah menemukannya. Itulah persis yang terjadi pada **Gambar 13.5**, yang rute sebenarnya ke a-nya (berjarak 1) melewati b (berjarak 3).

Langkah 6: kontradiksinya. Rangkaikan ketaksamaannya. Memakai Langkah 4 untuk kesamaannya, lalu Persamaan (13.9), lalu Persamaan (13.8):

$$d[y] \;=\; \delta(s, y) \;\leq\; \delta(s, u) \;<\; d[u].$$

Jadi $d[y] < d[u]$ secara tegas. Tetapi baik y maupun u sama-sama berada di dalam antrean prioritas pada saat pengeluaran itu, dan algoritmanya mengeluarkan u — simpul berkunci *minimum*. Hal itu menuntut $d[u] \leq d[y]$. Kedua pernyataan itu bertentangan satu sama lain. Dalam satu baris: *tetapi* $d[u] > d[y] \Rightarrow$ *algoritmanya akan memilih* y *dari antrean prioritas untuk diproses berikutnya, bukan* u *— kontradiksi.*

Karena itu u semacam itu tidak ada, dan setiap simpul dikeluarkan dengan $d[u] = \delta(s, u)$. Karena setiap simpul dikeluarkan tepat sekali dan gelungnya berjalan sampai Q kosong, pada saat berakhir $d[v] = \delta(s, v)$ untuk semua v, dan Lema 13.9 lalu berkata bahwa G_π adalah pohon lintasan terpendek. ■

Sebuah akibat yang berguna dalam praktik.

AKIBAT 13.11. Barisan kunci yang dikembalikan oleh panggilan `Extract-Min` yang berturut-turut di dalam Dijkstra bersifat tak menurun.

Bukti. Cukuplah membandingkan dua pengeluaran yang *berurutan*, sebab barisan yang tiap sukunya sekurang-kurangnya sebesar suku sebelumnya adalah barisan tak turun. Jadi misalkan u dikeluarkan pada satu putaran dan u' pada putaran berikutnya, lalu ikutilah kunci-kuncinya sepanjang putaran di antaranya, selangkah demi selangkah.

1. Ketika u dikeluarkan, ia memegang kunci terkecil di dalam Q. Jadi setiap simpul v yang masih di dalam Q memenuhi $d[v] \geq d[u]$.
2. Putaran itu lalu merelaksasi sisi-sisi yang keluar dari u. Relaksasi (u, v) entah tidak mengubah apa pun, entah menetapkan $d[v] \leftarrow d[u] + w(u, v)$. Karena $w(u, v) \geq 0$, nilai barunya tetap sekurang-kurangnya $d[u]$.
3. Jadi pada akhir putaran setiap kunci di dalam Q tetap sekurang-kurangnya $d[u]$ — kunci yang tak tersentuh karena langkah 1, kunci yang diturunkan karena langkah 2.
4. Pengeluaran berikutnya mengambil salah satu kunci itu, sehingga $d[u'] \geq d[u]$. ■

Perhatikan bahwa langkah 2 adalah satu-satunya tempat argumen ini memakai $w \geq 0$. Dengan sisi negatif, sebuah relaksasi dapat menanam kunci *di bawah* $d[u]$, dan pengeluaran berikutnya akan mundur — persis itulah yang terjadi pada graf ongkos Profitina Laundry di §13.12.2.

Kegunaan praktisnya: pada penelusuran §13.7.4 kunci yang dikeluarkan adalah 0,5,7,8,9 — monoton, sebagaimana dijanjikan. Hal ini juga mengesahkan **keluar lebih awal** untuk persoalan sepasang tunggal: bila Anda hanya menginginkan $\delta(s, t)$, Anda boleh menghentikan gelungnya pada saat t dikeluarkan, sebab setiap kunci yang dikeluarkan sesudahnya sekurang-kurangnya sebesar itu dan tidak ada relaksasi berikutnya yang dapat menurunkan $d[t]$.

13.7.9 Waktu jalannya

Pembukuannya menyegarkan karena sederhananya, sebab seluruh biaya algoritmanya bersemayam di dalam antrean prioritasnya.

Hitunglah operasinya. Baris 5 menyisipkan setiap simpul, sehingga ada $|V|$ operasi `Insert`. Gelung `while` berjalan sekali per simpul, sehingga ada $|V|$ operasi `Extract-Min`. Baris 9–11 memeriksa tiap sisi (u, v) tepat sekali, yaitu ketika pangkalnya u dikeluarkan — di sepanjang seluruh jalannya, $\sum_{u \in V} |Adj[u]| = |E|$ pemeriksaan — dan masing-masing dapat mengakibatkan satu `Decrease-Key`. Jadi ada paling banyak $|E|$ operasi `Decrease-Key`. Selebihnya adalah pembukuan seharga $O(V + E)$. Karena itu totalnya adalah

$$T = |V| \cdot T_{\text{Extract-Min}} + |E| \cdot T_{\text{Decrease-Key}}. \tag{13.10}$$

Sekarang sulihkan ketiga implementasi antrean dari Bab 11 ke dalam Persamaan (13.10), seperti yang dikerjakan **Tabel 13.8**.

Tabel 13.8 Waktu jalan Dijkstra untuk tiga implementasi antrean prioritas. Total *binary heap* menyederhana menjadi $O(E \lg V)$ bila setiap simpul terjangkau dari s, sehingga $E \geq V - 1$; baris *Fibonacci heap* berasal dari Fredman dan Tarjan [5].

Q Diimplementasikan sebagai	T(Extract-Min)	T(Decrease-Key)	Total
larik tak terurut	$\Theta(V)$	$\Theta(1)$	$\Theta(V^2)$
binary heap	$\Theta(\lg V)$	$\Theta(\lg V)$	$O((V + E)\lg V) = O(E \lg V)$
Fibonacci heap	$O(\lg V)$ teramortisasi	$\Theta(1)$ teramortisasi	$O(V \lg V + E)$

Bacalah tabelnya sebagai nasihat, bukan sebagai remeh-temeh.

Larik tak terurut, $\Theta(V^2)$. `Extract-Min` adalah penyapuan linear; `Decrease-Key` adalah satu kali menulis ke larik. Totalnya $|V| \cdot \Theta(V) + |E| \cdot \Theta(1) = \Theta(V^2 + E) = \Theta(V^2)$, sebab $E \leq V^2$. Inilah pilihan yang *tepat* untuk graf **padat**, yang $E = \Theta(V^2)$ toh sudah berlaku: Anda tidak dapat mengalahkan $\Theta(V^2)$ ketika Anda memang harus menengok $\Theta(V^2)$ sisi, dan lariknya sama sekali tidak punya biaya tambahan *heap*.

Binary heap, $O(E\lg V)$. Pilihan baku untuk graf **jarang**. Dengan $E = \Theta(V)$ ini menjadi $O(V\lg V)$, jauh lebih baik daripada $\Theta(V^2)$. Inilah yang diimplementasikan **Kode 13.2**, lewat `priority_queue` pustaka standar dan penghapusan malas.

Fibonacci heap, $O(V\lg V + E)$. Batas terbaiknya, sebab `Decrease-Key` — operasi yang dikerjakan sampai $|E|$ kali — menjadi konstan. Ia kemenangan teoretis sekaligus kekecewaan praktis: konstanta yang tersembunyi di dalam *Fibonacci heap* cukup besar sehingga untuk graf berukuran realistis sebuah *binary heap* biasanya lebih cepat menurut jam dinding. Pakailah ia ketika E sangat besar dibandingkan V dan Anda sudah mengukurnya, bukan sebelum itu.

Titik silang antara larik dan *heap* kira-kira berada di $E = \Theta(V^2/\lg V)$. Di bawahnya *heap*-nya menang; di atasnya lariknya yang menang.

13.7.10 Diukur

Asimtotik adalah ramalan. **Kode 13.3** mengujinya, dan sekaligus menguji sesuatu yang lebih penting: bahwa algoritma-algoritma bab ini sepakat satu sama lain. Ia menjalankan tiga implementasi — Dijkstra *heap*, Dijkstra larik, Bellman–Ford — pada graf contoh dan pada dua ribu graf berarah tak negatif yang dibangkitkan secara acak, memeriksa bahwa ketiganya menghasilkan vektor jarak yang identik setiap kalinya, lalu mengukur waktu kedua Dijkstra-nya pada graf jarang dan graf padat yang ukurannya membesar.

Kode 13.3 Sebuah pemeriksaan silang sekaligus tolok ukur: Dijkstra *heap*, Dijkstra larik, dan Bellman–Ford harus sepakat pada setiap graf tak negatif, lalu kedua Dijkstra-nya diukur waktunya pada masukan jarang ($m = 8n$) dan padat ($m = n^2/4$).

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <queue>
#include <random>
#include <chrono>
#include <climits>

using namespace std;
using namespace std::chrono;

const long long INF = LLONG_MAX / 4;
struct Edge { int u, v; long long w; };

vector<long long> dijkstraHeap(int n, const vector<Edge>& edges, int s) {
    vector<vector<pair<int,long long>>> adj(n);
    for (const Edge& e : edges) adj[e.u].push_back({e.v, e.w});
    vector<long long> d(n, INF);
    d[s] = 0;
    priority_queue<pair<long long,int>, vector<pair<long long,int>>,
                   greater<pair<long long,int>>> Q;
    Q.push({0, s});
    while (!Q.empty()) {
        auto [key, u] = Q.top(); Q.pop();
        if (key > d[u]) continue;                  // entri basi
        for (auto [v, w] : adj[u])
            if (d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
    }
    return d;
}

// Dijkstra dengan larik biasa alih-alih heap: EXTRACT-MIN adalah penyapuan linear.
vector<long long> dijkstraArray(int n, const vector<Edge>& edges, int s) {
    vector<vector<pair<int,long long>>> adj(n);
    for (const Edge& e : edges) adj[e.u].push_back({e.v, e.w});
    vector<long long> d(n, INF);
    vector<bool> done(n, false);
    d[s] = 0;
    for (int it = 0; it < n; ++it) {
        int u = -1;
        for (int v = 0; v < n; ++v)                // EXTRACT-MIN yang O(V)
            if (!done[v] && (u == -1 || d[v] < d[u])) u = v;
        if (u == -1 || d[u] >= INF) break;
        done[u] = true;
        for (auto [v, w] : adj[u])
            if (d[u] + w < d[v]) d[v] = d[u] + w;  // DECREASE-KEY berongkos O(1)
```

Kode 13.3 (Lanjutan) *Sebuah pemeriksaan silang sekaligus tolok ukur: Dijkstra heap, Dijkstra larik, dan Bellman–Ford harus sepakat pada setiap graf tak negatif, lalu kedua Dijkstra-nya diukur waktunya pada masukan jarang (m = 8n) dan padat (m = n^2/4).*

```
    }
    return d;
}

vector<long long> bellmanFord(int n, const vector<Edge>& edges, int s) {
    vector<long long> d(n, INF);
    d[s] = 0;
    for (int i = 1; i <= n - 1; ++i) {
        bool changed = false;
        for (const Edge& e : edges)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) {
                d[e.v] = d[e.u] + e.w; changed = true;
            }
        if (!changed) break;
    }
    return d;
}

int main() {
    // (1) graf contoh Dijkstra: ketiganya sepakat
    vector<Edge> sample = {{0,1,10},{0,3,5},{1,2,1},{1,3,2},
                           {3,1,3},{3,2,9},{3,4,2},{2,4,4},{4,2,6},{4,0,7}};
    auto a1 = dijkstraHeap(5, sample, 0);
    auto a2 = dijkstraArray(5, sample, 0);
    auto a3 = bellmanFord(5, sample, 0);
    cout << "example graph  heap=";
    for (long long x : a1) cout << x << " ";
    cout << " array=";
    for (long long x : a2) cout << x << " ";
    cout << " BF=";
    for (long long x : a3) cout << x << " ";
    cout << ((a1 == a2 && a2 == a3) ? " -> all agree\n\n" : " -> DISAGREE\n\n");

    // (2) 2000 graf berarah tak negatif acak: ketiganya harus selalu sepakat
    mt19937 rng(12345);
    int bad = 0;
    for (int trial = 0; trial < 2000; ++trial) {
        int n = 2 + (int)(rng() % 9);
        int m = (int)(rng() % (n * n));
        vector<Edge> e;
        for (int i = 0; i < m; ++i)
            e.push_back({(int)(rng() % n), (int)(rng() % n), (long long)(rng() % 20)});
        auto h = dijkstraHeap(n, e, 0), r = dijkstraArray(n, e, 0), b = bellmanFord(n, e,
0);
        if (!(h == r && r == b)) ++bad;
    }
    cout << "2000 random non-negative digraphs: " << bad << " disagreements\n\n";
```

Kode 13.3 (Lanjutan) *Sebuah pemeriksaan silang sekaligus tolok ukur: Dijkstra heap, Dijkstra larik, dan Bellman–Ford harus sepakat pada setiap graf tak negatif, lalu kedua Dijkstra-nya diukur waktunya pada masukan jarang (m = 8n) dan padat (m = n^2/4).*

```
    // (3) bagaimana kedua Dijkstra menskala, pada graf jarang dan pada graf padat
    cout << "        n      m    array(ms)   heap(ms)\n";
    for (int n : {1000, 2000, 4000}) {
        for (int deg : {8, 0}) {                    // deg 0 berarti "padat": m = n*8 atau
n*n/4
            long long m = deg ? (long long)n * deg : (long long)n * n / 4;
            vector<Edge> e;
            e.reserve(m);
            for (long long i = 0; i < m; ++i)
                e.push_back({(int)(rng() % n), (int)(rng() % n), (long long)(rng() %
1000)});
            auto t0 = steady_clock::now();
            auto r = dijkstraArray(n, e, 0);
            auto t1 = steady_clock::now();
            auto h = dijkstraHeap(n, e, 0);
            auto t2 = steady_clock::now();
            if (r != h) { cout << "  DISAGREE\n"; return 1; }
            cout << setw(9) << n << setw(8) << m
                 << setw(11) << fixed << setprecision(1)
                 << duration<double, milli>(t1 - t0).count()
                 << setw(11) << duration<double, milli>(t2 - t1).count() << "\n";
        }
    }
    return 0;
}
```

```
example graph  heap=0 8 9 5 7  array=0 8 9 5 7  BF=0 8 9 5 7  -> all agree

2000 random non-negative digraphs: 0 disagreements

        n      m    array(ms)   heap(ms)
     1000    8000        2.1        0.5
     1000  250000        7.2        4.5
     2000   16000        6.9        1.0
     2000 1000000       27.9       18.4
     4000   32000       25.9        2.1
     4000 4000000      113.7       83.8
```

Bacalah tabelnya. Pada baris-baris **jarang** ($m = 8n$), versi lariknya tumbuh seperti n^2: dari $n = 1000$ ke $n = 4000$ adalah kelipatan empat pada n dan 2.1 → 25.9 ms, yaitu kelipatan sekitar 12 — dekat dengan 16 yang diramalkan, dengan kekurangannya berasal dari efek tembolok pada ukuran kecil. Versi *heap*-nya pada rentang yang sama berjalan 0.5 → 2.1 ms, kelipatan sekitar 4, yang cocok dengan ramalan $O(E\lg V) = O(n\lg n)$ hampir persis. Pada graf jarang, *heap*-nya antara empat sampai dua belas kali lebih cepat, dan jaraknya melebar seiring membesarnya n. (Waktu-waktu ini berasal dari satu kali jalan pada satu mesin; bilangan mutlak Anda akan berbeda, tetapi perbandingannya sesuai dengan ramalan analisisnya.)

Pada baris-baris **padat** ($m = n^2/4$) keduanya jauh lebih berdekatan — 7.2 lawan 4.5, lalu 27.9 lawan 18.4, lalu 113.7 lawan 83.8 — sebab keduanya kini didominasi oleh penyapuan sisi seharga $\Theta(E)$ dan bukan oleh antreannya. Titik silang yang diramalkan belum benar-benar tercapai pada ukuran-ukuran ini, tetapi kecenderungannya persis yang diramalkan tabelnya: keunggulan *heap*-nya menyusut seiring memadatnya graf.

Dan kedua baris pertamanya adalah yang paling penting untuk memercayai apa pun yang lain dalam bab ini: pada graf contoh dan pada dua ribu graf acak, tiga implementasi yang saling bebas menghasilkan bilangan yang sama setiap kalinya.

13.7.11 Dijkstra, Prim, dan penelusuran melebar

Ada tiga algoritma di dalam buku ini yang berkerangka sama. Menjajarkan ketiganya, sebagaimana dikerjakan **Tabel 13.9**, adalah cara tercepat untuk berhenti mengacaukan mereka.

Tabel 13.9 Penelusuran melebar, Prim, dan Dijkstra dibandingkan. Satu baris yang memisahkan Prim dari Dijkstra adalah kuncinya: Prim mengukur *satu sisi*, Dijkstra mengukur *seluruh perjalanan*. Itu pula sebabnya Prim tahan terhadap bobot negatif sedangkan Dijkstra tidak — Prim tidak pernah menjumlahkan dua bobot, sehingga ia tidak pernah memerlukan ketaksamaan segitiga.

	BFS (Bab 10)	Prim–Jarník (Bab 11)	Dijkstra (Bab 13)
Pertanyaan yang dijawab	sisi paling sedikit dari s	pohon merentang termurah	rute termurah dari s
Struktur garis depan	antrean FIFO	antrean prioritas	antrean prioritas
Kunci v	(tidak ada; urutan kedatangan)	$\min\{w(u,v): u \in \text{pohon}\}$	$\min\{d[u] + w(u,v): u \in S\}$
Aturan pembaruan	antrekan bila putih	$key[v] \leftarrow w(u,v)$	$d[v] \leftarrow d[u] + w(u,v)$
Perlu $w \geq 0$?	sama sekali tak berbobot	tidak — bobot apa pun boleh	**ya**
Arah grafnya	mana saja	tak berarah	berarah (atau mana saja)
Keluaran	pohon BFS + cacah lompatan	pohon merentang minimum	pohon lintasan terpendek + jarak
Waktu (*binary heap*)	$O(V + E)$	$O(E \lg V)$	$O(E \lg V)$

Baris terakhir tabel itu berbalas bila direnungkan. Prim aman dengan bobot negatif sebab ia tidak pernah membentuk jumlah $d[u] + w(u,v)$; ia hanya pernah membandingkan bobot sisi satu per satu, dan pembandingan tidak peduli pada tanda. Kerakusan Dijkstra bertumpu pada klaim bahwa memperpanjang sebuah lintasan tidak dapat memendekkannya, dan itu pernyataan tentang jumlah. Kerangka yang sama, tulang yang berbeda.

13.8 Algoritma Bellman–Ford

13.8.1 Mengapa kita memerlukannya

Titik berangkatnya adalah sebuah pernyataan yang lugas: *algoritma Dijkstra tidak bekerja bila ada sisi negatif.* Lalu, dalam satu kalimat, alasannya: *kita tidak dapat bersikap rakus dengan pengandaian bahwa panjang lintasan hanya akan bertambah di masa depan.* Seluruh metode Dijkstra adalah memfinalkan simpul terdekat yang belum selesai lalu tidak pernah menoleh ke belakang, dan §13.7.3 sudah memperlihatkan graf tiga simpul yang membuat hal itu salah.

Algoritma Bellman–Ford — yang muncul secara terpisah dari Richard Bellman [6] pada 1958 dan Lester Ford [7] pada 1956, dengan Edward Moore [8] sampai pada metode yang sama pada 1957 — meninggalkan kerakusan sama sekali. Ia tidak berusaha pandai-pandai soal urutan ia merelaksasi sisi. Ia merelaksasi **setiap sisi, sebanyak** $|V| - 1$ **kali**, dan bersandar pada sifat relaksasi lintasan (Lema 13.8) untuk menjamin bahwa urutan yang benar sudah terjadi di suatu tempat di dalam semua pengulangan itu. Sebagai ganti kelambatannya ia memperoleh dua hal yang tidak dapat ditawarkan Dijkstra: ia benar dengan sisi negatif, dan ia dapat **mendeteksi** siklus negatif lalu mengatakannya, alih-alih mengembalikan omong kosong.

13.8.2 Algoritmanya

Gambar 13.13 memberikan pseudokodenya.

```
BELLMAN-FORD(G, w, s)
01  untuk tiap v ∈ V[G]
02      d[v] ← ∞
03      π[v] ← NIL
04  d[s] ← 0
05  untuk i ← 1 sampai |V[G]| − 1 lakukan
06      untuk tiap sisi (u, v) ∈ E[G] lakukan
07          RELAX(u, v, w)
08  untuk tiap sisi (u, v) ∈ E[G] lakukan
09      jika d[v] > d[u] + w(u, v) maka kembalikan false
10  kembalikan true
```

Gambar 13.13 Bellman–Ford: baris 01–04 menginisialisasi, baris 05–07 menjalankan $|V| - 1$ sapuan yang masing-masing merelaksasi setiap sisi, dan baris 08–09 adalah pendeteksinya, yang mengembalikan `false` bila masih ada sisi yang dapat direlaksasi. Baris 10 mengembalikan `true` ketika jawabannya layak dipercaya.

Baris demi baris:

Baris 01–04. `INITIALIZE-SINGLE-SOURCE`, persis **Pseudokode 13.1**: setiap estimasi ∞, setiap pendahulu NIL, dan sumbernya 0. **Baris 05.** Ulangi badannya $|V| - 1$ kali. Bukan $|V|$; bukan $|E|$. Cacahnya berasal dari Teorema 13.4: tidak ada lintasan terpendek yang punya lebih dari $|V| - 1$ sisi. **Baris 06.** Untuk **setiap** sisi grafnya, dalam urutan apa pun yang kebetulan diberikan daftar sisinya. **Baris 07.** `Relax(u,v,w)` — ketiga baris yang sama dengan di mana pun juga dalam bab ini. **Baris 08–09.** Pendeteksinya. Sapulah sisi-sisinya sekali lagi. Bila ada sisi (u, v) yang masih memenuhi $d[v] > d[u] + w(u, v)$, kembalikan `false`. **Baris 10.** Bila tidak, kembalikan `true`: tidak ada siklus negatif yang terjangkau, dan d memuat jarak yang sebenarnya.

> **WAWASAN KUNCI:** Bellman–Ford sama sekali tidak mengambil keputusan apa pun. Ia tidak memilih sisi mana yang direlaksasi berikutnya, sehingga ia tidak memerlukan antrean prioritas, tidak memerlukan pengurutan, dan tidak memerlukan pengandaian apa pun tentang tanda. Satu-satunya gagasannya adalah pengulangan, dan argumen pencacahan §13.8.3 menyebutkan berapa banyak pengulangan yang sudah cukup.

13.8.3 Mengapa $|V| - 1$ sapuan sudah cukup

> **TEOREMA 13.12.** Misalkan G tidak punya siklus berbobot negatif yang terjangkau dari s. Sesudah sapuan ke-i baris 05–07, $d[v] = \delta(s, v)$ untuk setiap simpul v yang lintasan terpendeknya dari s memakai paling banyak i sisi. Akibatnya, sesudah $|V| - 1$ sapuan, $d[v] = \delta(s, v)$ untuk setiap v.

Bukti. Tetapkan sebuah simpul v yang terjangkau dari s. Menurut Teorema 13.4, ada lintasan terpendek yang *sederhana* dari s ke v; sebutlah ia

$$p = \langle v_0, v_1, \ldots, v_k \rangle, \qquad v_0 = s,\ v_k = v,$$

dan karena p sederhana ia punya $k \leq |V| - 1$ sisi.

Sekarang terapkan sifat relaksasi lintasan, Lema 13.8. Ia hanya menuntut agar sisi $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ direlaksasi **dalam urutan relatif itu**, dengan apa pun di antaranya. Lihatlah apa yang dikerjakan Bellman–Ford: sapuan 1 merelaksasi setiap sisi G, sehingga ia khususnya merelaksasi (v_0, v_1) pada suatu titik. Sapuan 2 merelaksasi setiap sisi lagi, sehingga ia merelaksasi (v_1, v_2). Sapuan i merelaksasi (v_{i-1}, v_i). Sesudah k sapuan, seluruh k sisi p sudah direlaksasi dalam urutan yang dituntut, sehingga Lema 13.8 memberikan $d[v_k] = \delta(s, v_k)$ — dan menurut Lema 13.5 ia tetap demikian.

Karena $k \leq |V| - 1$ untuk setiap v, menjalankan $|V| - 1$ sapuan mencukupi bagi semuanya serentak. ■

> **Bacalah itu sekali lagi.** Buktinya *tidak* mengklaim bahwa sapuan i merelaksasi sisi ke-i milik p dan tidak ada lagi yang berguna. Ia hanya mengklaim bahwa sapuan i merelaksasi *sekurang-kurangnya* sisi ke-i-nya, di suatu tempat di antara $|E|$ relaksasi yang dikerjakannya. Sering kali satu sapuan saja membuat kemajuan yang jauh lebih banyak — bila daftar sisinya kebetulan berurutan dengan tepat, satu sapuan dapat menyelesaikan seluruh grafnya. Batas $|V| - 1$ adalah kasus terburuknya, yang tercapai ketika daftar sisinya berurutan dengan persis salah.

Bukti kedua menjalankan induksinya dengan cara yang sedikit berbeda, lewat lintasan termurah yang banyak sisinya dibatasi, dan ada baiknya kita punya keduanya.

> **TEOREMA 13.12′ (bentuk cacah sisi).** Misalkan $\delta_i(s, u)$ menyatakan bobot lintasan termurah dari s ke u *di antara semua lintasan yang memakai paling banyak i sisi* (∞ bila tidak ada). Bila tidak ada siklus berbobot negatif yang terjangkau dari s, maka sesudah sapuan ke-i Bellman–Ford, $\delta(s, u) \leq d[u] \leq \delta_i(s, u)$ untuk setiap u.

Bukti, dengan induksi atas i.

Kasus basis, $i = 0$. Sebelum sapuan apa pun, $d[s] = 0$ dan $d[u] = \infty$ untuk $u \neq s$. Dan $\delta_0(s, s) = 0$ (lintasan kosong) sedangkan $\delta_0(s, u) = \infty$ untuk $u \neq s$ (tidak ada lintasan bersisi nol yang mencapai tempat lain). Keduanya sepakat, sehingga $d[u] \leq \delta_0(s, u)$ berlaku dengan kesamaan. Kasus basis ini memang sepele.

Langkah induksi. Andaikan $d[u] \leq \delta_{i-1}(s, u)$ untuk semua u sesudah sapuan $i - 1$. Tetapkan sebuah simpul u lalu tinjaulah sebuah lintasan termurah ke u yang memakai paling banyak i sisi. Pengamatan kuncinya adalah bahwa ada tepat dua kemungkinan:

- **Entah** ia memakai paling banyak $i - 1$ sisi, yang dalam hal ini $\delta_i(s, u) = \delta_{i-1}(s, u)$, dan $d[u]$ sudah paling banyak sebesar nilai itu dan tidak pernah dinaikkan oleh relaksasi mana pun;
- **atau** ia memakai tepat i sisi, yang dalam hal ini kupaslah sisi terakhirnya (z, u): yang tersisa adalah lintasan ke z yang bersisi paling banyak $i - 1$, dan menurut substruktur optimal ia adalah lintasan semacam itu yang termurah, sehingga $\delta_i(s, u) = \delta_{i-1}(s, z) + w(z, u)$.

Kedua kasus itu bersama-sama menyatakan bahwa δ_i mematuhi rekurens

$$\delta_i(s, u) \;=\; \min\Big(\delta_{i-1}(s, u),\; \min_{(z,u)\in E}(\delta_{i-1}(s, z) + w(z, u))\Big). \tag{13.11}$$

Sekarang tinjaulah sapuan i. Sapuan itu merelaksasi setiap sisi, sehingga khususnya ia merelaksasi sisi (z, u) pada kasus kedua. Pada saat itu $d[z] \leq \delta_{i-1}(s, z)$: hal itu berlaku sesudah sapuan $i - 1$, dan $d[z]$ hanya mungkin turun sejak itu. Sesudah relaksasinya, karena itu, $d[u] \leq d[z] + w(z, u) \leq \delta_{i-1}(s, z) + w(z, u) = \delta_i(s, u)$. Pada kasus pertama $d[u] \leq \delta_{i-1}(s, u) = \delta_i(s, u)$ sudah berlaku. Bagaimanapun juga $d[u] \leq \delta_i(s, u)$ pada akhir sapuan i, dan itulah ketaksamaan

sebelah kanan; ketaksamaan sebelah kiri, $d[u] \geq \delta(s,u)$, adalah sifat batas atas, Lema 13.5. Dengan kata-kata: *setiap sapuan mencoba merelaksasi setiap sisi — termasuk* (z,u) *— sehingga ia menangkap kedua kasus* Persamaan (13.11). ■

Mengapa ketaksamaan dan bukan kesamaan? Sebab satu sapuan dapat mengerjakan *lebih* dari satu sisi: bila sisi-sisi sebuah lintasan kebetulan muncul di dalam daftar dalam urutan yang tepat, satu sapuan dapat membawa estimasi menyusuri beberapa sisi sekaligus, dan $d[u]$ lalu turun di bawah $\delta_i(s,u)$ — ia sudah merupakan jarak yang sebenarnya. Penelusuran §13.8.4 memperlihatkan hal ini pada sapuan 2, ketika $d[x]$ mencapai 4 lewat rute dua sisi $s \to y \to x$.

Akhirnya lingkarannya tertutup: sesudah $n-1$ sapuan, $\delta(s,u) \leq d[u] \leq \delta_{n-1}(s,u)$ untuk tiap simpul u, dan karena lintasan terpendek mana pun punya paling banyak $n-1$ sisi (Teorema 13.4), $\delta_{n-1}(s,u) = \delta(s,u)$. Kedua ujung ketaksamaan itu bertemu, sehingga $d[u] = \delta(s,u)$ dan selesailah kita.

13.8.4 Contohnya, sapuan demi sapuan

Sekarang kita menelusuri Bellman–Ford pada graf lima simpul dengan tiga sisi negatif. Simpulnya adalah s, t, x, y, z dan kesepuluh sisinya adalah

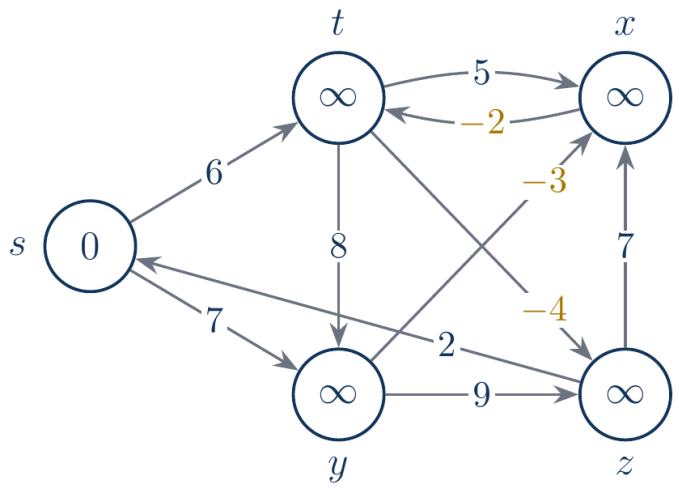
$$w(t,x) = 5,\ w(t,y) = 8,\ w(t,z) = -4,\ w(x,t) = -2,\ w(y,x) = -3,$$

$$w(y,z) = 9,\ w(z,x) = 7,\ w(z,s) = 2,\ w(s,t) = 6,\ w(s,y) = 7.$$

Urutan sisi-sisinya direlaksasi penting bagi *penelusurannya*, meskipun tidak bagi jawaban yang dicapainya, dan kita memakai urutan yang tertulis di atas: (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y).

Perhatikan bahwa kedua sisi yang keluar dari s berada **paling belakang** di dalam daftarnya. Itu sengaja dipilih sebagai kasus terburuk: tidak ada yang dapat terjadi di dalam sebuah sapuan sebelum sisi-sisi milik s sendiri direlaksasi, sehingga tiap sapuan mendorong garis depannya maju tepat satu simpul, dan keempat sapuannya diperlukan.

Panel (a) — sesudah inisialisasi. $d[s] = 0$, semua yang lain ∞.

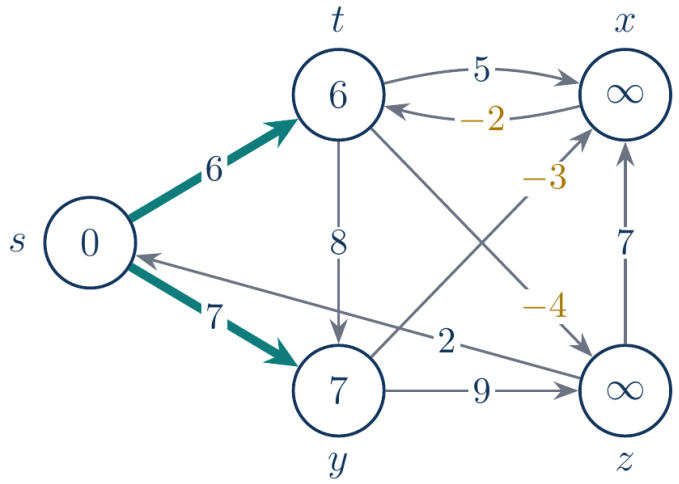


Gambar 13.14 Contoh Bellman–Ford, panel (a): inisialisasi. Tiga dari kesepuluh sisinya berbobot negatif — $w(x,t) = -2$, $w(y,x) = -3$, $w(t,z) = -4$ — dan itulah persis sebabnya Dijkstra tidak boleh dipakai di sini. Namun tidak ada *siklus* negatif: periksalah $t \to z \to s \to t$, yang berongkos $-4 + 2 + 6 = 4 > 0$, dan $t \to x \to t$, yang berongkos $5 + (-2) = 3 > 0$.

Sapuan 1. Relaksasikan kesepuluh sisinya berurutan. Delapan yang pertama tidak berbuat apa-apa, sebab $d[t]$, $d[x]$, $d[y]$, dan $d[z]$ semuanya masih ∞. Lalu:

- $(s,t), w = 6$: $d[t] = \infty > 0 + 6$, sehingga $d[t] \leftarrow 6, \pi[t] \leftarrow s$.
- $(s,y), w = 7$: $d[y] = \infty > 0 + 7$, sehingga $d[y] \leftarrow 7, \pi[y] \leftarrow s$.

Sesudah sapuan 1: $d = (s{:}\,0,\ t{:}\,6,\ x{:}\,\infty,\ y{:}\,7,\ z{:}\,\infty)$ — keadaan **Gambar 13.15**.

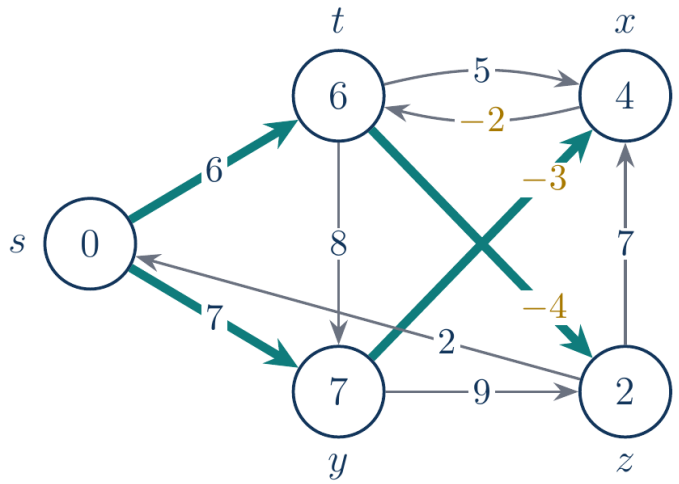


Gambar 13.15 Panel (b): sesudah sapuan 1. Hanya kedua sisi yang meninggalkan s yang berpengaruh, sebab hanya merekalah yang pangkalnya punya estimasi berhingga ketika mereka diperiksa. Anak panah tebal adalah sisi pendahulu yang sekarang.

Sapuan 2. Kini $d[t]$ dan $d[y]$ berhingga, sehingga lebih banyak sisi yang menggigit:

- $(t,x), w = 5$: $d[x] = \infty > 6 + 5 = 11$, sehingga $d[x] \leftarrow 11, \pi[x] \leftarrow t$.
- $(t,y), w = 8$: $6 + 8 = 14 > 7$ — tidak berubah.
- $(t,z), w = -4$: $d[z] = \infty > 6 + (-4) = 2$, sehingga $d[z] \leftarrow 2, \pi[z] \leftarrow t$.
- $(x,t), w = -2$: $11 + (-2) = 9 > 6$ — tidak berubah.
- $(y,x),\ w = -3$: $d[x] = 11 > 7 + (-3) = 4$, sehingga $d[x] \leftarrow 4,\ \pi[x] \leftarrow y$. **Estimasi bagi x diperbaiki dua kali di dalam sapuan yang sama.**
- $(y,z), w = 9$: $7 + 9 = 16 > 2$ — tidak berubah.
- $(z,x), w = 7$: $2 + 7 = 9 > 4$ — tidak berubah.
- $(z,s), w = 2$: $2 + 2 = 4 > 0$ — tidak berubah.
- $(s,t), (s,y)$: tidak berubah.

Sesudah sapuan 2: $d = (0,\ 6,\ 4,\ 7,\ 2)$, yang digambarkan pada **Gambar 13.16**.

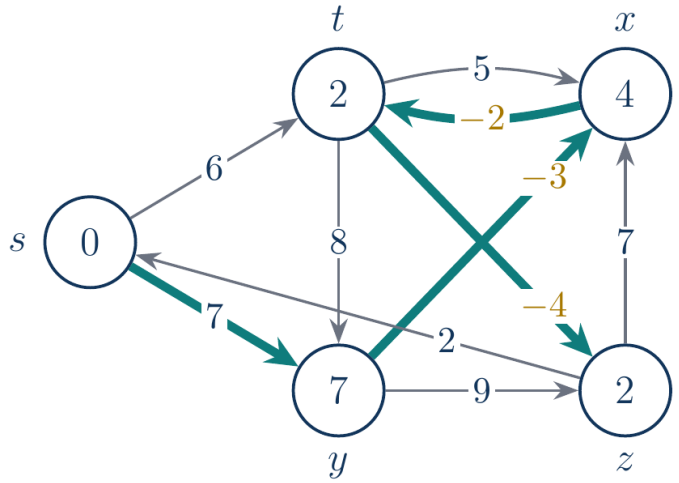


Gambar 13.16 Panel (c): sesudah sapuan 2. Setiap simpul kini punya estimasi berhingga. $d[x]$ mendarat di 4 lewat y dan sisi negatif (y, x), dan $d[z]$ di 2 lewat t dan sisi negatif (t, z). Dua dari kelima estimasinya masih salah.

Sapuan 3. Hanya satu sisi yang berbuat sesuatu:

- $(x, t), w = -2$: $d[t] = 6 > 4 + (-2) = 2$, sehingga $d[t] \leftarrow 2, \pi[t] \leftarrow x$.

Yang lain semuanya sudah sekurang-kurangnya sama baiknya. Sesudah sapuan 3: $d = (0,\ 2,\ 4,\ 7,\ 2)$, yang digambarkan pada **Gambar 13.17**.

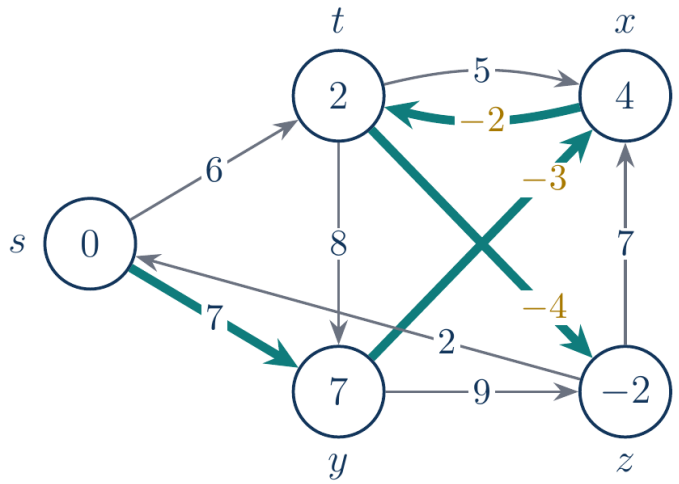


Gambar 13.17 Panel (d): sesudah sapuan 3. Estimasi bagi t turun dari 6 menjadi 2, sebab rute $s \to y \to x \to t$ berongkos $7 + (-3) + (-2) = 2$, yang mengalahkan sisi langsung berbobot 6. Anak panah pendahulunya yang tebal berayun dari s ke x. Perbaikan ini tak terlihat sampai sapuan 3 sebab ia memerlukan tiga sisi.

Sapuan 4. Sekali lagi tepat satu sisi yang berbuat sesuatu:

- $(t, z), w = -4$: $d[z] = 2 > 2 + (-4) = -2$, sehingga $d[z] \leftarrow -2, \pi[z]$ tetap t.

Sesudah sapuan 4: $d = (0,\ 2,\ 4,\ 7,\ -2)$, yang merupakan **Gambar 13.18**; inilah jarak-jarak yang sebenarnya.



Gambar 13.18 Panel (e), sesudah sapuan 4: $\delta(s, z) = -2$ lewat rute empat sisi $s \to y \to x \to t \to z$, berbobot $7 + (-3) + (-2) + (-4)$. Empat sisi adalah sebanyak-banyaknya sisi lintasan sederhana di dalam graf lima simpul, sehingga keempat sapuan $|V| - 1 = 4$ diperlukan.

Sapuan pendeteksinya. Sapulah kesepuluh sisinya sekali lagi. Periksalah sisi-sisi yang barangkali dapat menyala: (y, x) memberikan $7 - 3 = 4 = d[x]$, tidak kurang; (x, t) memberikan $4 - 2 = 2 = d[t]$, tidak kurang; (t, z) memberikan $2 - 4 = -2 = d[z]$, tidak kurang; (z, s) memberikan $-2 + 2 = 0 = d[s]$, tidak kurang. Tidak ada yang membaik. Algoritmanya mengembalikan `true`.

13.8.5 Penelusuran yang sama dalam bentuk tabel

Tabel 13.10 Jalannya Bellman–Ford selengkapnya pada graf contoh, satu baris per sapuan atas kesepuluh sisinya. Pada sapuan 2, $d[x]$ ditetapkan menjadi 11 lalu menjadi 4 *di dalam sapuan yang sama*; hanya nilai pada akhir tiap sapuan yang diperlihatkan.

Sesudah	$d[s]$	$d[t]$	$d[x]$	$d[y]$	$d[z]$	Yang berubah
inisialisasi	0	∞	∞	∞	∞	—
sapuan 1	0	6	∞	7	∞	(s, t), (s, y)
sapuan 2	0	6	4	7	2	$(t, x) \to 11$, $(t, z) \to 2$, $(y, x) \to 4$
sapuan 3	0	2	4	7	2	$(x, t) \to 2$
sapuan 4	0	2	4	7	−2	$(t, z) \to -2$
deteksi	0	2	4	7	−2	tidak ada — mengembalikan `true`

Jawaban akhirnya, dengan rutenya dibaca dari π, adalah **Tabel 13.11**.

Tabel 13.11 Keluaran Bellman–Ford pada graf contoh. Tiga dari keempat rute tak sepelenya memakai y dan sisi negatif (y, x), struktur yang mudah terlewat oleh metode rakus yang memfinalkan simpul terlalu dini.

v	s	t	x	y	z
$\delta(s, v)$	0	2	4	7	−2
$\pi[v]$	NIL	x	y	s	t
lintasan terpendek	s	s, y, x, t	s, y, x	s, y	s, y, x, t, z

13.8.6 Programnya

Kode 13.4 adalah seluruh algoritmanya, yang mencetak tiap sapuan sehingga keluarannya dapat diperiksa baris demi baris terhadap kelima panel di atas.

Kode 13.4 Bellman–Ford dengan pendeteksian siklus negatif pada graf contoh, yang mencetak setiap sapuannya. Separuh kedua `main` menurunkan $w(z, s)$ dari 2 menjadi -8, sehingga tercipta siklus $s \to t \to z \to s$ berbobot $6 + (-4) + (-8) = -6$, lalu memperlihatkan pendeteksinya menyala. Kompilasilah dengan `g++ -std=c++17 -O2 -o bellman bellman.cpp`.

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <climits>

using namespace std;

const long long INF = LLONG_MAX / 4;

struct Edge { int u, v; long long w; };

struct Graph {
    vector<string> name;
    vector<Edge>   edges;
    int n() const { return (int)name.size(); }
    int id(const string& s) const {
        return (int)(find(name.begin(), name.end(), s) - name.begin());
    }
};

string show(long long x) { return x >= INF ? "INF" : to_string(x); }

void printRow(const string& tag, const Graph& g, const vector<long long>& d) {
    cout << setw(9) << left << tag << right;
    for (int v = 0; v < g.n(); ++v)
        cout << setw(6) << (g.name[v] + "=" + show(d[v]));
    cout << "\n";
}

// Mengembalikan true ketika tidak ada siklus negatif yang terjangkau dari s.
// d dan pi terisi bagaimanapun juga.
bool bellmanFord(const Graph& g, int s, vector<long long>& d, vector<int>& pi,
                 bool verbose) {
    d.assign(g.n(), INF);
    pi.assign(g.n(), -1);
    d[s] = 0;
    if (verbose) printRow("init", g, d);
```

Kode 13.4 (Lanjutan) *Bellman–Ford dengan pendeteksian siklus negatif pada graf contoh, yang mencetak setiap sapuannya. Separuh kedua main menurunkan w(z,s) dari 2 menjadi -8, sehingga tercipta siklus s \to t \to z \to s berbobot 6 + (-4) + (-8) = -6, lalu memperlihatkan pendeteksinya menyala. Kompilasilah dengan g++ -std=c++17 -O2 -o bellman bellman.cpp.*

```
    for (int pass = 1; pass <= g.n() - 1; ++pass) {
        bool changed = false;
        for (const Edge& e : g.edges) {
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) {   // RELAX(u,v,w)
                d[e.v] = d[e.u] + e.w;
                pi[e.v] = e.u;
                changed = true;
                if (verbose)
                    cout << "   pass " << pass << ": relax ("
                         << g.name[e.u] << "," << g.name[e.v] << ") w=" << e.w
                         << " -> d[" << g.name[e.v] << "]=" << d[e.v] << "\n";
            }
        }
        if (verbose) printRow("pass " + to_string(pass), g, d);
        if (!changed) {                     // tak ada yang bergerak: sapuan berikutnya tak
mungkin menggerakkan apa pun
            if (verbose) cout << "   (no change in this pass; the rest are skipped)\n";
            break;
        }
    }

    for (const Edge& e : g.edges)           // sapuan ke-|V|: pendeteksinya
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) {
            if (verbose)
                cout << "   edge (" << g.name[e.u] << "," << g.name[e.v]
                     << ") still relaxes: " << d[e.u] << "+" << e.w
                     << " < " << d[e.v] << "  => NEGATIVE CYCLE\n";
            return false;
        }
    return true;
}

int main() {
    // Graf contoh Bellman-Ford, sisi dalam urutan yang dipakai pada penelusuran.
    Graph g;
    g.name = {"s", "t", "x", "y", "z"};
    auto E = [&](const char* a, const char* b, long long w) {
        g.edges.push_back({g.id(a), g.id(b), w});
    };
    E("t","x", 5); E("t","y", 8); E("t","z",-4);
    E("x","t",-2);
    E("y","x",-3); E("y","z", 9);
    E("z","x", 7); E("z","s", 2);
    E("s","t", 6); E("s","y", 7);
```

Kode 13.4 (Lanjutan) *Bellman–Ford dengan pendeteksian siklus negatif pada graf contoh, yang mencetak setiap sapuannya. Separuh kedua main menurunkan w(z,s) dari 2 menjadi -8, sehingga tercipta siklus s \to t \to z \to s berbobot 6 + (-4) + (-8) = -6, lalu memperlihatkan pendeteksinya menyala. Kompilasilah dengan g++ -std=c++17 -O2 -o bellman bellman.cpp.*

```
    cout << "=== the example graph ===\n";
    vector<long long> d; vector<int> pi;
    bool ok = bellmanFord(g, g.id("s"), d, pi, true);
    cout << "returns " << (ok ? "true" : "false") << "\n\n";

    cout << "vertex |  d  | pi | shortest path from s\n";
    cout << "-------+-----+----+----------------------\n";
    for (int v = 0; v < g.n(); ++v) {
        vector<string> path;
        for (int t = v; t != -1; t = pi[t]) path.push_back(g.name[t]);
        reverse(path.begin(), path.end());
        string p;
        for (size_t i = 0; i < path.size(); ++i) { if (i) p += " -> "; p += path[i]; }
        cout << "   " << g.name[v] << "   |" << setw(4) << d[v] << " |  "
             << (pi[v] == -1 ? string("-") : g.name[pi[v]]) << " | " << p << "\n";
    }

    // Graf yang sama, tetapi w(z,s) diturunkan dari 2 menjadi -8. Kini s->t->z->s
    // berongkos 6 + (-4) + (-8) = -6 < 0: sebuah siklus negatif.
    cout << "\n=== the same graph with w(z,s) = -8 ===\n";
    for (Edge& e : g.edges)
        if (e.u == g.id("z") && e.v == g.id("s")) e.w = -8;
    ok = bellmanFord(g, g.id("s"), d, pi, false);
    cout << "BELLMAN-FORD returns " << (ok ? "true" : "false") << "\n";
    return 0;
}
```

Ada dua rincian di dalam program itu yang layak disinggahi.

Pengawal `d[e.u] < INF` sebelum tiap relaksasi adalah sifat tanpa-lintasan (Lema 13.6) yang diubah menjadi kode. Tanpanya, sisi yang keluar dari simpul yang belum tercapai akan menghitung $\mathrm{INF} + w$, yang untuk w negatif *kurang* daripada INF, dan algoritmanya akan dengan riang merambatkan jarak sampah ke seluruh bagian graf yang tak terjangkau. Memakai `LLONG_MAX/4` alih-alih `LLONG_MAX` berarti bahwa seandainya pun pengawalnya terlupa, aritmetikanya tidak akan melimpah — pengaman berlapis.

`if (!changed) break;` adalah pengoptimalan cuma-cuma dengan bukti di belakangnya: bila satu sapuan penuh tidak merelaksasi apa pun, maka tidak ada nilai d yang berubah, sehingga sapuan berikutnya akan memeriksa nilai yang persis sama dan juga tidak akan mengubah apa pun, dan demikian pula setiap sapuan sesudahnya. Berhenti lebih awal tidak mungkin mengubah hasilnya. Pada daftar sisi yang tertata baik, hal ini mengubah $|V| - 1$ sapuan kasus terburuk menjadi dua atau tiga sapuan. Perhatikan bahwa programnya tetap menjalankan sapuan *pendeteksi* sesudahnya — melewatkannya berarti kehilangan uji siklus negatifnya.

13.8.7 Apa yang dicetaknya

```
=== the example graph ===
init        s=0 t=INF x=INF y=INF z=INF
   pass 1: relax (s,t) w=6 -> d[t]=6
   pass 1: relax (s,y) w=7 -> d[y]=7
pass 1      s=0   t=6 x=INF   y=7 z=INF
   pass 2: relax (t,x) w=5 -> d[x]=11
   pass 2: relax (t,z) w=-4 -> d[z]=2
   pass 2: relax (y,x) w=-3 -> d[x]=4
pass 2      s=0   t=6   x=4   y=7   z=2
   pass 3: relax (x,t) w=-2 -> d[t]=2
pass 3      s=0   t=2   x=4   y=7   z=2
   pass 4: relax (t,z) w=-4 -> d[z]=-2
pass 4      s=0   t=2   x=4   y=7  z=-2
returns true

vertex |  d  | pi | shortest path from s
-------+-----+----+---------------------
   s   |   0 |  - | s
   t   |   2 |  x | s -> y -> x -> t
   x   |   4 |  y | s -> y -> x
   y   |   7 |  s | s -> y
   z   |  -2 |  t | s -> y -> x -> t -> z

=== the same graph with w(z,s) = -8 ===
BELLMAN-FORD returns false
```

Setiap bilangannya cocok dengan **Tabel 13.10** dan dengan kelima panel di §13.8.4, termasuk $d[x] = 11$ yang sekejap di dalam sapuan 2.

13.8.8 Pendeteksian siklus negatif, dan mengapa sapuan tambahannya menemukannya

Baris 08–09 **Gambar 13.13** adalah empat baris kode yang memikul sebuah teorema.

> **TEOREMA 13.13.** Sesudah $|V| - 1$ sapuan, `Bellman-Ford` mengembalikan `false` jika dan hanya jika grafnya memuat siklus berbobot negatif yang terjangkau dari s.

Bukti, kedua arahnya.

Bila tidak ada siklus negatif yang terjangkau, ia mengembalikan `true`. Menurut Teorema 13.12, sesudah $|V| - 1$ sapuan $d[v] = \delta(s, v)$ untuk setiap v. Ambillah sembarang sisi (u, v). Bila $d[u] = \infty$, pengawalnya melewatinya. Bila tidak, menurut bentuk sisi ketaksamaan segitiga (Akibat 13.3),

$$d[v] = \delta(s, v) \;\leq\; \delta(s, u) + w(u, v) = d[u] + w(u, v),$$

sehingga uji $d[v] > d[u] + w(u, v)$ pada baris 09 bernilai salah. Tidak ada sisi yang menyala; algoritmanya mencapai baris 10 dan mengembalikan `true`.

Bila ada siklus negatif yang terjangkau, ia mengembalikan `false`. Arah ini adalah argumen penjumlahan. Andaikan algoritmanya mengembalikan `true`. Maka untuk setiap sisi (u, v) dengan $d[u]$ berhingga,

$$d[v] \;\leq\; d[u] + w(u, v). \tag{13.12}$$

Misalkan $c = \langle c_0, c_1, \ldots, c_k = c_0 \rangle$ sebuah siklus negatif yang terjangkau, sehingga $\sum_{i=1}^{k} w\,(c_{i-1}, c_i) < 0$. Setiap simpul pada c terjangkau dari s, sehingga setiap $d[c_i]$ berhingga dan Persamaan (13.12) berlaku bagi setiap sisi c. Jumlahkan Persamaan (13.12) sekeliling siklusnya:

$$\sum_{i=1}^{k} d\,[c_i] \;\le\; \sum_{i=1}^{k} d\,[c_{i-1}] \;+\; \sum_{i=1}^{k} w\,(c_{i-1}, c_i).$$

Karena siklusnya menutup, $c_0 = c_k$, sehingga kedua jumlah d-nya berjalan atas multihimpunan simpul yang persis sama dan keduanya sama besar. Coretlah keduanya:

$$0 \;\le\; \sum_{i=1}^{k} w\,(c_{i-1}, c_i) \;=\; w(c) \;<\; 0,$$

yang mustahil. Jadi algoritmanya tidak mungkin mengembalikan `true`. ■

Gagasan yang sama dapat pula diungkapkan lewat notasi δ_i pada Persamaan (13.11), dan layak pula kita punya: *bila masih ada sisi yang dapat direlaksasi di dalam grafnya, maka ada simpul u sedemikian sehingga $\delta_n(s,u) < \delta_{n-1}(s,u)$. Tetapi hanya ada n simpul di dalam G* — sehingga jalan termurah yang memakai n sisi pasti mengulang sebuah simpul, dan karenanya memuat sebuah siklus, dan karena ia *lebih murah* daripada apa pun yang bersisi $n-1$, siklus itu pasti berbobot negatif.

> **PERINGATAN:** `false` berarti "ada siklus negatif yang terjangkau dari s", bukan "grafnya punya siklus negatif". Siklus negatif yang duduk di dalam komponen yang tidak dapat Anda capai dari s tak terlihat oleh uji ini dan tidak berbahaya bagi jawabannya. Bila Anda perlu tahu apakah ada siklus negatif *mana pun* di mana pun, tambahkanlah sebuah simpul baru dengan sisi berbobot nol ke setiap simpul lainnya lalu jalankan Bellman–Ford darinya — Soal Latihan 13.14.

`false` juga tidak memberi tahu Anda *di mana* siklusnya, hanya bahwa siklus itu ada. Memulihkannya mudah begitu Anda tahu akalnya: ingatlah simpul v mana yang menyalakan pendeteksinya, berjalanlah mundur $|V|$ langkah menyusuri π (yang dijamin mendaratkan Anda *di atas* siklusnya, sebab $|V|$ langkah tidak mungkin tetap berada di luarnya), lalu susurilah π berkeliling sampai Anda kembali ke simpul itu. Soal Latihan 13.15 meminta Anda mengimplementasikan hal ini.

13.8.9 Waktu jalannya

Seluruh hitungannya muat dalam satu baris:

$$T_{\mathrm{BF}} \;=\; \underbrace{\Theta(V)}_{\text{inisialisasi}} \;+\; \underbrace{(|V|-1)\,|E|}_{\text{sapuan}} \;+\; \underbrace{|E|}_{\text{pendeteksi}} \;=\; \Theta(VE). \tag{13.13}$$

Secara rinci, dengan membaca Persamaan (13.13) suku demi suku: inisialisasinya $\Theta(V)$. Gelung utama baris 05–07 berjalan $|V|-1$ kali, dan tiap sapuan memeriksa seluruh $|E|$ sisi seharga $\Theta(1)$ masing-masing, yang memberikan $\Theta(VE)$. Pendeteksi baris 08–09 adalah satu penyapuan $\Theta(E)$ lagi, yang terserap. Totalnya $\Theta(VE)$, dan ini Θ, bukan O: tanpa pengoptimalan keluar-lebih-awal, algoritmanya selalu mengerjakan seluruh kerjanya, bagaimanapun rupa grafnya.

Seberapa buruk itu? Pada graf jarang dengan $E = \Theta(V)$ ia $\Theta(V^2)$ — sama dengan Dijkstra berbasis larik, dan lebih buruk daripada $O(V\lg V)$ milik Dijkstra *heap*. Pada graf padat dengan $E = \Theta(V^2)$ ia $\Theta(V^3)$, yang sama dengan menjalankan Floyd–Warshall dan memperoleh *semua pasangan* dengan harga yang sama. Itu alasan yang baik untuk meraih Bab 15 ketika grafnya padat dan punya sisi negatif.

> **Sebuah kehati-hatian tentang pengoptimalan yang populer.** Keluar lebih awal mengubah banyak kejadian praktis menjadi dua atau tiga sapuan, dan penghalusan lebih lanjut — memelihara antrean berisi hanya simpul-simpul yang d-nya berubah pada sapuan sebelumnya, yang dikenal sebagai SPFA atau algoritma Bellman–Ford–Moore — sering kali jauh lebih cepat. Tetapi kasus terburuknya tetap $\Theta(VE)$, dan graf lawan yang memaksa kasus terburuk itu mudah disusun dan rutin muncul di kompetisi pemrograman. Pakailah ia, tetapi jangan pernah menyebutkan batas yang lebih baik daripada $\Theta(VE)$ untuknya.

13.9 Lintasan Terpendek pada Graf Berarah Asiklik

13.9.1 Mengapa DAG itu mudah

Seluruh gagasannya muat dalam satu kalimat: *mencari lintasan terpendek pada DAG jauh lebih mudah, sebab urutan untuk melakukan relaksasinya mudah ditemukan — pengurutan topologis!*

Ingatlah kembali §13.3.4: setiap algoritma dalam bab ini adalah jawaban atas pertanyaan *dalam urutan apa sebaiknya saya merelaksasi sisi, supaya ketika saya memakai* $d[u]$ *nilainya sudah final?* Dijkstra menjawabnya dengan antrean prioritas dan membayar $O(\lg V)$ per operasi demi hak istimewa itu. Bellman–Ford menolak menjawabnya dan sebagai gantinya membayar faktor sebesar $|V|$. Pada graf berarah asiklik, jawabannya cuma-cuma: Bab 11 menunjukkan cara menghitung **urutan topologis** — sebuah daftar simpul yang membuat setiap sisi mengarah maju — dalam waktu $\Theta(V + E)$, entah lewat waktu penyelesaian pencarian mendalam-dahulu atau lewat metode derajat masuk Kahn.

Dan urutan topologis itulah persis yang kita perlukan. Bila kita memproses simpul dalam urutan topologis, maka pada saat kita sampai di u, setiap sisi yang barangkali dapat memperbaiki $d[u]$ sudah direlaksasi, sebab setiap sisi semacam itu datang dari simpul yang lebih awal di dalam urutannya. Jadi $d[u]$ sudah final, dan kita boleh merelaksasi sisi-sisi yang keluar dari u sekali lalu tidak pernah memikirkan u lagi.

13.9.2 Algoritmanya

Gambar 13.19 memberikan pseudokodenya.

```
DAG-SHORTEST-PATHS(G, w, s)
01  untuk tiap v ∈ V[G]
02      d[v] ← ∞
03  d[s] ← 0
04  urutkan simpul-simpul V[G] secara topologis
05  untuk tiap simpul u, diambil dalam urutan topologis, lakukan
06      untuk tiap simpul v ∈ Adj[u] lakukan
07          RELAX(u, v, w)
```

Gambar 13.19 `DAG-Shortest-Paths`: baris 01–03 menginisialisasi, baris 04 mengurutkan simpul secara topologis, dan baris 05–07 merelaksasi sisi keluar tiap simpul tepat sekali, dalam urutan itu. Tidak ada antrean dan tidak ada pengulangan.

Bandingkan ia dengan Dijkstra: baris 04 menggantikan antrean prioritasnya sama sekali, dan baris 05–07 menggantikan gelung `while` beserta `Extract-Min`-nya. Bandingkan ia dengan

Bellman–Ford: gelung luar sebanyak $|V| - 1$ sapuan itu lenyap, sebab satu sapuan dalam urutan yang benar mengerjakan pekerjaan semuanya.

13.9.3 Mengapa tidak diperlukan antrean prioritas

> **TEOREMA 13.14.** Bila G adalah graf berarah asiklik, `DAG-Shortest-Paths(G,w,s)` berakhir dengan $d[v] = \delta(s, v)$ untuk setiap $v \in V$, dan G_π adalah pohon lintasan terpendek yang berakar di s.

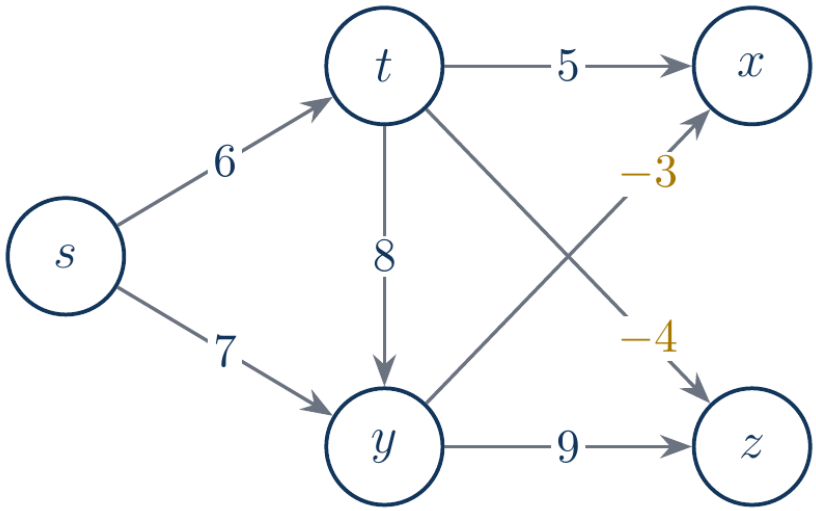
Bukti. Misalkan v terjangkau dari s dan misalkan $p = \langle v_0, v_1, \dots, v_k \rangle$, $v_0 = s$, $v_k = v$, sebuah lintasan terpendek dari s ke v. (DAG sama sekali tidak punya siklus, sehingga tentu saja tidak punya siklus negatif, dan Teorema 13.4 berlaku.)

Setiap sisi (v_{i-1}, v_i) milik p adalah sisi G, sehingga di dalam urutan topologis v_{i-1} datang benar-benar sebelum v_i. Karena itu simpul $v_0, v_1, \dots, v_k$ muncul di dalam urutan topologisnya persis dalam urutan relatif tersebut. Algoritmanya memproses simpul dalam urutan topologis dan, ketika ia memproses v_{i-1}, ia merelaksasi (v_{i-1}, v_i). Karena itu sisi-sisi p direlaksasi dalam urutan $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ — mungkin dengan banyak relaksasi lain di antaranya, yang diizinkan oleh sifat relaksasi lintasan. Menurut Lema 13.8, $d[v] = \delta(s, v)$ pada akhirnya, dan Lema 13.9 memberikan pohonnya. Bila v tak terjangkau, Lema 13.6 memberikan $d[v] = \infty = \delta(s, v)$. ■

Perhatikan apa yang **tidak** diperlukan buktinya: tidak ada pengandaian apa pun tentang tanda bobotnya. Algoritma DAG sepenuhnya senang dengan sisi negatif — sebuah DAG tidak mungkin memuat siklus, sehingga ia tidak mungkin memuat siklus negatif, dan penyakit §13.5.2 sama sekali tidak dapat muncul. Ia satu-satunya algoritma dalam bab ini yang sekaligus berwaktu linear dan aman terhadap bobot negatif, dengan harga hanya bekerja pada graf asiklik.

13.9.4 Sebuah contoh yang dikerjakan

Kita membangun contohnya dari graf yang sudah dikenal pembaca: ambillah graf contoh Bellman–Ford pada **Gambar 13.14** lalu hapuslah ketiga sisi *mundur*-nya (x, t), (z, x), dan (z, s). Yang tersisa, yang diperlihatkan pada **Gambar 13.20**, bersifat asiklik dan mempertahankan dua dari ketiga bobot negatifnya, $w(t, z) = -4$ dan $w(y, x) = -3$; yang ketiga, $w(x, t) = -2$, termasuk sisi yang dihapus.



Gambar 13.20 Graf contoh Bellman–Ford dengan ketiga sisi mundurnya dibuang: $s \to t$ (6), $s \to y$ (7), $t \to x$ (5), $t \to y$ (8), $t \to z$ (−4), $y \to x$ (−3), $y \to z$ (9). Setiap anak panah kini menunjuk dari kiri ke kanan atau ke bawah, sehingga grafnya sebuah DAG — dan ia masih memuat dua sisi negatif, yang ditangani algoritma DAG tanpa mengeluh.

Langkah 1: pengurutan topologis. Algoritma Kahn dari Bab 11 menghitung derajat masuknya: s punya 0, t punya 1 (dari s), y punya 2 (dari s dan t), x punya 2 (dari t dan y), z punya 2 (dari t dan y). Hanya s yang siap. Buang s: t turun menjadi 0 dan y turun menjadi 1. Buang t: y turun menjadi 0, x menjadi 1, z menjadi 1. Buang y: x dan z turun menjadi 0. Buang x, lalu z. Urutannya adalah

$$s,\ t,\ y,\ x,\ z.$$

Langkah 2: satu sapuan relaksasi.

Ambil s, $d[s] = 0$. Relaksasikan (s, t): $0 + 6 = 6 < \infty$, sehingga $d[t] \leftarrow 6$. Relaksasikan (s, y): $0 + 7 = 7 < \infty$, sehingga $d[y] \leftarrow 7$.

Ambil t, $d[t] = 6$ *— dan ia sudah final, sebab satu-satunya sisi yang masuk ke t adalah* (s, t)*, yang sudah direlaksasi.* Relaksasikan (t, x): $6 + 5 = 11 < \infty$, sehingga $d[x] \leftarrow 11$. Relaksasikan (t, y): $6 + 8 = 14$, tidak kurang daripada 7 — tidak berubah. Relaksasikan (t, z): $6 + (-4) = 2 < \infty$, sehingga $d[z] \leftarrow 2$.

Ambil y, $d[y] = 7$*, final.* Relaksasikan (y, x): $7 + (-3) = 4 < 11$, sehingga $d[x] \leftarrow 4$. Relaksasikan (y, z): $7 + 9 = 16$, tidak kurang daripada 2 — tidak berubah.

Ambil x, $d[x] = 4$*, final.* Tidak ada sisi keluar yang tersisa sesudah penghapusannya.

Ambil z, $d[z] = 2$*, final.* Tidak ada sisi keluar.

Selesai, dalam satu sapuan. Jawabannya adalah $\delta(s, s) = 0$, $\delta(s, t) = 6$, $\delta(s, y) = 7$, $\delta(s, x) = 4$, $\delta(s, z) = 2$. **Tabel 13.12** adalah jalan yang sama dalam bentuk tabel.

Tabel 13.12 `DAG-Shortest-Paths` pada **Gambar 13.20**, satu baris per simpul dalam urutan topologis. Bandingkan dengan **Tabel 13.10**: Bellman–Ford memerlukan empat sapuan atas sepuluh sisi — empat puluh relaksasi — untuk menyelesaikan graf selengkapnya. Di sini tujuh relaksasi sudah mencukupi. Perhatikan juga bahwa jawabannya berbeda ($\delta(s,t)$ adalah 6 di sini dan 2 di sana) sebab menghapus (x,t) membuang rute yang murah itu; keduanya adalah graf yang berbeda, bukan dua jawaban atas pertanyaan yang sama.

Simpul yang diambil	$d[s]$	$d[t]$	$d[x]$	$d[y]$	$d[z]$	Sisi yang direlaksasi
(inisialisasi)	0	∞	∞	∞	∞	—
s	0	6	∞	7	∞	$(s,t) \to 6$, $(s,y) \to 7$
t	0	6	11	7	2	$(t,x) \to 11$, (t,y) tidak, $(t,z) \to 2$
y	0	6	4	7	2	$(y,x) \to 4$, (y,z) tidak
x	0	6	4	7	2	tidak ada
z	0	6	4	7	2	tidak ada

13.9.5 Programnya

Kode 13.5 `DAG-Shortest-Paths`: pengurutan topologis Kahn disusul satu sapuan relaksasi. Uji `if ((int)order.size() != n)` adalah pendeteksi siklus yang cuma-cuma — algoritma Kahn tidak mungkin mengeluarkan setiap simpul bila ada siklus — sehingga programnya menolak memberikan jawaban pada graf yang pengandaiannya gagal. Kompilasilah dengan `g++ -std=c++17 -O2 -o dag dag.cpp`.

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>

using namespace std;

const long long INF = LLONG_MAX / 4;

struct Edge { int to; long long w; };

int main() {
    // Graf contoh Bellman-Ford dengan ketiga sisi mundurnya
    // (x,t), (z,x), dan (z,s) dihapus. Yang tersisa adalah sebuah DAG.
    vector<string> name = {"s", "t", "x", "y", "z"};
    int n = (int)name.size();
    auto id = [&](const string& s) {
        return (int)(find(name.begin(), name.end(), s) - name.begin());
    };
    vector<vector<Edge>> adj(n);
    auto E = [&](const char* a, const char* b, long long w) {
        adj[id(a)].push_back({id(b), w});
    };
    E("s","t", 6); E("s","y", 7);
    E("t","x", 5); E("t","y", 8); E("t","z",-4);
    E("y","x",-3); E("y","z", 9);

    // --- langkah 1: pengurutan topologis (algoritma derajat masuk Kahn, Bab 11) ---
    vector<int> indeg(n, 0);
    for (int u = 0; u < n; ++u) for (const Edge& e : adj[u]) ++indeg[e.to];
    queue<int> ready;
    for (int v = 0; v < n; ++v) if (indeg[v] == 0) ready.push(v);
    vector<int> order;
    while (!ready.empty()) {
        int u = ready.front(); ready.pop();
        order.push_back(u);
        for (const Edge& e : adj[u]) if (--indeg[e.to] == 0) ready.push(e.to);
    }
    if ((int)order.size() != n) { cout << "not a DAG\n"; return 1; }
```

Kode 13.5 (Lanjutan) *DAG-Shortest-Paths: pengurutan topologis Kahn disusul satu sapuan relaksasi. Uji if ((int)order.size() != n) adalah pendeteksi siklus yang cuma-cuma — algoritma Kahn tidak mungkin mengeluarkan setiap simpul bila ada siklus — sehingga programnya menolak memberikan jawaban pada graf yang pengandaiannya gagal. Kompilasilah dengan g++ -std=c++17 -O2 -o dag dag.cpp.*

```
    cout << "topological order:";
    for (int v : order) cout << " " << name[v];
    cout << "\n\n";

    // --- langkah 2: satu sapuan relaksasi, simpul dalam urutan topologis ---
    vector<long long> d(n, INF);
    vector<int> pi(n, -1);
    int s = id("s");
    d[s] = 0;

    for (int u : order) {
        cout << "take " << name[u] << " (d="
             << (d[u] >= INF ? string("INF") : to_string(d[u])) << ")\n";
        if (d[u] >= INF) { cout << "    unreachable; skip its edges\n"; continue; }
        for (const Edge& e : adj[u]) {
            long long cand = d[u] + e.w;
            cout << "    relax (" << name[u] << "," << name[e.to] << ") w=" << e.w
                 << ": " << d[u] << (e.w < 0 ? "+(" + to_string(e.w) + ")"
                                             : "+" + to_string(e.w))
                 << "=" << cand << " vs d["
                 << name[e.to] << "]="
                 << (d[e.to] >= INF ? string("INF") : to_string(d[e.to])) << " -> ";
            if (cand < d[e.to]) { d[e.to] = cand; pi[e.to] = u; cout << "improved\n"; }
            else                { cout << "no change\n"; }
        }
    }

    cout << "\nvertex |  d  | pi | shortest path from s\n";
    cout <<   "-------+-----+----+----------------------\n";
    for (int v = 0; v < n; ++v) {
        vector<string> path;
        for (int t = v; t != -1; t = pi[t]) path.push_back(name[t]);
        reverse(path.begin(), path.end());
        string p;
        for (size_t i = 0; i < path.size(); ++i) { if (i) p += " -> "; p += path[i]; }
        cout << "   " << name[v] << "   |" << setw(4) << d[v] << " |  "
             << (pi[v] == -1 ? string("-") : name[pi[v]]) << " | " << p << "\n";
    }
    return 0;
}
```

```
topological order: s t y x z

take s (d=0)
    relax (s,t) w=6: 0+6=6 vs d[t]=INF -> improved
    relax (s,y) w=7: 0+7=7 vs d[y]=INF -> improved
take t (d=6)
    relax (t,x) w=5: 6+5=11 vs d[x]=INF -> improved
    relax (t,y) w=8: 6+8=14 vs d[y]=7 -> no change
    relax (t,z) w=-4: 6+(-4)=2 vs d[z]=INF -> improved
take y (d=7)
    relax (y,x) w=-3: 7+(-3)=4 vs d[x]=11 -> improved
    relax (y,z) w=9: 7+9=16 vs d[z]=2 -> no change
take x (d=4)
take z (d=2)

vertex |  d  | pi | shortest path from s
-------+-----+----+----------------------
   s   |   0 |  - | s
   t   |   6 |  s | s -> t
   x   |   4 |  y | s -> y -> x
   y   |   7 |  s | s -> y
   z   |   2 |  t | s -> t -> z
```

13.9.6 Waktu jalannya

Jawabannya $\Theta(V + E)$ — *hanya satu relaksasi untuk tiap sisi, kira-kira V kali lebih cepat daripada Bellman–Ford.*

Pembukuannya: inisialisasi $\Theta(V)$; pengurutan topologis $\Theta(V + E)$ dengan metode mana pun dari Bab 11; gelung utamanya mengunjungi tiap simpul sekali dan tiap sisi sekali, $\Theta(V + E)$. Totalnya $\Theta(V + E)$, yang **linear terhadap ukuran masukannya** — Anda tidak dapat berbuat lebih baik, sebab Anda sekurang-kurangnya harus membaca grafnya. Tidak ada faktor lg sebab tidak ada antrean prioritas, dan tidak ada faktor $|V|$ sebab tidak ada pengulangan.

> **WAWASAN KUNCI:** Urutan topologis mengerjakan secara cuma-cuma apa yang dikerjakan antrean prioritas dengan biaya $O(\lg V)$ per simpul: ia memberi tahu Anda simpul mana yang aman difinalkan berikutnya. Ketika struktur grafnya sudah menyerahkan keterangan itu kepada Anda, janganlah membayar sebuah struktur data untuk menemukannya kembali.

13.9.7 Lintasan terpanjang dan lintasan kritis

Sebuah bonus yang jatuh cuma-cuma. Pada graf umum, persoalan *lintasan sederhana terpanjang* bersifat NP-sukar — §13.3.3 menerangkan mengapa argumen gunting-dan-tempelnya patah. Pada DAG ia sepele: negasikan setiap bobot, jalankan `DAG-Shortest-Paths`, lalu negasikan jawabannya. Setara dengan itu, gantilah `<` dengan `>` dan ∞ dengan $-\infty$ di dalam relaksasinya lalu jalankan gelung yang sama.

Inilah **metode lintasan kritis** dalam penjadwalan proyek. Simpulnya adalah tonggak, sebuah sisi (u, v) berbobot w berarti "v tidak dapat dimulai sampai w satuan waktu sesudah u", dan lintasan terpanjang dari "proyek mulai" ke "proyek selesai" adalah durasi sesingkat-singkatnya yang mungkin bagi seluruh proyeknya. Setiap sisi pada lintasan itu bersifat *kritis*: tundalah salah satunya dan seluruh proyeknya molor. Soal Latihan 13.11 meminta Anda menghitung satu untuk pipeline Profitina Laundry.

13.10 Menyusun Ulang Lintasannya, Bukan Sekadar Panjangnya

Segala sesuatu sejauh ini menghitung bilangan. Tidak ada yang menginginkan bilangan. Seorang kurir menginginkan rute; sebuah penghala menginginkan lompatan berikutnya; seorang manajer proyek menginginkan daftar tugas. Bagian ini mengubah π menjadi hal itu.

13.10.1 Larik pendahulu itulah jawabannya

Lema 13.9 sudah mengerjakan kerja teoretisnya: ketika setiap $d[v] = \delta(s, v)$, subgraf pendahulu G_π adalah pohon lintasan terpendek, dan lintasan tunggal dari s ke v di dalamnya adalah lintasan terpendek di dalam G. Yang tersisa hanyalah membacanya.

Membacanya dilakukan mundur. $\pi[v]$ adalah simpul *terakhir* sebelum v; $\pi[\pi[v]]$ adalah yang sebelum itu; dan seterusnya sampai kita mencapai s, yang π-nya NIL. Jadi kita mengumpulkan simpul dari v mundur ke s lalu membalikkan daftarnya.

Pseudokode 13.3 Mencetak lintasan terpendek dari s ke v secara rekursif lewat larik pendahulu.

```
PRINT-PATH(G, s, v)
1  bila v = s
2      cetak s
3  bila tidak, bila pi[v] = NIL
4      cetak "tidak ada lintasan dari s ke v"
5  bila tidak
6      PRINT-PATH(G, s, pi[v])
7      cetak v
```

Pseudokode 13.3 mencetak lintasannya dalam urutan yang benar tanpa pembalikan yang eksplisit: baris 6 mencetak segala sesuatu sebelum v, lalu baris 7 mencetak v. Kedalamannya adalah banyaknya sisi pada lintasan itu, paling banyak $|V| - 1$, sehingga pada graf besar versi iteratif dengan vektor yang eksplisit lebih aman. **Kode 13.6** memberikan keduanya.

13.10.2 Programnya

Kode 13.6 Penyusunan ulang lintasan dari π, secara iteratif dan rekursif, dengan dua pengaman: sebuah pengawal siklus di dalam `buildPath`, dan penghitungan ulang bobot tiap lintasan dari daftar sisinya untuk mencocokkannya dengan d. Grafnya adalah graf contoh Dijkstra ditambah satu simpul q yang tidak ditunjuk apa pun.

```
#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>

using namespace std;

const long long INF = LLONG_MAX / 4;
struct Edge { int to; long long w; };

// PRINT-PATH(G, s, v) dari larik pendahulu, ditulis tanpa rekursi.
// Mengembalikan false ketika v tak terjangkau dari s.
bool buildPath(const vector<int>& pi, int s, int v, vector<int>& path) {
    path.clear();
    int guard = 0, n = (int)pi.size();
    for (int t = v; t != -1; t = pi[t]) {
        path.push_back(t);
        if (++guard > n) return false;        // pi punya siklus: tolak berputar selamanya
    }
    reverse(path.begin(), path.end());
    return !path.empty() && path.front() == s;
}

// Bentuk rekursifnya, persis seperti yang dituliskan buku teks.
void printPathRec(const vector<string>& name, const vector<int>& pi, int s, int v) {
    if (v == s)            { cout << name[s]; return; }
    if (pi[v] == -1)       { cout << "no path from " << name[s] << " to " << name[v];
                             return; }
    printPathRec(name, pi, s, pi[v]);
    cout << " -> " << name[v];
}

int main() {
    // Graf yang sama dengan kode Dijkstra, ditambah simpul terpencil q yang
    // tidak ditunjuk apa pun, supaya kita melihat rupa "tanpa lintasan".
    vector<string> name = {"s", "u", "v", "x", "y", "q"};
    int n = (int)name.size();
    auto id = [&](const string& t) {
        return (int)(find(name.begin(), name.end(), t) - name.begin());
    };
    vector<vector<Edge>> adj(n);
    auto E = [&](const char* a, const char* b, long long w) {
        adj[id(a)].push_back({id(b), w});
    };
```

Kode 13.6 (Lanjutan) *Penyusunan ulang lintasan dari \pi, secara iteratif dan rekursif, dengan dua pengaman: sebuah pengawal siklus di dalam buildPath, dan penghitungan ulang bobot tiap lintasan dari daftar sisinya untuk mencocokkannya dengan d. Grafnya adalah graf contoh Dijkstra ditambah satu simpul q yang tidak ditunjuk apa pun.*

```
    E("s","u",10); E("s","x", 5);
    E("u","v", 1); E("u","x", 2);
    E("x","u", 3); E("x","v", 9); E("x","y", 2);
    E("v","y", 4);
    E("y","v", 6); E("y","s", 7);

    int s = id("s");
    vector<long long> d(n, INF);
    vector<int> pi(n, -1);
    d[s] = 0;
    priority_queue<pair<long long,int>, vector<pair<long long,int>>,
                   greater<pair<long long,int>>> Q;
    Q.push({0, s});
    while (!Q.empty()) {
        auto [key, u] = Q.top(); Q.pop();
        if (key > d[u]) continue;
        for (const Edge& e : adj[u])
            if (d[u] + e.w < d[e.to]) {
                d[e.to] = d[u] + e.w; pi[e.to] = u; Q.push({d[e.to], e.to});
            }
    }

    for (int v = 0; v < n; ++v) {
        cout << "s to " << name[v] << ": ";
        vector<int> path;
        if (!buildPath(pi, s, v, path)) { cout << "unreachable\n"; continue; }
        long long check = 0;
        for (size_t i = 0; i + 1 < path.size(); ++i) {       // jumlahkan ulang bobotnya
            for (const Edge& e : adj[path[i]])
                if (e.to == path[i+1]) { check += e.w; break; }
        }
        for (size_t i = 0; i < path.size(); ++i) {
            if (i) cout << " -> ";
            cout << name[path[i]];
        }
        cout << "   weight " << check << " (d = " << d[v] << ")"
             << (check == d[v] ? "  OK" : "  MISMATCH") << "\n";
    }

    cout << "\nthe recursive PRINT-PATH, for comparison:\n";
    for (int v = 0; v < n; ++v) { cout << "  "; printPathRec(name, pi, s, v); cout << "\n";
}
    return 0;
}
```

```
s to s: s   weight 0 (d = 0)  OK
s to u: s -> x -> u   weight 8 (d = 8)  OK
s to v: s -> x -> u -> v   weight 9 (d = 9)  OK
s to x: s -> x   weight 5 (d = 5)  OK
s to y: s -> x -> y   weight 7 (d = 7)  OK
s to q: unreachable

the recursive PRINT-PATH, for comparison:
  s
  s -> x -> u
  s -> x -> u -> v
  s -> x
  s -> x -> y
  no path from s to q
```

Setiap lintasan yang dilaporkan berbobot persis sebagaimana dikatakan d — dan itu bukan tautologi melainkan pemeriksaan yang sungguh-sungguh, sebab bobotnya dihitung ulang dari daftar ketetanggaannya lalu dibandingkan dengan bilangan yang dihasilkan potongan kode yang sama sekali berbeda.

13.10.3 Tiga jebakan

Membaca π sebelum algoritmanya selesai. Di tengah-tengah Dijkstra, $\pi[v]$ menunjuk pendahulu terbaik yang *ditemukan sejauh ini*. Pada penelusuran §13.7.4, $\pi[v]$ adalah x, lalu y, lalu u. Menyusuri π pada langkah 3 akan menghasilkan lintasan yang sungguh-sungguh ada, berbobot 13, yang sekadar bukan yang terpendek. Hanya ketika algoritmanya sudah berakhirlah Lema 13.9 tersedia.

Lupa menetapkan π sama sekali. Pseudokode Dijkstra pada **Gambar 13.4** hanya memperbarui d. Bila Anda menyalinnya secara harfiah, Anda akan memperoleh jarak yang benar beserta sebuah larik berisi NIL. Tetapkanlah selalu $\pi[v] \leftarrow u$ di dalam `if` yang sama yang menetapkan $d[v]$ — jangan pernah di luarnya, atau Anda akan mencatat pendahulu bagi perbaikan yang tidak terjadi.

Menyusuri π sesudah Bellman–Ford mengembalikan `false`. Dengan siklus negatif yang terjangkau, π dapat memuat siklus, dan jalan mundur yang naif akan berputar selamanya. Pengawal di dalam `buildPath` — yang mencacah langkah lalu menyerah sesudah $|V|$ langkah — tidak berongkos apa pun dan mengubah gelung tak hingga menjadi jawaban yang bersih. Pakailah ia di dalam kode mana pun yang mungkin menemui bobot yang tak dipercaya.

13.11 Algoritma Mana untuk Graf Mana

Tabel 13.13 adalah seluruh bab ini yang dimampatkan menjadi alat bantu keputusan; prosa di bawahnya adalah hal yang sama yang dinyatakan sebagai sebuah prosedur.

Tabel 13.13 Algoritma sumber tunggal mana yang dipakai. Bacalah dari atas ke bawah: tanyakan lebih dahulu apakah grafnya asiklik, lalu apakah ada bobot yang negatif, lalu seberapa padat ia. Kedua pertanyaan pertama adalah tentang *kebenaran* dan tidak menerima kompromi; hanya yang ketiga yang tentang kecepatan.

	Dijkstra	Bellman–Ford	Relaksasi DAG	BFS (Bab 10)
Grafnya harus	graf berarah apa pun	graf berarah apa pun	**asiklik**	graf apa pun
Bobot yang diizinkan	hanya $w \geq 0$	w real apa pun	w real apa pun	tidak ada (semuanya sama)
Siklus negatif	tak terdeteksi, jawabannya salah	**terdeteksi**, mengembalikan `false`	mustahil	mustahil
Waktu, jarang ($E = \Theta(V)$)	$O(V \lg V)$	$\Theta(V^2)$	$\Theta(V)$	$\Theta(V)$
Waktu, padat ($E = \Theta(V^2)$)	$\Theta(V^2)$ dengan larik	$\Theta(V^3)$	$\Theta(V^2)$	$\Theta(V^2)$
Struktur tambahan yang diperlukan	antrean prioritas	tidak ada	pengurutan topologis	antrean FIFO
Pemakaian yang khas	jaringan jalan, perutean OSPF	arbitrase mata uang, perutean RIP	penjadwalan proyek, *pipeline*	cacah lompatan, labirin tak berbobot

Prosedur keputusannya dengan kata-kata, dan ia layak dihafalkan:

1. **Apakah semua bobot sisinya sama?** Pakailah penelusuran melebar. $\Theta(V + E)$ dan sama sekali tanpa aritmetika.
2. **Apakah grafnya asiklik?** Pakailah `DAG-Shortest-Paths`. $\Theta(V + E)$, dan bobot negatif tidak apa-apa.
3. **Apakah semua bobotnya tak negatif?** Pakailah Dijkstra. *Binary heap* bila jarang, larik biasa bila padat.
4. **Bila tidak**, pakailah Bellman–Ford, dan *periksalah nilai kembaliannya* sebelum Anda memakai keluarannya.
5. **Apakah Anda memerlukan semua pasangan?** Bukalah Bab 15.

> **CONTOH (tiga keputusan):** *Sebuah jaringan jalan berisi 400.000 simpang dan 1.000.000 ruas jalan, dengan bobot berupa menit tempuh.* Bersiklus, tak negatif, sangat jarang — Dijkstra dengan *binary heap*. *Sebuah graf kebergantungan build berisi 5.000 tugas beserta durasinya.* Asiklik — `DAG-Shortest-Paths`, atau kembarannya untuk lintasan terpanjang bagi lintasan kritisnya. *Sebuah tabel berisi 40 mata uang dengan seluruh kurs berpasangannya, dengan bobot* $-\ln(\text{kurs})$. Bersiklus, padat, dan bobot negatif adalah seluruh intinya — Bellman–Ford, dijalankan justru untuk melihat apakah ia mengembalikan `false`.

13.12 Profitina dalam Praktik

13.12.1 Di mana lintasan terpendek berada di dalam sebuah platform visibilitas AI

Profitina adalah platform visibilitas AI (GEO): ia memindai sebuah situs web, menilai seberapa siap situs itu untuk *dikutip* oleh mesin jawaban AI, lalu menerangkan tiap sinyal yang diukurnya [9]. Tiga bagian produk itu adalah persoalan lintasan terpendek yang mengenakan pakaian berbeda.

Garis depan penjelajahannya adalah antrean prioritas yang berkunci biaya berjalan. Sebuah pemindaian tidak mengambil halaman-halaman sebuah situs dalam urutan ia menemukannya; ia mengambilnya dalam urutan sebuah biaya yang terakumulasi di sepanjang lintasan tautan dari URL masuknya — kedalaman klik, lompatan pengalihan, anggaran jeda robots, waktu yang sudah terpakai pada host ini. Itu persis $d[u] + w(u, v)$, dan garis depan yang selalu memunculkan halaman berikutnya yang termurah persis adalah antrean prioritas Dijkstra. Semua bobot yang terlibat tak negatif — tidak ada pengambilan yang pernah mengembalikan waktu — sehingga Dijkstra adalah algoritma yang tepat dan batas $O(E\lg V)$-nya adalah model biaya penjelajahannya. Bab 11 mengamati bahwa antrean yang sama melayani Prim; §13.7.11 menerangkan mengapa keduanya tidak dapat saling dipertukarkan di sini: anggaran penjelajahannya adalah jumlah di sepanjang sebuah rute, bukan biaya satu lompatan.

Pipeline pemindaiannya adalah sebuah DAG, dan jadwalnya adalah lintasan terpanjang. Bab 11 menunjukkan bahwa tahap-tahap sebuah pemindaian — ambil, ekstrak, urai data terstruktur, nilai tiap sinyal, agregasi, simpan, beri tahu — membentuk graf kebergantungan berarah asiklik. Bobotilah tiap sisi dengan durasi tahapnya dan lintasan *terpanjang* melalui DAG itu adalah waktu minimum yang dapat ditempuh sebuah pemindaian penuh, menurut argumen lintasan kritis §13.9.7. Ia dihitung dalam $\Theta(V + E)$ tanpa antrean prioritas, sebab urutan topologisnya sudah diketahui.

Tren skor mingguan dan irama pemindaian ulang. Papan analitik historisnya memindai ulang tiap merek menurut irama per jenjang. Memilih merek mana yang dipindai berikutnya, diberi

biaya per merek dan penalti kebasiannya, sekali lagi adalah keputusan langkah-berikutnya-yang-termurah atas sebuah graf yang bobotnya tak negatif.

13.12.2 Studi kasus yang dikerjakan: Profitina Laundry

Profitina Laundry adalah studi kasus pengajaran buku ini: sebuah laundry kecil di Surabaya dengan sebuah mobil bak, sebuah depot, dan segelintir titik pengambilan. Pertanyaan pagi ini adalah ketiga pertanyaan bab ini.

Pertanyaan 1. Berapa rute tercepat dari depot ke tiap titik pengambilan? Grafnya punya tujuh simpul — Depot, Kertajaya, Dharmahusada, Manyar, ITS, Mulyosari, Keputih — dan sebelas ruas satu arah, masing-masing berbobot menit berkendara. Menit tidak pernah negatif, sehingga ini pekerjaan Dijkstra.

Pertanyaan 2. Berapa *biaya* tiap rute, ketika satu perhentian membayar rabat? Mitra kampus di ITS membayar laundry-nya untuk mengambil di sana, sehingga ruas (ITS,Keputih) berongkos bersih *negatif* sebesar −15 ribu rupiah. Dijkstra tidak boleh lagi dipakai — §13.7.3 — sehingga ini pekerjaan Bellman–Ford.

Pertanyaan 3. Apa yang terjadi bila rabatnya ditetapkan terlalu murah hati? Bila rabatnya melampaui biaya gelung yang kembali kepadanya, seorang pengemudi dapat berputar selamanya dan dibayar untuk itu. Itulah siklus negatif, dan mendeteksinya adalah perbedaan antara sebuah lembar tarif dan sebuah kebangkrutan. **Kode 13.7** menjawab ketiga pertanyaan itu dalam satu program.

Kode 13.7 Studi kasus Profitina Laundry: Dijkstra atas menit berkendara, Bellman–Ford atas biaya bersih dengan satu ruas negatif, dan pendeteksi siklus negatifnya menyala ketika rabatnya dinaikkan terlalu jauh. Kompilasilah dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp`.

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <map>
#include <algorithm>
#include <climits>

using namespace std;

const long long INF = LLONG_MAX / 4;

struct Leg { int u, v; long long w; };

struct Net {                        // graf berarah berbobot mungil dengan simpul bernama
    vector<string> name;
    map<string,int> idx;
    vector<vector<pair<int,long long>>> adj;
    vector<Leg> edges;                          // sisi yang sama sebagai daftar datar

    int id(const string& s) {
        auto it = idx.find(s);
        if (it != idx.end()) return it->second;
        idx[s] = (int)name.size();
        name.push_back(s);
        adj.push_back({});
        return (int)name.size() - 1;
    }
    void add(const string& a, const string& b, long long w) {
        int u = id(a), v = id(b);
        adj[u].push_back({v, w});
        edges.push_back({u, v, w});
    }
    int n() const { return (int)name.size(); }
};

string pathOf(const Net& g, const vector<int>& pi, int v) {
    vector<string> p;
    for (int t = v; t != -1; t = pi[t]) p.push_back(g.name[t]);
    reverse(p.begin(), p.end());
    string out;
    for (size_t i = 0; i < p.size(); ++i) { if (i) out += " -> "; out += p[i]; }
    return out;
}
```

Kode 13.7 (Lanjutan) *Studi kasus Profitina Laundry: Dijkstra atas menit berkendara, Bellman–Ford atas biaya bersih dengan satu ruas negatif, dan pendeteksi siklus negatifnya menyala ketika rabatnya dinaikkan terlalu jauh. Kompilasilah dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
void dijkstra(const Net& g, int s, vector<long long>& d, vector<int>& pi) {
    d.assign(g.n(), INF);
    pi.assign(g.n(), -1);
    d[s] = 0;
    priority_queue<pair<long long,int>, vector<pair<long long,int>>,
                   greater<pair<long long,int>>> Q;
    Q.push({0, s});
    while (!Q.empty()) {
        auto [key, u] = Q.top(); Q.pop();
        if (key > d[u]) continue;
        for (auto [v, w] : g.adj[u])
            if (d[u] + w < d[v]) { d[v] = d[u] + w; pi[v] = u; Q.push({d[v], v}); }
    }
}

bool bellmanFord(const Net& g, int s, vector<long long>& d, vector<int>& pi) {
    d.assign(g.n(), INF);
    pi.assign(g.n(), -1);
    d[s] = 0;
    for (int i = 1; i <= g.n() - 1; ++i)
        for (const Leg& e : g.edges)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) { d[e.v] = d[e.u] + e.w; pi[e.v] =
e.u; }
    for (const Leg& e : g.edges)                 // sapuan tambahannya: pendeteksinya
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) return false;
    return true;
}

int main() {
    cout << "PROFITINA LAUNDRY - morning pickup round\n";
    cout << "========================================\n\n";

    // ---- bagian 1: menit berkendara antarperhentian. Semua tak negatif: Dijkstra. ----
    Net road;
    road.add("Depot",       "Kertajaya",   7);
    road.add("Depot",       "Dharmahusada",11);
    road.add("Kertajaya",   "Manyar",       4);
    road.add("Kertajaya",   "Dharmahusada", 3);
    road.add("Dharmahusada","ITS",          9);
    road.add("Manyar",      "ITS",          6);
    road.add("Manyar",      "Mulyosari",   10);
    road.add("ITS",         "Keputih",      5);
    road.add("ITS",         "Mulyosari",    4);
    road.add("Mulyosari",   "Keputih",      8);
    road.add("Keputih",     "Depot",       16);
```

Kode 13.7 (Lanjutan) *Studi kasus Profitina Laundry: Dijkstra atas menit berkendara, Bellman–Ford atas biaya bersih dengan satu ruas negatif, dan pendeteksi siklus negatifnya menyala ketika rabatnya dinaikkan terlalu jauh. Kompilasilah dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
    vector<long long> d; vector<int> pi;
    dijkstra(road, road.id("Depot"), d, pi);

    cout << "minutes from the depot, and the road to take:\n";
    cout << left << setw(15) << "stop" << right << setw(5) << "min" << "   route\n";
    for (int v = 0; v < road.n(); ++v)
        cout << left << setw(15) << road.name[v] << right << setw(5) << d[v]
             << "   " << pathOf(road, pi, v) << "\n";

    // ---- bagian 2: biaya bersih dalam ribuan rupiah. Titik antar mitra membayar
    //      rabat, sehingga satu sisi negatif. Dijkstra tak boleh dipakai di sini. ----
    cout << "\nnet cost per route leg (negative = partner rebate):\n";
    Net cost;
    cost.add("Depot",       "Kertajaya",   14);
    cost.add("Depot",       "Dharmahusada",22);
    cost.add("Kertajaya",   "Manyar",       8);
    cost.add("Kertajaya",   "Dharmahusada", 6);
    cost.add("Dharmahusada","ITS",         18);
    cost.add("Manyar",      "ITS",         12);
    cost.add("Manyar",      "Mulyosari",   20);
    cost.add("ITS",         "Keputih",    -15);   // mitra kampus membayar rabat
    cost.add("ITS",         "Mulyosari",    8);
    cost.add("Mulyosari",   "Keputih",     16);
    cost.add("Keputih",     "Depot",       30);

    vector<long long> dc; vector<int> pic;
    bool ok = bellmanFord(cost, cost.id("Depot"), dc, pic);
    cout << "BELLMAN-FORD returns " << (ok ? "true" : "false") << "\n";
    cout << left << setw(15) << "stop" << right << setw(6) << "cost" << "   route\n";
    for (int v = 0; v < cost.n(); ++v)
        cout << left << setw(15) << cost.name[v] << right << setw(6) << dc[v]
             << "   " << pathOf(cost, pic, v) << "\n";

    // ---- bagian 3: lembar tarif yang sama dengan rabat yang terlalu murah hati ----
    cout << "\nnow suppose the rebate is raised to -80:\n";
    for (Leg& e : cost.edges)
        if (cost.name[e.u] == "ITS" && cost.name[e.v] == "Keputih") e.w = -80;
    ok = bellmanFord(cost, cost.id("Depot"), dc, pic);
    cout << "BELLMAN-FORD returns " << (ok ? "true" : "false")
         << "  (a driver could circle for ever and be paid for it)\n";
    return 0;
}
```

```
PROFITINA LAUNDRY - morning pickup round
========================================

minutes from the depot, and the road to take:
stop             min   route
Depot              0   Depot
Kertajaya          7   Depot -> Kertajaya
Dharmahusada      10   Depot -> Kertajaya -> Dharmahusada
Manyar            11   Depot -> Kertajaya -> Manyar
ITS               17   Depot -> Kertajaya -> Manyar -> ITS
Mulyosari         21   Depot -> Kertajaya -> Manyar -> Mulyosari
Keputih           22   Depot -> Kertajaya -> Manyar -> ITS -> Keputih

net cost per route leg (negative = partner rebate):
BELLMAN-FORD returns true
stop             cost   route
Depot               0   Depot
Kertajaya          14   Depot -> Kertajaya
Dharmahusada       20   Depot -> Kertajaya -> Dharmahusada
Manyar             22   Depot -> Kertajaya -> Manyar
ITS                34   Depot -> Kertajaya -> Manyar -> ITS
Mulyosari          42   Depot -> Kertajaya -> Manyar -> Mulyosari
Keputih            19   Depot -> Kertajaya -> Manyar -> ITS -> Keputih

now suppose the rebate is raised to -80:
BELLMAN-FORD returns false  (a driver could circle for ever and be paid for it)
```

Ada tiga hal yang layak dibaca dari keluaran itu.

Ruas langsung Depot → Dharmahusada menghabiskan 11 menit, tetapi memutar jauh lewat Kertajaya menghabiskan $7 + 3 = 10$. Rute terpendek bukanlah rute dengan ruas paling sedikit — pelajaran §13.2.2, di dalam sebuah mobil bak.

Keputih berongkos 19 untuk dicapai sedangkan ITS, yang berada *di jalan menuju ke sana*, berongkos 34. Biaya yang turun ketika Anda makin jauh persis itulah yang dihasilkan sebuah sisi negatif, dan persis itu pula yang meruntuhkan jaminan Dijkstra: Akibat 13.11 menjanjikan bahwa kunci yang dikeluarkan tidak pernah turun, padahal di sini simpul berongkos 34 menghantar ke simpul berongkos 19. Pada graf yang satu ini, Dijkstra dengan penghapusan malas kebetulan masih akan mencetak bilangan yang benar, sebab Keputih sama sekali tidak dapat dicapai sebelum ITS dikeluarkan — tetapi itu keberuntungan, bukan teorema. Pada graf mana pun yang sebuah simpulnya difinalkan sebelum sisi masuk negatifnya yang murah diperiksa, jawabannya menjadi salah persis seperti yang diperlihatkan **Gambar 13.5**. Bellman–Ford tidak memerlukan keberuntungan.

Dan ketika rabatnya dinaikkan dari -15 menjadi -80, gelung Depot → Kertajaya → Manyar → ITS → Keputih → Depot berongkos $14 + 8 + 12 - 80 + 30 = -16$, dan pendeteksinya menyala. Tidak ada kepandaian sebesar apa pun di dalam kode peruteannya yang akan menangkap hal itu; hanya sapuan tambahan baris 08–09 yang menangkapnya.

13.13 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. Memakai Dijkstra pada graf yang punya sisi negatif. Galat yang paling sering terjadi dalam seluruh pokok bahasan ini, sebab programnya tidak mengeluh. Tambahkanlah sebuah asersi: bila ada bobot yang negatif, tolaklah menjalankan Dijkstra. Lihat **Gambar 13.5**.

2. Memercayai bahwa menambahkan konstanta kepada tiap bobot memperbaiki sisi negatif. Tidak, sebab lintasan bersisi k bertambah kc sedangkan lintasan bersisi 2 hanya bertambah $2c$, sehingga pembandingan di antara keduanya berubah. Soal Latihan 13.6.

3. Mengacaukan sisi negatif dengan siklus negatif. Sisi negatif itu biasa saja; Bellman–Ford dan algoritma DAG menanganinya persis seperti mereka menangani sisi yang lain. Hanya *siklus* yang bobot totalnya negatif yang menghancurkan persoalannya. **Gambar 13.2**.

4. Menjalankan Bellman–Ford sebanyak $|V|$ sapuan alih-alih $|V| - 1$, atau sebanyak $|E|$ sapuan. Yang pertama memboroskan satu sapuan; yang kedua memboroskan banyak. Teorema 13.4 menetapkan cacahnya pada $|V| - 1$ sebab lintasan sederhana punya paling banyak sebanyak itu sisi.

5. Melewatkan sapuan pendeteksinya. Tanpa baris 08–09 Anda memperoleh bilangan tanpa petunjuk apa pun apakah bilangan itu berarti sesuatu. Sapuan itu berongkos $\Theta(E)$ — satu bagian dari $|V|$ bagian totalnya — dan tidak pernah pantas dilewatkan.

6. Merelaksasi keluar dari simpul yang belum tercapai. `INF + (-4)` kurang daripada `INF`, sehingga tanpa pengawal `d[u] < INF` estimasi simpul yang tak terjangkau diam-diam menjadi berhingga. Lema 13.6 di dalam kode.

7. Limpahan bertanda dari nilai penjaga yang terlalu besar. Memakai `INT_MAX` sebagai ∞ lalu menghitung `d[u] + w` melingkar menjadi bilangan negatif yang besar, yang tampak sebagai jalan pintas yang menakjubkan bagi uji relaksasinya. Pakailah `LLONG_MAX/4`, atau kawallah lebih dahulu lalu jumlahkan kemudian — program-program bab ini mengerjakan keduanya.

8. Memperbarui d tanpa memperbarui π. Anda memperoleh panjang dan tidak memperoleh rute. §13.10.3.

9. Membaca π sebelum ia berakhir. Larik pendahulu yang setengah jadi memerikan lintasan yang sungguh-sungguh ada yang belum terpendek.

10. Melupakan `DecreaseKey`, atau memalsukannya dengan salah. Bila $d[v]$ turun dan antreannya tidak diberi tahu, antreannya akan menyerahkan kembali simpul dalam urutan yang salah dan jawabannya akan salah. Entah implementasikan `DecreaseKey` dengan semestinya atau pakailah akal-akalan penghapusan malas **Kode 13.2** — mendorong salinan kedua lalu membuang yang basi pada saat pengeluaran. Yang tidak boleh Anda lakukan adalah mendorong salinan kedua lalu *gagal* membuang yang basi, yang membuat algoritmanya memproses satu simpul dua kali.

11. Mengandaikan bahwa pohon lintasan terpendek adalah pohon merentang minimum. Keduanya benda berbeda yang menjawab pertanyaan berbeda. §13.1.2 serta Soal Latihan 13.4 dan 13.5.

12. Menjalankan algoritma DAG pada graf yang bukan DAG. Ia mengembalikan sebuah jawaban, dan jawaban itu salah. Pengurutan topologis Kahn mendeteksi siklusnya secara cuma-cuma — **Kode 13.5** memeriksa `order.size() != n` lalu menolak.

13. Menyebutkan batas yang lebih baik untuk SPFA. Penghalusan berbasis antrean atas Bellman–Ford itu cepat pada masukan yang khas dan $\Theta(VE)$ pada kasus terburuk. Jangan pernah menulis $O(E)$ di sebelahnya.

14. Memperlakukan $\delta(s, v) = \infty$ sebagai galat. Itu sebuah jawaban: tidak ada rute. Cetaklah "tak terjangkau" lalu lanjutkan.

13.14 Ringkasan Bab

Persoalan lintasan terpendek pada graf berarah berbobot $G = (V, E, w)$ menanyakan $\delta(s, v)$, yaitu bobot minimum lintasan mana pun dari s ke v, dengan bobot sebuah lintasan adalah $w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$. Dari keempat ragamnya — sepasang tunggal, sumber tunggal, tujuan tunggal, semua pasangan — kita menyelesaikan **sumber tunggal**, sebab yang pertama tereduksi kepadanya tanpa penghematan yang diketahui, yang ketiga adalah ia di atas graf transposnya, dan yang keempat entah $|V|$ kali menjalankannya atau menjadi pokok bahasan Bab 15.

Ada dua fakta struktural yang mendasari semuanya. **Substruktur optimal** (Teorema 13.1): setiap sublintasan sebuah lintasan terpendek adalah lintasan terpendek pula, dibuktikan dengan menggunting keluar sublintasan yang konon lebih baik lalu menempelkannya masuk. **Ketaksamaan segitiga** (Teorema 13.2, Akibat 13.3): $\delta(s, v) \leq \delta(s, u) + w(u, v)$ untuk setiap sisi, sebab rute melalui u adalah satu calon di antara semua rute ke v.

Sisi negatif tidak berbahaya; siklus negatif mematikan. Siklus terjangkau yang bobot totalnya negatif membuat $\delta(s, v) = -\infty$ bagi segala sesuatu di hilirnya, sebab tiap putaran tambahan lebih murah. Bila siklus semacam itu tidak ada, setiap lintasan terpendek boleh diambil sederhana dan karena itu punya paling banyak $|V| - 1$ sisi (Teorema 13.4) — fakta yang menetapkan cacah sapuan Bellman–Ford.

Relaksasi adalah satu-satunya primitifnya: bila $d[v] > d[u] + w(u, v)$, tetapkan $d[v] \leftarrow d[u] + w(u, v)$ dan $\pi[v] \leftarrow u$. Lima sifat membuatnya aman, dan setiap bukti kebenaran dalam bab ini tereduksi kepadanya: sifat **batas atas** ($d \geq \delta$ selalu, dan membeku begitu sama), sifat **tanpa-lintasan**, sifat **kekonvergenan** (jawaban yang benar untuk u ditambah satu relaksasi memberikan jawaban yang benar untuk v), sifat **relaksasi lintasan** (relaksasikan sisi-sisi sebuah lintasan terpendek berurutan dan titik ujungnya selesai), dan sifat **subgraf pendahulu** (G_π adalah pohon lintasan terpendek ketika semua $d = \delta$).

Dijkstra memfinalkan simpul dalam urutan jarak yang menaik, dengan memakai antrean prioritas yang berkunci d. Ia benar hanya untuk $w \geq 0$; Langkah 5 buktinya adalah tempat pengandaian itu dibelanjakan, dan **Gambar 13.5** adalah graf tiga simpul yang membuat pengabaiannya memberikan jawaban salah tanpa peringatan. Ongkosnya adalah $|V|$ kali `ExtractMin` ditambah $|E|$ kali `DecreaseKey`: $\Theta(V^2)$ dengan larik, $O(E \lg V)$ dengan *binary heap*, $O(V \lg V + E)$ dengan *Fibonacci heap*.

Bellman–Ford sama sekali tidak mengambil pilihan: ia merelaksasi setiap sisi $|V| - 1$ kali, yang menurut sifat relaksasi lintasan sudah cukup, lalu merelaksasi setiap sisi sekali lagi. Bila masih ada yang membaik, berarti ada siklus negatif yang terjangkau dan ia mengembalikan `false`. Ongkosnya $\Theta(VE)$.

Pada DAG, urutan topologis memberi tahu kita secara cuma-cuma apa yang dikenakan biaya $O(\lg V)$ oleh antrean prioritas untuk ditemukan, sehingga satu relaksasi per sisi dalam urutan itu menyelesaikan persoalannya dalam $\Theta(V + E)$, termasuk dengan bobot negatif. Membalikkan pembandingannya memberikan lintasan terpanjang dan lintasan kritis sebuah proyek.

Akhirnya, jawaban yang sebenarnya diinginkan siapa pun adalah **rutenya**, yang dipulihkan dengan menyusuri π mundur dari v ke s lalu membalikkannya — dan menurut Lema 13.9 hal itu sah hanya sesudah algoritmanya berakhir.

13.14.1 Menengok ke depan

Bab 14 melangkah menjauh dari graf menuju **analisis amortisasi**, yaitu teknik yang mengesahkan "$\Theta(1)$ teramortisasi" pada baris *Fibonacci heap* di **Tabel 13.8** — sebuah klaim yang sudah dua kali kita pakai tanpa dibuktikan. Bab 15 kembali ke lintasan terpendek dan menanyakan **semua pasangan** sekaligus, dengan pemrograman dinamis $\Theta(V^3)$ milik Floyd–Warshall dan dengan algoritma Johnson, yang seluruh tujuannya adalah *membobot ulang* graf bersisi negatif supaya Dijkstra — yang cepat, dan yang terlarang di sini — menjadi sah.

13.15 Soal Latihan

13.1 Hitunglah $w(p)$ untuk $p = \langle s, x, y, v \rangle$ di dalam graf contoh Dijkstra, lalu katakan apakah p adalah lintasan terpendek dari s ke v.

13.2 Berikan contoh graf berarah berbobot yang punya dua lintasan terpendek yang berbeda antara pasangan simpul yang sama.

13.3 Buktikan bahwa bila semua bobot sisinya positif (secara tegas), maka setiap lintasan terpendek bersifat sederhana.

13.4 Berikan sebuah graf tak berarah berbobot yang terhubung dengan empat simpul yang pohon lintasan terpendeknya berakar di suatu simpul s bukan merupakan pohon merentang minimum.

13.5 Buktikan atau sangkallah: bobot total sebuah pohon lintasan terpendek yang berakar di s sekurang-kurangnya sama dengan bobot total pohon merentang minimum graf yang sama.

13.6 Seorang mahasiswa mengusulkan menangani bobot negatif dengan menambahkan konstanta $c = |\min_e w(e)|$ kepada tiap sisi lalu menjalankan Dijkstra. Berikan graf yang membuat cara ini mengembalikan lintasan terpendek yang salah, dan terangkan cacatnya dalam satu kalimat.

13.7 Jalankan Dijkstra dengan tangan pada graf contoh Dijkstra dengan sumber x alih-alih s. Berikan tabel penelusurannya dalam bentuk **Tabel 13.6**.

13.8 Di dalam graf contoh Bellman–Ford, berapa sapuan paling sedikit yang tetap menghasilkan jawaban yang benar seandainya daftar sisinya ditata ulang secara optimal? Berikan tatanan semacam itu.

13.9 Buktikan bahwa di dalam Dijkstra, begitu sebuah simpul u dikeluarkan, tidak ada relaksasi berikutnya yang dapat mengubah $d[u]$ — dengan mengandaikan $w \geq 0$.

13.10 Lengkapilah bukti Lema 13.9 untuk kasus siklus berbobot nol di dalam G_π: tunjukkan bahwa siklus semacam itu tidak mungkin muncul, mengingat $\pi[s]$ tidak pernah diberi nilai.

13.11 Pipeline pemrosesan Profitina Laundry adalah: *terima* → *pilah* (10 menit), *pilah* → *cuci* (35), *pilah* → *cuci kering* (50), *cuci* → *keringkan* (40), *cuci kering* → *setrika* (15), *keringkan* → *setrika* (15), *setrika* → *lipat* (10), *lipat* → *kirim* (5). Gambarlah DAG-nya, carilah sebuah urutan topologis, lalu hitunglah lintasan kritis dari *terima* ke *kirim*.

13.12 Modifikasilah Dijkstra supaya, di antara semua lintasan terpendek, ia mengembalikan satu yang sisinya paling sedikit. Buktikan bahwa modifikasi Anda benar.

13.13 Modifikasilah Bellman–Ford supaya alih-alih mengembalikan `false` ia melabeli setiap simpul dengan $+\infty$, sebuah nilai berhingga, atau $-\infty$, secara benar. Berikan algoritmanya beserta waktu jalannya.

13.14 Berikan algoritma $O(VE)$ yang memutuskan apakah sebuah graf berarah memuat siklus negatif *mana pun*, terjangkau dari sebuah sumber tertentu atau tidak.

13.15 Perluaslah Bellman–Ford supaya ketika ia mendeteksi siklus negatif ia juga mencetak simpul-simpul salah satu siklus semacam itu, berurutan. Sahkanlah jalan $|V|$ langkah yang dipakai metode Anda.

13.16 Misalkan G berbobot tak negatif dan misalkan s sebuah sumber. Buktikan bahwa bila $\delta(s,u) = \delta(s,v)$ dan $(u,v) \in E$ maka $w(u,v) \geq 0$ — lalu berikan contoh yang menunjukkan bahwa kebalikannya gagal.

13.16 Penyelesaian Soal Latihan

13.1 $w(p) = w(s,x) + w(x,y) + w(y,v) = 5 + 2 + 6 = 13$. Ia *bukan* yang terpendek: $\delta(s,v) = 9$, yang dicapai oleh $\langle s,x,u,v\rangle$ — lihat **Tabel 13.1**.

13.2 Dua simpul s dan t yang disambungkan oleh dua rute sejajar berbobot total sama: $s \to a \to t$ dengan $w(s,a) = 1$, $w(a,t) = 1$, dan $s \to b \to t$ dengan $w(s,b) = 1$, $w(b,t) = 1$. Keduanya berbobot 2 dan keduanya terpendek. Setiap algoritma dalam bab ini mempertahankan rute yang *lebih dahulu* ditemukannya, sebab `RELAX` hanya mengganti rute bila ada perbaikan yang tegas; tidak ada yang lebih benar daripada yang lain.

13.3 Misalkan p sebuah lintasan terpendek dari s ke v dan andaikan ia mengulang sebuah simpul, sehingga ia memuat sebuah siklus c dengan sekurang-kurangnya satu sisi. Semua bobotnya positif secara tegas, sehingga $w(c) > 0$. Memotong keluar c meninggalkan lintasan p' dengan $w(p') = w(p) - w(c) < w(p)$, yang bertentangan dengan keminimalan p. Karena itu p tidak mengulang simpul mana pun. (Dengan bobot yang sekadar tak negatif, kesimpulannya melemah menjadi "*suatu* lintasan terpendek bersifat sederhana" — Teorema 13.4 — sebab siklus berbobot nol dapat hadir tanpa menambah ongkos sedikit pun.)

13.4 Simpul s,a,b,c; sisi $\{s,a\} = 1$, $\{s,b\} = 1$, $\{s,c\} = 1$, $\{a,b\} = 1$. Pohon merentang minimumnya boleh memakai $\{s,a\},\{a,b\},\{s,c\}$, berbobot total 3. Pohon lintasan terpendek dari s memakai $\{s,a\},\{s,b\},\{s,c\}$, berbobot total 3 — sama di sini, jadi perkuatlah: buatlah $\{a,b\} = 0.5$. Kini MST-nya adalah $\{s,a\},\{a,b\},\{s,c\}$ berbobot 2.5, sedangkan pohon lintasan terpendek dari s tetap $\{s,a\},\{s,b\},\{s,c\}$ berbobot 3, sebab $\delta(s,b) = 1$ lewat sisi langsung dan 1.5 lewat a. Kedua pohonnya berbeda, dan pohon lintasan terpendeknya lebih berat.

13.5 Benar, dan Soal Latihan 13.4 menunjukkan bahwa ketaksamaannya dapat tegas. Bukti: pohon lintasan terpendek yang berakar di s adalah pohon merentang bagi komponen yang terjangkau (Lema 13.9), dan pohon merentang minimum menurut definisinya adalah pohon merentang yang bobot totalnya minimum. Karena itu bobot total MST-nya paling banyak sama dengan bobot pohon merentang mana pun, khususnya pohon lintasan terpendeknya. Keduanya mengoptimalkan tujuan yang berbeda: MST meminimumkan *jumlah* bobot sisi yang dipilih, sedangkan pohon lintasan terpendek meminimumkan tiap *jarak akar-ke-simpul* secara satu per satu.

13.6 Ambillah tiga simpul s, a, t dan tiga sisi: $w(s,a) = -2$, $w(a,t) = -2$, dan $w(s,t) = -3$. Pada graf aslinya rute dua sisi $s \to a \to t$ berbobot -4 dan sisi langsungnya berbobot -3, sehingga $\delta(s,t) = -4$ lewat a. Bobot yang paling negatif adalah -3, jadi $c = 3$, dan sesudah ditambahkan setiap bobot menjadi tak negatif: $w(s,a) = 1$, $w(a,t) = 1$, $w(s,t) = 0$. Kini rute dua sisinya berbobot 2 dan sisi langsungnya berbobot 0, sehingga Dijkstra pada graf yang digeser dengan benar menemukan lintasan terpendek graf *yang digeser*, yaitu sisi langsung — yang merupakan

lintasan yang salah bagi graf aslinya. Cacatnya, dalam satu kalimat: lintasan bersisi k digelembungkan sebesar kc, sehingga pergeserannya menghukum lintasan sebanding dengan banyaknya sisi dan dapat mengubah lintasan mana yang termurah.

13.7 Dari sumber x: $d[x] = 0$ pada mulanya. Langkah 1 mengeluarkan x (0) lalu merelaksasi $(x, u) \to 3$, $(x, v) \to 9$, $(x, y) \to 2$; $Q = \{y = 2, u = 3, v = 9, s = \infty\}$. Langkah 2 mengeluarkan y (2) lalu merelaksasi (y, v): $2 + 6 = 8 < 9$ sehingga $d[v] \leftarrow 8$; (y, s): $2 + 7 = 9 < \infty$ sehingga $d[s] \leftarrow 9$; $Q = \{u = 3, v = 8, s = 9\}$. Langkah 3 mengeluarkan u (3) lalu merelaksasi (u, v): $3 + 1 = 4 < 8$ sehingga $d[v] \leftarrow 4$; (u, x): tidak berubah; $Q = \{v = 4, s = 9\}$. Langkah 4 mengeluarkan v (4), merelaksasi (v, y): $4 + 4 = 8 > 2$, tidak berubah; $Q = \{s = 9\}$. Langkah 5 mengeluarkan s (9), merelaksasi (s, u) dan (s, x): tidak berubah. Akhirnya: $\delta(x, x) = 0$, $\delta(x, y) = 2$, $\delta(x, u) = 3$, $\delta(x, v) = 4$, $\delta(x, s) = 9$.

13.8 Satu sapuan sudah cukup. Pohon lintasan terpendeknya di sini adalah satu lintasan $s \to y \to x \to t \to z$. Daftarkan keempat sisinya lebih dahulu, dalam urutan itu — $(s, y), (y, x), (x, t), (t, z)$ — disusul keenam sisi sisanya dalam urutan apa pun. Selama sapuan tunggal itu keempat sisi pohonnya direlaksasi menurut urutan lintasannya, sehingga sifat relaksasi lintasan (Lema 13.8) dipenuhi di dalam sapuan itu dan setiap d mencapai nilai sebenarnya. Nol sapuan tidak mungkin cukup, sebab $d[z] = \infty$ sebelum relaksasi apa pun. Penyapuan pendeteksi baris 08–09 tetap dijalankan sesudahnya, sebagaimana selalu wajib, untuk menegaskan bahwa tidak ada lagi yang dapat diperbaiki.

13.9 Andaikan u dikeluarkan pada suatu saat dengan $d[u] = \delta(s, u)$ (Teorema 13.10). Menurut sifat batas atas, $d[u] \geq \delta(s, u)$ selalu berlaku, dan relaksasi hanya pernah *menurunkan* $d[u]$. Sebuah penurunan akan memberikan $d[u] < \delta(s, u)$, yang bertentangan dengan sifat batas atas. Karena itu tidak ada relaksasi sesudah pengeluaran yang dapat mengubah $d[u]$. Perhatikan bahwa argumennya memerlukan $w \geq 0$ hanya melalui Teorema 13.10.

13.10 Andaikan G_π memuat sebuah siklus $\langle c_0, \ldots, c_k = c_0 \rangle$. Setiap simpul padanya punya π yang sudah diberi nilai, sehingga tidak satu pun di antaranya adalah s — sebab $\pi[s]$ tidak pernah diberi nilai oleh baris mana pun dari algoritma mana pun kita, s berderajat masuk 0 di dalam G_π dan tidak mungkin terletak pada sebuah siklus. Karena itu tiap c_i memiliki $\pi[c_i]$ yang ditetapkan oleh relaksasi (c_{i-1}, c_i), yang pada saat itu $d[c_i] = d[c_{i-1}] + w(c_{i-1}, c_i)$; karena $d[c_{i-1}]$ hanya mungkin turun sesudahnya, pada saat ini $d[c_i] \geq d[c_{i-1}] + w(c_{i-1}, c_i)$. Menjumlahkannya sekeliling siklus lalu mencoret suku d-nya memberikan $0 \geq w(\text{siklus})$. Bila $w(\text{siklus}) < 0$ kita bertentangan dengan hipotesis tanpa siklus negatifnya. Bila $w(\text{siklus}) = 0$, tinjaulah yang *terakhir* dari k pemberian nilai itu; pada saat itu ketaksamaan yang bersesuaian bersifat tegas, sebab uji relaksasinya menuntut $d[c_i] > d[c_{i-1}] + w(c_{i-1}, c_i)$ *sebelum* pemberian nilainya dan tidak ada perubahan berikutnya yang menyentuh $d[c_{i-1}]$. Sebuah ketaksamaan tegas di mana pun di dalam jumlahnya memberikan $0 > w(\text{siklus}) = 0$, yang mustahil. Jadi G_π asiklik.

13.11 Urutan topologis: *terima, pilah, cuci, cuci kering, keringkan, setrika, lipat, kirim* (juga sah: menukar *cuci kering* lebih awal). Relaksasi lintasan terpanjang dari *terima*: $\ell[\text{pilah}] = 10$; $\ell[\text{cuci}] = 45$; $\ell[\text{cuci kering}] = 60$; $\ell[\text{keringkan}] = 85$; $\ell[\text{setrika}] = \max(60 + 15, 85 + 15) = 100$; $\ell[\text{lipat}] = 110$; $\ell[\text{kirim}] = 115$. Lintasan kritisnya adalah *terima* → *pilah* → *cuci* → *keringkan* → *setrika* → *lipat* → *kirim*, 115 menit. Cuci kering punya kelonggaran $85 - 60 = 25$ menit: ia boleh dimulai terlambat atau berjalan lebih lama sampai 25 menit tanpa menunda pengirimannya.

13.12 Peliharalah larik kedua $h[v]$, yaitu banyaknya sisi pada rute yang tercatat. Inisialisasikan $h[s] = 0$, $h[v] = \infty$. Ubahlah relaksasinya menjadi: bila $d[u] + w(u, v) < d[v]$, atau $(d[u] + w(u, v) = d[v]$ **dan** $h[u] + 1 < h[v])$, maka tetapkan $d[v] \leftarrow d[u] + w(u, v)$, $h[v] \leftarrow h[u] + 1$,

$\pi[v] \leftarrow u$. Antrean prioritasnya juga harus berkunci *pasangan* $(d[v], h[v])$, dibandingkan secara leksikografis — pertama menurut d, dan menurut h hanya untuk memecah seri — sehingga di antara simpul-simpul yang berjarak sama, simpul yang dicapai dengan sisi lebih sedikit dikeluarkan lebih dahulu. *Kebenarannya:* bayangkan setiap sisi membawa bobot pasangan $(w(u, v),1)$ dan pasangan dijumlahkan komponen demi komponen. Pasangan yang dibandingkan secara leksikografis berperilaku persis seperti bilangan untuk segala hal yang dipakai §13.6 dan Teorema 13.10: menambahkan pasangan dengan komponen pertama tak negatif dan komponen kedua positif tidak pernah membuat sebuah rute lebih kecil secara leksikografis, sehingga satu-satunya tempat buktinya membelanjakan $w \geq 0$ (Langkah 5) tetap berlaku. Dijkstra dengan bobot pasangan karena itu menemukan, untuk setiap v, pasangan $(\delta(s, v), h)$ yang terkecil secara leksikografis — rute termurah, dan di antara yang termurah, rute dengan sisi paling sedikit. (Mengunci antrean dengan d saja tidak cukup: dengan sisi berbobot nol, sebuah simpul dapat dikeluarkan pada jarak yang benar tetapi dengan terlalu banyak sisi, sebelum rute seri dengan sisi lebih sedikit ditemukan.)

13.13 Jalankan $|V| - 1$ sapuannya seperti biasa. Lalu jalankan satu sapuan tambahan dan kumpulkan ke dalam sebuah himpunan N setiap simpul v yang untuknya ada sisi (u, v) yang masih terelaksasi. Kerjakan sebuah pencarian melebar-dahulu atau mendalam-dahulu *maju* dari setiap simpul N lalu tandai segala sesuatu yang dicapainya: semua simpul yang tertandai memperoleh $-\infty$. Dari yang tak tertandai, yang $d[v] = \infty$ memperoleh $+\infty$ dan sisanya mempertahankan $d[v]$ berhingganya. Waktu jalannya $\Theta(VE)$ untuk sapuan-sapuannya ditambah $\Theta(V + E)$ untuk pencarian penyebarannya, sehingga $\Theta(VE)$ seluruhnya. **Kode 13.8** mengimplementasikan persis itu.

Kode 13.8 Penyelesaian Soal Latihan 13.13: Bellman–Ford yang diperluas sehingga tiap simpul dilabeli $+\infty$, sebuah jarak berhingga, atau $-\infty$, menurut apakah ia tak terjangkau, terjangkau dengan aman, atau terjangkau dari sebuah siklus negatif.

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>

using namespace std;

const long long INF = LLONG_MAX / 4;
struct Edge { int u, v; long long w; };

int main() {
    // s -> a -> b -> c -> a adalah siklus berbobot 2 + (-5) + 1 = -2, dan c -> d
    // meninggalkannya, jadi a, b, c, dan d semuanya berdelta = -tak hingga.  e
    // terjangkau dari s tanpa menyentuh siklusnya, dan f sama sekali tak terjangkau.
    vector<string> name = {"s", "a", "b", "c", "d", "e", "f"};
    int n = (int)name.size();
    vector<Edge> edges = {
        {0,1,1}, {1,2,2}, {2,3,-5}, {3,1,1}, {3,4,3}, {0,5,4}, {6,5,1}
    };

    vector<long long> d(n, INF);
    d[0] = 0;
    for (int i = 1; i <= n - 1; ++i)
        for (const Edge& e : edges)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) d[e.v] = d[e.u] + e.w;

    // Satu sapuan lagi menandai tiap titik ujung yang masih membaik; lalu kita
    // menyebarkan tanda itu maju, sebab segala sesuatu yang terjangkau dari simpul
    // semacam itu juga terdorong ke -tak hingga.
    vector<bool> minusInf(n, false);
    queue<int> q;
    for (const Edge& e : edges)
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v] && !minusInf[e.v]) {
            minusInf[e.v] = true; q.push(e.v);
        }
    vector<vector<int>> adj(n);
    for (const Edge& e : edges) adj[e.u].push_back(e.v);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u]) if (!minusInf[v]) { minusInf[v] = true; q.push(v); }
    }
```

Kode 13.8 (Lanjutan) *Penyelesaian Soal Latihan 13.13: Bellman–Ford yang diperluas sehingga tiap simpul dilabeli +\infty, sebuah jarak berhingga, atau -\infty, menurut apakah ia tak terjangkau, terjangkau dengan aman, atau terjangkau dari sebuah siklus negatif.*

```
    cout << "vertex |  delta(s,v)\n";
    cout << "-------+------------\n";
    for (int v = 0; v < n; ++v) {
        cout << "   " << name[v] << "   |  ";
        if (minusInf[v])      cout << "-infinity\n";
        else if (d[v] >= INF) cout << "+infinity (unreachable)\n";
        else                  cout << d[v] << "\n";
    }
    return 0;
}
```

```
vertex |  delta(s,v)
-------+------------
   s   |  0
   a   |  -infinity
   b   |  -infinity
   c   |  -infinity
   d   |  -infinity
   e   |  4
   f   |  +infinity (unreachable)
```

Grafnya adalah graf **Gambar 13.2** ditambah sebuah simpul e yang terjangkau dari s tetapi tidak dari siklusnya, dan sebuah simpul f yang sama sekali tak terjangkau. Ketiga labelnya muncul.

13.14 Tambahkan sebuah simpul baru q dan sebuah sisi (q, v) berbobot 0 untuk setiap $v \in V$. Tidak ada siklus baru yang tercipta, sebab q tidak punya sisi masuk, dan setiap simpul aslinya kini terjangkau dari q. Jalankan Bellman–Ford dari q pada graf yang diperbesar itu, yang punya $|V| + 1$ simpul dan $|E| + |V|$ sisi; ia mengembalikan `false` persis ketika graf aslinya memuat siklus negatif di mana pun. Ongkosnya $\Theta((V + 1)(E + V)) = \Theta(VE)$ untuk graf terhubung.

13.15 Ketika pendeteksinya menemukan sisi (u, v) yang masih terelaksasi, tetapkan $x \leftarrow v$ lalu susurilah π mundur $|V|$ kali: $x \leftarrow \pi[x]$, diulang $|V|$ kali. *Mengapa ini mendarat di atas siklusnya:* rantai pendahulu dari v pasti memasuki siklusnya sesudah paling banyak $|V| - 1$ langkah, sebab hanya ada $|V|$ simpul yang berbeda dan rantainya tidak dapat meninggalkan sebuah siklus begitu ia berada di dalamnya; jadi sesudah $|V|$ langkah simpul yang sekarang pasti berada pada siklusnya. Lalu catatlah $y \leftarrow x$ dan susurilah π dari x sambil mengumpulkan simpul, sampai kembali ke y; balikkan daftar yang terkumpul itu untuk memperoleh siklusnya dalam urutan maju. Ongkosnya $\Theta(V)$ di atas Bellman–Ford.

13.16 Andaikan $\delta(s, u) = \delta(s, v)$ dan $(u, v) \in E$. Bentuk sisi ketaksamaan segitiga memberikan $\delta(s, v) \leq \delta(s, u) + w(u, v)$, sehingga $\delta(s, u) \leq \delta(s, u) + w(u, v)$, dan karenanya $w(u, v) \geq 0$. (Hipotesis bobot tak negatifnya bahkan tidak diperlukan untuk arah ini.) Kebalikannya gagal: pada graf contoh Dijkstra, $w(x, y) = 2 \geq 0$ tetapi $\delta(s, x) = 5 \neq 7 = \delta(s, y)$. Kesamaan jarak adalah syarat yang jauh lebih kuat daripada ketaknegatifan sisi yang menghubungkannya.

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, 1st ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1993.

[3] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[4] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[5] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[6] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[7] L. R. Ford, Jr., "Network flow theory," RAND Corporation, Santa Monica, CA, USA, Paper P-923, Aug. 1956.

[8] E. F. Moore, "The shortest path through a maze," in *Proc. International Symposium on the Theory of Switching*, Cambridge, MA, USA, 1959, pp. 285–292.

[9] Profitina, “Profitina — AI Visibility (GEO) platform,” product documentation, app.profitina.com, 2026.

Bab 14
Analisis Amortisasi

Di akhir bab ini Anda akan mampu memandang sebuah struktur data yang operasinya kadang murah dan sesekali membawa bencana, lalu membuktikan sebuah batas atas *seluruh barisannya* yang jauh lebih baik daripada batas yang Anda peroleh dengan menjumlahkan kasus terburuk satu per satu operasi. Anda akan mampu menerangkan, kepada seseorang yang belum pernah mendengar kata itu, apa arti *teramortisasi* — dan, yang sama pentingnya, apa yang bukan artinya: ia bukan analisis kasus rata-rata, tidak ada peluang muncul di dalamnya di mana pun, dan batas yang dibuktikannya berlaku untuk *setiap* barisan operasi, termasuk barisan yang dipilih oleh seorang lawan. Anda akan mampu menerapkan ketiga metode bakunya — metode agregat, metode akuntansi, dan metode potensial — pada struktur yang sama dan memperoleh jawaban yang sama dengan tiga cara. Anda akan mampu mengambil tiga soal klasik — yang dikenal dengan nomornya pada edisi ketiga buku Cormen dkk. sebagai 17.1-3, 17.2-2, dan 17.3-6, dan yang muncul dengan rumusan yang sama pada bab analisis amortisasi edisi keempat [1] — lalu mengerjakan masing-masingnya dari baris pertama sampai baris terakhir. Anda akan mampu mengimplementasikan sebuah tabel dinamis yang melipatduakan dirinya ketika penuh dan menyeparuhkan dirinya ketika kosong, mengatakan dengan tepat mengapa menyeparuhkan pada keadaan separuh penuh adalah bencana sedangkan menyeparuhkan pada seperempat penuh bukan, serta menuliskan barisan operasi yang membuktikannya. Dan Anda akan mampu menjawab pertanyaan yang digantung Bab 11: mengapa hutan himpunan saling lepas dengan gabung menurut peringkat dan pemampatan lintasan berbiaya $O(\alpha(n))$ tiap operasi, apa sebenarnya α itu, dan mengapa nilainya paling besar 4 untuk setiap persoalan yang akan pernah berjalan di atas mesin fisik.

14.1 Dari Mana Kita Melanjutkan

14.1.1 Sebuah utang dari Bab 11

Bab 11 membangun sebuah hutan himpunan saling lepas untuk menjalankan algoritma Kruskal. Ia menambahkan dua akal-akalan — gabung menurut peringkat dan pemampatan lintasan — lalu mengatakan sesuatu yang pasti terasa seperti mengelak:

Dengan kedua akal-akalan itu, biaya barisan berisi m operasi atas n elemen adalah $O(m \cdot \alpha(n))$, dengan α adalah fungsi Ackermann balikan… Anda tidak dapat membatasi satu `FindSet` tunggal dengan $\alpha(n)$, sebab satu pencarian sungguh-sungguh dapat berongkos $\Theta(\lg n)$; Anda hanya dapat membatasi *rata-ratanya atas seluruh barisan*.

Kalimat itu memuat seluruh gagasan bab ini dalam satu anak kalimat: *Anda hanya dapat membatasi rata-ratanya atas seluruh barisan*. Bab 11 juga berjanji bahwa metode potensial Bab 14 adalah alat untuk membuktikan hal semacam itu. Bab ini menepati janji tersebut. §14.8 kembali kepada union–find dan menerangkan, dengan kecepatan seorang pemula, dari mana α itu datang.

Bab yang sama membuat janji kedua. Tabel biaya antrean prioritasnya mencantumkan *Fibonacci heap* [2] dengan `DecreaseKey` pada "$O(1)$ teramortisasi", dan berkomentar bahwa itu "persis

klaim semacam itulah yang keberadaannya hendak dibuktikan oleh Bab 14". Di akhir bab ini Anda akan paham persis apa yang dinyatakan oleh baris tabel tersebut.

14.1.2 Apa yang perlu Anda bawa

Sangat sedikit. Anda perlu tahu apa itu tumpukan (setumpuk benda yang penambahan dan pembuangannya hanya dilakukan di puncak), apa itu larik, serta notasi O, Θ, dan Ω dari Bab 3. Anda perlu merasa nyaman dengan lambang penjumlahan $\sum$, dan Anda perlu mengingat satu fakta tentang deret geometri, yang akan kita turunkan ulang ketika kita memerlukannya. Selebihnya dibangun di sini.

Satu lambang akan muncul terus-menerus, jadi mari kita tetapkan sekarang. Sepanjang bab ini, n adalah **banyaknya operasi di dalam barisan yang sedang kita analisis** — bukan banyaknya elemen di dalam strukturnya. Ini perubahan kebiasaan. Di Bab 5, n adalah ukuran larik yang sedang diurutkan. Di sini ia mencacah *operasi*: push, pop, increment, penyisipan. Ketika kita juga memerlukan banyaknya elemen, kita akan memberinya nama lain, misalnya s untuk banyaknya item di sebuah tumpukan atau num untuk banyaknya item di sebuah tabel.

14.2 Pertanyaan yang Dijawab Analisis Amortisasi

14.2.1 Sebuah batas yang benar, dan tidak berguna

Berikut sebuah struktur data kecil. Ia sebuah tumpukan, jadi ia punya dua operasi yang biasa:

- `Push(S, x)` — letakkan item x di puncak tumpukan S;
- `Pop(S)` — ambil item teratas dari tumpukan S lalu kembalikan.

Keduanya jelas berwaktu konstan: masing-masing menyentuh tepat satu slot larik yang mendasarinya. Sekarang tambahkan operasi ketiga:

- `Multipop(S, k)` — buang k item teratas dari S, atau semuanya bila isinya kurang dari k.

`Multipop` tidak berwaktu konstan. Bila tumpukannya memuat s item dan Anda memanggil `Multipop(S, k)`, operasi itu membuang $\min(s, k)$ item, dan membuang satu item berbiaya satu satuan, sehingga biayanya $\min(s, k)$.

Sekarang andaikan seseorang menyerahkan kepada Anda sebuah program yang melakukan n operasi semacam itu, dalam urutan yang tidak Anda kendalikan, bermula dari tumpukan kosong, lalu bertanya: berapa lama program itu berjalan?

Inilah analisis yang dilakukan semua orang lebih dahulu. Operasi tunggal yang paling mahal adalah `Multipop` yang paling mahal. Sebuah `Multipop` berbiaya $\min(s, k)$, dan tumpukannya tidak pernah dapat memuat lebih dari n item (sebab tiap item harus di-push lebih dahulu, sedangkan operasinya paling banyak n). Jadi satu operasi berbiaya paling besar n. Ada n operasi. Maka totalnya paling besar

$$n \times n = n^2,$$

dan programnya berjalan dalam waktu $O(n^2)$.

Setiap kata di situ benar. Tidak ada kekeliruan di dalamnya. Dan ia nyaris tidak berguna, sebab jawaban yang sebenarnya adalah $O(n)$ — lebih baik dengan faktor n. Untuk $n = 1{,}000{,}000$ batas yang jujur-tetapi-kasar itu berkata "sampai satu triliun langkah", sedangkan kenyataannya "di bawah dua juta". Itulah beda antara program yang tidak berani Anda kirimkan dan program yang selesai sebelum tangan Anda lepas dari tetikus.

> **DI MANA BATAS KASARNYA KELIRU.** Ia mengalikan biaya terburuk *satu* operasi dengan banyaknya operasi, seolah-olah setiap operasi dapat menjadi yang terburuk *pada saat yang sama*. Tidak bisa. Sebuah `Multipop` yang membuang sejuta item hanya dapat terjadi sesudah sejuta `Push` terjadi — dan push-push itu adalah bagian dari jatah n operasi yang sama. Operasi yang mahal dan operasi yang murah tidaklah saling bebas: **yang mahal dibayar oleh yang murah yang datang sebelumnya**. Batas kasarnya membuang hubungan itu, dan seluruh isi bab ini adalah tentang tidak membuangnya.

14.2.2 Faktor n itu, dibuat nyata

Mari kita buat jurangnya kelihatan dengan dua barisan tegas yang masing-masing berisi tepat n operasi.

Barisan pertama. Push $n - 1$ item, lalu panggil `Multipop(S, n)` sekali.

- $n - 1$ push berbiaya 1 masing-masing, totalnya $n - 1$.
- Satu `Multipop` itu membuang $n - 1$ item, jadi ia berbiaya $n - 1$.
- Total keseluruhan: $(n - 1) + (n - 1) = 2n - 2$.

Satu `Multipop` itu sungguh-sungguh berbiaya $\Theta(n)$. Batas kasarnya tidak berbohong tentang *dia*. Tetapi barisannya secara keseluruhan berbiaya sekitar $2n$, bukan n^2, sebab hanya ada *satu* operasi mahal dan sisanya semurah mungkin.

Barisan kedua. Cobalah punya banyak operasi mahal. Push $n/2$ item, lalu lakukan $n/2$ pemanggilan `Multipop(S, n)`.

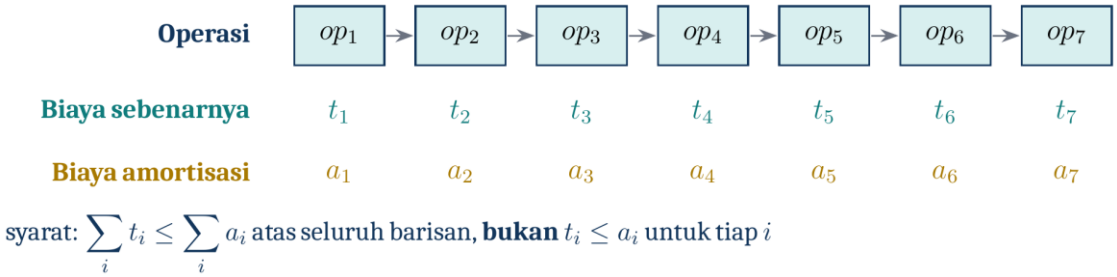
- $n/2$ push berbiaya $n/2$ seluruhnya.
- `Multipop` yang *pertama* mengosongkan tumpukannya, berbiaya $n/2$.
- Sisanya, $n/2 - 1$ buah `Multipop`, menemukan tumpukan kosong dan berbiaya 0 masing-masing.
- Total keseluruhan: $n/2 + n/2 = n$.

Si lawan ingin punya banyak `Multipop` yang mahal dan hanya bisa mendapat satu. **Itulah intinya.** Tidak ada barisan berisi n operasi pada struktur ini yang berbiaya lebih dari sekitar $2n$, dan §14.3.2 membuktikannya. Batas $O(n^2)$ meleset dengan faktor n — bukan dengan sebuah konstanta, bukan dengan sebuah logaritma, melainkan dengan seluruh n.

> **PERTANYAAN BAB INI.** *Diberikan sebuah barisan berisi n operasi pada sebuah struktur data, berapa biaya total seluruh barisannya pada kasus terburuk, dan karenanya berapa biaya kasus terburuk per operasi bila barisannya diperlakukan sebagai satu kesatuan?* Bilangan terakhir itu — biaya total dibagi n — itulah yang akan kita sebut **biaya amortisasi per operasi**. Kata "amortisasi" berasal dari dunia keuangan: mengamortisasi sebuah utang berarti menyebar satu pembayaran besar sekali-jalan menjadi banyak angsuran kecil yang rata. Di sini kita menyebar biaya operasi mahal yang jarang itu ke atas banyak operasi murah yang memungkinkannya terjadi.

14.2.3 Tiga biaya, dan satu-satunya pertidaksamaan yang penting

Gambar 14.1 menyajikan kosakatanya tanpa aljabar sama sekali, dan gambar itu layak dibaca pelan-pelan sebab seluruh pokok bahasannya ada di dalamnya.



Gambar 14.1 Operasi $op_1, \ldots, op_7$ beserta **biaya sebenarnya** t_i dan **biaya amortisasi** a_i yang kita tetapkan untuknya. Syaratnya adalah $\sum t_i \le \sum a_i$ atas seluruh barisan, dan sama sekali **bukan** $t_i \le a_i$ untuk tiap i.

Bacalah ketiga barisnya berurutan.

Baris teratas adalah barisan operasinya sendiri: $op_1, op_2, \ldots$ — pemanggilan sungguhan yang dilakukan programnya, dalam urutan ia melakukannya.

Baris tengah adalah **biaya sebenarnya** tiap operasi. Gambar itu menuliskannya t_i; Cormen dkk. [1] dan kebanyakan buku lain menuliskannya c_i, dan begitu pula kita mulai §14.3 dan seterusnya, tetapi maksud keduanya sama: banyaknya langkah dasar yang sungguh-sungguh dan jujur dilakukan operasi ke-i ketika programnya dijalankan. Biaya sebenarnya sebuah `Push` adalah 1. Biaya sebenarnya sebuah `Multipop` yang membuang 37 item adalah 37.

Baris terbawah adalah **biaya amortisasi**, ditulis a_i. Ini bilangan yang *kita karang*. Ia tidak diukur; ia ditetapkan oleh kita, para penganalisisnya, dan kita bebas menetapkan apa pun yang kita suka — asalkan kita menuruti satu aturan.

Aturan itu adalah syarat pertama yang diperlihatkan gambar tersebut:

$$\sum_{i=1}^{n} t_i \le \sum_{i=1}^{n} a_i \quad \text{atas seluruh barisannya, mulai dari } op_1. \tag{14.1}$$

Dengan kata lain, Persamaan (14.1) berkata: **biaya karangan itu, bila dijumlahkan, harus mencapai sekurang-kurangnya sebesar biaya sungguhannya bila dijumlahkan.** Bila kita dapat mengarang sehimpunan biaya amortisasi yang memenuhi ini, dan bila tiap biaya karangan kita kecil, maka kita telah membuktikan sebuah batas atas total sungguhannya: total sungguhannya paling besar sama dengan total karangannya, dan total karangannya kecil.

Pernyataan kedua pada gambar itu dicetak merah, sebab itulah yang sering salah dipahami pemula:

Tidak disyaratkan: $t_i \le a_i$ untuk tiap i.

Kita **tidak** memerlukan tiap biaya amortisasi perorangan untuk menutup biaya sebenarnya operasinya sendiri. Sebuah operasi tunggal boleh menerjang jatah amortisasinya — sebagaimana pasti dilakukan `Multipop` yang membuang sejuta item, ketika biaya amortisasinya 0. Yang harus berlaku hanyalah *total berjalannya*, dan hanya sejak awal barisannya. Sebagian operasi menabung; operasi berikutnya membelanjakan tabungan itu.

DEFINISI. Misalkan sebuah barisan berisi n operasi dilakukan pada sebuah struktur data, dan misalkan c_i (ditulis t_i pada **Gambar 14.1**) adalah **biaya sebenarnya** operasi ke-i. Suatu penetapan bilangan $a_1, a_2, \ldots, a_n$ merupakan himpunan **biaya amortisasi** yang sah bila

$$\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} a_i \qquad \text{untuk setiap } k = 1,2,\ldots,n. \tag{14.2}$$

Bila setiap a_i dibatasi oleh nilai yang sama, $\hat{c}$, kita katakan strukturnya mendukung operasi itu pada **biaya amortisasi** $\hat{c}$, dan biaya total sembarang barisan berisi n operasi paling besar $n\hat{c}$.

Perhatikan bahwa Persamaan (14.2) menuntut pertidaksamaan itu pada **setiap awalan** k, bukan hanya di ujungnya. Ini sedikit lebih kuat daripada pertidaksamaan tunggal Persamaan (14.1) pada $k = n$, dan inilah yang sesungguhnya diberikan metode akuntansi dan metode potensial. Ia penting: artinya Anda boleh menghentikan programnya kapan saja dan batasnya tetap berlaku, dan itu persis yang Anda inginkan bila programnya mungkin disela, atau bila Anda menginginkan batasnya untuk setiap k sekaligus.

14.2.4 Analisis amortisasi bukan analisis kasus rata-rata

Ucapkanlah ini dengan suara keras sekali sebelum membaca lebih jauh: **analisis amortisasi bukan analisis kasus rata-rata.**

Keduanya terdengar mirip. Keduanya menghasilkan sebuah bilangan yang tampak seperti "biaya per operasi secara rata-rata". Keduanya berbeda jenis sama sekali, dan mengacaukannya akan merusak kebenaran argumen Anda — Bab 16 membahas beberapa soal yang jawabannya justru ditentukan oleh perbedaan ini.

Analisis kasus rata-rata mengandaikan sebuah *sebaran peluang* atas masukannya. Ia berkata: bila masukannya diambil secara acak menurut sebaran ini, maka waktu jalannya yang *diharapkan* adalah sekian. Jawabannya adalah sebuah harapan. Quicksort teracak dari Bab 5 adalah contoh bakunya: $O(n\lg n)$-nya adalah harapan atas pemilihan pivot acaknya, dan ada jalan-jalan eksekusi — jarang, tetapi nyata — yang memakan $\Theta(n^2)$.

Analisis amortisasi tidak mengandaikan **apa-apa**. Tidak ada sebaran peluang. Tidak ada keacakan. Tidak ada "masukan yang lazim". Batas yang dibuktikannya adalah batas **kasus terburuk** atas biaya total barisannya, dan ia berlaku untuk *setiap* barisan berisi n operasi — termasuk barisan yang dipilih oleh seorang lawan yang sudah membaca kode sumber Anda khusus untuk menghancurkan Anda.

MITOS YANG SERING DIPERCAYA, DAN YANG HENDAK DILENYAPKAN BAB INI. "$O(1)$ teramortisasi" *tidak* berarti "$O(1)$ pada kebanyakan waktu, dan bila Anda sial ia lambat". Artinya: **tidak ada barisan berisi n operasi yang berbiaya lebih dari $O(n)$ seluruhnya.** Anda boleh sesial yang Anda mau. Anda boleh memilih sendiri barisannya, sambil mengetahui tiap baris implementasinya. Anda tetap tidak dapat membuat totalnya melampaui batasnya. Satu-satunya yang tidak dapat Anda simpulkan adalah bahwa suatu operasi *tertentu* cepat — dan itu keterbatasan yang sungguhan, yang dibahas §14.4 dengan jujur.

§14.4 mengatakan hal ini sekali lagi, sesudah Anda melihat metode agregat bekerja, sebab ia mendarat berbeda pada kali kedua.

14.2.5 Ketiga metodenya

Ada tiga cara baku untuk melakukan analisis amortisasi [1], [3], yang dirangkum pada **Gambar 14.2**.

Tiga cara menemukan biaya amortisasi

1. **Metode agregat.** Batasi biaya total $T(n)$ dari sembarang n operasi; tiap operasi ditagih $T(n)/n$.
2. **Metode akuntansi.** Beri operasi tagihan yang berbeda-beda; tagih sebagian lebih mahal agar membayar di muka, sebagai kredit tersimpan, untuk yang lain.
3. **Metode potensial.** Simpan pembayaran di muka itu sebagai "energi potensial" Φ struktur datanya sendiri: biaya amortisasi = biaya sebenarnya + perubahan potensial.

Gambar 14.2 Ketiga teknik yang umum: **metode agregat** membagi biaya total $T(n)$ dengan n; **metode akuntansi** menagih sebagian operasi terlalu mahal supaya membayar di muka untuk yang lain; **metode potensial** menyimpan pembayaran di muka itu sebagai "energi potensial" strukturnya sendiri.

Satu kalimat untuk masing-masingnya:

1. **Metode agregat.** Hitunglah biaya total $T(n)$ sembarang barisan berisi n operasi secara langsung, dengan sebuah argumen pencacahan. Lalu tiap operasi ditetapkan biaya amortisasi yang sama, $T(n)/n$. Inilah metode yang paling sederhana dan yang pertama patut dicoba, tetapi ia memberi setiap operasi biaya amortisasi yang sama, yang kadang terlalu tumpul.
2. **Metode akuntansi.** Tetapkan tagihannya sendiri bagi tiap *jenis* operasi. Tagihlah lebih besar daripada biaya operasinya, dan surplusnya disimpan sebagai **kredit** pada suatu objek tertentu di dalam strukturnya. Ketika sebuah operasi mahal datang, ia membayar dengan kredit yang disetorkan sebelumnya. Kewajibannya adalah membuktikan bahwa saldo kreditnya tidak pernah menjadi negatif.
3. **Metode potensial.** Gagasan yang sama dengan metode akuntansi, tetapi pembayaran di mukanya tidak ditempelkan pada objek perorangan; ia sebuah bilangan tunggal Φ — **potensialnya** — yang dihitung dari keadaan seluruh strukturnya. Biaya amortisasi didefinisikan sebagai biaya sebenarnya ditambah perubahan potensialnya. Inilah yang paling lentur di antara ketiganya, yang sanggup naik sampai ke struktur yang sukar seperti *Fibonacci heap* [2] dan pohon *splay* [4], dan inilah yang ditunjuk Bab 11.

Ketiganya memberikan jawaban yang sama pada contoh-contoh di dalam bab ini. Bedanya terletak pada berapa banyak kerja yang dituntut masing-masing, dan berapa banyak yang diceritakannya kepada Anda.

14.3 Metode Agregat

14.3.1 Apa itu metodenya

Metode agregat adalah yang paling tidak cerdik di antara ketiganya, dan itu sebuah pujian. Ia terdiri atas tepat dua langkah.

Langkah 1. Tunjukkan bahwa sembarang barisan berisi n operasi memakan waktu $T(n)$ pada kasus terburuk, untuk suatu fungsi T yang Anda hitung secara langsung.

Langkah 2. Nyatakan biaya amortisasi tiap operasi sebagai $T(n)/n$.

Sekian saja. Perhatikan apa yang *tidak* dilakukannya: ia tidak membedakan antar-operasi. Bila sebuah struktur mendukung tiga operasi yang berbeda, metode agregat memberi ketiganya biaya amortisasi yang sama, $T(n)/n$, sekalipun salah satunya jelas lebih murah daripada yang lain. Ketumpulan itulah satu-satunya kelemahan sejati metode ini.

Seluruh kerjanya ada di Langkah 1, dan seluruh seninya terletak pada menemukan argumen pencacahan yang menangkap kebergantungan antar-operasinya — hal yang dibuang oleh batas kasarnya.

14.3.2 Tumpukan dengan `Multipop`

Gambar 14.3 memberikan ketiga operasinya dalam pseudokode.

masing-masing $O(1)$

```
PUSH(S, x)
  top[S] ← top[S] + 1
  S[top[S]] ← x
```

```
POP(S)
  jika STACK-EMPTY(S)
      maka galat "underflow"
      selain itu top[S] ← top[S] − 1
      kembalikan S[top[S] + 1]
```

biaya = **banyaknya putaran gelung** = $\min(s, k)$

```
MULTIPOP(S, k)
  selama tidak STACK-EMPTY(S) dan k ≠ 0
      lakukan POP(S)
          k ← k − 1
```

Gambar 14.3 Operasi-operasi tumpukan: `Push` dan `Pop` masing-masing menyentuh satu slot larik dan satu indeks, sehingga keduanya $O(1)$. `Multipop(S,k)` memanggil `Pop` sampai tumpukannya kosong atau k pop sudah dilakukan, sehingga biayanya adalah banyaknya putaran gelungnya.

Bacalah ketiga prosedurnya baris demi baris, sebab kita akan menghitung biayanya dengan tepat.

`Push(S, x)` melakukan dua hal: ia menaikkan indeks `top[S]` sebanyak satu, dan ia menuliskan x ke slot yang kini ditunjuk indeks itu. Dua langkah, tidak bergantung pada seberapa besar tumpukannya. Biaya: $O(1)$. Kita akan menyebutnya tepat 1 satuan, yang hanya mengubah jawabannya sebesar faktor konstanta.

`Pop(S)` memeriksa apakah tumpukannya kosong; bila ya, ia melaporkan galat *underflow*; bila tidak, ia menurunkan `top[S]` sebanyak satu dan mengembalikan nilai yang tadinya berada di puncak. Lagi-lagi sejumlah langkah yang tetap. Biaya: 1 satuan.

`Multipop(S, k)` berputar selama tumpukannya tidak kosong *dan* $k \neq 0$, memanggil `Pop(S)` dan menurunkan k setiap kali berputar. Jadi banyaknya putaran gelungnya adalah mana yang habis lebih dahulu — itemnya atau jatah k-nya. Bila tumpukannya saat itu memuat s item, gelungnya berputar

$$\min(s, k) \text{ kali,}$$

dan karena tiap putaran adalah satu `Pop`, biaya sebenarnya pemanggilan itu tepat $\min(s, k)$ satuan.

Terus terang saja: bila tumpukannya memuat s elemen maka `Multipop(k)` berkompleksitas waktu $O(\min(s, k))$, **bukan** $O(1)$.

Batas naifnya, sekali lagi

Kita sudah melakukan ini di §14.2.1, tetapi mari kita tuliskan sekali lagi selengkapnya, supaya perbandingannya tepat.

Tumpukannya mulai kosong. Kita melakukan n operasi, masing-masingnya `Push`, `Pop`, atau `Multipop`, dalam urutan apa pun. Sebesar apa s dapat menjadi? Tiap item di tumpukan diletakkan di sana oleh sebuah `Push`, dan operasinya paling banyak n, sehingga push-nya paling banyak n, sehingga $s \leq n$ selalu. Karenanya satu `Multipop` berbiaya paling besar $\min(n, k) \leq n$. Karenanya satu operasi berbiaya $O(n)$. Karenanya n operasi berbiaya $O(n^2)$.

Inilah batas yang diperoleh dari kasus terburuk "untuk satu operasi Multipop tunggal", dan ia memberikan $O(n^2)$ bagi barisannya. Namun, dengan metode agregat, kita dapat menurunkan batas atas yang jauh lebih ketat.

Pengamatan yang memperbaikinya

Inilah pengamatan kuncinya:

Banyaknya operasi `Pop` seluruhnya (termasuk yang dilakukan di dalam `Multipop`) tidak mungkin melampaui banyaknya operasi `Push` seluruhnya — yaitu, paling banyak $O(n)$.

Mari kita buktikan itu dengan hati-hati, sebab ia jantung persoalannya dan ia jenis argumen yang akan perlu Anda tirukan.

Klaim. Di dalam sembarang barisan operasi yang bermula dari tumpukan kosong, banyaknya kali sebuah item dibuang dari tumpukannya paling besar sama dengan banyaknya kali sebuah item diletakkan padanya.

Bukti. Tengoklah sembarang item tunggal — sebutlah x. Bagaimana x sampai ke tumpukannya? Hanya ada satu cara: sebuah `Push` meletakkannya di sana. Bagaimana x dapat meninggalkan tumpukannya? Persis ada dua cara: sebuah `Pop`, atau satu putaran gelung di dalam sebuah `Multipop` — dan badan gelung itu *adalah* sebuah `Pop`. Jadi setiap pembuangan x adalah satu `Pop`. Dapatkah x dibuang dua kali? Tidak: sekali dibuang, x tidak lagi berada di tumpukannya, dan tidak ada yang pernah mengembalikannya ke sana kecuali sebuah `Push` lain, yang berarti push yang berbeda. Jadi:

- tiap item di-push tepat sekali oleh push yang menciptakannya, dan
- tiap item di-pop **paling banyak sekali** sesudah push itu — paling banyak, sebab barisannya boleh berakhir dengan x masih duduk di tumpukannya.

Karena itu pembuangannya berpadanan satu-satu dengan suatu upahimpunan push-nya, dan

$$(\text{banyaknya pembuangan}) \leq (\text{banyaknya push}). \qquad \blacksquare$$

> **"TIAP ELEMEN DI-POP PALING BANYAK SEKALI SESUDAH IA DI-PUSH."** Satu kalimat itu adalah seluruh analisis agregat tumpukannya. Ia layak dihafal dalam bentuk itu, sebab kalimat

> yang sama — dengan kata benda yang berbeda — membuktikan batas amortisasi bagi belasan struktur lain. Bentuk umumnya adalah: *temukan sebuah satuan kerja yang dapat ditagihkan kepada sebuah objek, lalu tunjukkan bahwa tiap objek hanya dapat ditagih sebanyak jumlah yang terbatas sepanjang seluruh barisannya.*

Totalnya, dihitung

Sekarang aritmetikanya. Misalkan barisannya memuat p push dan q pembuangan, dengan pembuangan berarti satu item meninggalkan tumpukannya (entah lewat sebuah `Pop` atau di dalam sebuah `Multipop`). Tiap dari n operasinya adalah `Push`, `Pop`, atau `Multipop`, sehingga $p \leq n$. Klaimnya memberikan $q \leq p \leq n$.

Biaya sebenarnya seluruhnya adalah

$$T(n) \;=\; \underbrace{p \times 1}_{\text{push-nya}} \;+\; \underbrace{q \times 1}_{\text{pembuangannya}} \;\leq\; n + n \;=\; 2n.$$

(Pembukuan di dalam `Multipop` — menguji syarat gelungnya, menurunkan k — menambahkan sebuah konstanta per putaran ditambah satu untuk pengujian terakhirnya, yang mengubah $2n$ menjadi cn untuk suatu konstanta kecil c. Kita bekerja dalam notasi O, jadi ini tidak penting; tetapi sadarilah bahwa "2" adalah angka ideal yang bersih, bukan cacah instruksi harfiah.)

Jadi $T(n) = O(n)$ untuk sembarang barisan berisi n operasi, dan menurut Langkah 2 metode agregat, biaya amortisasi per operasinya adalah

$$\frac{T(n)}{n} \;\leq\; \frac{2n}{n} \;=\; 2 \;=\; O(1).$$

Ketiga operasinya — `Push`, `Pop`, dan `Multipop` — berbiaya amortisasi $O(1)$. Termasuk `Multipop`, yang biaya sebenarnya dapat $\Theta(n)$. Itu bukan pertentangan; itu definisinya sedang bekerja.

Mengukurnya

Klaim tentang waktu jalan patut diuji dengan program yang sungguh-sungguh dijalankan. **Kode 14.1** adalah sebuah tumpukan dengan ketiga operasinya, diinstrumentasi sehingga tiap item yang meninggalkan tumpukannya ikut dihitung, lalu dijalankan pada tiga barisan yang berbeda: barisan si lawan, barisan yang murah secara sepele, dan campuran pseudo-acak.

Kode 14.1 Sebuah tumpukan dengan `Multipop`, diinstrumentasi. Kompilasi dengan `g++ -std=c++17 -O2 -o multipop multipop.cpp`.

```
// Sebuah tumpukan dengan PUSH, POP dan MULTIPOP, diinstrumentasi sehingga
// setiap slot larik yang tersentuh ikut dihitung.  Program ini lalu melaporkan
// biaya amortisasi terukur -- total kerja dibagi banyaknya
// operasi -- untuk tiga barisan yang berbeda.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

class CountingStack {
public:
    long long work = 0;        // tulis/baca slot dasar, yaitu biaya sebenarnya
    long long nPush = 0, nPop = 0, nMultipop = 0, nOps = 0;

    void push(int x) { data.push_back(x); ++work; ++nPush; ++nOps; }

    // Satu POP sebagai operasi tersendiri.
    void pop() { ++nPop; ++nOps; popOne(); }

    // MULTIPOP(S,k): buang min(s,k) item teratas.  Dihitung sebagai SATU
    // operasi, tetapi biaya sebenarnya adalah banyaknya item yang dibuangnya.
    int multipop(int k) {
        ++nMultipop; ++nOps;
        int removed = 0;
        while (!data.empty() && k != 0) { popOne(); --k; ++removed; }
        return removed;
    }

    size_t size() const { return data.size(); }
    bool empty() const { return data.empty(); }

private:
    vector<int> data;
    // Satu pembuangan dasar.  MULTIPOP dibangun dari ini, sehingga
    // `work` menghitung tepat satu satuan per item yang meninggalkan tumpukan.
    void popOne() { if (!data.empty()) { data.pop_back(); ++work; } }
};
```

Kode 14.1 (Lanjutan) *Sebuah tumpukan dengan Multipop, diinstrumentasi. Kompilasi dengan g++ -std=c++17 -O2 -o multipop multipop.cpp.*

```
void report(const string& name, const CountingStack& S) {
    cout << left << setw(30) << name
         << " ops=" << setw(8) << S.nOps
         << " (push " << setw(7) << S.nPush
         << " pop " << setw(7) << S.nPop
         << " multipop " << setw(7) << S.nMultipop << ")"
         << "  actual cost=" << setw(9) << S.work
         << "  cost/op=" << fixed << setprecision(6)
         << (double)S.work / (double)S.nOps << "\n";
}

int main() {
    const int n = 100000;

    // Barisan A: kesukaan si lawan.  Dorong n/2 item, lalu minta
    // MULTIPOP mengosongkan seluruh tumpukan dalam satu operasi.  Ulangi.
    {
        CountingStack S;
        while (S.nOps + 1 < n) {
            for (int i = 0; i < n / 2 && S.nOps + 1 < n; ++i) S.push(i);
            S.multipop(n);                        // satu operasi yang sangat mahal
        }
        report("A: fill, then MULTIPOP(n)", S);
    }

    // Barisan B: satu push, satu multipop, terus-menerus -- tiap MULTIPOP murah.
    {
        CountingStack S;
        while (S.nOps + 2 <= n) { S.push(1); S.multipop(n); }
        report("B: push/MULTIPOP alternating", S);
    }

    // Barisan C: campuran pseudo-acak, deterministik agar dapat diulang.
    {
        CountingStack S;
        unsigned long long seed = 12345;
        auto rnd = [&seed]() { seed = seed * 6364136223846793005ULL +
1442695040888963407ULL;
                                return (unsigned)(seed >> 33); };
        while (S.nOps < n) {
            unsigned r = rnd() % 100;
            if (r < 60) S.push((int)r);
            else if (r < 90) { if (!S.empty()) S.pop(); else S.push(0); }
            else S.multipop((int)(rnd() % 1000));
        }
        report("C: random 60/30/10 mix", S);
    }
```

Kode 14.1 (Lanjutan) *Sebuah tumpukan dengan Multipop, diinstrumentasi. Kompilasi dengan g++ -std=c++17 -O2 -o multipop multipop.cpp.*

```
    cout << "\nThe bound proved in the text is: actual cost < 2n, i.e. cost/op < 2.\n";
    cout << "With n = " << n << " that ceiling is " << 2 * n << ".\n";
    return 0;
}
```

Menjalankannya mencetak:

```
A: fill, then MULTIPOP(n)      ops=100000   (push 99998   pop 0       multipop 2      )
actual cost=199996     cost/op=1.999960
B: push/MULTIPOP alternating   ops=100000   (push 50000   pop 0       multipop 50000  )
actual cost=100000     cost/op=1.000000
C: random 60/30/10 mix         ops=100000   (push 64514   pop 25192   multipop 10294  )
actual cost=129027     cost/op=1.290270

The bound proved in the text is: actual cost < 2n, i.e. cost/op < 2.
With n = 100000 that ceiling is 200000.
```

Tengoklah barisan A. Ia melakukan push sebanyak yang ia bisa lalu mengosongkan tumpukannya dalam satu `Multipop` raksasa; operasi tunggal itu membuang hampir 50.000 item. Totalnya 199,996 melawan langit-langit 200,000 — si lawan telah mendorong strukturnya sampai empat satuan dari batas yang dibuktikan dan tidak dapat maju lagi. Barisan C, campuran yang kelihatan biasa saja, mendarat di 1.29 per operasi. Tidak sekali pun pengukurannya mencapai 2, dan tidak sekali pun ia mendekati $n = 100{,}000$, yang adalah yang diizinkan oleh batas $O(n^2)$.

14.3.3 Pencacah biner k-bit

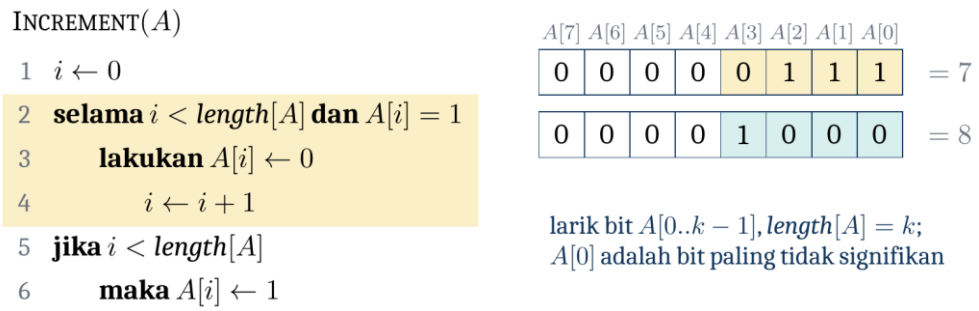
Contoh kedua inilah yang patut Anda simpan di kepala, sebab ia berulang di seluruh ilmu komputer: sebuah pencacah biner yang Anda naikkan mulai dari nol.

Strukturnya adalah sebuah larik $A[0 \ldots k-1]$ berisi k bit, dengan `length[A]` $=k$. Bit $A[0]$ adalah yang **paling tidak berarti** — tempat satuan — dan $A[k-1]$ yang paling berarti. Bilangan yang diwakili pencacahnya saat ini adalah

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i.$$

Jadi bila $k = 8$ dan bitnya, dari $A[7]$ turun ke $A[0]$, berbunyi 0 0 0 0 1 0 1 1, nilainya adalah $8 + 2 + 1 = 11$.

Untuk menambahkan satu ke pencacahnya kita melakukan apa yang Anda lakukan di atas kertas dalam basis sepuluh ketika Anda menambahkan 1 kepada 1999: ubahlah sembilan-sembilan di ekornya menjadi nol lalu naikkan yang bukan-sembilan pertama. Dalam basis dua, "sembilan"-nya adalah satu. Prosedur `Increment` adalah **Gambar 14.4**.



Gambar 14.4 `Increment(A)`: baris 2–4 berjalan naik dari $A[0]$ sambil mengubah tiap 1 di ekornya menjadi 0, dan baris 6 menyetel 0 pertama yang dicapainya. Bila setiap bitnya 1, pencacahnya diam-diam berputar kembali ke nol, persis seperti register mesin k-bit.

Telusurilah sekali dengan tangan pada $k = 4$, bermula dari $A = 0\,1\,1\,1$ (nilai 7, ditulis dari yang paling berarti lebih dahulu).

- $i = 0$. Apakah $i < 4$ dan $A[0] = 1$? Ya. Maka $A[0] \leftarrow 0$ dan $i \leftarrow 1$. Lariknya: 0 1 1 0.

- $i = 1$. Apakah $A[1] = 1$? Ya. Maka $A[1] \leftarrow 0$, $i \leftarrow 2$. Lariknya: 0 1 0 0.
- $i = 2$. Apakah $A[2] = 1$? Ya. Maka $A[2] \leftarrow 0$, $i \leftarrow 3$. Lariknya: 0 0 0 0.
- $i = 3$. Apakah $A[3] = 1$? Tidak, ia 0. Keluar dari gelungnya.
- Baris 5: apakah $3 < 4$? Ya. Baris 6: $A[3] \leftarrow 1$. Lariknya: 1 0 0 0.

Nilainya beranjak dari 7 ke 8, dan empat bit ditulisi. Secara umum, **biaya satu `Increment` adalah banyaknya bit yang diubahnya**: satu tulis untuk tiap 1 di ekor yang dinolkan, ditambah satu tulis untuk 0 yang disetel.

Tabel 8-bit

Gambar 14.5 memperlihatkan sebuah pencacah 8-bit yang digerakkan dari 0 naik sampai 16, lengkap dengan pola bitnya pada tiap langkah dan total biaya berjalannya.

Nilai pencacah	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Biaya total
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Gambar 14.5 Pencacah 8-bit sesudah tiap satu dari 16 increment; sel yang diarsir pada sebuah baris adalah bit yang dibalik oleh increment berikutnya, sehingga banyaknya adalah biaya sebenarnya increment itu. Total berjalannya sesudah 16 increment adalah 31, di bawah $2 \times 16 = 32$.

Berjalanlah menyusuri lajur arsirannya dengan jari Anda dan dua hal langsung mencuat.

Pertama, $A[0]$ terarsir pada *setiap* baris. Tentu saja: menambahkan 1 selalu mengubah bit satuannya. Jadi $A[0]$ berbalik n kali dalam n increment.

Kedua, baris-baris yang lebar — yang punya banyak sel terarsir — jarang muncul, dan semakin lebar semakin jarang. Beranjak dari 7 ke 8 mengarsir empat sel. Beranjak dari 15 ke 16 mengarsir lima. Tetapi itulah satu-satunya baris selebar itu dalam seluruh tabelnya. Di antara keduanya duduk tujuh baris, dan empat di antaranya hanya mengarsir satu sel.

Itulah kebergantungan yang diabaikan batas kasarnya, dalam wujud gambar. Batas kasarnya berkata: sebuah `Increment` dapat membalik sampai k bit, ada n buah, jadi totalnya $O(nk)$. Koreksinya cukup satu pengamatan: *tidak semua bit berubah setiap kali prosedur `Increment` dipanggil!*

Seberapa sering bit i berbalik?

Inilah argumen pencacahannya. Ambillah pelan-pelan.

Bit $A[0]$ berubah pada **setiap** increment. Dalam n increment ia berbalik n kali.

Bit $A[1]$ berubah hanya ketika $A[0]$ tadinya 1 dan harus dinolkan, yang terjadi pada setiap increment *kedua*: beranjak $1 \to 2$, $3 \to 4$, $5 \to 6$, dan seterusnya. Dalam n increment ia berbalik $\lfloor n/2 \rfloor$ kali.

Bit $A[2]$ berubah hanya ketika $A[0]$ dan $A[1]$ keduanya 1 — yaitu, pada setiap increment *keempat*: $3 \to 4$, $7 \to 8$, $11 \to 12$. Dalam n increment ia berbalik $\lfloor n/4 \rfloor$ kali.

Polanya jelas, dan dapat dinyatakan sebagai: *Secara umum, untuk* $i = 0,1, \ldots, \lfloor \log_2 n \rfloor$, *bit* $A[i]$ *akan berubah* $\lfloor n/2^i \rfloor$ *kali sepanjang* n *operasi.* Dan untuk $i > \lfloor \log_2 n \rfloor$ bitnya tidak pernah berubah sama sekali, sebab pencacahnya tidak pernah sebesar itu.

Mari kita lihat mengapa $\lfloor n/2^i \rfloor$ itu benar. Bit $A[i]$ berubah persis ketika increment-nya membawa simpanan sampai ke posisi i, yang terjadi persis ketika nilai yang sedang dinaikkan memiliki i bit rendahnya semua sama dengan 1 — yaitu, ketika nilainya kongruen dengan $2^i - 1$ modulo 2^i. Di antara n nilai $0,1, \ldots, n-1$ yang kita naikkan, banyaknya nilai semacam itu adalah $\lfloor n/2^i \rfloor$. (Satu dari tiap blok berisi 2^i bilangan bulat berurutan berbentuk demikian, dan ada $\lfloor n/2^i \rfloor$ blok utuh yang termuat di dalam n bilangan bulat pertamanya, sedangkan blok sebagian di ujungnya tidak menyumbang tambahan apa pun bagi lantainya.)

Menjumlahkannya

Banyaknya balikan bit seluruhnya sepanjang n increment adalah jumlah, atas semua bitnya, dari seberapa sering masing-masingnya berbalik:

$$T(n) = \sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor. \tag{14.3}$$

(Kita menulis lg untuk $\log_2$, sebagaimana ditetapkan Bab 3.)

Gambar 14.6 membatasi jumlah Persamaan (14.3) dalam dua baris.

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Gambar 14.6 Analisis agregat pencacah dalam satu tampilan: buang lantainya, perpanjang jumlahnya sampai tak hingga, keluarkan n, lalu jumlahkan deret geometrinya menjadi 2.

Inilah tampilan itu, satu langkah per baris, dengan alasan tiap langkahnya.

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor \le \sum_{i=0}^{\lfloor \lg n \rfloor} \frac{n}{2^i}$$

Mengapa: $\lfloor x \rfloor \le x$ untuk setiap bilangan real x. Membuang lantainya hanya dapat memperbesar tiap sukunya.

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \frac{n}{2^i} < \sum_{i=0}^{\infty} \frac{n}{2^i}$$

Mengapa: tiap sukunya positif, sehingga memperpanjang jumlahnya sampai tak hingga menambahkan lebih banyak suku positif dan menaikkan nilainya secara tegas.

$$\sum_{i=0}^{\infty} \frac{n}{2^i} = n \sum_{i=0}^{\infty} \frac{1}{2^i}$$

Mengapa: n tidak bergantung pada i, sehingga ia dapat ditarik keluar ke depan jumlahnya.

$$n \sum_{i=0}^{\infty} \frac{1}{2^i} = n \cdot 2 = 2n.$$

Mengapa: ini deret geometri tak hingga $1 + 1/2 + 1/4 + 1/8 + \cdots$. Bila Anda belum pernah menjumpainya, berikut buktinya dalam satu baris. Misalkan $S = 1 + 1/2 + 1/4 + \cdots$. Maka $1/2\, S = 1/2 + 1/4 + 1/8 + \cdots$, yaitu S dengan suku pertamanya dibuang, sehingga $1/2\, S = S - 1$. Dengan menyelesaikannya, $1/2\, S = 1$, karenanya $S = 2$.

Jadi

$$T(n) < 2n,$$

dan metode agregat memberikan biaya amortisasi per `Increment` sebesar

$$\frac{T(n)}{n} < \frac{2n}{n} = 2 = O(1).$$

> **DENGAN BAHASA SEHARI-HARI.** Sebuah increment dapat membalik sebanyak k bit, sehingga satu increment tunggal dapat lambat. Tetapi untuk membalik k bit Anda harus lebih dahulu menyetel k bit itu, satu per satu, sepanjang $2^k - 1$ increment sebelumnya. Increment yang mahal persis adalah yang sudah dibayari oleh rentetan panjang increment yang murah. Dirata-ratakan atas barisannya, kurang dari dua bit berbalik per increment — sebesar apa pun k.

Perhatikan satu hal lagi: batas $2n$ sama sekali tidak menyebut k. Pencacah 64-bit dan pencacah 8-bit berbiaya sama per increment secara rata-rata. Inilah sebabnya menaikkan pencacah gelung diperlakukan sebagai operasi berwaktu konstan dalam model RAM Bab 2, sekalipun sebuah simpanan pada prinsipnya dapat beriak menembus seluruh register.

Mengukurnya

Kode 14.2 mengimplementasikan `Increment` persis seperti yang ditulis **Gambar 14.4**, memasang pencacah balikan pada tiap bit, dan mula-mula menghasilkan ulang tabel 8-bit **Gambar 14.5** — lajur "Total Cost" yang sama, dihitung alih-alih disalin.

Kode 14.2 Pencacah biner k-bit dengan pencacah balikan per bit. Kompilasi dengan `g++ -std=c++17 -O2 -o counter counter.cpp`.

```
// Pencacah biner k-bit, diimplementasikan persis seperti pseudokode
// INCREMENT, dengan pencacah balik terpasang pada tiap bit.  Program ini
// mula-mula menghasilkan ulang tabel 8-bit, lalu mengukur biaya
// amortisasinya pada jalan panjang dan mencocokkannya dengan batas di teks.
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

class BitCounter {
public:
    explicit BitCounter(int k) : A(k, 0), flips(k, 0) {}

    // INCREMENT(A): mengembalikan biaya sebenarnya -- banyaknya bit yang berbalik.
    long long increment() {
        long long cost = 0;
        size_t i = 0;
        while (i < A.size() && A[i] == 1) {   // nolkan tiap 1 di ekornya
            A[i] = 0; ++flips[i]; ++cost; ++i;
        }
        if (i < A.size()) {                   // setel 0 pertama yang dicapai
            A[i] = 1; ++flips[i]; ++cost;
        }
        total += cost; ++nOps;
        return cost;
    }

    unsigned long long value() const {        // hanya sahih selama k <= 63
        unsigned long long v = 0;
        for (size_t i = A.size(); i-- > 0; ) v = 2 * v + A[i];
        return v;
    }
    int bit(int i) const { return A[i]; }
    long long flipsOf(int i) const { return flips[i]; }
    long long actualCost() const { return total; }
    long long ops() const { return nOps; }
    size_t width() const { return A.size(); }

private:
    vector<int> A;            // larik bit A[0..k-1]; A[0] paling tidak berarti
    vector<long long> flips;
    long long total = 0, nOps = 0;
};
```

Kode 14.2 (Lanjutan) *Pencacah biner k-bit dengan pencacah balikan per bit. Kompilasi dengan g++ -std=c++17 -O2 -o counter counter.cpp.*

```
int main() {
    // ---- Bagian 1: tabel 8-bit, kolom A[7]..A[0] dan Total Cost.
    cout << "Counter";
    for (int i = 7; i >= 0; --i) cout << "  A[" << i << "]";
    cout << "   Total Cost\n";
    BitCounter C(8);
    cout << setw(5) << 0 << "  ";
    for (int i = 7; i >= 0; --i) cout << setw(5) << C.bit(i) << " ";
    cout << setw(9) << 0 << "\n";
    for (int v = 1; v <= 16; ++v) {
        C.increment();
        cout << setw(5) << v << "  ";
        for (int i = 7; i >= 0; --i) cout << setw(5) << C.bit(i) << " ";
        cout << setw(9) << C.actualCost() << "\n";
    }

    // ---- Bagian 2: seberapa sering tiap bit berbalik, lawan floor(n/2^i).
    const long long n = 1000000;
    BitCounter D(32);
    for (long long t = 0; t < n; ++t) D.increment();
    cout << "\nAfter n = " << n << " increments on a 32-bit counter:\n";
    cout << " bit i   measured flips   predicted floor(n/2^i)\n";
    for (int i = 0; i < 24; ++i)
        cout << setw(6) << i << setw(17) << D.flipsOf(i)
             << setw(25) << (n >> i) << "\n";
    cout << "\ntotal actual cost = " << D.actualCost()
         << "   ceiling 2n = " << 2 * n << "\n";
    cout << "measured amortized cost per INCREMENT = " << fixed << setprecision(6)
         << (double)D.actualCost() / (double)D.ops() << "\n";
    return 0;
}
```

Separuh pertama keluarannya adalah tabel **Gambar 14.5**, dihitung ulang:

```
Counter  A[7]  A[6]  A[5]  A[4]  A[3]  A[2]  A[1]  A[0]   Total Cost
    0      0     0     0     0     0     0     0     0         0
    1      0     0     0     0     0     0     0     1         1
    2      0     0     0     0     0     0     1     0         3
    3      0     0     0     0     0     0     1     1         4
    4      0     0     0     0     0     1     0     0         7
    5      0     0     0     0     0     1     0     1         8
    6      0     0     0     0     0     1     1     0        10
    7      0     0     0     0     0     1     1     1        11
    8      0     0     0     0     1     0     0     0        15
    9      0     0     0     0     1     0     0     1        16
   10      0     0     0     0     1     0     1     0        18
   11      0     0     0     0     1     0     1     1        19
   12      0     0     0     0     1     1     0     0        22
   13      0     0     0     0     1     1     0     1        23
   14      0     0     0     0     1     1     1     0        25
   15      0     0     0     0     1     1     1     1        26
   16      0     0     0     1     0     0     0     0        31
```

Bandingkan lajur Total Cost itu dengan **Gambar 14.5**: 0,1,3,4,7,8,10,11,15,16,18,19,22,23,25,26,31. Sama persis. Tabel yang digambar tangan itu tepat sekali, dan itu hal yang menyenangkan untuk dapat dipastikan.

Separuh keduanya menguji rumus balikannya terhadap kenyataan pada sejuta increment:

```
After n = 1000000 increments on a 32-bit counter:
 bit i   measured flips   predicted floor(n/2^i)
     0          1000000                  1000000
     1           500000                   500000
     2           250000                   250000
     3           125000                   125000
     4            62500                    62500
     5            31250                    31250
     6            15625                    15625
     7             7812                     7812
     8             3906                     3906
     9             1953                     1953
    10              976                      976
    11              488                      488
    12              244                      244
    13              122                      122
    14               61                       61
    15               30                       30
    16               15                       15
    17                7                        7
    18                3                        3
    19                1                        1
    20                0                        0
    21                0                        0
    22                0                        0
    23                0                        0

total actual cost = 1999993   ceiling 2n = 2000000
measured amortized cost per INCREMENT = 1.999993
```

Tiap cacah balikan yang terukur sama persis dengan $\lfloor n/2^i \rfloor$ — termasuk baris-baris mulai $i = 20$, ketika pencacahnya memang tidak pernah setinggi itu dan bitnya tidak pernah bergerak. Totalnya, 1,999,993, berada tujuh satuan di bawah langit-langit $2n = 2{,}000{,}000$, sehingga batas $T(n) < 2n$ sekaligus benar dan ketat sampai selisih pembulatan. Perhatikan pula batas *kasar*nya sebagai pembanding: $O(nk) = 10^6 \times 32 = 3.2 \times 10^7$, enam belas kali lebih besar daripada kenyataannya.

14.3.4 Soal 17.1-3, dikerjakan tuntas

Soal ini adalah latihan baku bagi metode agregat, dan ia sepadan dengan setiap menit yang Anda berikan.

> **SOAL 17.1-3.** Sebuah barisan berisi n operasi dilakukan pada sebuah struktur data. Biaya operasi ke-i adalah i bila i tepat sebuah pangkat 2, dan 1 bila bukan. Gunakan metode agregat untuk menentukan biaya amortisasi per operasinya.

Sebuah tabel kecil, **Gambar 14.7**, membuat polanya nyata sebelum aljabarnya dimulai.

- Sebuah barisan n operasi dijalankan pada sebuah struktur data.
- Biaya operasi ke-i adalah i bila i tepat sebuah pangkat 2, dan 1 bila bukan.
- Gunakan **metode agregat** untuk menentukan biaya amortisasi per operasi.

Operasi i	**Biaya** c_i
1	$1 = 2^0$
2	$2 = 2^1$
3	1
4	$4 = 2^2$
5	1
Total	**9**

Gambar 14.7 Lima operasi pertama Soal 17.1-3 berbiaya 1,2,1,4,1: pangkat dua 1,2,4 menagih indeksnya sendiri dan operasi lain menagih 1. Totalnya 9, atau 1.8 per operasi.

Memahami fungsi biayanya

Tidak ada struktur data yang dijelaskan, dan memang tidak diperlukan: soalnya menyerahkan biaya tiap operasi kepada kita secara langsung. Tulislah c_i untuk biaya operasi i. Maka

$$c_i = \begin{cases} i & \text{bila } i = 2^j \text{ untuk suatu bilangan bulat } j \geq 0, \\ 1 & \text{bila tidak.} \end{cases} \tag{14.4}$$

Pada Persamaan (14.4), $c_1 = 1$ (dan $1 = 2^0$, sehingga ini pangkat 2 dan ia menagih $i = 1$, yang kebetulan memang 1), $c_2 = 2$, $c_3 = 1$, $c_4 = 4$, $c_5 = c_6 = c_7 = 1$, $c_8 = 8$, dan seterusnya. Ini karikatur sebuah struktur sungguhan — tabel berlipat dua dari §14.7 berperilaku hampir persis seperti ini — dan karikaturnya disengaja: ia mengasingkan polanya.

Biaya kasus terburuk satu operasi *tunggal* adalah n (ketika $i = n$ dan n sebuah pangkat 2), sehingga batas kasarnya berkata $O(n^2)$ lagi. Mari kita berbuat lebih baik.

Memecah jumlahnya

Biaya total n operasi adalah $\sum_{i=1}^{n} c_i$. Akal-akalannya — dan ia akal-akalan yang baku — adalah memecah operasinya menjadi yang mahal dan yang murah, lalu mencacah tiap kelompoknya terpisah.

Bagian murahnya. *Setiap* operasi, mahal atau tidak, ditagih sekurang-kurangnya 1. Ada n operasi. Jadi dasar sebesar n dibayar oleh barisannya, apa pun yang terjadi. Tulislah itu sebagai $\sum_{i=1}^{n} 1 = n$.

Bagian mahalnya. Operasi yang merupakan pangkat dua membayar *tambahan*. Operasi $i = 2^j$ membayar 2^j seluruhnya, yang 1 di antaranya sudah dicacah pada dasarnya, sehingga tambahannya adalah $2^j - 1$.

Pangkat dua mana saja yang muncul? Yaitu 2^j dengan $2^j \leq n$. Dengan mengambil lg kedua ruasnya, $j \leq \lg n$, dan karena j bilangan bulat, $j \leq \lfloor \lg n \rfloor$. Jadi j berjalan atas $0, 1, 2, \ldots, \lfloor \lg n \rfloor$.

Menyatukan kedua bagiannya memberikan baris pertama penyelesaian pada **Gambar 14.8**:

$$\text{Total} = \sum_{j=0}^{\lfloor \lg n \rfloor} (2^j - 1) + \sum_{i=1}^{n} 1. \tag{14.5}$$

$$\begin{aligned}
\textit{Total} &= \sum_{j=0}^{\lfloor \lg n \rfloor} (2^j - 1) + \sum_{i=1}^{n} 1 && \text{pangkat dua membayar } 2^j\text{, bukan 1} \\
&= \left(2^{\lfloor \lg n \rfloor + 1} - 1\right) - \left(\lfloor \lg n \rfloor + 1\right) + n && \text{deret geometri: } \sum_{j=0}^{m} x^j = \frac{x^{m+1} - 1}{x - 1} \\
&= 2^{\lfloor \lg n \rfloor + 1} + n - \lfloor \lg n \rfloor - 2 \\
&\leq 2n + n - \lfloor \lg n \rfloor - 2 && \text{sebab } 2^{\lfloor \lg n \rfloor} \leq 2^{\lg n} = n \\
&= 3n - \lfloor \lg n \rfloor - 2.
\end{aligned}$$

Biaya total n operasi adalah $O(n)$, sehingga biaya amortisasi per operasi adalah $O(1)$.

Gambar 14.8 Penyelesaian lengkap Soal 17.1-3, dari jumlah yang dipecah sampai Total $\leq 3n - \lfloor \lg n \rfloor - 2$, lengkap dengan rumus deret geometri dan kesamaan logaritma yang diperlukannya.

Menghitungnya, satu baris demi satu baris

Penyelesaian itu memakai bentuk tertutup deret geometri berhingga, yang sebaiknya kita buktikan alih-alih kita terima begitu saja. Untuk $x \neq 1$,

$$\sum_{k=0}^{m} x^k = \frac{x^{m+1} - 1}{x - 1}. \tag{14.6}$$

Bukti. Misalkan $S = 1 + x + x^2 + \cdots + x^m$. Kalikan dengan x: $xS = x + x^2 + \cdots + x^{m+1}$. Kurangkan: $xS - S = x^{m+1} - 1$, sebab tiap suku kecuali yang terakhir pada xS dan yang pertama pada S saling meniadakan. Jadi $S(x - 1) = x^{m+1} - 1$, dan membaginya dengan $x - 1$ memberikan rumusnya. ■

Dengan $x = 2$ dan $m = \lfloor \lg n \rfloor$, Persamaan (14.6) memberikan

$$\sum_{j=0}^{\lfloor \lg n \rfloor} 2^j = \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2 - 1} = 2^{\lfloor \lg n \rfloor + 1} - 1.$$

Angka -1 di dalam kurung pada Persamaan (14.5) dikurangkan sekali untuk tiap nilai j, dan j mengambil $\lfloor \lg n \rfloor + 1$ nilai (terhitung $j = 0$). Jadi

$$\sum_{j=0}^{\lfloor \lg n \rfloor} (2^j - 1) = (2^{\lfloor \lg n \rfloor + 1} - 1) - (\lfloor \lg n \rfloor + 1).$$

Dan dasarnya adalah $\sum_{i=1}^{n} 1 = n$. Jadi

$$\text{Total} = 2^{\lfloor \lg n \rfloor + 1} - 1 - (\lfloor \lg n \rfloor + 1) + n,$$

yang merupakan baris kedua **Gambar 14.8**. Dengan merapikan kedua konstantanya, $-1 - 1 = -2$:

$$\text{Total} = 2^{\lfloor \lg n \rfloor + 1} + n - \lfloor \lg n \rfloor - 2. \tag{14.7}$$

Itulah baris ketiganya, dan Persamaan (14.7) adalah rumus yang **eksak** — bukan sebuah batas. Kita akan mengujinya secara numerik sebentar lagi.

Mengubahnya menjadi sebuah batas

Satu-satunya suku yang canggung adalah $2^{\lfloor \lg n \rfloor + 1}$. Membuang lantainya hanya dapat memperbesar pangkatnya:

$$2^{\lfloor \lg n \rfloor + 1} \leq 2^{\lg n + 1}.$$

Kesamaan umum $a^{\log_b n} = n^{\log_b a}$, yang dicatat pada **Gambar 14.8**, dapat menanganinya. Dalam kasus kita ia lebih sederhana daripada itu: $2^{\lg n + 1} = 2^{\lg n} \cdot 2^1 = n \cdot 2 = 2n$, sebab $2^{\lg n} = n$ menurut definisi lg. Jadi

$$\text{Total} \leq 2n + n - \lfloor \lg n \rfloor - 2 = 3n - \lfloor \lg n \rfloor - 2,$$

baris terakhir penyelesaiannya. Karena $\lfloor \lg n \rfloor \geq 0$ untuk $n \geq 1$, kita boleh membuangnya juga lalu menyimpulkan

$$\text{Total} \;\leq\; 3n - 2 \;=\; O(n).$$

Menurut Langkah 2 metode agregat, biaya amortisasi per operasinya adalah

$$\frac{\text{Total}}{n} \;\leq\; \frac{3n-2}{n} \;<\; 3 \;=\; O(1),$$

dan kesimpulannya berbunyi: *biaya total untuk n operasi adalah $O(n)$; biaya amortisasi per operasi yang dihasilkannya adalah $O(1)$.*

> **MENGAPA 3 DAN BUKAN SESUATU YANG LEBIH BURUK.** Setiap operasi membayar 1. Yang mahal membayar tambahan, tetapi tambahannya adalah 1,2,4,8, … sampai paling besar n, dan deret itu **berlipat dua**, sehingga jumlahnya didominasi oleh suku terakhirnya dan berjumlah kurang dari $2n$. Satu satuan masing-masing untuk n operasi, ditambah paling besar $2n$ untuk seluruh tambahannya, memberikan $3n$. Setiap kali Anda melihat biaya yang berlipat dua, harapkanlah totalnya kira-kira dua kali yang terbesar, dan harapkanlah biaya amortisasi yang konstan.

Memeriksanya dengan sebuah program

Rumus yang eksak layak diuji secara eksak. **Kode 14.3** menjumlahkan biayanya secara langsung lalu membandingkan total berjalannya dengan bentuk tertutupnya sekaligus dengan batasnya.

Kode 14.3 Soal 17.1-3 diuji secara numerik terhadap bentuk tertutupnya dan batasnya. Kompilasi dengan `g++ -std=c++17 -O2 -o ex1713 ex1713.cpp`.

```
// Soal 17.1-3.  Operasi ke-i berbiaya i bila i tepat sebuah pangkat
// dari 2, dan 1 bila bukan.  Program ini menjumlahkan biayanya langsung lalu
// membandingkan total berjalannya dengan bentuk tertutup yang diturunkan di teks,
//     Total(n) = 2^(floor(lg n)+1) + n - floor(lg n) - 2,
// dan dengan batas  Total(n) <= 3n - floor(lg n) - 2.
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

bool isPowerOfTwo(long long i) { return i > 0 && (i & (i - 1)) == 0; }

long long cost(long long i) { return isPowerOfTwo(i) ? i : 1; }

int floorLg(long long n) {                  // floor(log2 n) untuk n >= 1
    int k = 0;
    while ((1LL << (k + 1)) <= n) ++k;
    return k;
}

long long closedForm(long long n) {
    int k = floorLg(n);
    return (1LL << (k + 1)) + n - k - 2;
}

int main() {
    cout << "   n   sum of costs   closed form   3n-floor(lg n)-2   amortized\n";
    long long running = 0;
    long long checkpoints[] = {1, 2, 3, 4, 5, 8, 16, 100, 1000, 1024, 100000, 1000000};
    int next = 0;
    for (long long i = 1; i <= 1000000; ++i) {
        running += cost(i);
        if (next < 12 && i == checkpoints[next]) {
            int k = floorLg(i);
            cout << setw(8) << i << setw(15) << running
                 << setw(14) << closedForm(i)
                 << setw(19) << 3 * i - k - 2
                 << setw(12) << fixed << setprecision(4)
                 << (double)running / (double)i << "\n";
            ++next;
        }
    }
    cout << "\nThe first two numeric columns agree at every checkpoint, so the\n"
            "closed form is right; the third is never smaller than the first,\n"
            "so the bound holds; the last column stays below 3, so the\n"
            "amortized cost is O(1).\n";
```

Kode 14.3 (Lanjutan) *Soal 17.1-3 diuji secara numerik terhadap bentuk tertutupnya dan batasnya. Kompilasi dengan g++ -std=c++17 -O2 -o ex1713 ex1713.cpp.*

```
    return 0;
}
```

```
   n   sum of costs   closed form   3n-floor(lg n)-2   amortized
       1              1             1                  1      1.0000
       2              3             3                  3      1.5000
       3              4             4                  6      1.3333
       4              8             8                  8      2.0000
       5              9             9                 11      1.8000
       8             19            19                 19      2.3750
      16             42            42                 42      2.6250
     100            220           220                292      2.2000
    1000           2013          2013               2989      2.0130
    1024           3060          3060               3060      2.9883
  100000         231054        231054             299982      2.3105
 1000000        2048555       2048555            2999979      2.0486

The first two numeric columns agree at every checkpoint, so the
closed form is right; the third is never smaller than the first,
so the bound holds; the last column stays below 3, so the
amortized cost is O(1).
```

Tiga hal yang perlu dibaca dari tabel itu. Pertama, lajur "sum of costs" dan "closed form" sama persis pada tiap titik periksanya, sehingga penurunannya benar. Kedua, batas $3n - \lfloor \lg n \rfloor - 2$ tidak pernah di bawah total sebenarnya, dan keduanya *sama* persis ketika n sebuah pangkat dua — tengoklah $n = 1,2,4,8,16,1024$. Jadi batasnya ketat, dan tidak dapat diperbaiki. Ketiga, biaya amortisasi terukurnya berkelana antara kira-kira 1,3 dan 2,99 tetapi tidak pernah mencapai 3, dan itulah arti "$O(1)$ dengan konstanta 3" dalam praktik.

Bandingkan baris $n = 5$ dengan tabel pada **Gambar 14.7**: tabel itu memberikan total 9 untuk lima operasi, dan programnya memberikan 9. Aritmetikanya terbukti benar.

14.3.5 Apa yang tidak dapat dilakukan metode agregat

Sebelum melangkah lebih jauh, pahamilah batas-batas metode ini dengan jelas.

Ia memberikan satu bilangan untuk semua operasi. Pada tumpukannya, `Push`, `Pop`, dan `Multipop` semuanya keluar pada biaya amortisasi 2, sekalipun sebuah `Push` jelas berbiaya tepat 1 dan akan selalu begitu. Bila Anda ingin berkata "push itu murah dan multipop itu gratis", metode agregat tidak dapat mengatakannya untuk Anda.

Ia memerlukan argumen pencacahan yang global. Pada tumpukannya dan pencacahnya, argumen semacam itu ada dan ia cantik. Pada pohon *splay* [4] atau *Fibonacci heap* [2], argumen semacam itu tidak dikenal, dan metode agregat sekadar gagal. Inilah sebabnya kedua metode lainnya ada.

Ia tidak memberi tahu Anda apa pun tentang *kapan* biayanya dikeluarkan. Bila program Anda punya tenggat — sistem waktu-nyata, antarmuka interaktif, penangan jaringan — mengetahui bahwa totalnya $O(n)$ tidak menghibur bila satu operasi di tengah memakan setengah detik. Batas amortisasi berbicara tentang *throughput*, bukan latensi. §14.11 kembali kepada hal ini.

14.4 Amortisasi Bukan Kasus Rata-Rata — Dikatakan Lagi

Anda kini sudah melihat dua analisis amortisasi yang lengkap. Inilah saatnya mengatakan hal itu lagi, sebab sekarang Anda punya sesuatu untuk menyangkutkannya.

Analisis amortisasi bukan analisis kasus rata-rata.

Kembalilah dan tengoklah bukti di §14.3.2 — "tiap elemen di-pop paling banyak sekali sesudah ia di-push". Di mana peluangnya? Tidak ada. Kita tidak mengandaikan barisannya acak. Kita tidak mengandaikan push lebih mungkin daripada pop. Kita membuktikan sebuah pernyataan tentang

setiap barisan: dalam urutan apa pun n operasi itu datang, dipilih oleh siapa pun Anda suka dengan alasan apa pun, totalnya paling besar $2n$.

Sekarang tengoklah argumen pencacahnya di §14.3.3. Lagi-lagi: di mana peluangnya? Tidak ada di mana-mana. Kita mencacah persis berapa kali tiap bit berbalik, untuk barisan tertentu "naikkan n kali dari nol". Hanya ada satu barisan semacam itu. Tidak ada apa pun untuk dirata-ratakan.

> **DUA KALIMAT YANG TERDENGAR SAMA DAN SEBENARNYA TIDAK.**
>
> *"Quicksort teracak berjalan dalam waktu harapan $O(n\lg n)$."* — Artinya: bila Anda menjalankannya berkali-kali pada masukan yang sama, rata-rata waktu jalannya akan $O(n\lg n)$. Sebagian jalan perorangan akan lebih buruk. Dengan nasib buruk, satu jalan tunggal dapat memakan $\Theta(n^2)$. Jaminannya bersifat peluang.
>
> *"`Multipop` berjalan dalam waktu $O(1)$ teramortisasi."* — Artinya: untuk setiap barisan berisi n operasi tumpukan, waktu totalnya adalah $O(n)$. Tidak ada nasib yang terlibat. Anda tidak dapat menyusun barisan yang mengalahkan batasnya, dan begitu pula orang lain. Jaminannya mutlak.

Berikut sebuah uji yang dapat Anda terapkan. Tanyakanlah: **dapatkah seorang lawan yang mengetahui seluruh implementasi saya mengalahkan batas ini?** Untuk batas kasus rata-rata, sering kali ya — seorang lawan yang mengetahui aturan pivot Anda dapat menyuapkan masukan yang membuat quicksort teracak menjadi lambat, bila ia juga dapat meramalkan bilangan acak Anda. Untuk batas amortisasi, tidak pernah. Batas amortisasi adalah batas kasus terburuk atas sebuah jumlah.

Ada kekeliruan kedua yang patut dicegat, sebab ia lebih halus. Analisis amortisasi juga **bukan** hal yang sama dengan "biasanya cepat". Sebuah struktur dapat punya batas amortisasi yang mengerikan dan tetap cepat dalam pemakaian sehari-hari, dan sebuah struktur dapat punya batas amortisasi yang indah dan tetap tersendat pada saat yang tidak nyaman. Batas amortisasi adalah pernyataan tentang *totalnya*, dan tidak lebih. Yang ia belikan untuk Anda adalah hak menuliskan $n \times O(1)$ alih-alih $n \times O(n)$ ketika Anda menghitung biaya sebuah gelung — yang biasanya persis yang Anda inginkan, dan sesekali tidak.

14.5 Metode Akuntansi

14.5.1 Tagihan dan kredit

Metode agregat menghitung sebuah total lalu membaginya. Metode akuntansi bekerja dengan arah sebaliknya: ia *memutuskan lebih dahulu* berapa tiap operasi akan ditagih, lalu membuktikan bahwa tagihannya memadai.

Metode ini bertumpu pada tiga aturan:

Biaya teramortisasi dalam metode akuntansi terdiri atas biaya sebenarnya dan kredit, yang dapat disetorkan atau ditarik.

Untuk memperlihatkan bahwa biaya rata-rata per operasi pada kasus terburuk itu rendah, total biaya teramortisasi atas sebuah barisan operasi harus berlaku sebagai batas atas bagi total biaya sebenarnya barisan tersebut.

Total kredit yang terkait dengan sebuah struktur data harus selalu tetap tak-negatif.

Inilah kiasannya, yang layak diambil secara harfiah sebab pembukuannya memang sungguh-sungguh pembukuan.

Bayangkan tiap langkah kerja dasar berbiaya satu rupiah untuk dikerjakan. Ketika sebuah operasi berjalan, kita menyerahkan sebuah jatah kepadanya — **biaya amortisasinya** $\hat{c}_i$, yang kita pilih. Operasi itu membelanjakan biaya sebenarnya c_i untuk mengerjakan kerjanya. Bila kita memberinya lebih daripada yang dibelanjakannya, surplus $\hat{c}_i - c_i$ tidak lenyap: kita menyetorkannya ke dalam strukturnya, ditempelkan pada suatu objek tertentu. Bila kita memberinya kurang daripada yang dibelanjakannya, ia menutup selisihnya dengan menarik setoran yang ditinggalkan operasi-operasi sebelumnya.

Setoran yang menumpuk itulah **kredit** di dalam strukturnya. Tulislah C_k sesudah k operasi:

$$C_k = \sum_{i=1}^{k} \hat{c}_i - \sum_{i=1}^{k} c_i.$$

14.5.2 Invariannya, dan mengapa itulah yang harus Anda buktikan

Segalanya berporos pada satu syarat.

INVARIAN KREDITNYA. Untuk setiap $k = 1,2, \dots, n$,

$$C_k = \sum_{i=1}^{k} \hat{c}_i - \sum_{i=1}^{k} c_i \geq 0.$$

(14.8)

Dengan kata lain: **strukturnya tidak pernah berutang.** Pada titik mana pun, kerja yang sudah dilakukan tidak boleh melampaui jatah yang sudah dibagikan.

Mengapa ini memberikan apa yang kita inginkan? Susunlah ulang pertidaksamaannya:

$$\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \hat{c}_i.$$

Itu persis syarat Persamaan (14.2) di §14.2.3. Jadi bila kreditnya tidak pernah negatif, biaya amortisasi yang dipilih itu sah, dan biaya sebenarnya seluruh barisannya dibatasi oleh total jatahnya. Bila tiap jatahnya paling besar suatu konstanta $\hat{c}$, totalnya paling besar $n\hat{c} = O(n)$.

Mengapa membuktikan $C_k \geq 0$ sudah cukup, dan mengapa ia harus berlaku pada setiap k dan bukan sekadar pada $k = n$? Sebab barisannya dapat berhenti di mana saja. Seandainya kreditnya boleh menyelam di bawah nol pada langkah 500 lalu pulih pada langkah 1000, maka awalan berisi 500 operasi itu akan berbiaya lebih daripada yang kita jatahkan untuknya — dan program yang hanya melakukan 500 operasi tersebut adalah program yang sepenuhnya sah. Batasnya harus berlaku untuk setiap awalan.

Bagaimana Anda sungguh-sungguh membuktikannya? Bukan dengan menghitung C_k dalam bentuk tertutup — itu biasanya sia-sia. Anda membuktikannya dengan menemukan sebuah **tafsiran fisik yang konkret atas kreditnya**: sebuah aturan yang mengatakan di mana tiap rupiah yang disimpan itu duduk, sedemikian sehingga jumlah yang tersimpan kelihatan tak-negatif. Inilah keterampilan metodenya, dan kedua contoh di bawah memperlihatkannya.

JEBAKANNYA. Menggiurkan untuk berkata "tagihannya jelas cukup besar" lalu melanjutkan. Ia tidak jelas apa-apa. Metode akuntansi hanya menjadi sebuah bukti bila Anda *menempatkan*

kreditnya — menyebutkan objek mana yang memegangnya — lalu mengamati bahwa cacah objek semacam itu tidak mungkin negatif. Sebuah pernyataan bahwa saldonya tetap positif, tanpa keterangan di mana uangnya berada, tidak membuktikan apa-apa.

14.5.3 Tumpukannya, dengan metode akuntansi

Gambar 14.9 memberikan penetapannya.

Operasi	Biaya sebenarnya	Biaya amortisasi (tagihan)
`push`	1	2
`pop`	1	0
`multipop`	$\min(s, k)$	0

- Setiap tagihan adalah konstanta, sehingga biaya amortisasi masing-masing dari ketiga operasi itu $O(1)$.
- Kreditnya sama dengan banyaknya item di tumpukan, yang tidak pernah negatif; jadi n operasi berbiaya total $O(n)$.

Gambar 14.9 Tagihan akuntansi untuk tumpukan: `Push` (biaya sebenarnya 1) ditagih 2, sedangkan `Pop` (biaya 1) dan `Multipop` (biaya $\min(s, k)$) ditagih 0. Ketiga tagihannya konstanta, sehingga n operasi berbiaya $O(n)$.

Penetapannya mengejutkan pada perjumpaan pertama. `Pop` ditagih **nol**. `Multipop`, yang dapat membuang sejuta item, ditagih **nol**. Hanya `Push` yang ditagih, dan ia ditagih dua kali lipat biayanya.

Inilah sebabnya ia berhasil, dan inilah "tafsiran fisik yang konkret" yang dituntut subbab sebelumnya.

Aturan kreditnya: tiap item di tumpukannya membawa tepat 1 rupiah kredit.

Sekarang periksalah bahwa aturannya bertahan melewati tiap operasi.

`Push`. Ditagih 2. Ia membelanjakan 1 untuk sungguh-sungguh meletakkan itemnya di tumpukan. Sisa yang 1 diletakkan *pada item itu* dan ikut bepergian bersamanya. Aturannya tetap berlaku: item barunya punya 1 rupiah, dan tidak ada yang lain berubah. Kreditnya naik tepat 1, begitu pula banyaknya item.

`Pop`. Ditagih 0. Ia harus membelanjakan 1 untuk membuang sebuah item. Dari mana rupiah itu datang? Dari item yang sedang dibuang — yang, menurut aturannya, sedang membawa tepat 1. Item itu membayar pembuangan dirinya sendiri lalu pergi tanpa membawa apa-apa. Kreditnya turun tepat 1, begitu pula banyaknya item. Aturannya tetap berlaku bagi item yang tersisa.

`Multipop(S,k)`. Ditagih 0. Ia membuang $\min(s, k)$ item dan harus membelanjakan $\min(s, k)$. Masing-masing item itu membawa tepat 1 rupiah menurut aturannya, sehingga masing-masing membayar pembuangan dirinya sendiri. Tidak ada yang dipinjam. Kreditnya turun $\min(s, k)$, persis sebagaimana banyaknya item.

Mengapa kreditnya tidak pernah dapat menjadi negatif. Sesudah sembarang awalan operasi, total kreditnya sama dengan banyaknya item yang sedang berada di tumpukan. Banyaknya item pada sebuah tumpukan tidak mungkin negatif. Karena itu kreditnya tidak mungkin negatif. Itulah seluruh buktinya, dan perhatikan bagaimana tafsiran fisiknya mengerjakan segalanya: kita tidak pernah perlu menghitung sebuah jumlah.

BACALAH DALAM SATU KALIMAT. *Tiap item membayar, pada saat ia di-push, untuk kedatangannya sekaligus untuk kepergiannya kelak.* Entah ia pergi lewat `Pop` atau di dalam sebuah `Multipop`, tidak ada bedanya — ongkosnya sudah dibayar di muka. Tagihan terbesar bagi operasi mana pun adalah 2, sehingga biaya amortisasi ketiga operasinya $O(1)$, dan n operasi berbiaya paling besar $2n$. Ini $2n$ yang sama dengan yang ditemukan metode agregat di §14.3.2 — dua metode, satu jawaban.

14.5.4 Pencacah binernya, dengan metode akuntansi

Penetapan untuk pencacahnya adalah:

- menyetel sebuah bit ke 1 — tagih **2**;
- menyetel sebuah bit ke 0 (sebuah penolan) — tagih **0**.

Dan alasannya dalam satu baris: *Banyaknya angka 1 di dalam pencacahnya tidak pernah negatif; karena itu, total kreditnya selalu tetap tak-negatif.*

Mari kita kembangkan itu menjadi argumen yang utuh.

Aturan kreditnya: tiap bit yang saat ini bernilai 1 membawa tepat 1 rupiah kredit.

Ingat kembali apa yang sungguh-sungguh dilakukan satu `Increment`: ia menolkan sejumlah t bit 1 di ekornya, lalu menyetel satu bit 0 menjadi 1 (kecuali ia meluap keluar lariknya, yang dalam hal itu ia tidak menyetel apa pun). Biaya sebenarnya adalah $t + 1$ tulisan pada kasus yang normal.

Satu penyetelan ke 1. Ditagih 2. Ia membelanjakan 1 untuk mengerjakan tulisannya. Yang 1 lagi disetorkan pada bit yang baru saja disetelnya, yang kini bernilai 1 — persis sebagaimana dituntut aturannya.

t penolan. Ditagih 0 masing-masing. Tiap penolan harus membelanjakan 1 untuk mengerjakan tulisannya. Tiap bit itu bernilai 1 sesaat sebelumnya, dan menurut aturannya ia membawa tepat 1 rupiah. Rupiah itu membayar penolannya. Bitnya menjadi 0 dan tidak membawa apa-apa, lagi-lagi sebagaimana dituntut aturannya.

Jadi tiap `Increment` ditagih paling besar 2, sebanyak apa pun bit yang dibaliknya. `Increment` yang membawa pencacahnya dari $2^{20} - 1$ ke 2^{20} membalik dua puluh satu bit dan tetap ditagih 2: dua puluh di antaranya penolan, dibayari kredit yang duduk pada dua puluh bit itu, yang disetorkan satu per satu sepanjang sejuta increment sebelumnya.

Mengapa kreditnya tidak pernah dapat menjadi negatif. Sesudah sembarang awalan increment, total kreditnya sama dengan banyaknya bit 1 di dalam pencacahnya. Cacah bit 1 tidak mungkin negatif. Karena itu kreditnya tidak mungkin negatif. ■

Total tagihan untuk n increment: paling besar $2n$. Jawaban yang sama dengan metode agregat, untuk ketiga kalinya.

SATU KASUS YANG TIDAK DICAKUP ATURANNYA. Bila pencacahnya meluap — setiap bitnya 1 dan `Increment` berjalan keluar dari ujung lariknya — maka tidak ada bit yang disetel ke 1, hanya penolan yang terjadi, dan operasinya ditagih 0 sementara ia membelanjakan k. Tetapi seluruh k penolan itu dibayari oleh k rupiah yang duduk pada k bit satunya, sehingga invariannya selamat; kreditnya sekadar merosot ke nol. Analisisnya tidak tercederai. Sebuah pencacah k-bit yang sudah meluap, dalam arti yang sungguhan, telah membelanjakan segala yang ditabungnya.

Mengamati kreditnya

Invarian yang dapat Anda amati adalah invarian yang Anda percayai. **Kode 14.4** menjalankan kedua strukturnya lalu mencetak saldo banknya sesudah tiap operasi, berdampingan dengan besaran yang menurut buktinya sama dengan saldo itu.

Kode 14.4 Metode akuntansi yang diinstrumentasi: saldo kreditnya dicetak sesudah tiap operasi, di samping besaran yang menurut buktinya sama dengannya. Kompilasi dengan `g++ -std=c++17 -O2 -o credit credit.cpp`.

```
// Metode akuntansi yang dibuat kasatmata.  Dua struktur dijalankan
// berdampingan -- tumpukan MULTIPOP dan pencacah biner -- dan sesudah tiap
// operasi tunggal program ini mencetak, atau setidaknya memeriksa, saldo
// banknya:  kredit = (amortisasi yang ditagih) - (biaya sebenarnya).
// Seluruh argumen metode akuntansi adalah bahwa bilangan ini tidak pernah
// turun di bawah nol.  Di sini kita menyaksikannya tidak turun di bawah nol.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;

// ---------- Struktur 1: tumpukannya, ditagih 2 untuk PUSH dan 0 untuk sisanya.
struct AccountedStack {
    vector<int> data;
    long long actual = 0, charged = 0;

    void push(int x)      { data.push_back(x); actual += 1; charged += 2; }
    void pop()            { if (!data.empty()) { data.pop_back(); actual += 1; } charged +=
0; }
    int  multipop(int k)  { int r = 0;
                            while (!data.empty() && k != 0) { data.pop_back(); ++actual; --
k; ++r; }
                            charged += 0; return r; }
    long long credit() const { return charged - actual; }
};

// ---------- Struktur 2: pencacahnya, ditagih 2 untuk tiap bit yang disetel ke 1.
struct AccountedCounter {
    vector<int> A;
    long long actual = 0, charged = 0;
    explicit AccountedCounter(int k) : A(k, 0) {}

    void increment() {
        size_t i = 0;
        while (i < A.size() && A[i] == 1) {   // menolkan sebuah 1 berbiaya 1, ditagih 0
            A[i] = 0; ++actual; ++i;
        }
        if (i < A.size()) {                   // menyetel sebuah 0 berbiaya 1, ditagih 2
            A[i] = 1; ++actual; charged += 2;
        }
    }
    long long ones() const { long long c = 0; for (int b : A) c += b; return c; }
    long long credit() const { return charged - actual; }
};
```

Kode 14.4 (Lanjutan) *Metode akuntansi yang diinstrumentasi: saldo kreditnya dicetak sesudah tiap operasi, di samping besaran yang menurut buktinya sama dengannya. Kompilasi dengan g++ -std=c++17 -O2 -o credit credit.cpp.*

```
int main() {
    cout << "PART 1  Stack: PUSH charged 2, POP and MULTIPOP charged 0.\n";
    cout << "op                    items  actual  charged  credit  (= items?)\n";
    AccountedStack S;
    auto show = [&](const string& what) {
        cout << left << setw(22) << what << right
             << setw(6) << S.data.size() << setw(8) << S.actual
             << setw(9) << S.charged << setw(8) << S.credit()
             << (S.credit() == (long long)S.data.size() ? "   yes" : "   NO") << "\n";
    };
    S.push(1); show("PUSH 1");
    S.push(2); show("PUSH 2");
    S.push(3); show("PUSH 3");
    S.push(4); show("PUSH 4");
    S.pop();   show("POP");
    S.multipop(2); show("MULTIPOP 2");
    S.push(9); show("PUSH 9");
    S.multipop(100); show("MULTIPOP 100");

    long long minCredit = 0;
    unsigned long long seed = 7;
    auto rnd = [&seed]() { seed = seed * 6364136223846793005ULL + 1442695040888963407ULL;
                           return (unsigned)(seed >> 33); };
    AccountedStack T;
    for (int t = 0; t < 200000; ++t) {
        unsigned r = rnd() % 100;
        if (r < 55) T.push((int)r);
        else if (r < 85) T.pop();
        else T.multipop((int)(rnd() % 500));
        if (T.credit() < minCredit) minCredit = T.credit();
    }
    cout << "\n200000 random stack operations: lowest credit ever seen = "
         << minCredit << "; final credit = " << T.credit()
         << ", final stack size = " << T.data.size() << "\n";

    cout << "\nPART 2  Counter: charge 2 for every bit set to 1, 0 for every reset.\n";
    cout << "  n   value  ones  actual  charged  credit  (= ones?)\n";
    AccountedCounter C(16);
    for (int n = 1; n <= 16; ++n) {
        C.increment();
        cout << setw(4) << n << setw(8) << n << setw(6) << C.ones()
             << setw(8) << C.actual << setw(9) << C.charged
             << setw(8) << C.credit()
             << (C.credit() == C.ones() ? "     yes" : "     NO") << "\n";
    }
    AccountedCounter D(32);
```

Kode 14.4 (Lanjutan) *Metode akuntansi yang diinstrumentasi: saldo kreditnya dicetak sesudah tiap operasi, di samping besaran yang menurut buktinya sama dengannya. Kompilasi dengan g++ -std=c++17 -O2 -o credit credit.cpp.*

```
    long long worst = 0;
    for (long long t = 0; t < 1000000; ++t) {
        D.increment();
        if (D.credit() < worst) worst = D.credit();
    }
    cout << "\n1000000 increments: lowest credit ever seen = " << worst
         << "; final credit = " << D.credit()
         << ", number of 1 bits = " << D.ones() << "\n";
    cout << "actual cost = " << D.actual << ", charged = " << D.charged
         << " (= 2n, the bound proved in the text)\n";
    return 0;
}
```

```
PART 1  Stack: PUSH charged 2, POP and MULTIPOP charged 0.
op                      items  actual  charged  credit  (= items?)
PUSH 1                      1       1        2       1   yes
PUSH 2                      2       2        4       2   yes
PUSH 3                      3       3        6       3   yes
PUSH 4                      4       4        8       4   yes
POP                         3       5        8       3   yes
MULTIPOP 2                  1       7        8       1   yes
PUSH 9                      2       8       10       2   yes
MULTIPOP 100                0      10       10       0   yes

200000 random stack operations: lowest credit ever seen = 0; final credit = 9, final stack
size = 9

PART 2  Counter: charge 2 for every bit set to 1, 0 for every reset.
  n   value  ones  actual  charged  credit  (= ones?)
   1       1     1       1        2       1     yes
   2       2     1       3        4       1     yes
   3       3     2       4        6       2     yes
   4       4     1       7        8       1     yes
   5       5     2       8       10       2     yes
   6       6     2      10       12       2     yes
   7       7     3      11       14       3     yes
   8       8     1      15       16       1     yes
   9       9     2      16       18       2     yes
  10      10     2      18       20       2     yes
  11      11     3      19       22       3     yes
  12      12     2      22       24       2     yes
  13      13     3      23       26       3     yes
  14      14     3      25       28       3     yes
  15      15     4      26       30       4     yes
  16      16     1      31       32       1     yes

1000000 increments: lowest credit ever seen = 0; final credit = 7, number of 1 bits = 7
actual cost = 1999993, charged = 2000000 (= 2n, the bound proved in the text)
```

Bacalah lajur “credit” terhadap lajur “items” pada Bagian 1, dan terhadap lajur “ones” pada Bagian 2. Keduanya sama pada setiap langkah tunggal — dan itulah aturan kreditnya, diuji alih-alih sekadar dinyatakan. Kredit terendah yang pernah teramati, sepanjang 200.000 operasi tumpukan acak dan 1.000.000 increment, adalah 0: saldonya menyentuh lantai berulang kali dan tidak pernah menembusnya.

Dan perhatikan lajur “actual” pada Bagian 2: 1,3,4,7,8,10,11,15, … ,31. Itu lajur Total Cost pada **Gambar 14.5** sekali lagi, dicapai oleh program yang sama sekali berbeda.

14.5.5 Soal 17.2-2, dikerjakan tuntas

Soalnya adalah: **kerjakan ulang Soal 17.1-3 dengan metode akuntansi.** **Gambar 14.10** merangkum penyelesaiannya.

Tagih setiap operasi 3. Kredit yang tersisa sesudah operasi ke-i, ditulis Total_i, adalah

$$\mathit{Total}_i = \sum_{j=1}^{i} 3 \;-\; \mathit{ActualCost}_i = 3i - \left(2^{\lfloor \lg i \rfloor + 1} + i - \lfloor \lg i \rfloor - 2\right).$$

Kasus 1: $i = 2^k$ dengan $k \geq 0$. Maka $2^{\lfloor \lg i \rfloor + 1} = 2i$ dan

$$\mathit{Total}_i = 3i - (2i + i - k - 2) = k + 2 \;\geq\; 2.$$

Kasus 2: $i = 2^k + j$ dengan $k \geq 0$ dan $0 \leq j < 2^k$. Maka $2^{\lfloor \lg i \rfloor + 1} = 2^{k+1} = 2(i - j)$ dan

$$\mathit{Total}_i = 3i - \left(2(i - j) + i - k - 2\right) = 2j + k + 2 \;\geq\; 2.$$

Pada kedua kasus kreditnya tidak negatif, sehingga biaya amortisasi total $3n$ menjadi batas atas biaya sebenarnya total: n operasi berbiaya $O(n)$, dan biaya amortisasi per operasi adalah $O(1)$.

Gambar 14.10 Penyelesaian Soal 17.2-2 dengan metode akuntansi: tiap operasi ditagih 3, dan kredit sesudah operasi i adalah $3i$ dikurangi biaya sebenarnya sejauh itu, dipecah menjadi dua kasus menurut letak i di antara pangkat-pangkat dua.

Tagihannya

Penyelesaiannya dibuka dengan sebuah keputusan: *biaya teramortisasi per operasi adalah 3.* Jadi aturannya sekadar **tagihlah tiap operasi 3**, apa pun ia. Operasi murah yang berbiaya 1 menyetorkan 2. Operasi mahal yang berbiaya i menarik $i - 3$.

Kita sekarang harus membuktikan bahwa kreditnya tidak pernah menjadi negatif.

Kredit sesudah i operasi

Menurut definisinya, kreditnya adalah total yang ditagih dikurangi total yang sungguh-sungguh dibelanjakan:

$$\text{Total}_i \;=\; \sum_{j=1}^{i} 3 \;-\; \text{ActualCost}_i \;=\; 3i - \text{ActualCost}_i,$$

yang merupakan baris tampilan pertama pada **Gambar 14.10**. Dan kita sudah tahu ActualCost_i — ia rumus eksak Persamaan (14.7) dari Soal 17.1-3, dengan i menggantikan n:

$$\text{ActualCost}_i \;=\; 2^{\lfloor \lg i \rfloor + 1} + i - \lfloor \lg i \rfloor - 2.$$

Mensubstitusikannya memberikan baris kedua:

$$\text{Total}_i \;=\; 3i - (2^{\lfloor \lg i \rfloor + 1} + i - \lfloor \lg i \rfloor - 2).$$

Sekarang pecahlah menjadi dua kasus menurut di mana i duduk relatif terhadap pangkat-pangkat dua. Pada keduanya, tulislah $k = \lfloor \lg i \rfloor$.

Kasus 1: i tepat sebuah pangkat dua, $i = 2^k$ dengan $k \geq 0$

Maka $\lfloor \lg i \rfloor = k$ persis, dan

$$2^{\lfloor \lg i \rfloor + 1} \;=\; 2^{k+1} \;=\; 2 \cdot 2^k \;=\; 2i.$$

Jadi

$$\begin{aligned} Total_i &= 3i - (2i + i - k - 2) \\ &= 3i - (3i - k - 2) \\ &= k + 2. \end{aligned}$$

Itu kasus pertama pada **Gambar 14.10**: $\text{Total}_i = 3i - (3i - k - 2) = k + 2$.

Karena $k \geq 0$, kita memperoleh $\text{Total}_i \geq 2 > 0$. **Kreditnya positif.** Bagus — dan perhatikan bahwa inilah saat yang *paling ketat*, tepat sesudah operasi termahal yang pernah dijumpai barisannya. Bila saldonya selamat di sini, ia selamat di mana-mana.

Kasus 2: i terletak tegas di antara dua pangkat, $i = 2^k + j$ dengan $k \geq 0$ dan $0 \leq j < 2^k$

Di sini $\lfloor \lg i \rfloor$ tetap k, sebab $2^k \leq i < 2^{k+1}$. Dan sekarang

$$2^{\lfloor \lg i \rfloor + 1} \;=\; 2^{k+1} \;=\; 2 \cdot 2^k \;=\; 2\,(i - j),$$

dengan memakai $2^k = i - j$ dari definisi j. Jadi

$$\begin{aligned} Total_i &= 3i - (2(i - j) + i - k - 2) \\ &= 3i - (2i - 2j + i - k - 2) \\ &= 3i - (3i - 2j - k - 2) \\ &= 2j + k + 2, \end{aligned} \tag{14.9}$$

yang merupakan kasus kedua: $\text{Total}_i = 2j + k + 2$.

Karena $j \geq 0$ dan $k \geq 0$, lagi-lagi $\text{Total}_i \geq 2 > 0$.

(Kasus 1 sebenarnya adalah kasus khusus $j = 0$ dari Kasus 2, dan itu pemeriksaan kewarasan yang berguna: memasukkan $j = 0$ ke dalam $2j + k + 2$ memberikan $k + 2$, yang cocok dengan Kasus 1. Keduanya tetap layak dipisahkan, sebab Kasus 1 adalah tempat sebuah operasi mahal baru saja meletus dan saldonya berada di titik terendahnya.)

Kesimpulannya

Kreditnya sekurang-kurangnya 2 sesudah tiap operasi, sehingga ia pasti tak-negatif, sehingga tagihan 3 per operasi itu sah. Karena itu: *total biaya teramortisasi sebesar $3n$ berlaku sebagai batas atas bagi total biaya sebenarnya, karena itu kompleksitas n operasi adalah $O(n)$; jadi biaya teramortisasi per operasinya adalah $O(1)$.*

Bandingkan dengan §14.3.4, tempat metode agregat memberikan batas $3n - \lfloor \lg n \rfloor - 2$. Metode akuntansi memberikan $3n$ yang sedikit lebih lemah. Itu khas: metode akuntansi menukar sedikit ketajaman dengan argumen yang jauh lebih pendek dan — yang lebih penting — dengan pernyataan yang bersifat *per operasi* alih-alih global.

Diperiksa dengan sebuah program

Kode 14.5 menjalankan barisannya, melacak saldonya, lalu membandingkannya dengan Persamaan (14.9) pada tiap langkah.

Kode 14.5 Soal 17.2-2: saldo kreditnya dilacak dan diuji terhadap $2j + k + 2$. Kompilasi dengan `g++ -std=c++17 -O2 -o ex1722 ex1722.cpp`.

```
// Soal 17.2-2: kerjakan ulang Soal 17.1-3 dengan metode akuntansi.
// Tiap operasi ditagih 3.  Program ini melacak saldo banknya sesudah
// tiap operasi dan mencocokkannya dengan bentuk tertutup dalam teks,
//     bila i = 2^k + j  dengan 0 <= j < 2^k, maka credit_i = 2j + k + 2.
#include <iostream>
#include <iomanip>
using namespace std;

bool isPowerOfTwo(long long i) { return i > 0 && (i & (i - 1)) == 0; }
long long actualCost(long long i) { return isPowerOfTwo(i) ? i : 1; }

int floorLg(long long n) { int k = 0; while ((1LL << (k + 1)) <= n) ++k; return k; }

int main() {
    const long long N = 1000000;
    long long credit = 0, minCredit = 0, actualTotal = 0;
    bool formulaHolds = true;
    long long firstBad = 0;

    cout << "  i  actual  charged  credit   2j+k+2\n";
    for (long long i = 1; i <= N; ++i) {
        long long c = actualCost(i);
        actualTotal += c;
        credit += 3 - c;                        // tagih 3, bayarkan c
        int k = floorLg(i);
        long long j = i - (1LL << k);           // sehingga i = 2^k + j, 0 <= j < 2^k
        long long predicted = 2 * j + k + 2;
        if (credit != predicted && formulaHolds) { formulaHolds = false; firstBad = i; }
        if (credit < minCredit) minCredit = credit;
        if (i <= 17)
            cout << setw(3) << i << setw(8) << c << setw(9) << 3
                 << setw(8) << credit << setw(9) << predicted << "\n";
    }
    cout << "\nclosed form 2j+k+2 agrees for every i up to " << N << ": "
         << (formulaHolds ? "yes" : "no, first failure at i = ") ;
    if (!formulaHolds) cout << firstBad;
    cout << "\nlowest credit ever seen = " << minCredit
         << "  (the accounting method needs this to be >= 0)\n";
    cout << "total actual cost = " << actualTotal
         << ", total charged = " << 3 * N
         << ", so 3n is indeed an upper bound.\n";
    return 0;
}
```

```
  i  actual  charged  credit   2j+k+2
  1       1        3       2        2
  2       2        3       3        3
  3       1        3       5        5
  4       4        3       4        4
  5       1        3       6        6
  6       1        3       8        8
  7       1        3      10       10
  8       8        3       5        5
  9       1        3       7        7
 10       1        3       9        9
 11       1        3      11       11
 12       1        3      13       13
 13       1        3      15       15
 14       1        3      17       17
 15       1        3      19       19
 16      16        3       6        6
 17       1        3       8        8

closed form 2j+k+2 agrees for every i up to 1000000: yes
lowest credit ever seen = 0  (the accounting method needs this to be >= 0)
total actual cost = 2048555, total charged = 3000000, so 3n is indeed an upper bound.
```

Rumus itu terbukti benar untuk sejuta operasi. Amatilah saldonya: ia naik 2 pada tiap operasi murah dan merosot tajam pada $i = 4,8,16$ — turun ke 4, 5, dan 6 berturut-turut, yang adalah $k + 2$ untuk $k = 2,3,4$, persis seperti kata Kasus 1. Ia tidak pernah mendekati nol, dan marginnya justru *tumbuh*: operasi yang mahal tertinggal di belakang tabungannya, dan itulah sebabnya batas amortisasinya lapang alih-alih ketat.

14.6 Metode Potensial

14.6.1 Gagasannya

Metode akuntansi menyimpan kredit *pada objek*: satu rupiah pada item tumpukan ini, satu rupiah pada bit 1 itu. Itu konkret dan mudah dibayangkan, tetapi ia menjadi canggung ketika pembayaran di mukanya tidak secara alami menempel pada satu objek mana pun — ketika yang Anda tabung adalah sifat *bentuk* strukturnya alih-alih sifat isinya.

Metode potensial menyelesaikan ini dengan menyimpan pembayaran di mukanya sebagai satu bilangan yang ditempelkan pada strukturnya secara keseluruhan. Garis besarnya:

Setiap struktur data dikaitkan dengan sebuah potensial yang tak-negatif. Struktur yang akan dipakai untuk n operasi ditentukan lebih dahulu. Barisan struktur data yang bersesuaian adalah $D_0, D_1, \ldots, D_n$, dengan potensial $\Phi(D_0), \Phi(D_1), \ldots, \Phi(D_n)$.

Mari kita siapkan notasinya dengan cermat, sebab lima lambang datang sekaligus.

> **METODE POTENSIAL, DISIAPKAN.** Kita melakukan n operasi pada sebuah struktur data.

- D_0 adalah strukturnya **sebelum** ada operasi apa pun — keadaan awalnya.
- D_i adalah strukturnya **sesudah** operasi ke-i, untuk $i = 1,2, \ldots, n$.
- c_i adalah **biaya sebenarnya** operasi ke-i: kerja sungguhan yang dilakukannya dalam mengubah D_{i-1} menjadi D_i.
- Φ (fi kapital, sebuah huruf Yunani) adalah **fungsi potensialnya**. Ia mengambil sebuah keadaan strukturnya lalu mengembalikan sebuah bilangan real, $\Phi(D_i)$. Kita yang memilihnya.
- $\hat{c}_i$ adalah **biaya amortisasi** operasi ke-i, didefinisikan sebagai

$$\hat{c}_i \;=\; c_i \;+\; \Phi(D_i) \;-\; \Phi(D_{i-1}). \tag{14.10}$$

Sebagian buku menuliskan biaya amortisasinya a_i alih-alih $\hat{c}_i$; keduanya hal yang sama, dan definisinya lalu berbunyi $a_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

Bacalah definisi Persamaan (14.10) dengan kata-kata. **Biaya amortisasi sebuah operasi adalah berapa ia sungguh-sungguh berbiaya, ditambah seberapa banyak ia menaikkan potensialnya** (atau dikurangi seberapa banyak ia menurunkannya). Sebuah operasi yang mengerjakan sedikit kerja tetapi meninggalkan strukturnya dalam keadaan yang lebih berbahaya — lebih banyak item bertumpuk, lebih banyak bit bernilai 1, sebuah tabel yang lebih penuh — ditagih untuk bahaya yang diciptakannya. Sebuah operasi yang mengerjakan banyak sekali kerja tetapi *menjinakkan* strukturnya memperoleh pengembalian atas bahaya yang dihapusnya.

Suku $\Phi(D_i) - \Phi(D_{i-1})$ disebut **perubahan potensialnya**, dan sering ditulis $\Delta\Phi_i$.

14.6.2 Jumlah teleskopiknya

Mengapa definisi ini memberikan biaya amortisasi yang sah? Jumlahkanlah Persamaan (14.10) atas seluruh barisannya, seperti yang dilakukan **Gambar 14.11**.

Bila biaya amortisasi dijumlahkan, suku-suku potensialnya saling meniadakan berpasangan (teleskopik):

$$\begin{aligned}\sum_{i=1}^{n} a_i &= \sum_{i=1}^{n}\big(c_i + \Phi(D_i) - \Phi(D_{i-1})\big)\\ &= \Big(\sum_{i=1}^{n} c_i\Big) + \big(\cancel{\Phi(D_1)} - \Phi(D_0)\big) + \big(\cancel{\Phi(D_2)} - \cancel{\Phi(D_1)}\big) + \cdots + \big(\Phi(D_n) - \cancel{\Phi(D_{n-1})}\big)\\ &= \Big(\sum_{i=1}^{n} c_i\Big) + \Phi(D_n) - \Phi(D_0).\end{aligned}$$

Jadi bila $\Phi(D_n) \geq \Phi(D_0)$, biaya amortisasi total $\sum a_i$ menjadi batas atas biaya sebenarnya total $\sum c_i$.

Gambar 14.11 Menjumlahkan biaya amortisasi atas $i = 1, \ldots, n$ membuat suku-suku potensialnya saling meniadakan berpasangan, menyisakan yang pertama dan yang terakhir. Perhitungan inilah yang membuat seluruh metodenya bekerja.

Inilah ia dengan tiap langkahnya dituliskan.

$$\sum_{i=1}^{n} \hat{c}_i \;=\; \sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

Mengapa: ini sekadar definisi $\hat{c}_i$, disubstitusikan ke dalam jumlahnya.

$$\sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) \;=\; \sum_{i=1}^{n} c_i \;+\; \sum_{i=1}^{n}(\Phi(D_i) - \Phi(D_{i-1}))$$

Mengapa: jumlah dari beberapa jumlah adalah jumlah dari jumlah-jumlahnya; kita sekadar mengelompokkan suku c bersama-sama dan suku Φ bersama-sama.

Sekarang tengoklah baik-baik jumlah yang kedua itu. Tuliskanlah untuk $n = 4$:

$$(\Phi(D_1) - \Phi(D_0)) + (\Phi(D_2) - \Phi(D_1)) + (\Phi(D_3) - \Phi(D_2)) + (\Phi(D_4) - \Phi(D_3)).$$

$+\Phi(D_1)$ pada kurung pertama meniadakan $-\Phi(D_1)$ pada kurung kedua. $+\Phi(D_2)$ pada kurung kedua meniadakan $-\Phi(D_2)$ pada kurung ketiga. Dan seterusnya. Segala yang di tengah saling meniadakan, dan hanya suku negatif yang paling awal dan suku positif yang paling akhir yang bertahan:

$$\sum_{i=1}^{n} (\Phi(D_i) - \Phi(D_{i-1})) \;=\; \Phi(D_n) - \Phi(D_0).$$

Ini disebut **jumlah teleskopik**, mengikuti cara teropong pelaut tua mengatup ke dalam dirinya sendiri.

Jadi

$$\sum_{i=1}^{n} \hat{c}_i \;=\; \sum_{i=1}^{n} c_i \;+\; \Phi(D_n) - \Phi(D_0), \tag{14.11}$$

yang persis baris kedua pada **Gambar 14.11**.

14.6.3 Satu syaratnya, dan mengapa ia diperlukan

Dengan menyusun ulang Persamaan (14.11) untuk mengasingkan hal yang kita pedulikan:

$$\sum_{i=1}^{n} c_i \;=\; \sum_{i=1}^{n} \hat{c}_i \;-\; \Phi(D_n) \;+\; \Phi(D_0).$$

Kita ingin total *sungguhannya* di ruas kiri paling besar sama dengan jumlah *karangannya* di ruas kanan. Itu menuntut

$$-\Phi(D_n) + \Phi(D_0) \;\leq\; 0,$$

yaitu,

$$\boxed{\Phi(D_n) \;\geq\; \Phi(D_0)} \tag{14.12}$$

Metode ini biasanya dinyatakan dengan syarat Persamaan (14.12) dituntut pada setiap langkah: $\Phi(D_i) \geq \Phi(D_0)$ *untuk semua i, sehingga total potensialnya tidak pernah jatuh di bawah nilai awalnya.* Menuntutnya untuk semua i yang lebih kuat daripada yang diperlukan untuk $i = n$ saja — dan, seperti di §14.5.2, versi yang lebih kuat itulah yang benar, sebab barisannya dapat berhenti di mana saja.

> **MENGAPA SYARAT ITU PERSIS INVARIAN KREDITNYA YANG MENYAMAR.** Pada metode akuntansi, kredit sesudah k operasi adalah $\sum \hat{c}_i - \sum c_i$, dan Persamaan (14.8) menuntutnya ≥ 0. Di sini, menurut kesamaan teleskopik yang diterapkan pada k operasi pertamanya, besaran yang sama itu adalah $\Phi(D_k) - \Phi(D_0)$. Jadi "kreditnya tidak pernah menjadi negatif" dan "potensialnya tidak pernah jatuh di bawah nilai awalnya" adalah **pernyataan yang sama**.

> Potensialnya *adalah* kreditnya; satu-satunya bedanya adalah potensialnya dihitung dari keadaan strukturnya alih-alih dilacak objek demi objek.

Jalan pintas praktisnya. Hampir selalu kita mengaturnya sehingga $\Phi(D_0) = 0$ — strukturnya mulai kosong, atau dalam keadaan diam — dan sehingga $\Phi(D) \geq 0$ untuk setiap keadaan D yang terjangkau. Maka syaratnya terpenuhi dengan sendirinya dan tidak ada lagi yang perlu diperiksa. Biasakanlah menyatakan kedua fakta itu secara tegas ketika Anda memakai metode ini; bukti yang melewatkannya belum lengkap.

Bagaimana menemukan Φ. Tidak ada algoritma untuk ini. Φ seharusnya mengukur *seberapa banyak kesulitan yang sedang ditimbun strukturnya* — seberapa jauh ia dari keadaan yang di dalamnya sebuah operasi mahal tidak mungkin terjadi. Pada sebuah tumpukan, kesulitannya adalah item yang menunggu di-pop. Pada sebuah pencacah, kesulitannya adalah bit 1 yang menunggu dinolkan. Pada tabel yang hampir penuh, kesulitannya adalah penyalinan yang akan datang. Anda menerka, Anda menghitung biaya amortisasinya, dan bila hasilnya konstan, terkaan Anda benar. Menerka dengan baik adalah keterampilan yang datang dari melihat beberapa contoh, dan itulah gunanya sisa subbab ini.

14.6.4 Tumpukannya, dengan metode potensial

Gambar 14.12 merangkum analisis potensial tumpukan.

- Potensial: $\Phi(D_i)$ = banyaknya objek di tumpukan sesudah operasi ke-i.
- Tumpukannya mula-mula kosong, jadi $\Phi(D_0) = 0$; tumpukan tidak pernah memuat objek sebanyak bilangan negatif, jadi $\Phi(D_i) \geq 0 = \Phi(D_0)$.
- `Push` ke tumpukan berisi s objek: $a_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (s+1) - s = 2$.
- `Multipop`(S, k) dengan $k' = \min(k, s)$: $a_i = k' + (s - k') - s = 0$; demikian pula `Pop` berbiaya $1 - 1 = 0$.

Gambar 14.12 Tumpukan dengan metode potensial: dengan Φ = banyaknya elemen di tumpukannya, `Push` berbiaya amortisasi $1 + (s+1) - s = 2$ dan `Multipop` berbiaya amortisasi $k' - k' = 0$, dengan $k' = \min(k, s)$.

Pilihannya. $\Phi(D)$ = banyaknya objek yang saat ini berada di tumpukannya.

Periksa kedua syaratnya. Tumpukannya mulai kosong, sehingga $\Phi(D_0) = 0$. Banyaknya elemen pada sebuah tumpukan tidak pernah negatif, sehingga $\Phi(D_i) \geq 0 = \Phi(D_0)$ untuk setiap i. Kedua syaratnya berlaku, sebagaimana dicatat **Gambar 14.12**.

Sekarang hitunglah biaya amortisasi tiap operasinya. Misalkan s banyaknya item di tumpukannya *sebelum* operasinya, sehingga $\Phi(D_{i-1}) = s$.

`Push`. Biaya sebenarnya $c_i = 1$. Sesudahnya tumpukannya memuat $s + 1$ item, sehingga $\Phi(D_i) = s + 1$. Karena itu

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}) \;=\; 1 + (s+1) - s \;=\; 1 + 1 \;=\; 2.$$

Ini baris pertama **Gambar 14.12**.

`Pop`. Biaya sebenarnya $c_i = 1$. Sesudahnya tumpukannya memuat $s - 1$ item. Karena itu

$$\hat{c}_i \;=\; 1 + (s-1) - s \;=\; 1 - 1 \;=\; 0.$$

`Multipop(S,k)`. Misalkan $k' = \min(k, s)$ — banyaknya item yang sungguh-sungguh dibuang. Biaya sebenarnya $c_i = k'$. Sesudahnya tumpukannya memuat $s - k'$ item. Karena itu

$$\hat{c}_i \;=\; k' + (s - k') - s \;=\; k' - k' \;=\; 0,$$

yang merupakan baris kedua **Gambar 14.12**.

Jadi biaya amortisasinya paling besar 2 untuk setiap operasi, dan n operasi berbiaya paling besar $2n$ — jawaban yang sama untuk ketiga kalinya, dengan metode yang ketiga. Itulah gunanya mengerjakannya tiga kali: jawabannya adalah sifat strukturnya, bukan sifat metodenya.

> **DENGAN BAHASA SEHARI-HARI.** Potensialnya mencacah berapa banyak item yang duduk di tumpukan menunggu dibuang — yaitu, seberapa banyak *kerja masa depan yang tertimbun*. Sebuah `Push` mengerjakan satu satuan kerja sekarang dan menimbun satu satuan untuk nanti, sehingga ia ditagih 2. Sebuah `Multipop` mengerjakan k' satuan kerja sekarang tetapi membuang k' satuan kerja masa depan yang tertimbun, sehingga keduanya saling meniadakan dan ia tidak ditagih apa-apa. Operasi yang mahal itu gratis justru karena ia sedang *membelanjakan tabungan*, bukan menciptakan kewajiban.

14.6.5 Pencacah binernya, dengan metode potensial

Gambar 14.13 merangkum analisis potensial pencacah.

- Potensial: b_i = banyaknya bit 1 sesudah `Increment` ke-i.
- Bila `Increment` ke-i menolkan t_i bit, biaya sebenarnya paling besar $t_i + 1$, dan $b_i \le b_{i-1} - t_i + 1$.
- Perubahan potensial: $\Phi(D_i) - \Phi(D_{i-1}) \le (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$.
- Biaya amortisasi: $a_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \le (t_i + 1) + (1 - t_i) = 2$.
- Jadi n increment yang dimulai dari nol berbiaya $O(n)$.

Gambar 14.13 Pencacah dengan metode potensial: dengan b_i = banyaknya bit 1 sesudah `Increment` ke-i dan t_i = banyaknya bit yang dinolkan increment itu, perubahan potensialnya paling besar $1 - t_i$ dan biaya amortisasinya paling besar $(t_i + 1) + (1 - t_i) = 2$.

Pilihannya. $\Phi(D_i) = b_i$, dengan b_i adalah banyaknya bit 1 di dalam pencacahnya sesudah `Increment` ke-i.

Periksa syaratnya. Pencacahnya mulai dari nol, sehingga $b_0 = 0$ dan $\Phi(D_0) = 0$. Cacah bit 1 tidak pernah negatif, sehingga $\Phi(D_i) \ge 0$. Keduanya berlaku.

Biaya sebenarnya. Andaikan `Increment` ke-i menolkan t_i bit dari 1 menjadi 0. Maka ia menyetel satu bit dari 0 menjadi 1, atau (pada luapan) tidak menyetel satu pun. Jadi biaya sebenarnya — banyaknya tulisan — paling besar

$$c_i \;\le\; t_i + 1.$$

Dengan kata-kata: *biaya sebenarnya terbesar bagi operasi tersebut adalah* $t_i + 1$.

Banyaknya bit 1 yang baru. Tadinya kita punya b_{i-1} satu. Operasinya membuang t_i di antaranya dan menambahkan paling banyak satu. Jadi

$$b_i \;\le\; b_{i-1} - t_i + 1,$$

yang merupakan baris berikutnya pada **Gambar 14.13**. (Ini pertidaksamaan alih-alih persamaan semata-mata karena kasus luapannya, yang di dalamnya tidak ada bit yang disetel dan $b_i = b_{i-1} - t_i$.)

Perubahan potensialnya. Kurangkan b_{i-1} dari kedua ruasnya:

$$\Phi(D_i) - \Phi(D_{i-1}) \;=\; b_i - b_{i-1} \;\leq\; (b_{i-1} - t_i + 1) - b_{i-1} \;=\; 1 - t_i.$$

Biaya amortisasinya. Satukanlah kedua kepingnya:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= t_i + 1 + 1 - t_i \\ &= 2.\end{aligned}$$

t_i-nya saling meniadakan. Peniadaan itulah seluruh analisisnya: **sebanyak apa pun bit yang harus dinolkan increment-nya, potensialnya jatuh persis sebanyak itu, dan kedua sukunya saling memusnahkan.** Jadi $\hat{c}_i \leq 2$ untuk setiap i, dan n increment berbiaya paling besar $2n$ — $2n$ yang sama seperti sebelumnya.

Karena itu, kompleksitas n operasi adalah $O(n)$.

Kedua potensialnya, diamati

Kode 14.6 menghitung potensialnya sesudah tiap operasi pada kedua strukturnya, mencetak c_i, perubahan Φ, dan $\hat{c}_i$, lalu akhirnya menguji kesamaan teleskopiknya $\sum \hat{c}_i = \sum c_i + \Phi(D_n) - \Phi(D_0)$ secara numerik.

Kode 14.6 Metode potensial yang diinstrumentasi pada tumpukannya sekaligus pada pencacahnya. Kompilasi dengan `g++ -std=c++17 -O2 -o potential potential.cpp`.

```
// Metode potensial yang dibuat kasatmata.  Untuk tumpukannya kita ambil
// Phi(D) = banyaknya objek di tumpukan; untuk pencacahnya kita ambil
// Phi(D) = banyaknya bit 1.  Sesudah tiap operasi program ini mencetak
// biaya sebenarnya c_i, perubahan potensialnya, dan biaya amortisasinya
//     a_i = c_i + Phi(D_i) - Phi(D_{i-1}),
// lalu memeriksa kesamaan  sum(a_i) = sum(c_i) + Phi(D_n) - Phi(D_0).
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;

struct PotStack {
    vector<int> data;
    long long sumC = 0, sumA = 0;
    long long phi() const { return (long long)data.size(); }

    long long push(int x) { long long b = phi(); data.push_back(x);
                            return acct(1, b); }
    long long pop()       { long long b = phi(); long long c = 0;
                            if (!data.empty()) { data.pop_back(); c = 1; }
                            return acct(c, b); }
    long long multipop(int k) { long long b = phi(); long long c = 0;
                            while (!data.empty() && k != 0) { data.pop_back(); --k; ++c; }
                            return acct(c, b); }
    long long lastC = 0;
private:
    long long acct(long long c, long long before) {
        lastC = c; long long a = c + phi() - before;
        sumC += c; sumA += a; return a;
    }
};

struct PotCounter {
    vector<int> A;
    long long sumC = 0, sumA = 0, lastC = 0, lastT = 0;
    explicit PotCounter(int k) : A(k, 0) {}
    long long phi() const { long long s = 0; for (int b : A) s += b; return s; }
```

Kode 14.6 (Lanjutan) *Metode potensial yang diinstrumentasi pada tumpukannya sekaligus pada pencacahnya. Kompilasi dengan g++ -std=c++17 -O2 -o potential potential.cpp.*

```
    long long increment() {
        long long before = phi(), c = 0, t = 0;
        size_t i = 0;
        while (i < A.size() && A[i] == 1) { A[i] = 0; ++c; ++t; ++i; }  // t bit dinolkan
        if (i < A.size()) { A[i] = 1; ++c; }
        lastC = c; lastT = t;
        long long a = c + phi() - before;
        sumC += c; sumA += a; return a;
    }
};

int main() {
    cout << "STACK, Phi = number of objects.  Phi(D_0) = 0.\n";
    cout << "op              size  c_i   Phi before  Phi after   a_i\n";
    PotStack S;
    auto line = [&](const string& w, long long a, long long before) {
        cout << left << setw(16) << w << right << setw(4) << S.data.size()
             << setw(6) << S.lastC << setw(12) << before
             << setw(11) << S.phi() << setw(7) << a << "\n";
    };
    { long long b = S.phi(); long long a = S.push(1);   line("PUSH", a, b); }
    { long long b = S.phi(); long long a = S.push(2);   line("PUSH", a, b); }
    { long long b = S.phi(); long long a = S.push(3);   line("PUSH", a, b); }
    { long long b = S.phi(); long long a = S.push(4);   line("PUSH", a, b); }
    { long long b = S.phi(); long long a = S.push(5);   line("PUSH", a, b); }
    { long long b = S.phi(); long long a = S.pop();     line("POP", a, b); }
    { long long b = S.phi(); long long a = S.multipop(3); line("MULTIPOP 3", a, b); }
    { long long b = S.phi(); long long a = S.multipop(9); line("MULTIPOP 9", a, b); }
    cout << "sum c_i = " << S.sumC << ", sum a_i = " << S.sumA
         << ", Phi(D_n) - Phi(D_0) = " << S.phi() - 0
         << "; identity holds: "
         << ((S.sumA == S.sumC + S.phi()) ? "yes" : "no") << "\n";

    cout << "\nCOUNTER, Phi = number of 1 bits.  Phi(D_0) = 0.\n";
    cout << "  n  value  t_i (bits reset)  c_i   Phi after   a_i\n";
    PotCounter C(8);
    for (int n = 1; n <= 16; ++n) {
        long long a = C.increment();
        cout << setw(3) << n << setw(7) << n << setw(15) << C.lastT
             << setw(7) << C.lastC << setw(11) << C.phi() << setw(7) << a << "\n";
    }
    cout << "sum c_i = " << C.sumC << ", sum a_i = " << C.sumA
         << " (= 2n = " << 2 * 16 << "), Phi(D_n) = " << C.phi() << "\n";
    cout << "every a_i above is exactly 2, which is the bound proved in the text\n";
    return 0;
}
```

```
STACK, Phi = number of objects.  Phi(D_0) = 0.
op              size  c_i   Phi before  Phi after   a_i
PUSH               1     1           0          1      2
PUSH               2     1           1          2      2
PUSH               3     1           2          3      2
PUSH               4     1           3          4      2
PUSH               5     1           4          5      2
POP                4     1           5          4      0
MULTIPOP 3         1     3           4          1      0
MULTIPOP 9         0     1           1          0      0
sum c_i = 10, sum a_i = 10, Phi(D_n) - Phi(D_0) = 0; identity holds: yes

COUNTER, Phi = number of 1 bits.  Phi(D_0) = 0.
  n  value  t_i (bits reset)  c_i   Phi after   a_i
  1      1              0      1          1      2
  2      2              1      2          1      2
  3      3              0      1          2      2
  4      4              2      3          1      2
  5      5              0      1          2      2
  6      6              1      2          2      2
  7      7              0      1          3      2
  8      8              3      4          1      2
  9      9              0      1          2      2
 10     10              1      2          2      2
 11     11              0      1          3      2
 12     12              2      3          2      2
 13     13              0      1          3      2
 14     14              1      2          3      2
 15     15              0      1          4      2
 16     16              4      5          1      2
sum c_i = 31, sum a_i = 32 (= 2n = 32), Phi(D_n) = 1
every a_i above is exactly 2, which is the bound proved in the text
```

Tabel pencacahnya layak dipelajari. Lajur c_i melompat-lompat — 1, 2, 1, 3, 1, 2, 1, 4, ..., 5 — dan lajur a_i adalah dinding rata berisi angka 2. Lajur t_i memperlihatkan sebabnya: setiap kali c_i besar, t_i juga besar, dan t_i persis yang ditiadakan oleh jatuhnya potensialnya. Dan totalnya memastikan kesamaan teleskopiknya: $\sum c_i = 31$, $\Phi(D_n) = 1$, dan $31 + 1 = 32 = \sum \hat{c}_i$.

Jalan tumpukannya melakukan hal yang sama dari sisi yang lain: `Multipop 3` sungguh-sungguh berbiaya 3, dan ditagih 0, sebab potensialnya jatuh dari 4 ke 1.

14.6.6 Soal 17.3-6: antrean dari dua tumpukan

Inilah soal terbaik dalam bab ini, sebab fungsi potensialnya tidak gamblang dan strukturnya adalah struktur yang mungkin sungguh-sungguh ingin Anda bangun.

> **SOAL 17.3-6.** Tunjukkan bagaimana mengimplementasikan sebuah antrean dengan dua tumpukan biasa sehingga biaya amortisasi tiap `Enqueue` dan tiap `Dequeue` adalah $O(1)$.

Persoalannya

Sebuah **tumpukan** bersifat masuk-terakhir-keluar-pertama (*last-in-first-out*): item yang Anda peroleh kembali adalah yang paling baru Anda masukkan. Sebuah **antrean** bersifat masuk-pertama-keluar-pertama (*first-in-first-out*): item yang Anda peroleh kembali adalah yang paling lama menunggu — seperti barisan di bank. Andaikan yang diberikan kepada Anda hanyalah tumpukan. Dapatkah Anda membangun sebuah antrean?

Bisa, dan akal-akalannya adalah bahwa membalik sebuah tumpukan mengubahnya menjadi sebuah antrean. Bila Anda mem-pop segalanya dari tumpukan A lalu mem-push-nya ke tumpukan

B sambil jalan, maka item di *dasar* A — yang tertua — berakhir di *puncak* B, tempat ia dapat di-pop lebih dahulu.

Rancangannya

Rancangannya cukup tiga kalimat:

Untuk mengimplementasikan sebuah antrean memakai tumpukan, dipakailah dua tumpukan: satu sebagai tumpukan masukan (S_{IN}) dan satunya sebagai tumpukan keluaran (S_{OUT}).

Untuk operasi `Enqueue`, elemennya di-push ke tumpukan masukan (S_{IN}).

Untuk operasi `Dequeue`, sebuah elemen di-pop dari tumpukan keluaran (S_{OUT}). Bila S_{OUT} kosong, semua elemen dari S_{IN} lebih dahulu dipindahkan ke S_{OUT} (dengan mem-pop dari S_{IN} dan mem-push ke S_{OUT}), sesudah itu satu elemen di-pop dari S_{OUT}.

Gambar 14.14 memberikan pseudokodenya.

```
ENQUEUE(e)
1  PUSH(S_IN, e)
```

```
DEQUEUE()
1  jika STACK-EMPTY(S_OUT) maka
2      selama tidak STACK-EMPTY(S_IN) lakukan
3          PUSH(S_OUT, POP(S_IN))
4  jika STACK-EMPTY(S_OUT) maka
5      galat "underflow"
6  selain itu
7      kembalikan POP(S_OUT)
```

Gambar 14.14 `Enqueue` dan `Dequeue` untuk antrean dua-tumpukan: `Enqueue` adalah satu push, dan `Dequeue` mem-pop dari S_{OUT} sesudah lebih dahulu menguras seluruh S_{IN} ke dalamnya, **hanya bila** S_{OUT} **kosong**. Pengujian *underflow*-nya datang sesudah pemindahannya.

Dua rincian pseudokodenya layak dikomentari, sebab salah mengerjakannya adalah kutu yang klasik.

Pemindahannya terjadi hanya ketika S_{OUT} kosong. Bila Anda memindahkan pada setiap `Dequeue`, Anda akan membalik urutannya berulang-ulang dan strukturnya akan salah sekaligus lambat. Seluruh argumen amortisasinya bergantung pada pemindahannya yang jarang.

Pengujian *underflow*-nya datang sesudah pemindahannya. Sebuah antrean kosong hanya bila *kedua* tumpukannya kosong. Menguji S_{OUT} sebelum pemindahannya akan keliru melaporkan *underflow* setiap kali itemnya kebetulan sedang duduk di S_{IN}.

Mengapa ia benar

Item masuk ke S_{IN} menurut urutan kedatangan, sehingga item tertua di S_{IN} berada di dasar. Sebuah pemindahan mem-pop mereka satu per satu — yang terbaru lebih dahulu — lalu mem-push-nya ke S_{OUT}, sehingga yang tertua berakhir di puncak S_{OUT}. Selama S_{OUT} tidak kosong, `Dequeue` melayani dari puncaknya, yang selalu merupakan item tertua yang masih berada di dalam strukturnya. Pendatang baru masuk ke S_{IN} dan tidak dapat menyerobot antrean, sebab mereka hanya akan mencapai S_{OUT} sesudah ia terkuras habis. Jadi strukturnya sungguh-sungguh masuk-pertama-keluar-pertama.

Mengapa batas biaya naifnya buruk

Sebuah `Dequeue` yang memicu pemindahan menggerakkan setiap item di S_{IN}. Bila antreannya memuat m item, itu dapat berarti kerja $\Theta(m)$, sehingga kasus terburuk bagi satu `Dequeue` tunggal adalah $\Theta(n)$ dalam sebuah barisan berisi n operasi, dan batas kasarnya $O(n^2)$ — jawaban salah yang sama seperti biasa.

Fungsi potensialnya

Gambar 14.15 memberikan analisis potensial selengkapnya.

- Potensial: $\Phi(D_i) = 2 \times \mathit{elements}_i$, dengan $\mathit{elements}_i$ banyaknya item di tumpukan **masukan** S_{IN} sesudah operasi ke-i.
- `Enqueue`: $a_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2 = 3$.
- `Dequeue` tanpa pemindahan (S_{OUT} tidak kosong): $a_i = 1 + 0 = 1$.
- `Dequeue` dengan pemindahan $e = \mathit{elements}_{i-1}$ item: $a_i = (2e + 1) + 0 - 2e = 1$.
- Jadi `Enqueue` maupun `Dequeue` berbiaya amortisasi $O(1)$.

Gambar 14.15 Analisis potensial antrean dua-tumpukan. Dengan $\Phi(D_i) = 2 \times \text{elements}_i$, dengan elements_i mencacah item di tumpukan **masukan** saja, `Enqueue` berbiaya amortisasi 3 dan kedua jenis `Dequeue` berbiaya 1.

Pilihannya. $\Phi(D_i) = 2 \times \text{elements}_i$, dengan elements_i adalah banyaknya elemen **di dalam tumpukan masukan** S_{IN} sesudah operasi ke-i. Perhatikan baik-baik: item yang duduk di S_{OUT} menyumbang *nol*.

Mengapa itu terkaan yang benar. Peristiwa yang mahal adalah sebuah pemindahan, dan sebuah pemindahan menyentuh persis item-item di S_{IN} — dua kali masing-masing, sekali untuk mem-pop dan sekali untuk mem-push. Item di S_{OUT} tidak akan pernah dipindahkan lagi; mereka sudah membayar. Jadi "kesulitan yang tertimbun" di dalam strukturnya persis dua satuan per item yang menunggu di S_{IN}, dan itulah potensialnya.

Periksa syaratnya. Kedua tumpukannya mulai kosong, sehingga $\Phi(D_0) = 0$. Cacah elemen tidak pernah negatif, sehingga $\Phi(D_i) \geq 0$. Keduanya berlaku.

Sekarang ketiga kasusnya, mengikuti **Gambar 14.15** baris demi baris. Tulislah e untuk elements_{i-1}, ukuran S_{IN} sebelum operasinya.

Kasus 1: `Enqueue`. Satu push ke S_{IN}: biaya sebenarnya $c_i = 1$. Sesudahnya S_{IN} memuat $e + 1$ item, sehingga potensialnya beranjak dari $2e$ ke $2(e + 1)$, naik sebesar 2.

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}) \;=\; 1 + 2 \;=\; 3.$$

Ini baris pertama **Gambar 14.15**.

Kasus 2: `Dequeue` dengan S_{OUT} tidak kosong — tidak perlu pemindahan. Satu pop dari S_{OUT}: biaya sebenarnya $c_i = 1$. S_{IN} tidak tersentuh, sehingga potensialnya sama sekali tidak berubah.

$$\hat{c}_i \;=\; 1 + 0 \;=\; 1.$$

Ini baris keduanya.

Kasus 3: `Dequeue` dengan S_{OUT} kosong — sebuah pemindahan terjadi. Inilah yang mahal. Pemindahannya menggerakkan seluruh e item keluar dari S_{IN} dan masuk ke S_{OUT}. Tiap item

berbiaya 2 untuk digerakkan: satu pop dan satu push. Lalu satu pop terakhir dari S_{OUT} mengembalikan jawabannya. Jadi

$$c_i = 2e + 1.$$

Sesudahnya S_{IN} kosong, sehingga $\Phi(D_i) = 0$, sementara $\Phi(D_{i-1}) = 2e$.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= (2e + 1) + 0 - 2e \\ &= 1.\end{aligned}$$

Ini baris ketiganya, yang pada gambar ditulis sebagai $(2 \times \text{elements}_{i-1} + 1) + 0 - 2 \times \text{elements}_{i-1} = 1$.

Kesimpulannya. Tiap operasi berbiaya amortisasi paling besar 3. Jadi n operasi berbiaya paling besar $3n = O(n)$, dan biaya amortisasi per operasinya adalah $O(1)$: *biaya teramortisasi bagi operasi* `Enqueue` *maupun* `Dequeue` *adalah* $O(1)$.

> **DENGAN BAHASA SEHARI-HARI.** Tiap item menempuh tepat satu perjalanan: di-push ke S_{IN}, di-pop dari S_{IN}, di-push ke S_{OUT}, di-pop dari S_{OUT}. Empat langkah, sekali masing-masing, sepanjang seluruh hidupnya. `Enqueue` membayar tiga di antaranya di muka — satu untuk push yang dilakukannya dan dua ditabung untuk pemindahannya — dan yang keempat, pop terakhir, dibayar oleh tagihan 1 pada `Dequeue` yang membuangnya. Sebuah `Dequeue` yang menemukan seribu item menunggu mengerjakan seribu satuan kerja dan ditagih 1, sebab seribu item itu masing-masing meninggalkan dua satuan kredit ketika mereka datang.

Programnya

Kode 14.7 mengimplementasikan pseudokode **Gambar 14.14** persis, menghitung Φ sesudah tiap operasi, dan mencetak biaya sebenarnya berdampingan dengan biaya amortisasinya.

Kode 14.7 Soal 17.3-6: antrean dua-tumpukan, dengan fungsi potensialnya dihitung sesudah tiap operasi. Kompilasi dengan `g++ -std=c++17 -O2 -o queue2 queue2.cpp`.

```
// Soal 17.3-6: antrean FIFO yang dibangun dari dua tumpukan LIFO, dengan
// fungsi potensial Phi(D) = 2 * (banyaknya item di tumpukan masukan)
// dihitung sesudah tiap operasi, sehingga biaya amortisasinya
//     a_i = c_i + Phi(D_i) - Phi(D_{i-1})
// dapat dicetak di samping biaya sebenarnya dan diuji terhadap klaim
// bahwa ia tidak pernah melampaui 3.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;

class TwoStackQueue {
public:
    long long lastActual = 0, lastAmortized = 0;
    long long totalActual = 0, totalAmortized = 0;
    long long worstActual = 0, worstAmortized = 0;

    void enqueue(int x) {
        long long before = potential();
        sIn.push_back(x);                    // ENQUEUE(e, S_IN): push(S_IN, e)
        account(1, before);
    }

    // Mengembalikan nilai yang dikeluarkan; menyetel `ok` ke false bila antreannya kosong.
    int dequeue(bool& ok) {
        long long before = potential();
        long long c = 0;
        if (sOut.empty()) {                  // pindahkan semuanya ke seberang
            // Tiap elemen yang pindah berbiaya 2: satu POP dari S_IN, satu PUSH ke S_OUT.
            while (!sIn.empty()) { sOut.push_back(sIn.back()); sIn.pop_back(); c += 2; }
        }
        if (sOut.empty()) { ok = false; account(c, before); return 0; }
        int v = sOut.back(); sOut.pop_back(); ++c;
        ok = true;
        account(c, before);
        return v;
    }

    long long potential() const { return 2 * (long long)sIn.size(); }
    size_t size() const { return sIn.size() + sOut.size(); }
    size_t inSize() const { return sIn.size(); }
    size_t outSize() const { return sOut.size(); }
```

Kode 14.7 (Lanjutan) *Soal 17.3-6: antrean dua-tumpukan, dengan fungsi potensialnya dihitung sesudah tiap operasi. Kompilasi dengan g++ -std=c++17 -O2 -o queue2 queue2.cpp.*

```
private:
    vector<int> sIn, sOut;
    void account(long long c, long long before) {
        lastActual = c;
        lastAmortized = c + potential() - before;
        totalActual += lastActual;
        totalAmortized += lastAmortized;
        if (lastActual > worstActual) worstActual = lastActual;
        if (lastAmortized > worstAmortized) worstAmortized = lastAmortized;
    }
};

int main() {
    TwoStackQueue Q;
    cout << "op          returned  |S_IN|  |S_OUT|   Phi   actual   amortized\n";
    auto line = [&](const string& what, const string& ret) {
        cout << left << setw(12) << what << setw(10) << ret << right
             << setw(6) << Q.inSize() << setw(9) << Q.outSize()
             << setw(7) << Q.potential()
             << setw(8) << Q.lastActual << setw(12) << Q.lastAmortized << "\n";
    };
    for (int i = 1; i <= 5; ++i) { Q.enqueue(i); line("ENQUEUE " + to_string(i), "-"); }
    bool ok;
    for (int i = 0; i < 3; ++i) { int v = Q.dequeue(ok); line("DEQUEUE", to_string(v)); }
    Q.enqueue(6); line("ENQUEUE 6", "-");
    for (int i = 0; i < 3; ++i) { int v = Q.dequeue(ok); line("DEQUEUE", to_string(v)); }

    // Jalan panjang yang memusuhi: n/2 enqueue, lalu n/2 dequeue, berulang.
    TwoStackQueue R;
    const int n = 200000;
    long long ops = 0;
    while (ops < n) {
        for (int i = 0; i < 1000 && ops < n; ++i) { R.enqueue(i); ++ops; }
        for (int i = 0; i < 1000 && ops < n; ++i) { R.dequeue(ok); ++ops; }
    }
    cout << "\n" << n << " operations in blocks of 1000:\n";
    cout << "  worst single actual cost   = " << R.worstActual << "\n";
    cout << "  worst single amortized cost= " << R.worstAmortized << "\n";
    cout << "  total actual cost          = " << R.totalActual << "\n";
    cout << "  total amortized cost       = " << R.totalAmortized << "\n";
    cout << "  actual cost per operation  = " << fixed << setprecision(6)
         << (double)R.totalActual / ops << "\n";
    cout << "The single worst operation costs " << R.worstActual
         << ", yet the average is at most 3: that gap is what amortization buys.\n";
    return 0;
}
```

```
op          returned  |S_IN|  |S_OUT|   Phi   actual   amortized
ENQUEUE 1   -              1        0      2       1           3
ENQUEUE 2   -              2        0      4       1           3
ENQUEUE 3   -              3        0      6       1           3
ENQUEUE 4   -              4        0      8       1           3
ENQUEUE 5   -              5        0     10       1           3
DEQUEUE     1              0        4      0      11           1
DEQUEUE     2              0        3      0       1           1
DEQUEUE     3              0        2      0       1           1
ENQUEUE 6   -              1        2      2       1           3
DEQUEUE     4              1        1      2       1           1
DEQUEUE     5              1        0      2       1           1
DEQUEUE     6              0        0      0       3           1

200000 operations in blocks of 1000:
  worst single actual cost   = 2001
  worst single amortized cost= 3
  total actual cost          = 400000
  total amortized cost       = 400000
  actual cost per operation  = 2.000000
The single worst operation costs 2001, yet the average is at most 3: that gap is what
amortization buys.
```

Bacalah lajur `returned` lebih dahulu: 1, 2, 3, 4, 5, 6 — antreannya sungguh-sungguh masuk-pertama-keluar-pertama. Sekarang bacalah lajur `actual` terhadap lajur `amortized`. Baris keenam adalah `Dequeue` yang mahal itu: ia memindahkan lima item dengan biaya 2 masing-masing ditambah pop terakhirnya, sehingga $c_i = 11$ — dan biaya amortisasinya 1, sebab potensialnya jatuh dari 10 ke 0. Setiap `Enqueue` ditagih tepat 3 dan setiap `Dequeue` tepat 1, persis seperti yang diramalkan ketiga kasusnya.

Jalan yang panjang menegaskan maksudnya pada skala besar: satu operasi berbiaya 2001 satuan, dan rata-ratanya atas 200.000 operasi adalah 2.

14.7 Tabel Dinamis

14.7.1 Persoalannya

Segala yang sudah lewat adalah pemanasan bagi yang satu ini. Tabel dinamis adalah struktur yang Anda pakai tiap hari tanpa menyadarinya: ia adalah `std::vector`, ia adalah `list` Python, ia adalah `ArrayList` Java — larik yang dapat diubah ukurannya menurut Sedgewick dan Wayne [5]. Memahami biaya amortisasinya adalah beda antara menulis `v.push_back(x)` di dalam gelung dengan percaya diri dan menulisnya dengan gelisah yang samar.

Sasarannya dapat dinyatakan dalam dua kalimat:

Dengan analisis amortisasi, akan diperlihatkan bahwa biaya teramortisasi penyisipan dan penghapusan item di dalam tabelnya adalah $O(1)$, sekalipun biaya sebenarnya operasi-operasi tersebut bisa lebih tinggi ketika mereka memicu perluasan atau penyusutan. Lebih jauh, akan diperlihatkan bahwa banyaknya ruang yang tak terpakai di dalam tabelnya tidak pernah melampaui suatu faktor konstanta tertentu relatif terhadap seluruh ruang yang dialokasikan.

Jadi ada dua sasaran. **Kecepatan**: $O(1)$ teramortisasi per operasi. **Ruang**: jangan pernah menyia-nyiakan lebih dari sepenggal konstan dari yang kita alokasikan. Kita harus mengenai keduanya; salah satunya saja mudah.

Inilah situasinya. Kita menginginkan sebuah tabel — sebuah larik — yang memuat sejumlah item yang tidak diketahui. Kita tidak dapat meminta “larik yang tumbuh” kepada sistem operasinya,

sebab memori dialokasikan dalam blok berukuran tetap. Yang dapat kita lakukan adalah mengalokasikan sebuah blok berukuran tertentu, mengisinya, dan ketika ia penuh, mengalokasikan blok yang lebih besar, menyalin segalanya ke sana, lalu membebaskan yang lama.

Kosakatanya adalah:

- `TABLE-INSERT` menyisipkan sebuah item ke sebuah slot.
- `TABLE-DELETE` membuang sebuah item dari tabelnya dan membebaskan slotnya.
- $num[T]$ adalah banyaknya item yang saat ini tersimpan, $size[T]$ banyaknya slot yang dialokasikan.
- **Faktor muatan** (*load factor*) $\alpha(T)$ adalah $num[T]/size[T]$ — penggal dari ruang teralokasi yang sungguh-sungguh terpakai. (Menurut kesepakatan, tabel kosong dengan $size[T] = 0$ memiliki $\alpha(T) = 1$, sehingga pengujian "tabelnya penuh" bekerja sejak awal sekali.)

> **BENTURAN NOTASI.** Subbab ini memakai α untuk faktor muatan, dan §14.8 akan memakai α untuk fungsi Ackermann balikan. Keduanya tidak berkaitan. Ini malang, tetapi itulah notasi yang dipakai kedua kepustakaannya, jadi Anda harus hidup dengannya; konteksnya selalu memperjelas mana yang dimaksud. Di subbab ini, α adalah sebuah pecahan antara 0 dan 1.

Kebijakan pertama yang dianalisis adalah ini: *bila semua slotnya terisi, yakni bila $\alpha(T)$ sama dengan 1, tabelnya diperluas menjadi dua kali banyaknya slot tabel yang lama.* **Pelipatan** (*doubling*).

14.7.2 `TABLE-INSERT`

Gambar 14.16 memberikan pseudokodenya.

```
TABLE-INSERT(T, x)
 1  jika size[T] = 0
 2      maka alokasikan table[T] dengan 1 slot
 3          size[T] ← 1
 4  jika num[T] = size[T]
 5      maka alokasikan new-table dengan 2 · size[T] slot
 6          sisipkan semua item table[T] ke new-table
 7          bebaskan table[T]
 8          table[T] ← new-table
 9          size[T] ← 2 · size[T]
10  sisipkan x ke table[T]
11  num[T] ← num[T] + 1
```

Gambar 14.16 `TABLE-INSERT(T,x)`: baris 1–3 menangani penyisipan pertama ke dalam tabel tanpa slot, baris 4–9 memperluas (alokasikan dua kali lipat, salin, bebaskan blok lama), dan baris 10–11 menyimpan x lalu mencacahnya.

Bacalah dengan cermat.

Baris 1–3. Bila tabelnya sama sekali belum punya slot, alokasikan satu slot. Inilah kasus pemancingnya: Anda tidak dapat melipatduakan nol.

Baris 4. Uji apakah tabelnya penuh: $num[T] = size[T]$.

Baris 5–9. Bila ia penuh: alokasikan blok baru dengan $2 \cdot size[T]$ slot; salin tiap item dari blok lama ke blok baru; bebaskan blok lamanya; buat $table[T]$ menunjuk ke blok baru; dan catat ukuran barunya.

Baris 10–11. Sisipkan x ke blok mana pun yang kini ditunjuk $table[T]$, dan naikkan $num[T]$ sebanyak satu.

Biayanya. Kita mencacah satu satuan per **penyisipan dasar**: satu satuan untuk menuliskan satu item ke satu slot. Jadi:

- Bila masih ada ruang, biayanya tepat 1 — baris 10 menuliskan satu item.
- Bila sebuah perluasan terjadi, baris 6 menyalin $num[T]$ item (yang sama dengan $size[T]$ yang lama, sebab tabelnya penuh) lalu baris 10 menuliskan satu lagi. Jadi biayanya $num[T] + 1$, dan karena $num[T]$ tadinya $size[T]$ dan sesudah penyisipannya tabelnya memuat i item, biayanya tepat i dengan i adalah indeks penyisipan ini.

Dalam satu kalimat: *biaya operasi ke-i, bila masih tersedia ruang di dalam tabelnya, adalah $c_i = 1$. Bila tabelnya penuh, sebuah perluasan akan terjadi, yang menghasilkan biaya $c_i = i$.*

Kapan sebuah perluasan terjadi? Tabelnya penuh pada ukuran 1,2,4,8,16, … — ia berlipat dua setiap kali. Jadi penyisipan yang memicu perluasan adalah yang ke-2 (tabel berukuran 1 penuh), yang ke-3 (ukuran 2 penuh), yang ke-5 (ukuran 4 penuh), yang ke-9, yang ke-17, … Polanya adalah: **penyisipan i memicu perluasan persis ketika $i - 1$ tepat sebuah pangkat 2** (dengan menganggap penyisipan ke-1 sebagai kasus pemancing yang khusus). Sebutlah ini **syarat perluasan**.

Periksalah: $i = 2 \Rightarrow i - 1 = 1 = 2^0$ ✓; $i = 3 \Rightarrow 2 = 2^1$ ✓; $i = 5 \Rightarrow 4 = 2^2$ ✓; $i = 9 \Rightarrow 8 = 2^3$ ✓; $i = 4 \Rightarrow 3$, bukan pangkat 2, jadi tidak ada perluasan ✓.

Batas kasarnya. Satu penyisipan dapat berbiaya n, ada n di antaranya, totalnya $O(n^2)$. Seperti biasa, kita dapat berbuat jauh lebih baik.

14.7.3 Metode agregat diterapkan pada `TABLE-INSERT`

Gambar 14.17 mengerjakannya dalam satu tampilan.

$$\begin{aligned}\sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n\end{aligned}$$

Gambar 14.17 Analisis agregat `TABLE-INSERT`. Satu satuan untuk tiap penyisipan, ditambah biaya perluasannya $2^0 + 2^1 + \cdots + 2^{\lfloor \lg n \rfloor}$, yang kurang dari $2n$; totalnya di bawah $3n$.

Inilah penurunannya, dikembangkan.

Dasarnya. Tiap penyisipan menuliskan sekurang-kurangnya satu item, sehingga n penyisipan menyumbang sekurang-kurangnya n. Tulislah itu sebagai n di ruas kanan pertidaksamaannya.

Perluasannya. Ketika penyisipan i memicu sebuah perluasan, biaya tambahan di luar dasar yang 1 itu adalah banyaknya item yang disalin, yaitu $i-1$ — dan $i-1$ adalah sebuah pangkat 2, katakanlah 2^j. Jadi perluasan yang terjadi pada penyisipan 2^j+1 berbiaya tambahan 2^j.

j mana saja yang muncul? Yaitu yang memenuhi $2^j+1 \le n$, yakni $2^j \le n-1 < n$, yakni $j \le \lfloor \lg n \rfloor$.

Karena itu

$$\sum_{i=1}^{n} c_i \;\le\; n \;+\; \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j, \tag{14.13}$$

yang merupakan baris pertama **Gambar 14.17**.

Hitunglah jumlah geometri pada Persamaan (14.13) dengan rumus Persamaan (14.6) yang dibuktikan di §14.3.4, $\sum_{k=0}^{m} x^k = (x^{m+1}-1)/(x-1)$, dengan $x=2$ dan $m=\lfloor \lg n \rfloor$:

$$\sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \;=\; 2^{\lfloor \lg n \rfloor + 1} - 1 \;<\; 2^{\lg n + 1} \;=\; 2n.$$

Jadi

$$\sum_{i=1}^{n} c_i \;<\; n + 2n \;=\; 3n,$$

baris kedua dan ketiga **Gambar 14.17**. Karena itu biaya total n penyisipan adalah $O(n)$, dan biaya amortisasi per penyisipannya adalah

$$\frac{3n}{n} \;=\; 3 \;=\; O(1).$$

Singkatnya: *untuk n operasi, kompleksitas kasus terburuknya adalah $O(n)$; biaya teramortisasi per operasinya adalah $O(1)$.*

> **MENGAPA MELIPATDUAKAN DAN BUKAN MENAMBAHKAN.** Seluruh argumennya bertumpu pada biaya perluasannya yang membentuk deret *geometri*: 1,2,4,8, …, masing-masing dua kali yang sebelumnya. Deret geometri yang berlipat dua berjumlah kurang dari dua kali suku terbesarnya, sehingga jumlah *seluruh* penyalinan yang pernah dikerjakan kurang dari dua kali ukuran tabel akhirnya. Seandainya sebagai gantinya Anda menumbuhkan tabelnya sebanyak jumlah tetap — katakanlah 100 slot setiap kali — biaya perluasannya akan menjadi 100,200,300, …, sebuah deret *aritmetika*, yang jumlahnya $\Theta(n^2/200)$. Soal 14.10 dan **Kode 14.13** membuat hal ini nyata. **Pelipatannya bukan rincian implementasi; ia seluruh gagasannya.**

14.7.4 Metode akuntansi diterapkan pada `TABLE-INSERT`

Argumen akuntansinya padat dengan indah:

Dengan Metode Akuntansi, satu operasi `TABLE-INSERT` tunggal memiliki biaya teramortisasi 3. Tiap item membayar tiga peristiwa yang berkaitan dengan penyisipan:

1. penyisipan item itu sendiri ke dalam tabelnya,
2. pemindahan item itu ketika tabelnya diperluas,
3. pemindahan item lain yang sebelumnya sudah pernah dipindahkan sekali.

Mari kita buat itu konkret, sebab anak kalimat ketiganya yang cerdik dan mudah terlewat begitu saja.

Tagihlah tiap penyisipan 3 satuan. Satu dibelanjakan seketika untuk menuliskan itemnya ke slotnya. Dua ditabung *pada item itu*.

Sekarang tinjaulah perluasan berikutnya. Tabelnya penuh pada ukuran s lalu berlipat dua menjadi $2s$. Seluruh s item harus disalin. Siapa yang membayar?

Tengoklah saat tabelnya mencapai ukuran s. Tepat sesudah perluasan *itu*, tabelnya memuat $s/2$ item dan punya s slot. Jadi $s/2$ dari item yang hadir pada perluasan berikutnya adalah item **lama** — mereka sudah ada sebelumnya — dan $s/2$ adalah item **baru**, yang disisipkan sejak itu. Hanya yang baru yang masih punya tabungan; yang lama sudah membelanjakan tabungannya pada perluasan sebelumnya.

Tiap item baru membawa 2 satuan tabungan. Ada $s/2$ di antaranya, sehingga bersama-sama mereka punya $2 \times (s/2) = s$ satuan — persis yang diperlukan untuk menyalin seluruh s item. Jadi tiap item baru membayar untuk **dirinya sendiri** disalin (anak kalimat 2) dan untuk **satu item lama** disalin (anak kalimat 3). Itulah peristiwa ketiga dalam daftar di atas, dan itulah sebabnya tagihannya 3 dan bukan 2.

Invarian kreditnya. Sesudah sembarang awalan penyisipan, kredit yang tersimpan di dalam tabelnya adalah $2 \times$ (banyaknya item yang disisipkan sejak perluasan terakhir), yang merupakan sebuah cacah dan karena itu tidak mungkin negatif. Karena itu kreditnya tidak pernah menjadi negatif, karena itu tagihan 3 itu sah, karena itu n penyisipan berbiaya paling besar $3n$.

Jadi untuk n operasi kompleksitas kasus terburuknya adalah $O(n)$, dan biaya teramortisasi per operasinya adalah $O(1)$. Batas yang sama, metode kedua.

14.7.5 Metode potensial diterapkan pada `TABLE-INSERT`

Metode potensial menjatuhkan pilihan ini:

$$\Phi(T) \;=\; 2 \cdot num[T] \;-\; size[T]. \tag{14.14}$$

Potensial Persamaan (14.14) adalah salah satu fungsi potensial yang paling termasyhur dalam pokok bahasan ini, dan ia membalas semenit perenungan tentang *mengapa* ia benar.

Membacanya. Tepat sesudah sebuah perluasan, tabelnya separuh penuh: $num = size/2$, sehingga $\Phi = 2(size/2) - size = 0$. Tabelnya berada dalam keadaan teraman dan potensialnya nol. Seiring item disisipkan, num mendaki sementara $size$ diam di tempat, sehingga Φ mendaki — tepat 2 per penyisipan. Ketika tabelnya menjadi penuh, $num = size$ dan $\Phi = 2\,size - size = size$: potensialnya kini sama dengan banyaknya item yang harus disalin oleh perluasan berikutnya. **Potensialnya telah menabung persis sebesar biaya bencana yang sudah terlihat mendekat.**

Periksa syaratnya. *Nilai awal fungsi potensial ini adalah 0* — sebab tabel kosong punya $num = size = 0$. *Karena tabelnya selalu memuat sekurang-kurangnya separuh kapasitasnya, yakni* $num[T] \geq size[T]/2$, *maka* $\Phi(T)$ *selalu tak-negatif.* Dan mengapa tabelnya selalu sekurang-kurangnya separuh penuh? Sebab ia persis separuh penuh tepat sesudah perluasan terakhirnya

(kita melipatduakannya dari keadaan penuh), dan penyisipan hanya menaikkan num. Jadi $2\,num \geq size$, yakni $\Phi \geq 0$. Kedua syaratnya berlaku.

Sekarang kedua kasusnya, yang dirangkum **Gambar 14.18**. Tulislah num_i dan $size_i$ untuk nilainya sesudah penyisipan ke-i.

Potensial: $\Phi(T) = 2 \cdot num[T] - size[T]$.

Tanpa perluasan pada penyisipan ke-i ($size_i = size_{i-1}$, biaya sebenarnya 1):

$$a_i = 1 + (2\,num_i - size_i) - (2\,num_{i-1} - size_{i-1}) = 1 + 2 = 3.$$

Dengan perluasan pada penyisipan ke-i ($size_i/2 = size_{i-1} = num_i - 1$, biaya sebenarnya num_i):

$$\begin{aligned} a_i &= num_i + (2\,num_i - size_i) - (2\,num_{i-1} - size_{i-1}) \\ &= num_i + \big(2\,num_i - 2(num_i - 1)\big) - \big(2(num_i - 1) - (num_i - 1)\big) \\ &= num_i + 2 - (num_i - 1) = 3. \end{aligned}$$

Pada kedua kasus biaya amortisasi `TABLE-INSERT` adalah 3, yakni $O(1)$.

Gambar 14.18 Kedua kasus analisis potensial `TABLE-INSERT` menghasilkan tepat 3. Pada kasus tanpa perluasan, biaya sebenarnya 1 dan potensialnya naik 2; pada kasus dengan perluasan, biaya sebenarnya num_i dan potensialnya ambruk sebesar $num_i - 3$.

Kasus 1: tidak ada perluasan. Maka $size_i = size_{i-1}$ dan $num_i = num_{i-1} + 1$, dan biaya sebenarnya $c_i = 1$.

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (2\,num_i - size_i) - (2\,num_{i-1} - size_{i-1}) \\ &= 1 + 2\,num_i - size_i - 2(num_i - 1) + size_i \\ &= 1 + 2\,num_i - 2\,num_i + 2 - size_i + size_i \\ &= 1 + 2 \\ &= 3. \end{aligned}$$

Dalam satu baris: $\hat{c}_i = 1 + (2\,num_i - size_i) - (2\,num_{i-1} - size_{i-1}) = 3$.

Kasus 2: sebuah perluasan terjadi. Ada dua hubungan yang berlaku:

$$\frac{size_i}{2} \;=\; size_{i-1} \;=\; num_i - 1.$$

Bacalah itu sebagai dua fakta. *Pertama*, $size_i = 2\,size_{i-1}$ — kita melipatduakannya. *Kedua*, $size_{i-1} = num_i - 1$ — tabelnya penuh sebelum penyisipan ini, sehingga ia memuat $size_{i-1}$ item, dan sesudah penyisipannya ia memuat satu lagi, yaitu num_i. Juga $num_{i-1} = num_i - 1$.

Biaya sebenarnya adalah $c_i = num_i$: kita menyalin $num_i - 1$ item yang sudah ada di sana, lalu menuliskan yang baru.

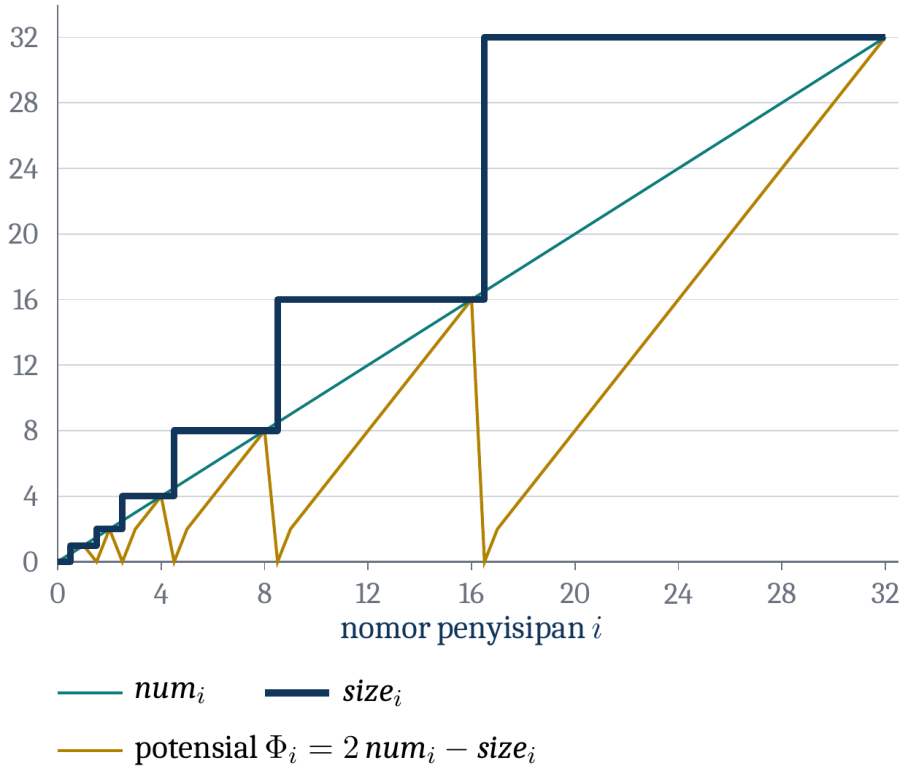
$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= num_i + (2\,num_i - size_i) - (2\,num_{i-1} - size_{i-1}) \\
&= num_i + 2\,num_i - 2\,size_{i-1} - 2(num_i - 1) + size_{i-1} \\
&= num_i + 2\,num_i - 2(num_i - 1) - 2\,num_i + 2 + (num_i - 1) \\
&= num_i + 2\,num_i - 2\,num_i + 2 - 2\,num_i + 2 + num_i - 1 \\
&= 3.
\end{aligned}
$$

(Pada baris keempat, baik $size_i = 2\,size_{i-1}$ maupun $size_{i-1} = num_i - 1$ sudah disubstitusikan. Kumpulkan suku num_i: $1 + 2 - 2 - 2 + 1 = 0$. Kumpulkan konstantanya: $2 + 2 - 1 = 3$.)

Kedua kasusnya menghasilkan tepat **3**. Itu bukan batas atas yang masih longgar; itu sebuah persamaan. Karena itu biaya teramortisasi operasi `TABLE-INSERT` adalah $O(1)$.

Gambarnya

Gambar 14.19 memplot ketiga besarannya sepanjang jalan berisi 32 penyisipan, dan inilah gambar paling berguna dalam bab ini.



Gambar 14.19 Sebuah barisan penyisipan: diagonal tipis adalah *num*, tangga tebal adalah *size*, dan gigi gergajinya adalah $\Phi = 2\,num - size$, yang jatuh ke 0 pada tiap perluasan lalu mendaki dengan kemiringan 2 sampai menemui *size* lagi.

Telusurilah gigi gergajinya dengan jari Anda. Tepat sebelum sebuah perluasan, $num = size$ dan $\Phi = size$ — potensialnya sudah mendaki sampai setinggi tangganya. Perluasannya meletus: *size* melonjak ke $2\,size$, dan Φ jatuh seketika ke 0, sebab tabel yang baru berlipat dua itu persis separuh penuh. Lalu pendakiannya dimulai lagi, dengan kemiringan 2, sebab tiap penyisipan menambahkan 1 kepada *num* dan karenanya 2 kepada $2\,num$.

Gambarnya adalah buktinya. Potensialnya adalah rekening tabungan. Ia terisi mantap selama tidak ada yang dramatis terjadi, dan ia dikosongkan dalam sekejap oleh perluasannya — yang persis gunanya tabungan.

Programnya

Kode 14.8 mengimplementasikan `TABLE-INSERT` dengan `malloc`, `memcpy`, dan `free` yang sungguhan, sehingga penyalinannya adalah kerja nyata dan bukan simulasi, lalu mencetak c_i, Φ sebelum, Φ sesudah, dan $\hat{c}_i$ untuk tujuh belas penyisipan pertamanya.

Kode 14.8 `TABLE-INSERT` dengan pelipatan, diinstrumentasi dengan fungsi potensialnya. Kompilasi dengan `g++ -std=c++17 -O2 -o dyntable dyntable.cpp`.

```
// TABLE-INSERT dengan pelipatan, diinstrumentasi tiga cara sekaligus:
//   * biaya sebenarnya c_i dari tiap penyisipan,
//   * total berjalannya, yang dibatasi metode agregat oleh 3n,
//   * potensial Phi(T) = 2*num - size dan biaya amortisasi
//     a_i = c_i + Phi(T_i) - Phi(T_{i-1}), yang dibuktikan teks bernilai 3.
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <cstring>
using namespace std;

class DynamicTable {
public:
    ~DynamicTable() { free(slot); }

    // Mengembalikan biaya sebenarnya penyisipan ini: 1 bila masih ada ruang,
    // bila tidak num_i (satu tulis untuk x ditambah num_{i-1} salinan).
    long long insert(int x) {
        long long c = 0;
        if (size_ == 0) { slot = (int*)malloc(sizeof(int)); size_ = 1; }
        if (num_ == size_) {
            int* fresh = (int*)malloc(sizeof(int) * 2 * size_);
            memcpy(fresh, slot, sizeof(int) * size_);   // num_ == size_ salinan
            c += size_;
            free(slot); slot = fresh; size_ *= 2;
            ++expansions;
        }
        slot[num_++] = x; c += 1;
        totalActual += c; ++nOps;
        return c;
    }

    long long potential() const { return 2 * num_ - size_; }
    long long num() const { return num_; }
    long long size() const { return size_; }
    long long actual() const { return totalActual; }
    long long ops() const { return nOps; }
    long long expansionCount() const { return expansions; }

private:
    int* slot = nullptr;
    long long num_ = 0, size_ = 0, totalActual = 0, nOps = 0, expansions = 0;
};
```

Kode 14.8 (Lanjutan) *TABLE-INSERT dengan pelipatan, diinstrumentasi dengan fungsi potensialnya. Kompilasi dengan g++ -std=c++17 -O2 -o dyntable dyntable.cpp.*

```
int main() {
    DynamicTable T;
    cout << " i   num  size  actual c_i   Phi before  Phi after  amortized a_i  running
total\n";
    for (int i = 1; i <= 17; ++i) {
        long long before = T.potential();
        long long c = T.insert(i);
        long long after = T.potential();
        cout << setw(3) << i << setw(6) << T.num() << setw(6) << T.size()
             << setw(12) << c << setw(13) << before << setw(11) << after
             << setw(15) << c + after - before << setw(15) << T.actual() << "\n";
    }

    DynamicTable U;
    const long long n = 1000000;
    for (long long i = 0; i < n; ++i) U.insert((int)i);
    cout << "\nn = " << n << " insertions into a table that starts empty:\n";
    cout << "  expansions performed   = " << U.expansionCount() << "\n";
    cout << "  total actual cost      = " << U.actual() << "\n";
    cout << "  aggregate bound 3n     = " << 3 * n << "\n";
    cout << "  measured cost per insert = " << fixed << setprecision(6)
         << (double)U.actual() / (double)U.ops() << "\n";
    cout << "  final num = " << U.num() << ", final size = " << U.size()
         << ", final potential = " << U.potential() << "\n";
    return 0;
}
```

```
 i   num  size  actual c_i   Phi before  Phi after  amortized a_i  running total
 1     1     1          1            0          1              2              1
 2     2     2          2            1          2              3              3
 3     3     4          3            2          2              3              6
 4     4     4          1            2          4              3              7
 5     5     8          5            4          2              3             12
 6     6     8          1            2          4              3             13
 7     7     8          1            4          6              3             14
 8     8     8          1            6          8              3             15
 9     9    16          9            8          2              3             24
10    10    16          1            2          4              3             25
11    11    16          1            4          6              3             26
12    12    16          1            6          8              3             27
13    13    16          1            8         10              3             28
14    14    16          1           10         12              3             29
15    15    16          1           12         14              3             30
16    16    16          1           14         16              3             31
17    17    32         17           16          2              3             48

n = 1000000 insertions into a table that starts empty:
  expansions performed   = 20
  total actual cost      = 2048575
  aggregate bound 3n     = 3000000
  measured cost per insert = 2.048575
  final num = 1000000, final size = 1048576, final potential = 951424
```

Lajur `amortized a_i` adalah dinding berisi angka 3 — penurunannya berkata angkanya akan tepat 3, dan ia tepat 3, pada tiap langkah kecuali penyisipan pemancingnya yang tidak punya tabel untuk disalin. Lajur `actual c_i` adalah 1,2,3,1,5,1,1,1,9, … ,17: mahal persis pada $i = 2,3,5,9,17$, yang adalah $1 + 2^j$ sebagaimana diramalkan. Amatilah potensialnya pada $i = 16$: ia mencapai 16, persis banyaknya item yang harus disalin perluasan berikutnya — dan pada $i = 17$ ia ambruk menjadi 2.

Sejuta penyisipan berbiaya 2.048.575 satuan melawan langit-langit terbukti sebesar 3.000.000, dan hanya melakukan 20 perluasan. Dua puluh. Pelipatan itu luar biasa mujarab.

14.7.6 Penghapusan, dan godaan menyeparuhkan pada separuh

Sekarang tambahkan `TABLE-DELETE`, yang membuang sebuah item dan membebaskan slotnya. Bila kita tidak pernah menyusutkan tabelnya, biaya amortisasi sebuah penghapusan secara sepele adalah $O(1)$ — tetapi kemudian sebuah tabel yang tumbuh sampai sejuta entri lalu 999.999 di antaranya dihapus akan tetap memegang sejuta slot memori. Itu melanggar sasaran kedua di §14.7.1: *banyaknya ruang yang tak terpakai di dalam tabelnya tidak pernah melampaui suatu faktor konstanta tertentu relatif terhadap seluruh ruang yang dialokasikan.*

Jadi kita harus menyusutkannya. Kebijakan yang gamblang adalah bayangan cermin dari perluasannya:

Kebijakan A. Ketika sebuah penghapusan meninggalkan tabelnya kurang dari separuh penuh, separuhkanlah ukurannya: alokasikan blok berisi $size/2$ slot, salin num item yang tersisa ke sana, lalu bebaskan blok lamanya.

Ia kelihatan benar. Ia salah, dan cara ia salah layak dipahami secara rinci, sebab ia jebakan yang baku dan ia alasan mengapa implementasi sungguhan memakai konstanta yang berbeda.

Barisan *thrashing*-nya, dituliskan

Berikut sebuah barisan operasi yang menghancurkan Kebijakan A. Andaikan tabelnya baru saja menjadi tepat penuh: $num = size = s$, dengan s sebuah pangkat dua. Sekarang lakukanlah, berulang-ulang, blok empat operasi

$$\text{INSERT}, \quad \text{DELETE}, \quad \text{DELETE}, \quad \text{INSERT}.$$

Ikutilah satu operasi demi satu operasi.

1. `INSERT`. Tabelnya penuh ($num = size = s$), sehingga ia meluas. Ukuran baru $2s$; s item disalin; satu ditulis. Biaya $s + 1$. Keadaan: $num = s + 1$, $size = 2s$.

2. `DELETE`. Buang satu item. Kini $num = s$ dan $size = 2s$, sehingga faktor muatannya tepat $1/2$. "Kurang dari separuh penuh" **salah** — s tidak kurang dari s — sehingga tidak ada penyusutan. Biaya 1. Keadaan: $num = s$, $size = 2s$.

3. `DELETE`. Buang satu lagi. Kini $num = s - 1$ dan $size = 2s$, dan $s - 1 < s$, sehingga tabelnya *memang* kurang dari separuh penuh. Susutkan: alokasikan s slot, salin $s - 1$ item yang tersisa. Biaya $1 + (s - 1) = s$. Keadaan: $num = s - 1$, $size = s$.

4. `INSERT`. $num = s - 1$ kurang dari $size = s$, sehingga masih ada ruang. Biaya 1. Keadaan: $num = s$, $size = s$ — **persis di tempat kita bermula**.

Bloknya adalah sebuah daur. Ia berbiaya

$$(s + 1) + 1 + s + 1 \;=\; 2s + 3$$

dan terdiri atas empat operasi. Jadi biaya per operasinya kira-kira $s/2$.

Sekarang buatlah s tumbuh bersama persoalannya. Ambillah sebuah barisan berisi n operasi: pakailah $n/4$ yang pertama untuk mengisi sebuah tabel sampai $num = size = n/4$, lalu pakailah

$3n/4$ operasi sisanya untuk mengulang bloknya. Tiap blok berisi 4 operasi berbiaya sekitar $2 \times (n/4) = n/2$, dan ada sekitar $3n/16$ blok, sehingga totalnya sekitar

$$\frac{3n}{16} \times \frac{n}{2} = \frac{3n^2}{32} = \Theta(n^2).$$

Kebijakan A berbiaya $\Theta(n)$ teramortisasi per operasi, bukan $O(1)$. Batas kasarnya ternyata tidak terlalu pesimistis — untuk kebijakan ini ia tercapai.

> **MENGAPA IA GAGAL: STRUKTURNYA TIDAK PUNYA KELONGGARAN.** Sesudah sebuah perluasan, tabelnya duduk pada faktor muatan tepat 1/2 — yang merupakan titik yang *sama* dengan pemicu penyusutannya. Satu penghapusan sudah cukup untuk mendorongnya melewati tepinya, dan satu penyisipan sudah cukup untuk mendorongnya kembali. Strukturnya berdiri di atas mata pisau, dan tiap operasi menjungkirkannya. Perluasannya dan penyusutannya masing-masing mengerjakan kerja $\Theta(s)$, dan tidak satu pun sudah dibayari, sebab tidak ada rentetan operasi murah di antaranya untuk melakukan pembayaran itu. **Sebuah batas amortisasi memerlukan ruang di antara titik-titik pemicunya.**

Perbaikannya

Kebijakan B. Luaskan (lipatduakan) ketika tabelnya menjadi penuh, seperti sebelumnya. Susutkan (separuhkan) hanya ketika sebuah penghapusan meninggalkan tabelnya **kurang dari seperempat penuh**.

Sekarang ada kelonggaran. Tepat sesudah sebuah perluasan, faktor muatannya 1/2; pemicu penyusutannya berada pada 1/4. Untuk beranjak dari yang satu ke yang lain Anda harus menghapus seperempat kapasitas tabelnya — artinya, $size/4$ operasi harus berlalu sebelum sebuah penyusutan dapat meletus. Tepat sesudah sebuah penyusutan, faktor muatannya 1/2 lagi (kita menyeparuhkan tabel yang seperempat penuh, sehingga $num = size_{\text{baru}}/2$); untuk mencapai pemicu perluasannya pada faktor muatan 1 Anda harus menyisipkan $size/2$ item lagi.

Pada kedua arah, sebuah peristiwa $\Theta(size)$ hanya dapat terjadi sesudah $\Theta(size)$ operasi murah membayarinya. Itulah persis syarat yang diperlukan sebuah batas amortisasi.

Faktor muatannya kini terkurung: ia tidak pernah naik di atas 1 dan tidak pernah jatuh di bawah 1/4 (kecuali untuk tabel yang begitu kecil sehingga ia hanya punya satu slot). Jadi sekurang-kurangnya seperempat ruang teralokasinya selalu terpakai, dan **ruang yang terbuang tidak pernah melampaui tiga perempat alokasinya** — sebuah faktor konstanta, yang merupakan sasaran kedua di §14.7.1, tercapai.

Mengukur kedua kebijakannya

Kode 14.9 mengimplementasikan kedua kebijakannya lalu menjalankan barisan *thrashing*-nya pada masing-masing, pada lima skala yang berbeda.

Kode 14.9 Kedua kebijakan penyusutan dibandingkan pada barisan *thrashing*-nya. Kompilasi dengan `g++ -std=c++17 -O2 -o dyntable2 dyntable2.cpp`.

```
// Tabel dinamis yang menyusut sekaligus tumbuh.  Dua kebijakan penyusutan
// dijalankan pada barisan yang persis sama:
//   KEBIJAKAN A -- separuhkan tabelnya begitu ia kurang dari separuh penuh;
//   KEBIJAKAN B -- separuhkan tabelnya hanya bila kurang dari seperempat penuh.
// Barisannya adalah barisan thrashing dari teks: isi tabelnya sampai
// persis penuh, lalu ulangi blok  INSERT, DELETE, DELETE, INSERT.
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

struct Table {
    long long num = 0, size = 0;
    long long actual = 0, ops = 0, expansions = 0, contractions = 0, worstOp = 0;
    int shrinkDenominator;                // 2 untuk kebijakan A, 4 untuk kebijakan B

    explicit Table(int d) : shrinkDenominator(d) {}

    void charge(long long c) { actual += c; ++ops; if (c > worstOp) worstOp = c; }

    void insert() {
        long long c = 0;
        if (size == 0) size = 1;
        if (num == size) { c += size; size *= 2; ++expansions; }  // salin `size` item
        ++num; c += 1;
        charge(c);
    }

    void remove() {
        if (num == 0) { charge(0); return; }
        long long c = 1;                                          // bebaskan satu slot
        --num;
        if (size > 1 && num * shrinkDenominator < size) {
            c += num;                                             // salin `num` item
            size /= 2; ++contractions;
        }
        charge(c);
    }
```

Kode 14.9 (Lanjutan) *Kedua kebijakan penyusutan dibandingkan pada barisan thrashing-nya. Kompilasi dengan g++ -std=c++17 -O2 -o dyntable2 dyntable2.cpp.*

```
    // Potensial untuk kasus gabungannya, sebagaimana diturunkan di teks.
    long long potential() const {
        if (size == 0) return 0;
        if (2 * num >= size) return 2 * num - size;
        return size / 2 - num;
    }
};

void run(const string& name, int denom, long long full, long long blocks) {
    Table T(denom);
    while (T.num < full) T.insert();          // kini num == size == full
    long long c0 = T.actual, o0 = T.ops, e0 = T.expansions, k0 = T.contractions;
    for (long long b = 0; b < blocks; ++b) { T.insert(); T.remove(); T.remove();
T.insert(); }
    long long cost = T.actual - c0, nops = T.ops - o0;
    cout << left << setw(22) << name << right
         << " table=" << setw(8) << full
         << " ops=" << setw(9) << nops
         << " cost=" << setw(12) << cost
         << " cost/op=" << setw(11) << fixed << setprecision(2)
         << (double)cost / (double)nops
         << "  expansions=" << setw(8) << T.expansions - e0
         << " contractions=" << T.contractions - k0 << "\n";
}

int main() {
    cout << "THRASHING TEST.  Fill the table to exactly full, then repeat the\n"
            "block  INSERT, DELETE, DELETE, INSERT  until n/2 operations are done.\n\n";
    for (long long n : {1024LL, 4096LL, 16384LL, 65536LL, 262144LL}) {
        long long full = n / 4, blocks = (n / 2) / 4;
        cout << "n = " << setw(7) << n << "\n";
        run("  A: shrink below 1/2", 2, full, blocks);
        run("  B: shrink below 1/4", 4, full, blocks);
    }

    cout << "\nPotential and worst case on a long mixed run "
            "(400000 inserts, then 400000 deletes):\n";
    for (int denom : {2, 4}) {
        Table T(denom);
        long long lowest = 0;
        for (int i = 0; i < 400000; ++i) { T.insert(); if (T.potential() < lowest) lowest =
T.potential(); }
        for (int i = 0; i < 400000; ++i) { T.remove(); if (T.potential() < lowest) lowest =
T.potential(); }
        cout << "  shrink below 1/" << denom
             << "   cost=" << setw(9) << T.actual
             << "  cost/op=" << fixed << setprecision(4)
             << (double)T.actual / (double)T.ops
             << "  worst single op=" << setw(8) << T.worstOp
```

Kode 14.9 (Lanjutan) *Kedua kebijakan penyusutan dibandingkan pada barisan thrashing-nya. Kompilasi dengan g++ -std=c++17 -O2 -o dyntable2 dyntable2.cpp.*

```
             << "  lowest potential=" << lowest
             << "  final num=" << T.num << " size=" << T.size << "\n";
    }
    return 0;
}
```

```
THRASHING TEST.  Fill the table to exactly full, then repeat the
block  INSERT, DELETE, DELETE, INSERT  until n/2 operations are done.

n =    1024
  A: shrink below 1/2  table=     256 ops=      512 cost=       65920 cost/op=     128.75
expansions=     128 contractions=128
  B: shrink below 1/4  table=     256 ops=      512 cost=         768 cost/op=       1.50
expansions=       1 contractions=0
n =    4096
  A: shrink below 1/2  table=    1024 ops=     2048 cost=     1050112 cost/op=     512.75
expansions=     512 contractions=512
  B: shrink below 1/4  table=    1024 ops=     2048 cost=        3072 cost/op=       1.50
expansions=       1 contractions=0
n =   16384
  A: shrink below 1/2  table=    4096 ops=     8192 cost=    16783360 cost/op=    2048.75
expansions=    2048 contractions=2048
  B: shrink below 1/4  table=    4096 ops=     8192 cost=       12288 cost/op=       1.50
expansions=       1 contractions=0
n =   65536
  A: shrink below 1/2  table=   16384 ops=    32768 cost=   268460032 cost/op=    8192.75
expansions=    8192 contractions=8192
  B: shrink below 1/4  table=   16384 ops=    32768 cost=       49152 cost/op=       1.50
expansions=       1 contractions=0
n =  262144
  A: shrink below 1/2  table=   65536 ops=   131072 cost=  4295065600 cost/op=   32768.75
expansions=   32768 contractions=32768
  B: shrink below 1/4  table=   65536 ops=   131072 cost=      196608 cost/op=       1.50
expansions=       1 contractions=0

Potential and worst case on a long mixed run (400000 inserts, then 400000 deletes):
  shrink below 1/2   cost=  1848555  cost/op=2.3107  worst single op=  262145  lowest
potential=0  final num=0 size=1
  shrink below 1/4   cost=  1586412  cost/op=1.9830  worst single op=  262145  lowest
potential=0  final num=0 size=2
```

Inilah percobaan yang paling jernih dalam bab ini. Bacalah lajur `cost/op` bagi Kebijakan A dari atas ke bawah: 128.75, 512.75, 2048.75, 8192.75, 32768.75. Setiap kali n dikalikan 4, biaya per operasinya dikalikan 4. Itu pertumbuhan linear dalam n: biaya amortisasi per operasi Kebijakan A adalah $\Theta(n)$, persis sebagaimana diramalkan analisis tangannya. Perhatikan pula lajur `contractions` — Kebijakan A melakukan 32.768 penyusutan pada jalan yang terbesar, satu pada tiap blok, masing-masing menyalin puluhan ribu item.

Sekarang bacalah Kebijakan B: 1.50 pada tiap skala. Konstan. Ia melakukan **satu** perluasan dalam seluruh jalannya dan **nol** penyusutan, sebab blok empat operasinya tidak pernah menggiring faktor muatannya turun sejauh 1/4.

> **PELAJARAN MORALNYA, YANG BERLAKU JAUH DI LUAR TABEL.** Ketika Anda membangun sebuah struktur yang menata ulang dirinya pada sebuah ambang, **ambang untuk tumbuh dan ambang untuk menyusut tidak boleh berada di titik yang sama.** Sisakanlah jarak — histeresis, kata seorang insinyur — sehingga strukturnya harus didorong jauh sebelum ia menata ulang dirinya lagi. Termostat bekerja atas asas yang sama, dan itulah sebabnya penyejuk udara Anda tidak menyala dan mati sekali tiap detik.

14.7.7 Fungsi potensial untuk kasus gabungannya

Sebagai catatan, berikut fungsi potensial yang membuktikan Kebijakan B $O(1)$ teramortisasi. Inilah yang diimplementasikan pada `Table::potential()` di **Kode 14.9**. Misalkan $\alpha = num/size$ adalah faktor muatannya. Definisikan

$$\Phi(T) = \begin{cases} 2\,num[T] - size[T] & \text{bila } \alpha(T) \geq \frac{1}{2}, \\ \frac{size[T]}{2} - num[T] & \text{bila } \alpha(T) < \frac{1}{2}. \end{cases} \tag{14.15}$$

Tiga hal yang perlu diperhatikan pada Persamaan (14.15).

Ia nol ketika tabelnya tepat separuh penuh. Masukkan $num = size/2$ ke salah satu cabangnya: yang pertama memberikan $2(size/2) - size = 0$, yang kedua memberikan $size/2 - size/2 = 0$. Jadi kedua cabangnya sepakat di sambungannya, dan fungsinya tidak melompat.

Ia tak-negatif di mana-mana. Pada cabang atasnya, $\alpha \geq 1/2$ berarti $2\,num \geq size$, sehingga $2\,num - size \geq 0$. Pada cabang bawahnya, $\alpha < 1/2$ berarti $num < size/2$, sehingga $size/2 - num > 0$.

Ia naik menuju kedua bencananya. Seiring tabelnya terisi menuju $\alpha = 1$, cabang pertamanya mendaki ke $size$ — biaya perluasannya. Seiring tabelnya kosong menuju $\alpha = 1/4$, cabang keduanya mendaki ke $size/2 - size/4 = size/4$ — biaya penyusutannya, yang menyalin $size/4$ item yang bertahan. Pada kedua arah, potensialnya menabung persis sebesar biaya peristiwa yang akan datang.

Analisis kasus lengkapnya panjang — ada empat kasus untuk penyisipan (meluas atau tidak, melintasi garis separuh-penuh atau tidak) dan empat untuk penghapusan — dan ia diberikan sebagai Soal 14.12 lengkap dengan penyelesaiannya. Hasil utamanya, yang kita kutip dan yang didukung pengukuran di atas, adalah:

> **TEOREMA (tabel dinamis).** Pada sebuah tabel dinamis yang melipatduakan dirinya ketika menjadi penuh dan menyeparuhkan dirinya ketika jatuh di bawah seperempat penuh, bermula dari tabel kosong, sembarang barisan berisi n operasi `TABLE-INSERT` dan `TABLE-DELETE` memakan waktu $O(n)$ seluruhnya. Biaya amortisasi tiap operasinya adalah $O(1)$, dan faktor muatannya tidak pernah jatuh di bawah $1/4$, sehingga tidak pernah lebih dari tiga perempat ruang teralokasinya terbuang.

Itulah kedua sasaran di §14.7.1, tertunaikan.

14.8 Union–Find: Janji dari Bab 11, Ditepati

14.8.1 Apa yang sedang kita analisis

Bab 11 membangun sebuah **hutan himpunan saling lepas** — gagasan yang bermula dari Galler dan Fisher [6], dan diterangkan dengan lembut oleh Kleinberg dan Tardos [7] — untuk membuat algoritma Kruskal efisien. Ingatlah kembali strukturnya, secara ringkas, sehingga subbab ini berdiri sendiri.

Kita punya n elemen. Mereka dipartisi menjadi himpunan-himpunan yang saling lepas. Tiap himpunan disimpan sebagai sebuah pohon, dan akar pohonnya adalah **wakil** himpunan itu. Tiap elemen x punya satu penunjuk, $parent[x]$; sebuah akar menunjuk ke dirinya sendiri. Tiga operasi:

- `MakeSet(x)` — ciptakan himpunan berisi satu elemen: $parent[x] \leftarrow x$.
- `FindSet(x)` — kembalikan wakil himpunan x, dengan menyusuri penunjuk induk naik sampai ke akarnya.
- `Union(x,y)` — leburkan kedua himpunan yang memuat x dan y, dengan membuat satu akar menunjuk ke akar yang lain.

Biaya `FindSet(x)` adalah banyaknya penunjuk yang disusurinya, yaitu kedalaman x di dalam pohonnya. Jadi seluruh pertanyaannya adalah: **seberapa dalam pohonnya dapat menjadi?**

Tanpa kehati-hatian, sangat dalam. Bila Anda menggabungkan $\{1\}$ dengan $\{2\}$, lalu hasilnya dengan $\{3\}$, lalu dengan $\{4\}$, selalu mengarahkan akar lamanya ke elemen barunya, Anda membangun sebuah lintasan sepanjang $n-1$, dan tiap `FindSet` atas elemen terbawahnya berbiaya $\Theta(n)$.

14.8.2 Akal-akalan 1: gabung menurut peringkat

Simpanlah, bersama tiap akar x, sebuah bilangan bulat $rank[x]$ — sebuah batas atas bagi tinggi pohon x. Pada mulanya tiap peringkat 0. Di dalam `Union`, **buatlah akar berperingkat lebih kecil menunjuk ke akar berperingkat lebih besar**; bila peringkatnya sama, pilihlah salah satunya, lalu naikkan peringkat pemenangnya sebanyak 1.

Mengapa ia berhasil. Kedalaman pohon gabungannya adalah kedalaman yang lebih dalam di antara keduanya, kecuali bila keduanya sama dalam, yang dalam hal itu ia bertambah satu. Gabung menurut peringkat membuat pohon yang lebih dangkal tergantung di bawah yang lebih dalam, sehingga kedalaman yang lebih dalam itu sama sekali tidak berubah. Kedalamannya hanya dapat tumbuh ketika dua pohon berperingkat *sama* bertemu.

Batas yang diberikannya. Sebuah pohon berperingkat r memuat sekurang-kurangnya 2^r simpul. Bukti dengan induksi atas r. *Basis:* sebuah pohon berperingkat 0 adalah satu simpul tunggal, dan $2^0 = 1$. *Langkah:* sebuah pohon berperingkat r hanya tercipta ketika dua pohon berperingkat $r-1$ digabungkan; menurut hipotesis induksinya masing-masing memuat sekurang-kurangnya 2^{r-1} simpul, sehingga gabungannya memuat sekurang-kurangnya $2^{r-1} + 2^{r-1} = 2^r$ simpul. ■

Karena tidak ada pohon yang dapat memuat lebih dari n simpul, $2^r \leq n$, sehingga $r \leq \lg n$. Tingginya paling besar sama dengan peringkatnya, sehingga **tiap `FindSet` berbiaya paling besar** $\lg n$, dan ini batas kasus terburuk yang sungguhan tanpa amortisasi apa pun. Itu saja sudah membawa Kruskal dari $O(E\,V)$ ke $O(E \lg V)$.

14.8.3 Akal-akalan 2: pemampatan lintasan

Berikut gagasan keduanya, dan ia nyaris tidak berongkos untuk diimplementasikan. Ketika `FindSet(x)` berjalan dari x naik ke akarnya r, ia melewati tiap simpul pada lintasan itu. Dalam perjalanan turunnya, **arahkanlah setiap simpul itu langsung ke r.**

Pencarian yang melakukan ini membayar dua kali lipat — satu sapuan naik, satu sapuan turun — tetapi tiap simpul pada lintasan itu kini berada pada kedalaman 1, dan tiap pencarian berikutnya yang menyentuh salah satunya berlangsung seketika. Kerjanya tidak terbuang; ia *diinvestasikan*.

Pemampatan lintasan adalah alasan mengapa struktur ini sama sekali memerlukan analisis amortisasi. Satu `FindSet` tunggal tetap dapat berbiaya $\Theta(\lg n)$ — yang pertama menuruni lintasan panjang memang begitu. Tetapi ia hanya dapat melakukannya *sekali*, sebab ia meratakan lintasan di belakangnya. Biaya strukturnya bukanlah biaya terburuk satu operasi; ia biaya total barisannya, dan perataannya berarti operasi berikutnya mewarisi kerja yang dilakukan operasi sebelumnya.

> **MENGAPA INI PERSIS SITUASI BAB INI.** Bandingkan dengan tumpukannya. Sebuah `Multipop` yang membuang seribu item berbiaya seribu — tetapi seribu item itu tidak akan pernah dibuang lagi. Sebuah `FindSet` yang menyusuri lintasan sepanjang dua puluh berbiaya dua

puluh — tetapi dua puluh simpul itu tidak akan pernah disusuri lagi. Pada kedua kasusnya, operasi yang mahal itu mahal justru karena ia sedang *melahap* struktur yang dibangun operasi-operasi sebelumnya, dan pada kedua kasusnya pertanyaan yang benar adalah totalnya atas barisannya, bukan yang terburuk atas operasinya. Alat untuk keduanya adalah metode potensial.

14.8.4 Apa jawabannya

Tarjan membuktikan pada 1975 [8] bahwa dengan kedua akal-akalan itu, sebuah barisan berisi m operasi atas n elemen berbiaya

$$O(m\,\alpha(n)),$$

dengan α adalah **fungsi Ackermann balikan**, dinamai menurut fungsi yang tumbuh sangat cepat yang diperkenalkan Ackermann pada 1928 [9]. Buktinya memakai sebuah fungsi potensial — yang canggih, menetapkan sebuah potensial bagi tiap simpul berdasarkan peringkatnya dan peringkat induknya — dan panjangnya beberapa halaman; monografnya [10] mengembangkannya panjang lebar. Kita tidak akan memuatnya ulang. Yang akan kita lakukan adalah (a) mengatakan dengan jujur apa itu α, (b) menunjukkan bahwa nilainya paling besar 4 untuk setiap n yang akan pernah Anda jumpai, dan (c) mengukur biaya nyata strukturnya sehingga Anda dapat melihat bahwa teorinya tidak melebih-lebihkan.

HAL YANG DIPERINGATKAN BAB 11. $O(\alpha(n))$ adalah batas **teramortisasi**, bukan batas kasus-terburuk-per-operasi. Satu `FindSet` tunggal sungguh-sungguh dapat berbiaya $\Theta(\lg n)$ — susuran pertama menuruni lintasan yang panjang. Yang tidak dapat terjadi adalah *banyak* pencarian berbiaya sebesar itu, sebab yang pertama meratakan lintasannya. Seperti selalu: batasnya adalah batas atas sebuah jumlah.

14.8.5 Apa sebenarnya α itu

Kebanyakan buku teks berkata "fungsi Ackermann balikan, yang tumbuh teramat lambat" lalu melangkah pergi. Berikut definisi yang jujur, yang tidak sukar, hanya tidak akrab.

Definisikanlah sebuah keluarga fungsi $A_k(j)$, untuk bilangan bulat $k \geq 0$ dan $j \geq 1$:

$$A_k(j) \;=\; \begin{cases} j+1 & \text{bila } k=0, \\ A_{k-1}^{(j+1)}(j) & \text{bila } k \geq 1, \end{cases} \tag{14.16}$$

dengan $A_{k-1}^{(j+1)}$ berarti "terapkan fungsi A_{k-1} pada keluarannya sendiri, sebanyak $j+1$ kali berturut-turut".

Maka

$$\alpha(n) \;=\; \min\{\,k : A_k(1) \geq n\,\}. \tag{14.17}$$

Dengan kata lain, Persamaan (14.17) berarti: $\alpha(n)$ adalah tingkat k terkecil pada keluarga itu yang $A_k(1)$-nya akhirnya tumbuh sebesar n.

Mari kita hitung $A_k(1)$ dari Persamaan (14.16) untuk beberapa k yang pertama, sebab bilangan-bilangannya adalah seluruh ceritanya.

$A_0(1)$**.** Menurut baris pertamanya, $A_0(j) = j+1$, sehingga $A_0(1) = 2$.

$A_1(1)$. $A_1(j) = A_0^{(j+1)}(j)$ — terapkan "tambah satu" sebanyak $j+1$ kali bermula dari j — yang memberikan $j + (j+1) = 2j + 1$. Jadi $A_1(1) = 3$.

$A_2(1)$. $A_2(j) = A_1^{(j+1)}(j)$. Karena A_1 melipatduakan-dan-menambah-satu, menerapkannya i kali pada j memberikan $2^i(j+1) - 1$. Dengan $i = j + 1$: $A_2(j) = 2^{j+1}(j+1) - 1$. Jadi $A_2(1) = 2^2 \cdot 2 - 1 = 7$.

$A_3(1)$. $A_3(1) = A_2^{(2)}(1) = A_2(A_2(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2048 - 1 = 2047$.

$A_4(1)$. $A_4(1) = A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047)$, yaitu A_2 yang diterapkan pada 2047 sebanyak 2048 kali. Penerapan yang paling pertama saja sudah memberikan $2^{2048} \cdot 2048 - 1$, sebuah bilangan berdigit sekitar 620 — dan sesudah itu kita melakukannya 2047 kali lagi, tiap langkahnya memangkatkan lagi.

Nilai-nilai $A_k(1)$ dikumpulkan pada **Tabel 14.1**.

Tabel 14.1 Nilai-nilai $A_k(1)$ yang mendefinisikan α. Lompatan dari $k = 3$ ke $k = 4$ bukan lompatan dalam ukuran, melainkan lompatan dalam *jenis*: 2047 adalah bilangan yang dapat Anda tuliskan, sedangkan $A_4(1)$ adalah bilangan yang uraian desimalnya tidak akan muat di alam semesta teramati.

k	$A_k(1)$
0	2
1	3
2	7
3	2047
4	jauh lebih besar daripada 2^{2048}

Membaca definisi α dari **Tabel 14.1** memberikan persis tabel yang dikutip Bab 11, dinyatakan ulang di sini sebagai **Tabel 14.2**.

Tabel 14.2 Fungsi Ackermann balikan. Agar $\alpha(n)$ mencapai 5, Anda memerlukan lebih dari $A_4(1) > 2^{2048}$ elemen.

Rentang n	$\alpha(n)$
$n \leq 2$	0
$n \leq 3$	1
$n \leq 7$	2
$n \leq 2047$	3
$n \leq A_4(1)$, yang melampaui 2^{2048}	4

14.8.6 Mengapa $\alpha(n) \leq 4$ untuk sembarang n yang muat di alam semesta

Taksiran para fisikawan bagi banyaknya atom di alam semesta teramati adalah sekitar 10^{80}, yang kira-kira 2^{266}. Bab 11 memakai angka yang sama.

Nah, $A_4(1)$ melampaui 2^{2048}. Dan 2^{2048} tak terbayangkan lebih besar daripada 2^{266} — nisbahnya 2^{1782}, yang sendirinya merupakan bilangan berdigit lebih dari lima ratus.

Jadi: seandainya tiap atom di alam semesta teramati menjadi sebuah elemen terpisah pada struktur himpunan saling lepas Anda, n akan sekitar 2^{266}, yang jauh di bawah $A_4(1)$, dan karena itu

$$\alpha(n) \;=\; 4.$$

> **APA ARTINYA INI.** $\alpha(n) \leq 4$ bukanlah "benar untuk masukan yang realistis" dalam arti yang mengawang-awang. Ia benar untuk setiap masukan yang dapat diwakilkan secara fisik, oleh mesin apa pun, kapan pun. Jadi hutan himpunan saling lepas dengan gabung menurut peringkat dan pemampatan lintasan berbiaya **paling banyak empat kali sebuah konstanta per operasi, secara teramortisasi** — dan dalam praktik jauh kurang dari itu, sebab kebanyakan pencariannya mendarat di simpul yang sudah termampatkan. Anda boleh memperlakukan `FindSet` dan `Union` sebagai $O(1)$ dalam taksiran rekayasa, sambil tahu bahwa batas yang jujur adalah $O(\alpha(n))$ dan bahwa α tidak benar-benar konstan. Fredman dan Saks [11] membuktikan pada 1989 bahwa $\Omega(\alpha(n))$ teramortisasi tak terhindarkan bagi struktur berbasis penunjuk mana pun, jadi tidak akan ada yang memperbaikinya.

14.8.7 Mengukurnya

Teori seekstrem ini layak diukur. **Kode 14.10** membangun hutannya dengan empat cara — dengan dan tanpa masing-masing akal-akalannya — dan mencacah tiap penunjuk yang disusuri, pada dua beban kerja: sebuah rantai, yang merupakan kasus terburuk bagi gabung yang naif, dan sebuah peleburan berimbang, yang merupakan kasus terburuk bagi gabung menurut peringkat.

Kode 14.10 Union–find dengan kedua akal-akalannya, tiap penunjuknya dicacah. Kompilasi dengan `g++ -std=c++17 -O2 -o dsu dsu.cpp`.

```
// Union-find yang diukur.  Tiap penunjuk yang disusuri ikut dihitung, sehingga
// "biaya amortisasi" yang dicetak di bawah adalah pengukuran, bukan kutipan.
// Dua beban kerja dipakai: sebuah rantai, yang menghancurkan struktur naifnya,
// dan peleburan berimbang, kasus terburuk bagi gabung menurut peringkat.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

struct Forest {
    vector<int> parent, rnk;
    bool byRank, compress;
    long long hops = 0, ops = 0;           // hops = penunjuk yang disusuri = biaya
sebenarnya

    Forest(int n, bool r, bool c) : parent(n), rnk(n, 0), byRank(r), compress(c) {
        for (int i = 0; i < n; ++i) parent[i] = i;
    }

    int findSet(int x) {
        ++ops;
        int root = x;
        while (parent[root] != root) { root = parent[root]; ++hops; }
        if (compress)                       // sapuan kedua: arahkan ulang tiap simpul
            while (parent[x] != root) { int nxt = parent[x]; parent[x] = root; x = nxt;
++hops; }
        return root;
    }

    void unionSets(int a, int b) {
        int ra = findSet(a), rb = findSet(b);
        if (ra == rb) return;
        if (!byRank) { parent[ra] = rb; return; }        // sembarang: cara yang buruk
        if (rnk[ra] < rnk[rb]) parent[ra] = rb;
        else if (rnk[ra] > rnk[rb]) parent[rb] = ra;
        else { parent[rb] = ra; ++rnk[ra]; }             // seri menaikkan peringkat satu
    }

    int height(int x) const { int h = 0; while (parent[x] != x) { x = parent[x]; ++h; }
return h; }
    int tallest() const {
        int t = 0; for (size_t i = 0; i < parent.size(); ++i) t = max(t, height((int)i));
return t;
    }
};
```

Kode 14.10 (Lanjutan) *Union–find dengan kedua akal-akalannya, tiap penunjuknya dicacah. Kompilasi dengan g++ -std=c++17 -O2 -o dsu dsu.cpp.*

```
// alpha(n), dari A_k(1) = 2, 3, 7, 2047, A_4(1) > 2^2048.
int alphaOf(long long n) {
    if (n <= 2) return 0;
    if (n <= 3) return 1;
    if (n <= 7) return 2;
    if (n <= 2047) return 3;
    return 4;                               // benar untuk tiap n yang dapat dialamati
mesin
}

void row(const string& name, const Forest& F, int tallBefore, int tallAfter) {
    cout << left << setw(30) << name << right
         << " ops=" << setw(9) << F.ops
         << " hops=" << setw(12) << F.hops
         << " hops/op=" << setw(11) << fixed << setprecision(3)
         << (double)F.hops / (double)F.ops
         << "   height " << tallBefore << " -> " << tallAfter << "\n";
}

// Beban kerja 1: gabungkan 0 dengan 1, 2, 3, ... berturut-turut.  Tiap gabungan
// menjadikan elemen barunya akar, sehingga tanpa gabung menurut peringkat tumbuh
// lintasan sepanjang n-1.  Lalu cari tiap elemen sekali, terburuk dahulu.
void chain(const string& name, bool r, bool c, int n) {
    Forest F(n, r, c);
    for (int i = 1; i < n; ++i) F.unionSets(0, i);
    int before = F.tallest();
    for (int i = 0; i < n; ++i) F.findSet(i);
    row(name, F, before, F.tallest());
}

// Beban kerja 2: kasus terburuk bagi gabung menurut peringkat.  Gabungkan akar
// berperingkat sama berpasangan, melipatduakan ukuran pohonnya tiap putaran,
// satu-satunya cara membuat peringkat -- dan karenanya tinggi -- mencapai lg n.
// Lalu cari tiap elemen sekali, terdalam dahulu.
void balanced(const string& name, bool r, bool c, int n) {
    Forest F(n, r, c);
    for (int step = 1; step < n; step *= 2)
        for (int i = 0; i + step < n; i += 2 * step) F.unionSets(i, i + step);
    int before = F.tallest();
    vector<int> order(n);
    for (int i = 0; i < n; ++i) order[i] = i;
    sort(order.begin(), order.end(),
         [&F](int a, int b) { return F.height(a) > F.height(b); });
    for (int x : order) F.findSet(x);
    row(name, F, before, F.tallest());
}
```

Kode 14.10 (Lanjutan) *Union–find dengan kedua akal-akalannya, tiap penunjuknya dicacah. Kompilasi dengan g++ -std=c++17 -O2 -o dsu dsu.cpp.*

```
int main() {
    const int n = 32768;
    cout << "n = " << n << " elements.  alpha(" << n << ") = " << alphaOf(n) << ".\n";
    cout << "lg n = 15, so union by rank alone promises height at most 15.\n";

    cout << "\nWORKLOAD 1  unite 0 with 1,2,3,...,n-1, then find every element.\n";
    chain("  naive (no rank, no compress)", false, false, n);
    chain("  union by rank only",           true,  false, n);
    chain("  path compression only",        false, true,  n);
    chain("  rank + path compression",      true,  true,  n);

    cout << "\nWORKLOAD 2  unite equal-rank roots in pairs, then find every element.\n";
    balanced("  naive (no rank, no compress)", false, false, n);
    balanced("  union by rank only",           true,  false, n);
    balanced("  path compression only",        false, true,  n);
    balanced("  rank + path compression",      true,  true,  n);

    cout << "\nHow alpha grows, from A_k(1) = 2, 3, 7, 2047, ... :\n"
            "   n <= 2       alpha(n) = 0\n"
            "   n <= 3       alpha(n) = 1\n"
            "   n <= 7       alpha(n) = 2\n"
            "   n <= 2047    alpha(n) = 3\n"
            "   n <= A_4(1)  alpha(n) = 4     and A_4(1) is larger than 2^2048\n";
    return 0;
}
```

```
n = 32768 elements.  alpha(32768) = 4.
lg n = 15, so union by rank alone promises height at most 15.

WORKLOAD 1  unite 0 with 1,2,3,...,n-1, then find every element.
  naive (no rank, no compress) ops=    98302 hops=  1073676289 hops/op=  10922.222   height
32767 -> 32767
  union by rank only           ops=    98302 hops=       32767 hops/op=      0.333   height
1 -> 1
  path compression only        ops=    98302 hops=      196595 hops/op=      2.000   height
32766 -> 1
  rank + path compression      ops=    98302 hops=       32767 hops/op=      0.333   height
1 -> 1

WORKLOAD 2  unite equal-rank roots in pairs, then find every element.
  naive (no rank, no compress) ops=    98302 hops=      311264 hops/op=      3.166   height
15 -> 15
  union by rank only           ops=    98302 hops=      245760 hops/op=      2.500   height
15 -> 15
  path compression only        ops=    98302 hops=      163775 hops/op=      1.666   height
14 -> 1
  rank + path compression      ops=    98302 hops=       98271 hops/op=      1.000   height
15 -> 1

How alpha grows, from A_k(1) = 2, 3, 7, 2047, ... :
   n <= 2       alpha(n) = 0
   n <= 3       alpha(n) = 1
   n <= 7       alpha(n) = 2
   n <= 2047    alpha(n) = 3
   n <= A_4(1)  alpha(n) = 4     and A_4(1) is larger than 2^2048
```

Lima hal yang perlu dibaca dari situ.

Struktur yang naif itu membawa bencana. Pada beban kerja 1 ia rata-rata 10.922 lompatan penunjuk per operasi. Ia membangun sebuah lintasan setinggi 32.767 dan tidak pernah meratakannya. Biaya terukur per operasinya sekitar $n/3$, yang berarti $\Theta(n)$.

Gabung menurut peringkat sendirian sudah cukup untuk menghentikan bencananya. Pada beban kerja 1 ia tidak pernah membiarkan tingginya melampaui 1.

Tetapi gabung menurut peringkat sendirian belum akhir ceritanya. Beban kerja 2 dirancang untuk mengalahkannya: dengan menggabungkan akar berperingkat sama secara berpasangan, kita memaksa peringkatnya naik satu tingkat demi satu tingkat sampai ia mencapai 15. Dan 15 persis lg32768 — batas yang dibuktikan di §14.8.2, tercapai sampai ke satuannya. Gabung menurut peringkat tidak *menjanjikan* yang lebih baik daripada lg$n$, dan di sini ia memang tidak memberikannya.

Pemampatan lintasan menuntaskan pekerjaannya. Pada beban kerja 2, "rank + path compression" meratakan hutan setinggi 15 itu menjadi setinggi 1 dan rata-rata tepat 1,000 lompatan per operasi, melawan 2,500 bagi gabung menurut peringkat sendirian.

Pengukurannya jauh dari $\alpha(n) = 4$. Batas yang dibuktikan adalah $O(m\,\alpha(n))$ dengan $\alpha(32768) = 4$; biaya terukurnya 1,000 lompatan per operasi. Batasnya adalah batas atas dengan ruang berlebih, dan dalam pemakaian sehari-hari strukturnya berperilaku seolah-olah ia berwaktu konstan datar — yang, untuk keperluan rekayasa, memang demikian.

Itulah utang dari Bab 11 yang kini terlunasi. `FindSet` dan `Union` berbiaya $O(\alpha(n))$ teramortisasi, $\alpha(n) \leq 4$ selalu, dan alasan mengapa ia batas teramortisasi alih-alih batas kasus terburuk adalah bahwa pemampatan lintasan membuat biaya tiap operasi bergantung pada apa yang dilakukan operasi-operasi sebelumnya — yang merupakan situasi yang dibicarakan seluruh bab ini.

14.9 Memilih Metode

Tiga metode, satu jawaban. Mana yang sebaiknya Anda raih? **Tabel 14.3** menaruh ketiganya berdampingan.

Tabel 14.3 Ketiga metodenya dibandingkan. Ketiganya setara kekuatannya untuk contoh-contoh di dalam bab ini; bedanya terletak pada berapa banyak kerja yang dituntut masing-masing dan berapa banyak yang diceritakannya kepada Anda.

	Agregat	Akuntansi	Potensial
Apa yang harus Anda temukan	sebuah argumen pencacahan bagi total $T(n)$	sebuah tagihan per jenis operasi, dan aturan yang menyebutkan di mana kreditnya duduk	sebuah fungsi Φ atas keadaan strukturnya
Apa yang Anda peroleh	satu biaya amortisasi, sama untuk tiap operasi	biaya amortisasi yang berbeda untuk tiap jenis operasi	biaya amortisasi yang berbeda untuk tiap operasi, yang boleh bergantung pada keadaannya
Apa yang harus Anda buktikan	argumen pencacahannya	kreditnya tidak pernah menjadi negatif	$\Phi(D_i) \geq \Phi(D_0)$ untuk semua i
Paling baik ketika	ada argumen bersih "tiap objek disentuh paling banyak k kali"	pembayaran di mukanya menempel secara alami pada objek perorangan	pembayaran di mukanya adalah sifat bentuk strukturnya
Paling lemah ketika	operasinya sangat berbeda-beda biayanya	tidak ada objek yang gamblang untuk menempelkan kreditnya	Anda tidak dapat menerka Φ
Contoh dalam bab ini	tumpukan §14.3.2, pencacah §14.3.3, Soal 17.1-3 §14.3.4	tumpukan §14.5.3, pencacah §14.5.4, Soal 17.2-2 §14.5.5, tabel §14.7.4	tumpukan §14.6.4, pencacah §14.6.5, antrean-dari-tumpukan §14.6.6, tabel §14.7.5 dan §14.7.7
Naik sampai ke	struktur yang sederhana saja	struktur yang cukup rumit	pohon *splay*, *Fibonacci heap*, union–find

Urutan serangan yang praktis:

1. **Cobalah metode agregat lebih dahulu.** Ia yang terpendek, dan ketika sebuah argumen pencacahan yang bersih ada, ia yang paling meyakinkan. Bila Anda dapat berkata "tiap elemen disentuh paling banyak dua kali sepanjang seluruh barisannya", berhentilah — Anda sudah selesai.

2. **Raihlah metode akuntansi ketika operasinya sungguh-sungguh berbeda biayanya** dan Anda ingin mengatakannya. Ia juga yang paling mudah diterangkan kepada seseorang yang belum mempelajari pokok bahasan ini, sebab kiasannya adalah pembukuan yang jujur.

3. **Raihlah metode potensial ketika kedua yang lain gagal**, atau ketika peristiwa mahal strukturnya bergantung pada *bentuk*nya alih-alih pada objek tertentu mana pun — sebuah tabel yang hendak berlipat dua, sebuah pohon yang hendak menyeimbangkan diri, sebuah *heap* yang hendak dikonsolidasikan. Ia juga metode yang harus dipelajari dengan sungguh-sungguh, sebab hanya ia di antara ketiganya yang naik sampai ke struktur yang sukar.

KETIGANYA BUKAN TEOREMA YANG BERBEDA. Ketiganya membuktikan pernyataan berbentuk sama: $\sum c_i \leq \sum \hat{c}_i$. §14.6.3 memperlihatkan bahwa syarat metode potensial dan invarian kredit metode akuntansi secara harfiah adalah pertidaksamaan yang sama. Jadi bila Anda memperoleh jawaban yang berbeda dari dua metode, salah satunya memuat kekeliruan aritmetika — bukan kebenaran yang lebih dalam.

14.10 Profitina dalam Praktik

14.10.1 Di mana amortisasi hidup di dalam sebuah platform visibilitas AI

Profitina adalah platform Visibilitas AI (GEO — *Generative Engine Optimization*): ia memindai situs web sebuah merek, menilai seberapa siap situs itu untuk **dikutip oleh sebuah mesin**

jawaban AI, menerangkan tiap sinyal yang diukurnya, lalu membantu memperbaiki apa yang ditemukannya. Ia berjalan di produksi pada `app.profitina.com`, di atas aplikasi web Python dengan PostgreSQL, dengan inferensi model lokal melalui Ollama.

Tiga bagian sistem itu adalah persoalan analisis amortisasi dalam arti yang ketat menurut bab ini.

Batas jelajahnya adalah sebuah tabel dinamis. Ketika pemindainya mulai bekerja pada sebuah situs, ia tidak tahu berapa banyak halaman yang akan ditemukannya. Ia menemukan pranala sambil jalan, dan tiap URL yang ditemukan ditambahkan ke sebuah batas jelajah. Batas jelajah itu adalah larik yang tumbuh, dan ia persis struktur §14.7: penambahan berbiaya $O(1)$ teramortisasi, sesekali $\Theta(m)$ ketika bloknya berlipat dua. Akibat rekayasanya eksak dan praktis: **total kerja penyalinan memori pemindainya sepanjang penjelajahan m halaman adalah $O(m)$, bukan** $O(m^2)$, dan sebuah halaman yang memicu pelipatan bukanlah laporan kutu. Tanpa argumen amortisasinya, penjelajahan 50.000 halaman kelihatan seolah-olah dapat mengerjakan 2.5×10^9 satuan penyalinan; dengan argumen itu, kita tahu ia mengerjakan kurang dari 1.5×10^5.

Penyangga sinyalnya adalah antrean dua-tumpukan. Sinyal — keterambilan isi, kehadiran data terstruktur, penanda kewenangan teknis — dihasilkan oleh penjelajahnya dalam urutan halaman mana pun yang selesai lebih dahulu, dan dikonsumsi oleh penilainya dalam urutan yang tetap. Penyangga di antara keduanya harus FIFO, dan ia dibangun dari dua tumpukan persis seperti di §14.6.6, sehingga sang produsen tidak pernah terhalang: `Enqueue` adalah satu push, teramortisasi 3. Ketika konsumennya menguras, satu `Dequeue` mengerjakan kerja $\Theta(e)$ dan sisanya mengerjakan $O(1)$, dan *totalnya* linear. Anggaran latensi papan kendalinya ditulis terhadap angka amortisasinya, bukan terhadap angka terburuk tunggalnya — yang sah untuk *throughput*, dan §14.11 menerangkan kapan ia tidak akan sah.

Penjadwal pindai-ulangnya adalah persoalan pencacah yang menyamar. Tiap merek yang dilacak dipindai ulang pada irama yang bergantung pada tingkat langganannya, dan penjadwalnya harus memutuskan, tiap jam, merek mana yang jatuh tempo. Penjadwal yang naif memeriksa tiap merek tiap jam, dengan biaya $\Theta(B)$ per detak untuk B merek. Yang lebih baik memelihara sebuah struktur berember yang di dalamnya tiap merek muncul di tepat satu ember dan embernya dinaikkan sebagaimana bit sebuah pencacah biner: kebanyakan detak menyentuh satu ember, dan detak yang jarang mengalir menembus beberapa. Analisisnya adalah pencacah k-bit §14.3.3 dengan kata benda yang berbeda, dan ia memberikan **kerja penjadwalan $O(1)$ teramortisasi per detak, berapa pun banyaknya tingkat langganan yang ada**.

Pendeteksian hampir-duplikatnya memakai ulang union–find. Bab 11 mencatat bahwa Profitina mengelompokkan halaman yang hampir identik sebelum menilainya, sehingga sebuah situs dengan seribu halaman yang dibangkitkan otomatis tidak dinilai seribu kali untuk isi yang sama. Kelompok-kelompok itu adalah himpunan saling lepas, mereka melebur seiring kemiripan ditemukan, dan strukturnya adalah hutan §14.8. Batas $O(\alpha(n))$-lah yang memungkinkan sapuan penghilangan duplikatnya dihitung berbiaya linear terhadap banyaknya pasangan kemiripan.

14.10.2 Studi kasus yang dikerjakan: Profitina Laundry

Studi kasus praktik di dalam buku ini selalu **Profitina Laundry**, sebuah usaha kecil untuk pengajaran, bukan produk yang sungguhan.

Profitina Laundry menerima pakaian di sebuah meja. Tiga struktur dari bab ini mengerjakan tiga tugas.

1. **Register penerimaannya** adalah sebuah tabel dinamis. Tiap helai pakaian yang datang dicatat di dalamnya. Ia berlipat dua ketika penuh.
2. **Tumpukan tertundanya** adalah sebuah tumpukan. Pakaian menunggu di sana untuk sebuah mesin. Memuat sebuah mesin mengambil sampai dua belas helai sekaligus — sebuah `Multipop`.
3. **Baris pengambilannya** adalah sebuah antrean yang dibangun dari dua tumpukan, sehingga pelanggan mengambil menurut urutan kedatangan mereka meskipun tumpukan yang mendasarinya bersifat masuk-terakhir-keluar-pertama.

Pertanyaan sang manajer bukan pertanyaan akademis. Ia ingin tahu apakah perangkat lunak mejanya akan sanggup mengikuti **pada hari yang terburuk** — hari ketika sebuah hotel mengirim seluruh simpanan linennya pukul delapan pagi dan tidak ada satu pun yang dicuci sampai pukul lima sore. **Kode 14.11** menyimulasikan sebuah Selasa biasa dan hari terburuk itu, sambil mencacah tiap satuan kerja.

Kode 14.11 Profitina Laundry: sebuah tabel dinamis, sebuah tumpukan `Multipop`, dan sebuah antrean dua-tumpukan, semuanya diinstrumentasi, sepanjang satu hari kerja. Kompilasi dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp`.

```
// Profitina Laundry -- satu hari kerja di meja penerimaan, dengan setiap
// struktur dalam bab ini mengerjakan satu tugas dan tiap satuan kerja dihitung.
//
//   * REGISTER PENERIMAAN adalah tabel dinamis yang melipatdua saat penuh;
//   * TUMPUKAN TERTUNDA adalah tumpukan, dan memuat mesin adalah MULTIPOP;
//   * BARIS PENGAMBILAN adalah antrean FIFO yang dibangun dari dua tumpukan.
//
// Program ini menjalankan hari biasa lalu hari terburuk yang dapat dibayangkan
// sang manajer, dan melaporkan biaya amortisasi terukur per peristiwa.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

struct Register {                       // tabel dinamisnya
    vector<string> slot;
    long long size = 0, work = 0, expansions = 0;
    void add(const string& tag) {
        if (size == 0) { size = 1; slot.reserve(1); }
        if ((long long)slot.size() == size) {
            work += size;               // salin tiap entri yang sudah ada
            size *= 2; ++expansions;
        }
        slot.push_back(tag); work += 1;
    }
    long long potential() const { return 2 * (long long)slot.size() - size; }
};

struct Pile {                           // tumpukannya, dengan MULTIPOP
    vector<string> item;
    long long work = 0;
    void put(const string& t) { item.push_back(t); work += 1; }
    int load(int capacity) {            // MULTIPOP: isi satu mesin
        int taken = 0;
        while (!item.empty() && taken < capacity) { item.pop_back(); ++work; ++taken; }
        return taken;
    }
    long long potential() const { return (long long)item.size(); }
};
```

Kode 14.11 (Lanjutan) *Profitina Laundry: sebuah tabel dinamis, sebuah tumpukan Multipop, dan sebuah antrean dua-tumpukan, semuanya diinstrumentasi, sepanjang satu hari kerja. Kompilasi dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
struct Line {                           // antrean dari dua tumpukan
    vector<int> in, out;
    long long work = 0;
    void join(int ticket) { in.push_back(ticket); work += 1; }
    int serve(bool& ok) {
        if (out.empty())
            while (!in.empty()) { out.push_back(in.back()); in.pop_back(); work += 2; }
        if (out.empty()) { ok = false; return -1; }
        int t = out.back(); out.pop_back(); work += 1; ok = true; return t;
    }
    long long potential() const { return 2 * (long long)in.size(); }
};

struct Day {
    Register reg; Pile pile; Line line;
    long long events = 0;
    long long work() const { return reg.work + pile.work + line.work; }
    long long potential() const { return reg.potential() + pile.potential() +
line.potential(); }
};

void report(const string& name, const Day& d, int garments) {
    cout << left << setw(26) << name << right
         << " garments=" << setw(7) << garments
         << " events=" << setw(8) << d.events
         << " work=" << setw(9) << d.work()
         << " work/event=" << setw(9) << fixed << setprecision(4)
         << (double)d.work() / (double)d.events
         << "  expansions=" << d.reg.expansions << "\n";
}

int main() {
    const int MACHINE = 12;             // satu mesin memuat 12 helai

    // ---- Selasa biasa: pakaian datang sedikit-sedikit, satu mesin dimuat
    // ---- setiap kali dua belas helai menunggu, pelanggan mengambil berurutan.
    {
        Day d; int garments = 20000; int ticket = 0; bool ok;
        for (int g = 1; g <= garments; ++g) {
            d.reg.add("G" + to_string(g)); ++d.events;
            d.pile.put("G" + to_string(g)); ++d.events;
            if ((int)d.pile.item.size() >= MACHINE) {
                int n = d.pile.load(MACHINE); ++d.events;
                for (int i = 0; i < n; ++i) { d.line.join(++ticket); ++d.events; }
            }
            if (g % 3 == 0) { d.line.serve(ok); ++d.events; }
        }
```

Kode 14.11 (Lanjutan) *Profitina Laundry: sebuah tabel dinamis, sebuah tumpukan Multipop, dan sebuah antrean dua-tumpukan, semuanya diinstrumentasi, sepanjang satu hari kerja. Kompilasi dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
        report("ordinary Tuesday", d, garments);
    }

    // ---- Hari terburuk: sebuah hotel mengirim seluruh linennya pukul 08:00,
    // ---- tidak ada yang dicuci sampai pukul 17:00, lalu semuanya sekaligus.
    {
        Day d; int garments = 20000; int ticket = 0; bool ok;
        for (int g = 1; g <= garments; ++g) {
            d.reg.add("H" + to_string(g)); ++d.events;
            d.pile.put("H" + to_string(g)); ++d.events;
        }
        while (!d.pile.item.empty()) {                  // satu rentetan MULTIPOP raksasa
            int n = d.pile.load(garments); ++d.events; // kapasitas = semuanya
            for (int i = 0; i < n; ++i) { d.line.join(++ticket); ++d.events; }
        }
        for (int i = 0; i < garments; ++i) { d.line.serve(ok); ++d.events; }
        report("worst day (one big load)", d, garments);
    }

    // ---- Berapa biaya satu peristiwa tunggal pada hari terburuk, dan berapa
    // ---- biaya sehari penuh per peristiwa.
    {
        Day d; int garments = 20000;
        long long worst = 0;
        for (int g = 1; g <= garments; ++g) {
            long long before = d.reg.work;
            d.reg.add("W" + to_string(g)); ++d.events;
            worst = max(worst, d.reg.work - before);
        }
        cout << "\nIntake register alone: " << garments << " garments logged.\n";
        cout << "  most expensive single log entry = " << worst
             << " slot writes (a table expansion)\n";
        cout << "  total work                      = " << d.reg.work << "\n";
        cout << "  amortized cost per garment      = " << fixed << setprecision(4)
             << (double)d.reg.work / (double)garments << " slot writes\n";
        cout << "  aggregate bound proved in 14.7  = 3 per garment, i.e. "
             << 3 * garments << " in total\n";
        cout << "  final potential 2*num - size    = " << d.reg.potential() << "\n";
    }
    return 0;
}
```

```
ordinary Tuesday           garments=  20000 events=   68324 work=   145646 work/event=
2.1317  expansions=15
worst day (one big load)   garments=  20000 events=   80001 work=   172767 work/event=
2.1596  expansions=15

Intake register alone: 20000 garments logged.
  most expensive single log entry = 16385 slot writes (a table expansion)
  total work                      = 52767
  amortized cost per garment      = 2.6383 slot writes
  aggregate bound proved in 14.7  = 3 per garment, i.e. 60000 in total
  final potential 2*num - size    = 7232
```

Bacalah itu sebagaimana sang manajer akan membacanya.

Hari terburuknya berbiaya **2,16 satuan kerja per peristiwa**. Selasa yang biasa berbiaya **2,13**. Selisihnya kurang dari dua persen. Bencana yang beliau khawatirkan — dua puluh ribu helai datang sekaligus dan tidak ada yang dicuci sampai petang — tidak berbiaya berarti lebih besar bagi perangkat lunaknya daripada hari yang sepi, sebab setiap struktur yang terlibat punya batas amortisasi yang tidak peduli pada *urutan* peristiwanya, hanya pada banyaknya.

Dan tengoklah peristiwa tunggal yang terburuk: **16.385 tulisan slot**, dari satu perluasan tabel. Satu entri catatan itu enam belas ribu kali lebih mahal daripada entri biasa. Seandainya perangkat lunak mejanya punya tenggat keras, katakanlah satu milidetik per helai, entri tunggal itu mungkin melesetinya. Ia tidak, sebab kerjanya adalah penyalinan memori 16.385 penunjuk, yang memakan mikrodetik — tetapi *bentuk* risikonya nyata, dan §14.11 mengatakan apa yang harus dilakukan terhadapnya.

Akhirnya, bandingkan 2,64 tulisan per helai yang terukur itu dengan batas terbukti sebesar 3. Teorinya benar dan tidak jauh dari ketat.

> **APA YANG SEBAIKNYA DIBAWA PULANG SANG MANAJER.** Beliau tidak perlu membeli komputer yang lebih besar untuk hari-hari yang sibuk. Beliau perlu tahu bahwa biaya perangkat lunaknya tumbuh **linear terhadap banyaknya helai pakaian**, dan bahwa puncak laju kedatangan tidak mengubah hal itu. Bila ada yang menawarkan sistem penerimaan yang "lebih cepat" yang menumbuhkan registernya seratus slot tetap setiap kali, beliau sebaiknya menolak: menurut **Kode 14.13** sistem semacam itu berbiaya $\Theta(n)$ per helai, dan pada hari berisi dua puluh ribu helai registernya saja akan mengerjakan $20{,}000 + 100 \cdot \frac{200 \cdot 199}{2} = 2{,}010{,}000$ operasi tulis slot — hampir empat puluh kali 52,767 operasi tulis pada sistem yang beliau punya sekarang.

14.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. Mengatakan "teramortisasi" padahal yang Anda maksud "kasus rata-rata". Kekeliruan yang paling sering, dan yang paling merusak. Analisis amortisasi tidak melibatkan peluang sama sekali. Batasnya berlaku untuk setiap barisan, termasuk yang terburuk. *Obatnya:* sebelum menuliskan "teramortisasi", tanyakanlah kepada diri Anda sendiri — di mana sebaran peluangnya? Bila ada, kata yang Anda inginkan adalah "harapan".

2. Mengatakan "teramortisasi" padahal yang Anda maksud "biasanya cepat". Sebuah struktur dengan biaya amortisasi $O(1)$ tetap dapat punya satu operasi di dalam barisannya yang berbiaya $\Theta(n)$. Batasnya berlaku atas totalnya, bukan atas operasi perorangan mana pun. *Obatnya:* bila Anda memerlukan jaminan tentang satu operasi *tunggal* — tenggat waktu-nyata, tanggapan interaktif, penangan interupsi — batas amortisasi bukanlah jaminan yang Anda perlukan. Carilah batas *kasus terburuk*, atau pakailah teknik inkremental yang menyebar kerja mahalnya ke banyak operasi alih-alih mengerjakan semuanya sekaligus.

3. Lupa bahwa batas amortisasi tidak tersusun bersama gelung bersarang sebagaimana Anda harapkan. Bila sebuah struktur memberi Anda $O(1)$ teramortisasi per operasi lalu Anda melakukan n operasi, totalnya $O(n)$ — itu tidak apa-apa. Tetapi bila Anda melakukan n operasi *di dalam* sebuah gelung luar yang berjalan n kali, **dan strukturnya disetel ulang di antara iterasi luarnya**, Anda memperoleh $n \times O(n) = O(n^2)$, yang mungkin memang benar. Amortisasinya hanya berlaku di dalam satu barisan tanpa sela pada satu struktur. *Obatnya:* kenalilah dengan tepat barisan mana yang dibicarakan batasnya.

4. Memilih sebuah fungsi potensial lalu lupa memeriksa $\Phi(D_0)$. Tiap bukti metode potensial punya dua kewajiban: menunjukkan $\Phi(D_i) \geq \Phi(D_0)$ untuk semua i, sebagaimana dituntut Persamaan (14.12), dan menghitung $\hat{c}_i$. Pemula mengerjakan yang kedua lalu melewati yang pertama. Tanpa yang pertama, buktinya tidak berharga — Anda dapat "membuktikan" batas apa pun juga dengan sebuah fungsi potensial yang dibiarkan menjadi negatif sesuka hati. *Obatnya:* tuliskanlah kedua baris "$\Phi(D_0) = 0$" dan "$\Phi(D) \geq 0$ untuk setiap keadaan yang terjangkau, sebab ia mencacah sesuatu" sebelum Anda menghitung satu pun biaya amortisasi.

5. Menyatakan bahwa kreditnya tetap tak-negatif alih-alih membuktikannya. Metode akuntansi baru menjadi sebuah bukti ketika Anda menyebutkan *di mana kreditnya duduk*: pada item ini, pada bit 1 itu, pada entri tabel yang baru disisipkan itu. Maka ketak-negatifannya menjadi gamblang sebab Anda sedang mencacah objek. *Obatnya:* nyatakanlah aturan kreditnya sebagai sebuah invarian, dalam satu kalimat, lalu ujilah terhadap tiap operasinya secara terpisah.

6. Menyeparuhkan tabel dinamis pada keadaan separuh penuh. Jebakan §14.7.6, dan ia kutu nyata yang pernah dikirimkan di dalam pustaka yang nyata. Ambang perluasan dan ambang penyusutan harus dipisahkan. *Obatnya:* luaskan pada faktor muatan 1, susutkan pada faktor muatan 1/4, sehingga strukturnya duduk pada 1/2 sesudah salah satu peristiwanya dan harus didorong sejauh penggal konstan dari ukurannya untuk memicu peristiwa berikutnya.

7. Menumbuhkan tabel sebanyak sebuah konstanta alih-alih sebesar faktor konstanta. `size += 100` kelihatan lebih hemat daripada `size *= 2` dan secara asimtotik membawa bencana: penyalinan totalnya $\Theta(n^2)$ alih-alih $\Theta(n)$. *Obatnya:* tumbuhkanlah selalu sebesar sebuah *faktor*. Faktornya tidak harus 2 — 1,5 lazim dipakai, dan Soal 14.9 memperlihatkan bahwa sembarang faktor konstan yang lebih besar dari 1 berhasil — tetapi ia harus berupa faktor.

8. Memperlakukan argumen `Multipop` seolah-olah ia menuntut tumpukannya mulai kosong. Ia tidak menuntutnya kosong; ia menuntutnya berupa keadaan awal yang *diketahui* yang potensialnya dapat Anda hitung. Bila tumpukannya mulai dengan s_0 item, potensialnya mulai pada s_0 alih-alih 0, dan totalnya menjadi $2n + s_0$. *Obatnya:* ketika strukturnya tidak mulai kosong, tambahkanlah $\Phi(D_0)$ ke totalnya lalu katakanlah demikian.

9. Mengacaukan kedua α-nya. Faktor muatan sebuah tabel dinamis dan fungsi Ackermann balikan sama-sama ditulis α. Keduanya tidak ada hubungannya satu sama lain. *Obatnya:* bacalah kalimat di sekelilingnya. Bila ia sebuah pecahan antara 0 dan 1, ia sebuah faktor muatan.

10. Percaya bahwa $O(\alpha(n))$ berarti $O(1)$. Ia tidak, secara formal — α tidak terbatas, ia hanya tumbuh begitu lambat sehingga nilainya 4 untuk setiap masukan yang dapat dimuat sebuah mesin. Menuliskan $O(1)$ di dalam sebuah bukti adalah keliru; menuliskan "efektifnya konstan" di dalam catatan rekayasa tidak apa-apa. *Obatnya:* simpanlah batas yang jujur di dalam analisisnya dan bacaan praktisnya di dalam komentarnya, sebagaimana dilakukan §14.8.6.

14.12 Ringkasan Bab

Pertanyaannya. Diberikan sebuah barisan berisi n operasi pada sebuah struktur data, berapa biaya *seluruh barisannya*? Mengalikan operasi tunggal yang terburuk dengan n dapat melebih-lebihkan jawabannya sebesar faktor n, sebab operasi yang mahal dibayari oleh operasi murah yang memungkinkannya.

Definisinya. Penetapan biaya amortisasi $a_1, \ldots, a_n$ sah bila $\sum_{i \le k} c_i \le \sum_{i \le k} a_i$ untuk setiap awalan k. **Gambar 14.1** menyatakan syaratnya dan — dengan tinta merah — hal yang *tidak* disyaratkan: $c_i \le a_i$ untuk i perorangan.

Bukan kasus rata-rata. Dinyatakan dua kali, sebab inilah salah paham yang dibawa-bawa pemula: tidak ada peluang di dalam analisis amortisasi, dan batasnya berlaku untuk setiap barisan, termasuk barisan seorang lawan.

Metode agregat. Hitunglah $T(n)$ langsung lalu bagi. Pada tumpukan dengan `Multipop`, "tiap elemen di-pop paling banyak sekali sesudah ia di-push" memberikan $T(n) \le 2n$. Pada pencacah k-bit, bit i berbalik $\lfloor n/2^i \rfloor$ kali dan $\sum_{i \ge 0} n/2^i = 2n$. Pada Soal 17.1-3, memecah jumlahnya

menjadi dasar sebesar n dan deret geometri berisi tambahannya memberikan total eksak $2^{\lfloor \lg n \rfloor + 1} + n - \lfloor \lg n \rfloor - 2$, karenanya $\leq 3n - \lfloor \lg n \rfloor - 2$.

Metode akuntansi. Tagihlah tiap jenis operasi sejumlah tetap lalu simpan surplusnya sebagai kredit pada objek tertentu; buktikanlah kreditnya tidak pernah menjadi negatif dengan mengenali apa yang dicacahnya. Tumpukan: tagih 2 untuk `Push`, 0 untuk `Pop` dan `Multipop`, kredit = banyaknya item. Pencacah: tagih 2 per bit yang disetel ke 1, kredit = banyaknya bit 1. Soal 17.2-2: tagih 3 per operasi, kredit $= 2j + k + 2$ dengan $i = 2^k + j$.

Metode potensial. Definisikan Φ atas keadaan strukturnya lalu tetapkan $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$. Menjumlahkannya berteleskop menjadi $\sum \hat{c}_i = \sum c_i + \Phi(D_n) - \Phi(D_0)$, sehingga metodenya sah asalkan $\Phi(D_i) \geq \Phi(D_0)$. Tumpukan: Φ = banyaknya item. Pencacah: Φ = banyaknya bit 1. Antrean dua-tumpukan: $\Phi = 2 \times |S_{IN}|$, yang memberikan `Enqueue` 3 dan `Dequeue` 1.

Tabel dinamis. `TABLE-INSERT` dengan pelipatan: perluasan terjadi ketika $i - 1$ sebuah pangkat 2 dan berbiaya i; totalnya di bawah $3n$; tagihan akuntansinya 3 per item (dirinya sendiri, pemindahan dirinya, dan pemindahan satu item yang lebih tua); potensialnya $\Phi = 2\,num - size$, yang memberikan tepat 3 pada kedua kasusnya dan jatuh ke 0 pada tiap perluasan, sebagaimana diperlihatkan **Gambar 14.19**. Dengan penghapusan, menyeparuhkan pada separuh penuh mengakibatkan *thrashing* — blok `INSERT, DELETE, DELETE, INSERT` berbiaya $\Theta(size)$ tiap empat operasi, terukur lebih dari 32.000 satuan per operasi pada $n = 262{,}144$ — sedangkan menyeparuhkan pada seperempat penuh berbiaya 1,50 per operasi pada tiap skala dan menjaga faktor muatannya di dalam [1/4,1].

Union–find. Gabung menurut peringkat sendirian membatasi tingginya oleh $\lg n$, dibuktikan oleh "sebuah pohon berperingkat r punya sekurang-kurangnya 2^r simpul". Pemampatan lintasan membuat analisisnya menjadi teramortisasi, sebab sebuah pencarian yang mahal meratakan lintasan di belakangnya. Bersama-sama keduanya memberikan $O(m\,\alpha(n))$, dengan α didefinisikan dari keluarga Ackermann $A_k(1) = 2,3,7,2047, \ldots$, sehingga $\alpha(n) \leq 4$ untuk setiap n di bawah $A_4(1) > 2^{2048}$ — yaitu setiap n yang dapat ada secara fisik.

Memilihnya. Agregat ketika sebuah argumen pencacahan yang bersih ada; akuntansi ketika kreditnya menempel pada objek; potensial ketika kesulitannya ada pada bentuk strukturnya — dan potensial ketika strukturnya sukar, sebab hanya ia di antara ketiganya yang naik sampai ke sana.

14.12.1 Menengok ke depan

Bab 15 kembali kepada lintasan terpendek dan meminta semua pasangan sekaligus, yang di sana tekniknya adalah relaksasi berulang bergaya matriks alih-alih amortisasi. Tetapi kebiasaan yang dibangun bab ini — *tanyakanlah berapa biaya seluruh barisannya, bukan berapa biaya langkah terburuknya* — adalah kebiasaan yang kini akan Anda pakai di mana-mana. Setiap kali Anda menulis gelung yang memanggil `push_back`, setiap kali Anda memakai tabel hash yang melakukan *rehash*, setiap kali Anda menggabungkan sebuah *Fibonacci heap*, Anda sedang bersandar pada sebuah batas amortisasi. Anda kini tahu apa arti batas-batas itu, bagaimana ia dibuktikan, dan — yang sama pentingnya — apa yang tidak dijanjikannya.

14.13 Soal Latihan

14.1 Sebuah tumpukan, yang mula-mula kosong, menerima n operasi `Push` yang disusul satu `Multipop(S, n)` tunggal. Ada berapa operasi di dalam barisannya, berapa biaya sebenarnya seluruhnya, dan berapa biaya rata-rata per operasinya? Metode mana di antara ketiganya yang paling wajar dipakai di sini, dan mengapa?

14.2 Tuliskanlah, sebagai sebuah injeksi formal, argumen §14.3.2 bahwa banyaknya pembuangan dari sebuah tumpukan tidak pernah melampaui banyaknya push. Nyatakanlah dengan jelas apa yang dipetakan fungsinya, lalu buktikan bahwa ia terdefinisi dengan baik dan satu-satu.

14.3 Sebuah pencacah biner k-bit **tidak** mulai dari nol: ia mulai memuat suatu nilai yang representasi binernya memuat b_0 buah angka satu. Dengan metode potensial, berikanlah batas terbaik yang dapat Anda berikan bagi biaya total n operasi `Increment` dari keadaan itu. Di bawah syarat apa atas b_0 totalnya masih $O(n)$?

14.4 Sebuah barisan berisi n operasi dilakukan pada sebuah struktur data. Biaya operasi ke-i adalah i bila i tepat sebuah pangkat **3**, dan 1 bila bukan. Gunakan metode agregat untuk menemukan biaya amortisasi per operasinya, dengan menyebutkan konstantanya secara tegas.

14.5 Tambahkanlah sebuah operasi `Multipush(S, k)` pada tumpukannya, yang mem-push k salinan suatu item. Apakah biaya amortisasi ketiga operasi `Push`, `Multipop`, dan `Multipush` masih $O(1)$? Bela jawaban Anda dengan sebuah barisan yang tegas, dan katakanlah apa yang salah dengan fungsi potensial Φ = banyaknya item.

14.6 Metode potensial sebagaimana dinyatakan mengandaikan $\Phi(D_0) = 0$. Andaikan ia tidak. Turunkanlah hubungan yang benar antara $\sum c_i$ dan $\sum \hat{c}_i$, lalu nyatakanlah dengan tepat apa yang harus diandaikan agar biaya amortisasinya tetap membatasi biaya sebenarnya.

14.7 Rancanglah sebuah pencacah k-bit yang mendukung `Increment` dan juga `Reset`, yang mengembalikan pencacahnya ke nol. Kedua operasinya harus $O(1)$ teramortisasi. Berikan datanya, kedua prosedurnya secara garis besar, sebuah fungsi potensial, dan biaya amortisasi tiap operasinya. (Sebuah `Reset` yang sekadar menuliskan k angka nol tidak memadai: tunjukkanlah mengapa.)

14.8 Tunjukkan bahwa metode agregat adalah kasus khusus metode potensial, dengan memperlihatkan sebuah fungsi potensial yang, bagi sembarang struktur dengan batas agregat $T(n)$, menetapkan biaya amortisasi $T(n)/n$ bagi tiap operasinya. Syarat tambahan apa yang harus dipenuhi barisannya agar Φ Anda sah?

14.9 Sebuah tabel dinamis tumbuh sebesar faktor 3 alih-alih 2: ketika ia menjadi penuh, sebuah tabel baru dengan $3 \cdot size$ slot dialokasikan. Temukanlah sebuah fungsi potensial berbentuk $\Phi(T) = a \cdot num[T] - b \cdot size[T]$ yang berhasil, lalu hitunglah biaya amortisasi `TABLE-INSERT`. Kemudian rampatkanlah ke faktor pertumbuhan $\gamma > 1$.

14.10 Sebuah tabel dinamis tumbuh dengan menambahkan 100 slot tetap setiap kali ia terisi penuh, alih-alih mengalikan ukurannya. Tunjukkan bahwa biaya amortisasi `TABLE-INSERT` adalah $\Theta(n)$, lalu berikanlah biaya total eksak n penyisipan.

14.11 Andaikan kita menyusutkan tabelnya ketika faktor muatannya jatuh di bawah 1/3 alih-alih 1/4. Apakah batas amortisasinya bertahan? Berikanlah sebuah fungsi potensial untuk kasus gabungannya lalu benarkanlah, atau berikanlah sebuah barisan yang mematahkannya.

14.12 Lengkapilah analisis kasus bagi tabel dinamis gabungan §14.7.7. Tunjukkan bahwa dengan fungsi potensial yang dinyatakan itu, `TABLE-INSERT` berbiaya amortisasi paling besar 3 dan `TABLE-DELETE` paling besar 2, pada setiap kasusnya.

14.13 Pada antrean dua-tumpukan Soal 17.3-6, andaikan bahwa ketika S_{OUT} kosong kita memindahkan hanya *separuh* item dari S_{IN}, meninggalkan sisanya. Apa yang salah? Jawablah dengan tepat: apakah persoalannya pada biayanya, pada kebenarannya, atau pada keduanya?

14.14 Buktikanlah dengan induksi bahwa di dalam sebuah hutan himpunan saling lepas yang dipelihara dengan gabung menurut peringkat, sebuah pohon yang akarnya berperingkat r memuat sekurang-kurangnya 2^r simpul. Simpulkan bahwa tidak ada pohon yang tingginya lebih dari $\lfloor \lg n \rfloor$.

14.15 Hitunglah $A_3(1)$ secara eksak dari definisi di §14.8.5, dengan memperlihatkan tiap penerapan A_2. Kemudian tunjukkan bahwa $A_4(1) > 2^{2048}$, lalu simpulkan bahwa $\alpha(n) \leq 4$ untuk setiap n yang kurang dari 2^{2048}.

14.16 Tulislah sebuah program yang mengukur apa yang sungguh-sungguh dilakukan `std::vector<int>::push_back`: catatlah kapasitasnya sesudah tiap push, laporkan tiap realokasi beserta nisbah pertumbuhannya, dan cacahlah total banyaknya penyalinan elemen sepanjang sejuta push. Bandingkan pengukurannya dengan batas $3n$ dari §14.7.3.

14.17 Sebuah pencacah k-bit mendukung `Increment` dan `Decrement`. Tunjukkan bahwa biaya amortisasi per operasinya **bukan** $O(1)$, dengan memperlihatkan sebuah barisan berisi n operasi yang berbiaya $\Theta(nk)$. Terangkan mengapa fungsi potensial Φ = banyaknya bit 1 gagal.

14.14 Penyelesaian Soal Latihan

14.1 Ada $n + 1$ operasi: n push dan satu `Multipop`.

n push berbiaya 1 masing-masing, totalnya n. `Multipop(S, n)` menemukan n item di tumpukannya lalu membuang semuanya, berbiaya n. Jadi biaya sebenarnya seluruhnya adalah

$$T = n + n = 2n.$$

Rata-rata per operasinya adalah $2n/(n + 1) < 2$.

Metode yang wajar adalah **metode agregat**, sebab seluruh barisannya diberikan secara tegas dan totalnya dapat dihitung langsung; tidak ada yang perlu disebar. Perhatikanlah bahwa jawabannya, < 2, persis konstanta yang diramalkan analisis agregat §14.3.2 bagi barisan sembarang — barisan inilah yang mencapainya.

14.2 Misalkan R himpunan peristiwa pembuangan di dalam barisannya (tiap pembuangan satu item dari tumpukannya, entah ia terjadi di dalam sebuah `Pop` atau di dalam gelung sebuah `Multipop`), dan misalkan P himpunan operasi `Push`. Definisikanlah

$$f: R \to P$$

dengan: $f(\rho)$ adalah operasi `Push` yang meletakkan di tumpukannya item yang dibuang oleh pembuangan ρ.

f terdefinisi dengan baik. Tiap item di tumpukannya sampai ke sana lewat tepat satu `Push`; tidak ada jalan lain ke atas tumpukannya. Jadi untuk tiap peristiwa pembuangan, item yang dibuangnya punya satu push asal yang tunggal.

f satu-satu. Andaikan $f(\rho_1) = f(\rho_2) = \pi$ untuk pembuangan $\rho_1 \neq \rho_2$. Maka ρ_1 dan ρ_2 keduanya membuang item yang diletakkan oleh π. Tetapi sekali item itu dibuang, ia tidak lagi berada di tumpukannya, dan tidak ada yang mengembalikannya — sebuah `Push` bernilai sama yang datang kemudian adalah push yang *berbeda*, yang meletakkan kemunculan yang *berbeda*. Jadi mana pun di antara ρ_1, ρ_2 yang datang belakangan membuang sebuah item yang tidak ada di sana, yang mustahil. Karenanya $\rho_1 = \rho_2$, sebuah kontradiksi.

Sebuah injeksi dari R ke dalam P memberikan $|R| \leq |P|$, yang merupakan klaimnya. Karena $|P| \leq n$, kita memperoleh $|R| \leq n$, dan biaya totalnya $|P| + |R| \leq 2n$. ■

14.3 Pertahankanlah potensialnya $\Phi(D_i) = b_i$, banyaknya bit 1 sesudah operasi i. Analisis §14.6.5 tidak berubah: tiap increment tetap punya $\hat{c}_i \leq 2$, sebab penurunan itu hanya memakai $c_i \leq t_i + 1$ dan $b_i \leq b_{i-1} - t_i + 1$, yang keduanya tidak menyebut nilai awalnya.

Yang berubah adalah pembukuan di kedua ujungnya. Dari kesamaan teleskopik Persamaan (14.11),

$$\sum_{i=1}^{n} c_i \;=\; \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) \;\leq\; 2n - b_n + b_0 \;\leq\; 2n + b_0,$$

dengan memakai $b_n \geq 0$. Jadi biaya totalnya paling besar $2n + b_0$.

Karena pencacahnya hanya punya k bit, $b_0 \leq k$, yang memberikan batas aman $2n + k$.

Totalnya $O(n)$ asalkan $b_0 = O(n)$ — khususnya setiap kali $k = O(n)$, yang berlaku pada situasi realistis mana pun. Hanya bila pencacahnya lebar secara keterlaluan dibandingkan banyaknya operasinya (katakanlah $k = n^2$) konfigurasi awalnya akan mendominasi, dan bahkan pada saat itu pun biayanya $O(n + k)$, yang merupakan jawaban yang jujur: **Anda mungkin harus membayar sekali untuk keadaan yang diserahkan kepada Anda, tetapi hanya sekali.**

14.4 Tulislah c_i untuk biaya operasi i, sehingga $c_i = i$ ketika $i = 3^j$ dan $c_i = 1$ bila bukan.

Pecahlah sebagaimana di §14.3.4. Tiap operasi membayar dasar sebesar 1, menyumbang n. Operasi pada $i = 3^j$ membayar tambahan $3^j - 1$. Pangkat-pangkat 3 yang tidak melampaui n adalah yang memenuhi $j \leq \lfloor \log_3 n \rfloor$; tulislah $m = \lfloor \log_3 n \rfloor$.

$$\text{Total} \;=\; n + \sum_{j=0}^{m} (3^j - 1) \;=\; n + \left(\sum_{j=0}^{m} 3^j \right) - (m + 1).$$

Dengan rumus deret geometri untuk $x = 3$,

$$\sum_{j=0}^{m} 3^j \;=\; \frac{3^{m+1} - 1}{3 - 1} \;=\; \frac{3^{m+1} - 1}{2}.$$

Karena $3^m \leq n$, kita punya $3^{m+1} \leq 3n$, sehingga ini paling besar $(3n - 1)/2 < 3/2\, n$.

Karena itu

$$\text{Total} \;<\; n + 3/2\,n \;=\; 5/2\,n \;=\; O(n),$$

dan biaya amortisasi per operasinya kurang dari $5/2 = 2.5$, jadi $O(1)$.

Pola umumnya. Dengan basis β menggantikan 2 atau 3, jumlah geometrinya adalah $(\beta^{m+1} - 1)/(\beta - 1) \leq \beta n/(\beta - 1)$, sehingga biaya amortisasinya paling besar $1 + \beta/(\beta - 1)$. Untuk $\beta = 2$ itu 3, cocok dengan §14.3.4; untuk $\beta = 3$ ia 2.5; dan seiring β membesar ia turun menuju 2. **Semakin jarang operasi mahalnya, semakin kecil konstantanya — tetapi ia tidak pernah turun di bawah 2.**

14.5 Tidak. Biaya amortisasinya bukan $O(1)$; ia $\Theta(n)$ pada kasus terburuk.

Barisannya. Lakukanlah $n/2$ operasi, masing-masing `Multipush(S, n/2)`, lalu $n/2$ operasi, masing-masing `Multipop(S, 1)`. Tiap `Multipush` sungguh-sungguh mem-push $n/2$ item, sehingga biaya sebenarnya $n/2$. Biaya total separuh pertamanya adalah

$$\frac{n}{2} \times \frac{n}{2} \;=\; \frac{n^2}{4},$$

dan separuh keduanya hanya menambahkan $n/2$. Jadi n operasi berbiaya $\Theta(n^2)$, dan biaya amortisasi per operasinya $\Theta(n)$.

Mengapa Φ = banyaknya item gagal. Dengan potensial itu, `Multipush(S,k)` punya

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}) \;=\; k + k \;=\; 2k,$$

yang tidak dibatasi oleh sebuah konstanta — ia tumbuh bersama k. Potensialnya *melipatduakan* tagihannya alih-alih meniadakannya.

Alasannya, dengan kata-kata. `Multipop` berhasil sebab ia **melahap** potensial yang tersimpan: ia mengerjakan k' satuan kerja dan memusnahkan k' satuan kewajiban yang tersimpan, sehingga keduanya saling meniadakan. `Multipush` melakukan kebalikannya: ia mengerjakan k satuan kerja **dan** menciptakan k satuan kewajiban yang tersimpan. Tidak ada apa pun untuk ditiadakan, dan tidak ada akal-akalan pembukuan yang dapat menyembunyikan kerja yang sungguh-sungguh sedang dikerjakan untuk pertama kalinya.

> **ATURAN UMUM YANG DIPERAGAKAN SOAL INI.** Amortisasi hanya dapat membuat sebuah operasi tampak murah bila operasi itu sedang *memusnahkan* struktur yang dibayar operasi-operasi sebelumnya untuk dibangun. Ia tidak akan pernah dapat membuat sebuah operasi yang *membangun* k satuan struktur berbiaya kurang dari k. Bila ada yang mengklaim batas $O(1)$ teramortisasi bagi sebuah operasi yang menciptakan keadaan baru tanpa batas, ia keliru.

14.6 Mulailah dari kesamaan teleskopik Persamaan (14.11) di §14.6.2, yang berlaku berapa pun $\Phi(D_0)$:

$$\sum_{i=1}^{n} \hat{c}_i \;=\; \sum_{i=1}^{n} c_i \;+\; \Phi(D_n) - \Phi(D_0).$$

Disusun ulang:

$$\sum_{i=1}^{n} c_i \;=\; \sum_{i=1}^{n} \hat{c}_i \;-\; \Phi(D_n) \;+\; \Phi(D_0).$$

Agar total amortisasinya membatasi total sebenarnya kita memerlukan $-\Phi(D_n) + \Phi(D_0) \leq 0$, yakni $\Phi(D_n) \geq \Phi(D_0)$. **Itulah satu-satunya andaian yang diperlukan** — dan ia harus berlaku untuk setiap awalan, yakni $\Phi(D_k) \geq \Phi(D_0)$ untuk semua k, bila batasnya hendak berlaku untuk setiap awalan.

Bila sebagai gantinya kita hanya tahu $\Phi \geq 0$ di mana-mana, kita memperoleh yang lebih lemah tetapi masih berguna

$$\sum_{i=1}^{n} c_i \;\leq\; \sum_{i=1}^{n} \hat{c}_i \;+\; \Phi(D_0),$$

yang berkata: **total amortisasinya membatasi total sebenarnya, sampai sebuah biaya tambahan sekali-jalan sebesar potensial awalnya.** Soal 14.3 persis situasi ini dengan $\Phi(D_0) = b_0$.

Praktik yang biasa — bersikeras $\Phi(D_0) = 0$ dan $\Phi \geq 0$ — sekadar kasus khusus yang nyaman yang di dalamnya biaya tambahannya lenyap.

14.7 Rancangannya. Pertahankanlah larik bit $A[0 \ldots k-1]$ seperti sebelumnya, ditambah satu bilangan bulat tambahan $high$, yang mencatat **satu lebih dari indeks tertinggi yang pernah disetel ke 1 sejak reset terakhir**. Pada mulanya $high = 0$.

- `Increment`: persis seperti sebelumnya — nolkan angka 1 di ekornya, setel 0 yang pertama — dan sebagai tambahan, bila bit yang disetel berada di indeks i, perbaruilah $high \leftarrow \max(high,\, i+1)$.
- `Reset`: tuliskan 0 ke $A[0], A[1], \ldots, A[high-1]$ saja, lalu setel $high \leftarrow 0$. Tiap bit pada indeks $\geq high$ sudah bernilai 0 dan tidak perlu disentuh.

Mengapa `Reset` yang naif tidak memadai. Bila `Reset` menuliskan seluruh k angka nol, seorang lawan dapat menyelang-nyelingkan `Increment` dan `Reset`: tiap pasangannya berbiaya $1 + k$, sehingga n operasi berbiaya $\Theta(nk)$, dan biaya amortisasinya $\Theta(k)$, bukan $O(1)$.

Potensialnya.

$$\Phi(D) \;=\; (\text{banyaknya bit 1 di } A) \;+\; high.$$

Syaratnya. Pada mulanya pencacahnya nol dan $high = 0$, sehingga $\Phi(D_0) = 0$. Kedua sukunya adalah cacah, sehingga $\Phi \geq 0$ selalu. Bagus.

Biaya amortisasi `Increment`. Andaikan ia menolkan t bit dan menyetel bit di indeks i. Biaya sebenarnya $c = t + 1$. Banyaknya bit 1 berubah sebesar $1 - t$. Dan $high$ naik paling banyak 1 — ia naik hanya bila $i + 1 > high$, dan karena bit di i tadinya 0 sedangkan semua bit di bawahnya 1, kita punya $high \geq i$, sehingga kenaikannya paling banyak 1. Karenanya

$$\hat{c} \;\leq\; (t+1) + (1-t) + 1 \;=\; 3.$$

Biaya amortisasi `Reset`. Biaya sebenarnya $c = high$ (kita menuliskan $high$ angka nol; bila $high = 0$ tagihlah 1 untuk pemanggilannya sendiri). Banyaknya bit 1 jatuh ke 0, turun sekurang-kurangnya 0; dan $high$ jatuh dari nilainya ke 0, turun tepat sebesar $high$. Jadi

$$\hat{c} \;=\; high + (0 - (\text{ones} + high)) \;=\; -\text{ones} \;\leq\; 0.$$

(Pada kasus sepele $high = 0$ pencacahnya memang sudah nol, tidak ada yang berubah, dan biaya amortisasinya adalah 1 yang ditagihkan untuk pemanggilannya.) Jadi `Reset` berbiaya paling besar 1 secara teramortisasi — biaya sebenarnya sudah dibayar di muka oleh increment yang menaikkan $high$ sejak semula.

Kedua operasinya $O(1)$ teramortisasi, sehingga n operasi berbiaya $O(n)$. **Kode 14.12** mengimplementasikan dan mengukurnya.

Kode 14.12 Sebuah pencacah dengan `Increment` dan `Reset`, keduanya $O(1)$ teramortisasi. Kompilasi dengan `g++ -std=c++17 -O2 -o resetcounter resetcounter.cpp`.

```
// Penyelesaian Soal 14.7.  Pencacah k-bit yang mendukung INCREMENT dan
// juga RESET, yang mengembalikan pencacahnya ke nol.  Akal-akalannya adalah
// mengingat indeks tertinggi yang pernah disetel, sehingga RESET cukup
// membersihkan bit di bawahnya.  Dengan potensial Phi = (banyaknya bit 1) + high,
// kedua operasinya O(1) teramortisasi.  Program ini mengukurnya.
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

struct ResetCounter {
    vector<int> A;
    int high = 0;                       // satu lebih dari indeks tertinggi yang pernah
disetel
    long long actual = 0, ops = 0, worstOne = 0, resetCost = 0;

    explicit ResetCounter(int k) : A(k, 0) {}

    long long phi() const {
        long long ones = 0; for (int b : A) ones += b;
        return ones + high;             // satu satuan per bit 1, satu per indeks hidup
    }

    void increment() {
        long long c = 0; size_t i = 0;
        while (i < A.size() && A[i] == 1) { A[i] = 0; ++c; ++i; }
        if (i < A.size()) { A[i] = 1; ++c; if ((int)i + 1 > high) high = (int)i + 1; }
        charge(c);
    }

    void reset() {
        long long c = 0;
        for (int i = 0; i < high; ++i) { A[i] = 0; ++c; }   // hanya awalan yang hidup
        high = 0;
        if (c == 0) c = 1;              // RESET yang kosong pun berbiaya satu langkah
        resetCost += c;
        charge(c);
    }

private:
    void charge(long long c) { actual += c; ++ops; if (c > worstOne) worstOne = c; }
};
```

Kode 14.12 (Lanjutan) *Sebuah pencacah dengan Increment dan Reset, keduanya O(1) teramortisasi. Kompilasi dengan g++ -std=c++17 -O2 -o resetcounter resetcounter.cpp.*

```
int main() {
    ResetCounter C(32);
    long long lowestPhi = C.phi();
    // Si lawan: naik ke nilai besar, RESET, naik lagi, ...
    for (int round = 0; round < 2000; ++round) {
        for (int t = 0; t < 500; ++t) C.increment();
        C.reset();
        if (C.phi() < lowestPhi) lowestPhi = C.phi();
    }
    cout << "2000 rounds of (500 INCREMENTs then one RESET)\n";
    cout << "  operations               = " << C.ops << "\n";
    cout << "  total actual cost        = " << C.actual << "\n";
    cout << "  most expensive single op = " << C.worstOne << "\n";
    cout << "  measured cost per op     = " << fixed << setprecision(4)
         << (double)C.actual / (double)C.ops << "\n";
    cout << "  lowest potential seen    = " << lowestPhi
         << "  (it must never go below Phi(D_0) = 0)\n";
    cout << "  final potential          = " << C.phi() << "\n";

    cout << "\n  cost of the 2000 RESETs alone = " << C.resetCost << "\n";
    cout << "  a RESET that blindly cleared all 32 bits would have cost "
         << 2000 * 32 << "\n";
    return 0;
}
```

```
2000 rounds of (500 INCREMENTs then one RESET)
  operations               = 1002000
  total actual cost        = 2006000
  most expensive single op = 9
  measured cost per op     = 2.0020
  lowest potential seen    = 0  (it must never go below Phi(D_0) = 0)
  final potential          = 0

  cost of the 2000 RESETs alone = 18000
  a RESET that blindly cleared all 32 bits would have cost 64000
```

Dua hal memastikan analisisnya. Biaya terukur per operasinya adalah 2,002, nyaman di bawah 3 yang dibuktikan. Dan 2.000 reset itu berbiaya 18.000 satuan seluruhnya — sembilan masing-masing, sebab 500 increment hanya pernah menyetel bit sampai indeks 8 — melawan 64.000 bagi versi yang membuta.

14.8 Misalkan batas agregatnya $T(n)$, yang memberi tiap operasi biaya amortisasi $\hat{c} = T(n)/n$. Definisikanlah

$$\Phi(D_i) \;=\; \sum_{j=1}^{i} (\hat{c} - c_j),$$

dengan $\Phi(D_0) = 0$ (jumlah kosongnya).

Maka untuk tiap i,

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}) \;=\; c_i + (\hat{c} - c_i) \;=\; \hat{c},$$

sebab kedua jumlahnya berbeda tepat sebesar satu suku tunggal $\hat{c} - c_i$. Jadi tiap operasi memperoleh biaya amortisasi $\hat{c} = T(n)/n$, yang persis diberikan metode agregat.

Syarat tambahannya. Φ harus memenuhi $\Phi(D_i) \geq \Phi(D_0) = 0$, yaitu

$$\sum_{j=1}^{i} c_j \;\le\; i \cdot \hat{c} \qquad \text{untuk setiap } i.$$

Dengan kata lain: **batas agregatnya harus berlaku untuk setiap awalan barisannya, bukan sekadar untuk keseluruhannya.** Ini persis syarat awalan yang ditandai di §14.2.3. Untuk tumpukannya dan pencacahnya ia terpenuhi dengan sendirinya, sebab argumen pencacahan di sana bekerja kata demi kata pada sembarang awalan. Sebuah struktur yang dibuat-buat yang seluruh biayanya dijejalkan ke beberapa operasi pertamanya akan memenuhi batas agregatnya pada $i = n$ dan melanggarnya lebih awal; bagi struktur semacam itu "batas" agregatnya sama sekali bukan batas amortisasi yang sah.

14.9 Ambillah

$$\Phi(T) \;=\; \frac{3}{2}\, num[T] \;-\; \frac{1}{2}\, size[T],$$

yaitu, $a = 3/2$ dan $b = 1/2$.

Pemeriksaan kewarasannya. Tepat sesudah sebuah perluasan sebesar faktor 3, tabelnya memuat $size/3$ item, sehingga $\Phi = 3/2 \cdot \frac{size}{3} - 1/2\, size = \frac{size}{2} - \frac{size}{2} = 0$. Ketika tabelnya penuh, $num = size$ dan $\Phi = 3/2\, size - 1/2\, size = size$ — persis banyaknya item yang akan disalin perluasan berikutnya. Dan $\Phi \ge 0$ setiap kali $num \ge size/3$, yang berlaku sebab tabelnya sekurang-kurangnya sepertiga penuh sejak saat ia diciptakan oleh sebuah perluasan. Tabel kosong punya $\Phi = 0$.

Tanpa perluasan. $c_i = 1$, $size_i = size_{i-1}$, $num_i = num_{i-1} + 1$:

$$\begin{aligned}\hat{c}_i \;&=\; 1 + 3/2\,(num_i - num_{i-1}) - 1/2\,(size_i - size_{i-1}) \;=\; 1 + 3/2 \\ &=\; 5/2.\end{aligned}$$

Dengan perluasan. Di sini $num_{i-1} = size_{i-1}$, $size_i = 3\, size_{i-1}$, $num_i = num_{i-1} + 1$, dan $c_i = num_i$:

$$\begin{aligned}\hat{c}_i &= num_i + \left(\tfrac{3}{2} num_i - \tfrac{1}{2} size_i\right) - \left(\tfrac{3}{2} num_{i-1} - \tfrac{1}{2} size_{i-1}\right) \\ &= num_i + \tfrac{3}{2} num_i - \tfrac{3}{2} size_{i-1} - \tfrac{3}{2}(num_i - 1) + \tfrac{1}{2} size_{i-1} \\ &= num_i + \tfrac{3}{2} - size_{i-1} \\ &= (size_{i-1} + 1) + \tfrac{3}{2} - size_{i-1} \\ &= \tfrac{5}{2}.\end{aligned}$$

Kedua kasusnya memberikan tepat $5/2$, sehingga `TABLE-INSERT` adalah $O(1)$ teramortisasi dan n penyisipan berbiaya paling besar $2.5n$ — lebih murah daripada $3n$ milik pelipatan, dengan harga memboroskan lebih banyak memori.

Faktor umum $\gamma > 1$. Ambillah

$$\Phi(T) \;=\; \frac{\gamma}{\gamma - 1}\, num[T] \;-\; \frac{1}{\gamma - 1}\, size[T],$$

dan kedua perhitungan yang sama memberikan biaya amortisasi

$$1 + \frac{\gamma}{\gamma - 1}$$

pada kedua kasusnya: 3 untuk $\gamma = 2$, 2.5 untuk $\gamma = 3$, dan 4 untuk $\gamma = 1.5$. Batasnya tetap konstan untuk tiap $\gamma > 1$ yang tetap dan meledak ketika $\gamma \to 1$ — yang persis Soal 14.10 pada limitnya.

Kode 14.13 mengukur kebijakan $\gamma = 2$ dan $\gamma = 3$ berdampingan dengan kebijakan penambahan tetapnya.

Kode 14.13 Tiga kebijakan pertumbuhan dibandingkan: melipatduakan, melipattigakan, dan menambahkan 100 slot tetap. Kompilasi dengan `g++ -std=c++17 -O2 -o growth growth.cpp`.

```
// Penyelesaian Soal 14.9 dan 14.10.  Tiga kebijakan pertumbuhan bagi sebuah
// tabel dinamis dijalankan pada n penyisipan yang sama dan kerjanya dihitung:
//     lipatduakan ukurannya,  lipattigakan ukurannya,  tambah 100 slot tetap.
// Dua yang pertama O(1) teramortisasi; yang ketiga Theta(n).
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

struct Growing {
    long long num = 0, size = 0, work = 0, expansions = 0;
    int factor;              // kalikan dengan `factor`, atau 0 untuk menambah `step`
    long long step;

    Growing(int f, long long s) : factor(f), step(s) {}

    void insert() {
        if (num == size) {
            work += num;                                // salin semuanya ke seberang
            size = (factor > 0) ? (size == 0 ? 1 : size * factor) : size + step;
            ++expansions;
        }
        ++num; work += 1;
    }
};

void run(const string& name, int factor, long long step, long long n) {
    Growing T(factor, step);
    for (long long i = 0; i < n; ++i) T.insert();
    cout << left << setw(22) << name << right
         << " n=" << setw(9) << n
         << " work=" << setw(13) << T.work
         << " work/insert=" << setw(11) << fixed << setprecision(3)
         << (double)T.work / (double)n
         << "  expansions=" << T.expansions
         << "  final size=" << T.size << "\n";
}
```

Kode 14.13 (Lanjutan) *Tiga kebijakan pertumbuhan dibandingkan: melipatduakan, melipattigakan, dan menambahkan 100 slot tetap. Kompilasi dengan g++ -std=c++17 -O2 -o growth growth.cpp.*

```
int main() {
    cout << "Each row inserts n items into an initially empty table.\n\n";
    for (long long n : {1000LL, 10000LL, 100000LL, 1000000LL}) {
        run("double the size",  2, 0,   n);
        run("triple the size",  3, 0,   n);
        run("add 100 slots",    0, 100, n);
        cout << "\n";
    }
    cout << "Doubling stays below its bound of 3 units per insert and tripling\n"
            "below its bound of 2.5: both constant.  Adding a fixed 100 slots grows\n"
            "without bound -- about n/200 per insert, which is Theta(n).\n";
    return 0;
}
```

```
Each row inserts n items into an initially empty table.

double the size        n=     1000 work=         2023 work/insert=      2.023
expansions=11  final size=1024
triple the size        n=     1000 work=         2093 work/insert=      2.093  expansions=8
final size=2187
add 100 slots          n=     1000 work=         5500 work/insert=      5.500
expansions=10  final size=1000

double the size        n=    10000 work=        26383 work/insert=      2.638
expansions=15  final size=16384
triple the size        n=    10000 work=        19841 work/insert=      1.984
expansions=10  final size=19683
add 100 slots          n=    10000 work=       505000 work/insert=     50.500
expansions=100  final size=10000

double the size        n=   100000 work=       231071 work/insert=      2.311
expansions=18  final size=131072
triple the size        n=   100000 work=       188573 work/insert=      1.886
expansions=12  final size=177147
add 100 slots          n=   100000 work=     50050000 work/insert=    500.500
expansions=1000  final size=100000

double the size        n=  1000000 work=      2048575 work/insert=      2.049
expansions=21  final size=1048576
triple the size        n=  1000000 work=      1797161 work/insert=      1.797
expansions=14  final size=1594323
add 100 slots          n=  1000000 work=   5000500000 work/insert=   5000.500
expansions=10000  final size=1000000

Doubling stays below its bound of 3 units per insert and tripling
below its bound of 2.5: both constant.  Adding a fixed 100 slots grows
without bound -- about n/200 per insert, which is Theta(n).
```

Baris pelipatan dan pelipattigaannya bertahan mantap antara 1,8 dan 2,7 seiring n tumbuh tiga orde besaran, selalu di bawah langit-langit terbuktinya, 3 dan 2,5. Baris penambahan tetapnya dikalikan 10 setiap kali n dikalikan 10: 5.5, 50.5, 500.5, 5000.5. Itulah tanda tangan $\Theta(n)$.

14.10 Dengan penambahan tetap sebesar $s = 100$ slot, tabelnya mulai tanpa slot lalu tumbuh menjadi $s, 2s, 3s, \ldots$ Mari kita cacah selangkah demi selangkah.

Kapan perluasan terjadi? Perluasan terjadi setiap kali sebuah penyisipan menjumpai tabel yang penuh, yakni ketika $num = size$. Keadaan itu terjadi sebelum penyisipan 1 (dengan $num = size = 0$), sebelum penyisipan $s + 1$ (dengan $num = size = s$), sebelum penyisipan $2s + 1$, dan

seterusnya. Jadi selama n penyisipan ada $m = \lceil n/s \rceil$ perluasan, dan perluasan ke-j terjadi ketika tabelnya memuat $(j-1)s$ item.

Berapa biaya masing-masing? Perluasan ke-j menyalin $(j-1)s$ item yang sudah ada. Yang pertama tidak menyalin apa pun; yang kedua menyalin s; yang terakhir menyalin $(m-1)s$.

Jumlahkan. Dengan memakai $1 + 2 + \cdots + (m-1) = \frac{m(m-1)}{2}$,

$$\sum_{j=1}^{m} (j-1)\ s \ = \ s \cdot \frac{m(m-1)}{2}, \qquad m = \left\lceil \frac{n}{s} \right\rceil.$$

Dengan menambahkan n operasi tulis untuk item itu sendiri, total yang **eksak** adalah

$$T(n) \ = \ n + s \cdot \frac{m(m-1)}{2} \ \approx \ n + \frac{n^2}{2s} \ = \ n + \frac{n^2}{200} \ = \ \Theta(n^2), \tag{14.18}$$

sehingga biaya amortisasi per penyisipannya kira-kira $n/(2s)$, yaitu $\Theta(n)$ — **linear, bukan konstan.**

Ujilah Persamaan (14.18) terhadap pengukuran di **Kode 14.13**. Pada $n = 10^6$, $m = 10^4$ dan rumusnya memberikan $10^6 + 100 \cdot \frac{10^4 \cdot 9999}{2} = 10^6 + 4{,}999{,}500{,}000 = 5{,}000{,}500{,}000$; programnya melaporkan 5,000,500,000. Pada $n = 10^5$, $m = 1000$ dan rumusnya memberikan $10^5 + 100 \cdot \frac{1000 \cdot 999}{2} = 50{,}050{,}000$; programnya melaporkan 50,050,000. Pada $n = 1000$ rumusnya memberikan $1000 + 100 \cdot \frac{10 \cdot 9}{2} = 5500$, lagi-lagi angka programnya. Persis.

Alasannya, sekali lagi: biaya perluasannya membentuk deret **aritmetika** $s, 2s, 3s, \ldots$, yang jumlahnya kuadratik, alih-alih deret **geometri**, yang jumlahnya linear terhadap suku terbesarnya.

14.11 Ya, ia bertahan. Sembarang ambang penyusutan τ dengan $0 < \tau < 1/2$ berhasil, dan $\tau = 1/3$ adalah ambang semacam itu.

Mengapa kelonggarannya cukup. Sesudah sebuah perluasan, faktor muatannya 1/2. Sesudah sebuah penyusutan dengan menyeparuhkan pada ambang 1/3, tabelnya memuat sedikit di bawah $size/3$ item di dalam $size/2$ slot, sehingga faktor muatan barunya sedikit di bawah 2/3 — khususnya, sekurang-kurangnya 1/3 dan paling besar 2/3. Dari titik istirahat mana pun, mencapai pemicu berikutnya menuntut penggal konstan dari *size* operasi: mencapai faktor muatan 1 dari 1/2 memakan $size/2$ penyisipan; mencapai faktor muatan 1/3 dari 1/2 memakan $size/6$ penghapusan. Jadi tiap peristiwa $\Theta(size)$ didahului oleh $\Theta(size)$ operasi murah, yang merupakan syarat sebuah batas amortisasi.

Fungsi potensialnya.

$$\Phi(T) \ = \ \begin{cases} 2\,num[T] - size[T] & \text{bila } \alpha(T) \geq \frac{1}{2}, \\ size[T] - 2\,num[T] & \text{bila } \alpha(T) < \frac{1}{2}. \end{cases}$$

Setara dengan itu, $\Phi(T) = |\,2\,num[T] - size[T]\,|$.

Mengapa itu fungsi yang benar. Ia 0 persis ketika tabelnya separuh penuh — keadaan istirahat sesudah sebuah perluasan — dan tak-negatif di mana-mana, karena ia sebuah nilai mutlak. Ia naik

ke *size* seiring tabelnya terisi, cocok dengan biaya perluasannya. Dan ia naik seiring tabelnya mengosong: pada pemicu penyusutannya, $num \approx size/3$, sehingga $\Phi \approx size - 2\,size/3 = size/3$, yang persis banyaknya item yang harus disalin penyusutannya. Pada kedua arah, potensialnya sudah menabung biaya peristiwa yang terlihat mendekat, dan hanya itulah yang harus dilakukan sebuah fungsi potensial.

Ambang umumnya. Untuk ambang penyusutan $\tau < 1/2$, ambillah cabang bawahnya sebagai $\frac{\tau}{\frac{1}{2}-\tau}(1/2\,size - num)$; koefisiennya dipilih sedemikian sehingga Φ sama dengan biaya penyusutan $\tau \cdot size$ pada saat pemicunya meletus. Dengan $\tau = 1/4$ koefisiennya 1, yang memulihkan §14.7.7; dengan $\tau = 1/3$ ia 2, yang memulihkan rumus di atas.

14.12 Tulislah α_i, num_i, $size_i$ untuk faktor muatan, cacah item, dan ukuran sesudah operasi i, lalu ingatlah potensial Persamaan (14.15):

$$\Phi = \begin{cases} 2\,num - size & \text{bila } \alpha \geq \frac{1}{2}, \\ \frac{1}{2}size - num & \text{bila } \alpha < \frac{1}{2}. \end{cases}$$

`TABLE-INSERT`, **kasus A:** $\alpha_{i-1} \geq 1/2$**.** Maka $\alpha_i \geq \alpha_{i-1} \geq 1/2$ juga, sehingga kedua potensialnya memakai cabang atasnya dan perhitungannya kata demi kata sama dengan §14.7.5: $\hat{c}_i = 3$, terjadi perluasan atau tidak.

`TABLE-INSERT`, **kasus B:** $\alpha_{i-1} < 1/2$**.** Tabelnya kurang dari separuh penuh, sehingga ia pasti tidak penuh, sehingga tidak ada perluasan dan $c_i = 1$, $size_i = size_{i-1} = s$.

Anak kasus B1: $\alpha_i < 1/2$ *juga.* Kedua potensialnya memakai cabang bawahnya:

$$\begin{aligned} \hat{c}_i \;=\; 1 + \left(\frac{s}{2} - num_i\right) - \left(\frac{s}{2} - num_{i-1}\right) \;&=\; 1 + (num_{i-1} - num_i) \\ &=\; 1 - 1 \;=\; 0. \end{aligned}$$

Anak kasus B2: $\alpha_{i-1} < 1/2 \leq \alpha_i$. Potensialnya berpindah cabang:

$$\begin{aligned} \hat{c}_i &= 1 + (2\,num_i - s) - \left(\frac{s}{2} - num_{i-1}\right) \\ &= 1 + 2(num_{i-1} + 1) - s - \frac{s}{2} + num_{i-1} \\ &= 3\,num_{i-1} + 3 - \frac{3}{2}s. \end{aligned}$$

Karena $\alpha_{i-1} < 1/2$ memberikan $num_{i-1} < s/2$,

$$\hat{c}_i \;<\; 3 \cdot \frac{s}{2} + 3 - 3/2\,s \;=\; 3.$$

Jadi `TABLE-INSERT` punya $\hat{c}_i \leq 3$ pada setiap kasusnya.

`TABLE-DELETE`, **kasus C:** $\alpha_{i-1} \geq 1/2$**.** Tabelnya sekurang-kurangnya separuh penuh, sehingga satu penghapusan tunggal tidak dapat membawanya di bawah seperempat (asalkan $size \geq 4$); tidak ada penyusutan, $c_i = 1$, $size_i = size_{i-1} = s$, $num_i = num_{i-1} - 1$.

Anak kasus C1: $\alpha_i \geq 1/2$. Kedua cabangnya atas:

$$\hat{c}_i \;=\; 1 + (2\,num_i - s) - (2\,num_{i-1} - s) \;=\; 1 + 2(num_i - num_{i-1}) \\ =\; 1 - 2 \;=\; -1.$$

Anak kasus C2: $\alpha_i < 1/2 \le \alpha_{i-1}$. Perpindahan cabang ke arah sebaliknya:

$$\begin{aligned}\hat{c}_i &= 1 + \left(\frac{s}{2} - num_i\right) - (2\,num_{i-1} - s)\\ &= 1 + \frac{3}{2}s - num_i - 2(num_i + 1)\\ &= \frac{3}{2}s - 3\,num_i - 1.\end{aligned}$$

Dari $\alpha_{i-1} \ge 1/2$ kita memperoleh $s \le 2\,num_{i-1} = 2\,num_i + 2$, sehingga

$$\hat{c}_i \;\le\; \frac{3}{2}(2\,num_i + 2) - 3\,num_i - 1 \;=\; 3\,num_i + 3 - 3\,num_i - 1 \;=\; 2.$$

`TABLE-DELETE`**, kasus D:** $\alpha_{i-1} < 1/2$**.** Kedua potensialnya memakai cabang bawahnya.

Anak kasus D1: tidak ada penyusutan. $c_i = 1$, $size_i = size_{i-1} = s$:

$$\hat{c}_i \;=\; 1 + \left(\frac{s}{2} - num_i\right) - \left(\frac{s}{2} - num_{i-1}\right) \;=\; 1 + (num_{i-1} - num_i) \\ =\; 1 + 1 \;=\; 2.$$

Anak kasus D2: sebuah penyusutan meletus. Ia meletus ketika penghapusannya meninggalkan $num_i < s/4$; dengan s sebuah pangkat dua ini berarti $num_i = \frac{s}{4} - 1$, dan kemudian $size_i = \frac{s}{2}$ dan $c_i = num_i + 1$ (satu penghapusan ditambah num_i salinan).

$$\begin{aligned}\hat{c}_i &= (num_i + 1) + \left(\frac{s}{4} - num_i\right) - \left(\frac{s}{2} - num_{i-1}\right)\\ &= (num_i + 1) + \frac{s}{4} - num_i - \frac{s}{2} + (num_i + 1)\\ &= num_i + 2 - \frac{s}{4}\\ &= \left(\frac{s}{4} - 1\right) + 2 - \frac{s}{4}\\ &= 1.\end{aligned}$$

Jadi `TABLE-DELETE` punya $\hat{c}_i \le 2$ pada setiap kasusnya. Biaya amortisasi terbesar atas kedelapan kasusnya adalah 3, sehingga sembarang barisan berisi n penyisipan dan penghapusan berbiaya paling besar $3n + \Phi(D_0) = 3n$ bila bermula dari tabel kosong. ■

14.13 Kebenarannya — dan itu saja sudah fatal. Batas biayanya, secara mengejutkan, tetap bertahan.

Kebenarannya. Item di S_{IN} duduk dengan yang tertua di dasar. Mem-pop separuhnya memindahkan separuh yang **terbaru** ke S_{OUT}, tempat mereka mendarat di puncak dan dilayani lebih dahulu. Jadi `Dequeue` yang paling berikutnya mengembalikan seorang pendatang baru sementara item yang lebih tua masih terkubur di S_{IN}. Strukturnya tidak lagi sebuah antrean.

Secara konkret: `Enqueue 1, 2, 3, 4`, lalu `Dequeue`. S_{IN} adalah [1,2,3,4] dari dasar ke puncak. Memindahkan separuhnya berarti mem-pop 4 dan 3 lalu mem-push-nya ke S_{OUT}, yang menjadi [4,3] dari dasar ke puncak, sehingga puncaknya 3. `Dequeue`-nya mengembalikan **3**, padahal ia seharusnya mengembalikan 1.

Biayanya. Pertahankan potensial $\Phi = 2 \times |S_{IN}|$ lalu telusurilah sebuah pemindahan separuh selangkah demi selangkah. Andaikan S_{OUT} kosong dan S_{IN} memuat e item.

1. Pemindahannya menggeser $e/2$ item, masing-masing seharga 2 satuan (satu pop, satu push): e satuan. Pop terakhirnya menambah 1. Jadi biaya sebenarnya $c_i = e + 1$.
2. Sesudahnya S_{IN} masih memuat $e/2$ item, sehingga potensialnya jatuh dari $2e$ ke $2 \cdot \frac{e}{2} = e$: turun sebesar e.
3. Karena itu biaya amortisasinya $\hat{c}_i = (e + 1) - e = 1$, persis seperti pada pemindahan penuh.

`Enqueue` tetap ditagih 3 dan `Dequeue` biasa tetap 1, sehingga batas amortisasi $O(1)$ per operasi tetap bertahan. Alasannya, tiap item tetap menyeberang dari S_{IN} ke S_{OUT} **paling banyak sekali** sepanjang hidupnya: item yang tertinggal di S_{IN} tidak dipindahkan dua kali, ia sekadar dipindahkan belakangan. Jadi perubahan ini murni cacat kebenaran — dan itu justru lebih buruk, sebab program yang cepat tetapi salah tidak memberi peringatan apa pun. **Pemindahan penuh tidak diperlukan demi kecepatan; ia diperlukan demi pembalikan yang membuat urutannya masuk-pertama-keluar-pertama.**

14.14 Klaim. Di dalam sebuah hutan himpunan saling lepas yang dipelihara dengan gabung menurut peringkat, sembarang pohon yang akarnya berperingkat r memuat sekurang-kurangnya 2^r simpul.

Bukti dengan induksi atas banyaknya operasi `Union` yang dilakukan.

Basis. Sebelum ada penggabungan apa pun, tiap pohon adalah satu simpul tunggal berperingkat 0, dan $2^0 = 1$ simpul. Klaimnya berlaku.

Langkah induksi. Andaikan klaimnya berlaku bagi tiap pohon di dalam hutan yang sekarang. Tinjaulah `Union` berikutnya, yang menggabungkan pohon berakar x dan y, berperingkat r_x dan r_y serta berukuran $n_x \geq 2^{r_x}$, $n_y \geq 2^{r_y}$ menurut hipotesisnya. Gabung menurut peringkat melakukan salah satu dari tiga hal.

- Bila $r_x < r_y$: x digantungkan di bawah y. Akar barunya adalah y, peringkatnya tidak berubah pada r_y, dan pohon barunya punya $n_x + n_y > n_y \geq 2^{r_y}$ simpul. Klaimnya berlaku.
- Bila $r_x > r_y$: setangkup.
- Bila $r_x = r_y = r$: salah satu akarnya digantungkan di bawah yang lain dan peringkat yang bertahan menjadi $r + 1$. Pohon barunya punya $n_x + n_y \geq 2^r + 2^r = 2^{r+1}$ simpul. Klaimnya berlaku.

Tidak ada operasi lain yang mengubah sebuah peringkat. (Pemampatan lintasan memindahkan simpul tetapi tidak pernah mengubah sebuah peringkat dan tidak pernah membuang sebuah simpul dari pohonnya, sehingga ia tidak mungkin melanggar klaimnya.) ■

Penyimpulannya. Misalkan r peringkat sembarang akar. Pohonnya punya sekurang-kurangnya 2^r simpul, dan tidak ada pohon yang dapat punya lebih dari n simpul, sehingga $2^r \leq n$ dan karenanya $r \leq \lg n$; karena r bilangan bulat, $r \leq \lfloor \lg n \rfloor$. Karena peringkat adalah batas atas bagi tingginya (ia memang didefinisikan demikian, dan ketiga kasus di atas memeliharanya), **tidak**

ada pohon yang tingginya lebih dari $\lfloor \lg n \rfloor$, sehingga `FindSet` menyusuri paling banyak $\lfloor \lg n \rfloor$ penunjuk. ■

Ini dipastikan oleh pengukuran: di **Kode 14.10**, beban kerja 2 dengan gabung menurut peringkat saja melaporkan pohon tertinggi setinggi tepat 15 pada $n = 32768 = 2^{15}$ elemen.

14.15 Ingatlah $A_0(j) = j + 1$ dan $A_k(j) = A_{k-1}^{(j+1)}(j)$ untuk $k \geq 1$.

Bentuk tertutup ketiga tingkat pertamanya.

$A_0(j) = j + 1$.

$A_1(j) = A_0^{(j+1)}(j)$ — tambahkan 1 kepada j, sebanyak $j + 1$ kali — $= j + (j + 1) = 2j + 1$.

$A_2(j) = A_1^{(j+1)}(j)$. Menerapkan A_1 sekali mengirim $j \mapsto 2j + 1$, yang dalam koordinat "tambah satu" adalah $j + 1 \mapsto 2(j + 1)$; sehingga menerapkannya i kali mengirim $j \mapsto 2^i(j + 1) - 1$. Dengan $i = j + 1$:

$$A_2(j) \;=\; 2^{j+1}(j + 1) - 1.$$

Periksa: $A_2(1) = 2^2 \cdot 2 - 1 = 7$. ✓

$A_3(1)$**, secara eksak.**

$$A_3(1) \;=\; A_2^{(2)}(1) \;=\; A_2(A_2(1)) \;=\; A_2(7).$$

Dan $A_2(7) = 2^8 \cdot 8 - 1 = 256 \cdot 8 - 1 = 2048 - 1 = \boxed{2047}$.

$A_4(1) > 2^{2048}$**.**

$$A_4(1) \;=\; A_3^{(2)}(1) \;=\; A_3(A_3(1)) \;=\; A_3(2047) \;=\; A_2^{(2048)}(2047).$$

Penerapan yang paling pertama dari 2048 penerapan itu sudah memberikan

$$A_2(2047) \;=\; 2^{2048} \cdot 2048 - 1 \;>\; 2^{2048},$$

dan tiap penerapan berikutnya hanya menaikkan nilainya, sebab $A_2(j) > j$ untuk semua $j \geq 1$. Jadi $A_4(1) > 2^{2048}$ — sesungguhnya jauh lebih besar, sebab 2047 penerapan sisanya masing-masing memangkatkan lagi.

Kesimpulannya. Menurut definisinya $\alpha(n) = \min\{k : A_k(1) \geq n\}$. Untuk sembarang $n \leq 2^{2048}$ kita punya $A_4(1) > 2^{2048} \geq n$, sehingga $k = 4$ sudah berhasil, dan karena itu $\alpha(n) \leq 4$.

Alam semesta teramati memuat kira-kira 2^{266} atom. Karena $2^{266} \ll 2^{2048}$, **setiap persoalan himpunan saling lepas yang akan pernah dapat diwakilkan secara fisik punya** $\alpha(n) \leq 4$**.** Agar α mencapai 5 Anda memerlukan lebih dari $A_4(1)$ elemen, yang bukan kekurangan rekayasa melainkan kekurangan alam semesta.

14.16 **Kode 14.14** mencatat kapasitasnya sesudah tiap `push_back`, mencetak tiap realokasi beserta nisbah pertumbuhannya, lalu menjumlahkan salinannya.

Kode 14.14 Mengukur apa yang sungguh-sungguh dilakukan `std::vector<int>::push_back`. Kompilasi dengan `g++ -std=c++17 -O2 -o vectorgrow vectorgrow.cpp`.

```
// Penyelesaian Soal 14.16.  Apa yang sebenarnya dilakukan std::vector ketika
// ia tumbuh?  Program ini mengamati capacity() sepanjang sejuta pemanggilan
// push_back, mencatat tiap realokasi, dan menghitung total banyaknya
// elemen yang disalin, membandingkannya dengan batas 3n dari §14.7.
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

int main() {
    vector<int> v;
    size_t cap = v.capacity();
    long long copied = 0, reallocations = 0;

    cout << "first twelve reallocations:\n";
    cout << "  at push #   old capacity   new capacity   growth ratio\n";
    for (long long i = 1; i <= 1000000; ++i) {
        v.push_back((int)i);
        if (v.capacity() != cap) {
            copied += i - 1;                // elemen yang sudah ada di sana
            ++reallocations;
            if (reallocations <= 12)
                cout << setw(11) << i << setw(15) << cap
                     << setw(15) << v.capacity() << setw(15)
                     << fixed << setprecision(3)
                     << (cap == 0 ? 0.0 : (double)v.capacity() / (double)cap) << "\n";
            cap = v.capacity();
        }
    }

    cout << "\nover 1000000 push_back calls:\n";
    cout << "  reallocations              = " << reallocations << "\n";
    cout << "  elements copied in total   = " << copied << "\n";
    cout << "  total element writes       = " << copied + 1000000
         << "  (copies plus the pushes themselves)\n";
    cout << "  writes per push_back       = " << fixed << setprecision(6)
         << (double)(copied + 1000000) / 1000000.0 << "\n";
    cout << "  the 3n bound of Section 14.7 = " << 3 * 1000000 << "\n";
    cout << "  final size = " << v.size() << ", final capacity = " << v.capacity() << "\n";
    return 0;
}
```

```
first twelve reallocations:
  at push #   old capacity   new capacity   growth ratio
          1              0              1          0.000
          2              1              2          2.000
          3              2              4          2.000
          5              4              8          2.000
          9              8             16          2.000
         17             16             32          2.000
         33             32             64          2.000
         65             64            128          2.000
        129            128            256          2.000
        257            256            512          2.000
        513            512           1024          2.000
       1025           1024           2048          2.000

over 1000000 push_back calls:
  reallocations              = 21
  elements copied in total   = 1048575
  total element writes       = 2048575  (copies plus the pushes themselves)
  writes per push_back       = 2.048575
  the 3n bound of Section 14.7 = 3000000
  final size = 1000000, final capacity = 1048576
```

Segala yang diramalkan bab ini kelihatan di situ.

Realokasinya terjadi pada push ke-1,2,3,5,9,17,33,65, … — yaitu, pada $2^j + 1$ — yang persis syarat perluasan di §14.7.2: **penyisipan i memicu perluasan persis ketika $i - 1$ sebuah pangkat 2.**

Nisbah pertumbuhannya tepat 2,000 setiap kali: implementasi ini melipatduakan, sebagaimana dilakukan `TABLE-INSERT` pada **Gambar 14.16**. Standar C++ [12] hanya menuntut agar `push_back` berwaktu konstan teramortisasi; faktor 2 adalah pilihan pustaka ini, dan pustaka lain memakai 1,5, yang menurut Soal 14.9 sama sahnya.

Sejuta push hanya menyebabkan **21** realokasi, dan menyalin 1.048.575 elemen seluruhnya, yang adalah $2^{20} - 1$ — satu kurang dari kapasitas akhirnya, persis sebagaimana dituntut jumlah geometrinya $1 + 2 + 4 + \cdots + 2^{19} = 2^{20} - 1$.

Total tulisan elemennya: 2.048.575, atau 2,049 per `push_back`, melawan langit-langit terbukti sebesar 3. Bandingkan dengan **Kode 14.8**, yang mengukur 2.048.575 untuk n yang sama dari implementasi yang berdiri sendiri. Pustaka bakunya dan buku teksnya sepakat sampai ke satuannya.

14.17 Barisannya. Misalkan k lebar pencacahnya dan misalkan n genap. Setellah pencacahnya ke $2^{k-1} - 1$, yang representasi binernya adalah $k - 1$ angka satu. Sekarang selang-selingkanlah:

$$\text{Increment},\ \text{Decrement},\ \text{Increment},\ \text{Decrement},\ \ldots$$

- `Increment`-nya membawa $0\underbrace{11\cdots1}_{k-1}$ ke $1\underbrace{00\cdots0}_{k-1}$: ia menolkan $k-1$ angka satu lalu menyetel satu bit, berbiaya k.
- `Decrement`-nya membawa $1\underbrace{00\cdots0}_{k-1}$ kembali ke $0\underbrace{11\cdots1}_{k-1}$: ia harus meminjam menembus $k-1$ angka nol, menyetel masing-masingnya menjadi 1, lalu membersihkan bit teratasnya, berbiaya k.

Tiap operasi berbiaya k, sehingga n operasi berbiaya nk. Dengan $k = \lg n$ itu $\Theta(n\lg n)$, sehingga biaya amortisasi per operasinya $\Theta(k)$, **bukan** $O(1)$. (Bila pencacahnya harus mulai dari nol, hal yang sama terjadi dengan lebih sederhana lagi: satu `Decrement` dari nol berputar menjadi semua satu dengan biaya k, `Increment` berikutnya berputar kembali ke nol dengan biaya k, dan pergantian keduanya membalik seluruh k bit setiap kali.)

Mengapa Φ = banyaknya bit 1 gagal. Hitunglah biaya amortisasi `Decrement` di atas. Biaya sebenarnya $c_i = k$. Sebelumnya pencacahnya punya 1 bit satu; sesudahnya, $k-1$ bit satu. Jadi

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}) \;=\; k + (k-1) - 1 \;=\; 2k - 2,$$

yang tumbuh bersama k alih-alih terbatas. Potensialnya *naik* hampir sebesar k pada saat yang sama ketika operasinya mengerjakan k satuan kerja, sehingga alih-alih saling meniadakan, kedua sukunya justru saling menambah.

Alasan yang lebih dalam. Potensial $= $ (banyaknya bit 1) berhasil untuk `Increment` sendirian sebab `Increment` adalah satu-satunya operasinya, dan satu-satunya cara menimbun bit 1 adalah membayar 2 untuk masing-masingnya. Menambahkan `Decrement` memberi si lawan sebuah jalan untuk *menciptakan* bit 1 secara murah relatif terhadap potensial yang tersimpan — lalu memusnahkannya lagi — sehingga strukturnya dapat digiring bolak-balik melintasi sebuah batas yang mahal tanpa henti. Inilah versi pencacah dari *thrashing* tabel di §14.7.6, dan diagnosisnya sama: **tidak ada kelonggaran antara keadaan yang mahal dan tetangganya.**

(Sebagai catatan, sebuah pencacah *dapat* mendukung kedua operasi itu dalam waktu $O(1)$ teramortisasi, tetapi tidak dalam representasi ini. Representasi biner berlebih, yang mengizinkan digit 2 di samping 0 dan 1, memang mendukung increment dan decrement dalam waktu $O(1)$ *kasus terburuk* — yang merupakan ilustrasi bagus bagi pokok di §14.11 bahwa ketika sebuah batas amortisasi tidak memadai, jawabannya biasanya representasi yang berbeda alih-alih analisis yang lebih cerdik.)

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] M. L. Fredman and R. E. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[3] R. E. Tarjan, “Amortized computational complexity,” *SIAM Journal on Algebraic and Discrete Methods*, vol. 6, no. 2, pp. 306–318, Apr. 1985.

[4] D. D. Sleator and R. E. Tarjan, “Self-adjusting binary search trees,” *Journal of the ACM*, vol. 32, no. 3, pp. 652–686, Jul. 1985.

[5] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[6] B. A. Galler and M. J. Fisher, “An improved equivalence algorithm,” *Communications of the ACM*, vol. 7, no. 5, pp. 301–303, May 1964.

[7] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[8] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, Apr. 1975.

[9] W. Ackermann, "Zum Hilbertschen Aufbau der reellen Zahlen" ["On Hilbert's construction of the real numbers"], *Mathematische Annalen*, vol. 99, no. 1, pp. 118–133, Dec. 1928.

[10] R. E. Tarjan, *Data Structures and Network Algorithms*. Philadelphia, PA, USA: SIAM, 1983.

[11] M. L. Fredman and M. E. Saks, "The cell probe complexity of dynamic data structures," in *Proc. 21st Annual ACM Symposium on Theory of Computing (STOC)*, Seattle, WA, USA, 1989, pp. 345–354.

[12] ISO/IEC 14882:2017, *Programming Languages — C++*, International Organization for Standardization, Geneva, Switzerland, 2017.

Bab 15
Lintasan Terpendek Semua Pasangan

Di akhir bab ini Anda akan mampu mengambil sebuah graf berarah berbobot lalu menghasilkan, dalam satu tabel, rute termurah antara *setiap* pasangan terurut simpulnya — bukan satu perjalanan, melainkan semuanya sekaligus. Anda akan mampu menyebutkan dengan tepat berapa biaya pendekatan naifnya, dan mengapa pendekatan itu kadang-kadang justru jawaban yang benar. Anda akan mampu menurunkan, bermodalkan tak lebih daripada definisi lintasan terpendek, rekurensi $l_{ij}^{(m)} = \min(l_{ij}^{(m-1)}, \min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\})$, mengenali di dalamnya bentuk perkalian matriks biasa, lalu memanfaatkan kemiripan itu dua kali: sekali untuk memperoleh sebuah algoritma, dan sekali lagi — lewat pemangkatan berulang (*repeated squaring*) — untuk membuatnya lebih cepat. Anda akan mampu menyatakan, menurunkan, mengimplementasikan, menelusuri, dan membuktikan kebenaran algoritma Floyd–Warshall, serta menyebutkan dengan persis mengapa gelung atas k haruslah yang terluar di antara ketiganya. Anda akan mampu merekonstruksi rute yang sesungguhnya, bukan sekadar ongkosnya, dari sebuah matriks pendahulu Π. Anda akan mampu menghitung penutupan transitif sebuah graf berarah dengan mengubah dua operator dan tidak lebih. Anda akan mampu membobot ulang sebuah graf bersisi negatif sehingga Dijkstra kembali sah dipakai, membuktikan bahwa pembobotan ulang itu tidak mengubah satu pun lintasan terpendek, dan dengan begitu mengimplementasikan algoritma Johnson untuk graf jarang. Dan Anda akan mampu menjawab, ketika seseorang menyerahkan sebuah graf kepada Anda lalu bertanya algoritma mana di antara keempatnya yang harus dijalankan, dengan sebuah angka, bukan dengan sebuah pendapat.

15.1 Dari Mana Kita Melanjutkan

Acuan bakunya adalah Bab 23 Cormen, Leiserson, Rivest, dan Stein [1] (Bab 25 pada edisi ketiga, yang nomor gambarnya disebut oleh beberapa keterangan gambar di bawah); makalah-makalah aslinya adalah catatan setengah halaman Floyd tahun 1962 [2], makalah matriks boolean Warshall pada tahun yang sama [3], dan makalah pembobotan ulang Johnson tahun 1977 [4]. Kleinberg dan Tardos [5] memberikan uraian yang teramat enak dibaca untuk sisi Bellman–Ford ceritanya, dan Sedgewick serta Wayne [6] untuk sisi implementasinya.

15.1.1 Apa yang Bab 13 berikan kepada kita

Bab 13 menyelesaikan **persoalan lintasan terpendek sumber tunggal**. Latarnya adalah sebuah graf berarah berbobot $G = (V, E)$ beserta sebuah **fungsi bobot** $w: E \to \mathbb{R}$ yang melekatkan sebuah bilangan riil pada setiap sisi. **Bobot sebuah lintasan** $p = \langle v_0, v_1, \dots, v_k\rangle$ adalah jumlah bobot sisi di sepanjangnya,

$$w(p) = \sum_{i=1}^{k} w\,(v_{i-1}, v_i),$$

dan **bobot lintasan terpendek** dari u ke v adalah

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \rightsquigarrow v \text{ sepanjang } p\} & \text{jika lintasan semacam itu a} \\ \infty & \text{jika tidak.} \end{cases}$$

Lambang $\rightsquigarrow$ berarti "ada lintasan dari … ke …", dan δ (huruf Yunani *delta*) disediakan di sepanjang buku ini untuk bobot lintasan terpendek yang *sesungguhnya* — jawabannya, benda yang sedang berusaha kita hitung. Setiap algoritma di Bab 13 bekerja dengan memelihara sebuah **taksiran** $d[v]$, yaitu bilangan yang mula-mula terlalu besar lalu berulang kali ditekan turun, bersama sebuah **pendahulu** $\pi[v]$, yaitu simpul tempat v untuk saat ini paling baik dicapai. Satu-satunya operasi yang menekan taksiran itu turun adalah **relaksasi**:

> **DEFINISI (diulang dari Bab 13). Merelaksasi** sebuah sisi (u, v) berarti menanyakan apakah pergi ke v lewat u lebih baik daripada rute terbaik menuju v yang sejauh ini ditemukan, lalu mencatat perbaikannya bila memang lebih baik:

$$\text{jika } d[u] + w(u,v) < d[v] \text{ maka } d[v] \leftarrow d[u] + w(u,v) \text{ dan } \pi[v] \leftarrow u.$$

> Setiap algoritma lintasan terpendek yang benar di buku ini adalah sebuah siasat untuk memutuskan sisi *yang mana* yang direlaksasi berikutnya, dan *berapa kali*.

Bab 13 memberikan tiga siasat semacam itu:

- **Algoritma Dijkstra**, yang menuntut setiap bobot tak negatif, mengeluarkan simpul dari antrean prioritas-min menurut d yang menaik, dan merelaksasi tiap sisi persis sekali. Dengan larik linear sebagai antreannya ia berongkos $O(V^2 + E)$; dengan tumpukan-min biner, $O(E \lg V)$; dengan *Fibonacci heap*, $O(V \lg V + E)$.
- **Bellman–Ford**, yang tahan terhadap bobot negatif, merelaksasi *setiap* sisi sebanyak $|V| - 1$ kali, berongkos $O(VE)$, dan mendeteksi siklus berbobot negatif dengan menguji apakah masih ada relaksasi yang bisa dilakukan.
- **Algoritma DAG**, yang mengurutkan graf asiklik secara topologis lalu merelaksasi menurut urutan itu, berongkos $\Theta(V + E)$.

Bab 13 juga membuktikan dua fakta yang akan terus-menerus kita sandari. Yang pertama adalah **teorema substruktur optimal**, yang di sana dibuktikan sebagai Teorema 13.1. Pada edisi ketiga buku Cormen dkk. ia adalah Lema 24.1; di edisi keempat [1] ia menjadi Lema 22.1. Dengan label mana pun Anda menjumpainya, isinya:

> **TEOREMA 13.1, DIULANG (sublintasan lintasan terpendek adalah lintasan terpendek; Lema 22.1 pada** [1]**).** Misalkan $p = \langle v_0, v_1, \ldots, v_k \rangle$ sebuah lintasan terpendek dari v_0 ke v_k, dan untuk sembarang $0 \le a \le b \le k$ misalkan $p_{ab} = \langle v_a, v_{a+1}, \ldots, v_b \rangle$ potongan p antara v_a dan v_b. Maka p_{ab} adalah lintasan terpendek dari v_a ke v_b.

Buktinya satu baris hitungan dan sebuah kontradiksi: uraikanlah $w(p) = w(p_{0a}) + w(p_{ab}) + w(p_{bk})$. Seandainya ada lintasan p'_{ab} dari v_a ke v_b yang lebih ringan daripada p_{ab}, maka menyisipkannya akan memberikan lintasan dari v_0 ke v_k berbobot $w(p_{0a}) + w(p'_{ab}) + w(p_{bk}) < w(p)$, bertentangan dengan anggapan bahwa p terpendek. Lema inilah seluruh alasan mengapa pemrograman dinamis dapat diterapkan pada lintasan terpendek, dan ia akan dipakai, dalam dua samaran yang berbeda, di §15.4 dan §15.6.

Fakta yang kedua menyangkut **siklus berbobot negatif**. Jika sebuah siklus berbobot total negatif terletak pada rute dari u ke v, maka mengitarinya sekali lagi menurunkan ongkosnya, dan mengitarinya seribu kali menurunkannya lebih jauh lagi; tidak ada minimum, dan $\delta(u, v)$ bernilai $-\infty$. Segala sesuatu di bab ini menganggap masukannya **tanpa siklus berbobot negatif**. *Sisi* negatif dipersilakan — contoh berjalan bab ini punya dua — tetapi *siklus* negatif tidak. §15.6.14 memperlihatkan cara mendeteksinya tanpa biaya tambahan.

15.1.2 Pertanyaan yang baru

Bab 13 menjawab "bagaimana saya pergi dari sini ke mana-mana?" Bab ini menjawab "bagaimana setiap orang sampai ke setiap orang?" Dinyatakan dengan persis:

> **DEFINISI (persoalan lintasan terpendek semua pasangan).** Diberikan sebuah graf berarah berbobot $G = (V, E)$ dengan fungsi bobot $w: E \to \mathbb{R}$, carilah, untuk **setiap** pasangan terurut simpul $u, v \in V$, sebuah lintasan terpendek dari u ke v beserta bobotnya $\delta(u, v)$. Keluarannya secara alamiah berbentuk **tabel**: entri pada baris u dan kolom v adalah bobot lintasan terpendek dari u ke v.

Perhatikanlah kata *terurut*. Pada graf berarah, perjalanan dari u ke v dan perjalanan dari v ke u adalah dua perjalanan yang berbeda dengan biaya yang umumnya berbeda pula. Sistem jalan satu arah membuat hal ini kentara; begitu pula jadwal penerbangan. Maka tabel keluarannya punya $|V|^2$ entri, bukan $\binom{|V|}{2}$, dan ia tidak simetris.

Mengapa ada orang yang menginginkan semuanya? Sebab amat banyak pertanyaan praktis yang sesungguhnya merupakan pertanyaan tentang seluruh tabelnya, bukan tentang satu entrinya.

- **Di mana depotnya sebaiknya ditaruh?** Untuk menjawabnya Anda memerlukan jarak dari setiap calon lokasi ke setiap pelanggan. Satu sumber tidaklah cukup; Anda belum tahu sumber mana yang nanti akan Anda pilih.
- **Berapa diameter jaringan ini?** Diameternya adalah yang terbesar di antara semua $\delta(u, v)$ — perjalanan terburuk di dalam sistemnya. Anda tidak dapat menemukan yang terbesar dari sebuah himpunan tanpa menghitung himpunannya.
- **Dua halaman mana di situs ini yang paling berjauhan dalam satuan klik?** Pertanyaan yang sama, jaringan yang berbeda.
- **Apakah relasi ini transitif, dan bila tidak, apa penutupan transitifnya?** §15.7 memperlihatkan bahwa ini adalah persoalan semua pasangan dengan dua operator ditukar.
- **Prakomputasi.** Bila sebuah tabel akan ditanyai jutaan kali, menghitungnya sekali lalu mencari entrinya dalam $O(1)$ mengalahkan menjalankan ulang algoritma sumber tunggal tiap kali ditanya, asalkan grafnya muat di memori.

Mulai dari sini kita menanggalkan garis di dalam notasi asimtotiknya, persis seperti yang dilakukan Bab 13: $O(V^3)$ berarti $O(|V|^3)$. Dan karena bab ini bekerja dengan matriks, kita memberi banyaknya simpul nama pendek n, sehingga $n = |V|$ di sepanjang bab.

15.2 Persoalan Semua Pasangan, Dengan Persis

15.2.1 Masukannya: sebuah matriks ketetanggaan

Algoritma sumber tunggal di Bab 13 mengambil grafnya sebagai **daftar ketetanggaan**, sebab mereka melangkah keluar dari satu simpul dan tak pernah perlu bertanya "adakah sisi dari i ke j?" secara tiba-tiba. Algoritma di bab ini justru menanyakan pertanyaan itu terus-menerus, dan mereka menghasilkan sebuah matriks sebagai keluaran. Maka wajarlah kalau masukannya pun diambil sebagai matriks.

Nomorilah simpulnya 1,2, … , n. (Penomoran apa pun boleh; algoritmanya tidak peduli yang mana, walaupun §15.6.6 akan memperlihatkan bahwa *urutannya* teramat menentukan *bagaimana* Floyd–Warshall bekerja, sekalipun ia tidak mengubah jawabannya.) Masukannya kemudian adalah sebuah matriks $n \times n$, $W = (w_{ij})$, yang digambarkan pada **Gambar 15.1**.

Beri nomor simpul-simpulnya $1, 2, \ldots, n$ dan simpan bobot sisi graf berarah $G = (V, E)$ dalam matriks $n \times n$ $W = (w_{ij})$:

$$w_{ij} = \begin{cases} 0 & \text{jika } i = j \\ \text{bobot sisi } (i,j) & \text{jika } i \neq j \text{ dan } (i,j) \in E \\ \infty & \text{jika } i \neq j \text{ dan } (i,j) \notin E \end{cases}$$

Sisi berbobot negatif diperbolehkan; siklus berbobot negatif tidak.

Gambar 15.1 Masukan setiap algoritma di bab ini: matriks ketetanggaan $n \times n$, $W = (w_{ij})$, dari Persamaan (15.1). Sisi berbobot negatif diperbolehkan; siklus berbobot negatif tidak.

Ditulis lengkap, definisinya adalah

$$w_{ij} = \begin{cases} 0 & \text{jika } i = j, \\ \text{bobot sisi berarah } (i,j) & \text{jika } i \neq j \text{ dan } (i,j) \in E, \\ \infty & \text{jika } i \neq j \text{ dan } (i,j) \notin E. \end{cases} \tag{15.1}$$

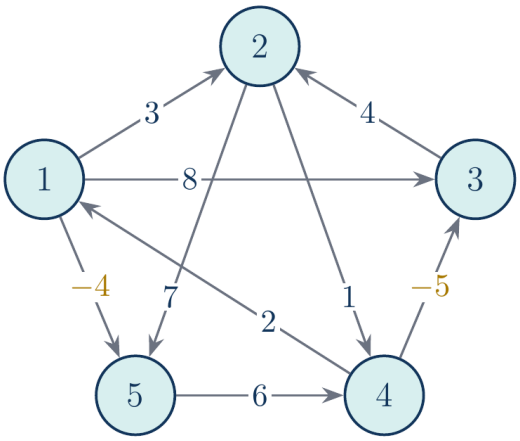
Tiga baris, dan masing-masing pantas menempati tempatnya.

Mengapa $w_{ii} = 0$. Biaya berdiam di tempat Anda berada adalah nol. Ini bukan kelaziman yang dianut demi kerapian; ia dipakai sebagai *fakta yang menanggung beban* dua kali di bab ini — sekali di §15.4.5, tempat $w_{jj} = 0$ memungkinkan kita meringkas minimum berbagian dua menjadi satu, dan sekali di §15.6.14, tempat entri diagonal yang menjadi negatif merupakan bukti bahwa ada siklus berbobot negatif.

Mengapa ∞ untuk sisi yang tidak ada. Karena $\min(x, \infty) = x$ dan $\infty + c = \infty$, lambang ∞ berperilaku persis sebagaimana seharusnya "Anda tidak bisa ke sana lewat situ" berperilaku. Ia tak pernah memenangi sebuah minimum melawan bilangan sungguhan, dan ia meracuni setiap penjumlahan yang dimasukinya. Di dalam program ia kita wakili dengan sebuah bilangan bulat besar — setiap kode di bab ini memakai `const long long INF = 1000000000LL;` — dan kita lalu harus berhati-hati agar tidak pernah menghitung `INF + INF`, yang pada `int` 32-bita akan meluap menjadi bilangan negatif dan disangka sebagai rute yang amat bagus. Setiap kode di sini menjaga penjumlahannya secara eksplisit.

Mengapa $i \neq j$ di baris kedua. Sebuah gelung-diri (i, i) berbobot positif tidak berguna: ia hanya bisa membuat rute menjadi lebih panjang. Sebuah gelung-diri berbobot negatif adalah siklus berbobot negatif, yang sudah kita kecualikan. Jadi $w_{ii} = 0$ sekaligus aman dan benar.

Untuk seluruh bab ini contoh berjalannya adalah graf bersimpul lima pada **Gambar 15.2**. Sisinya adalah $1 \to 2$ berbobot 3, $1 \to 3$ berbobot 8, $1 \to 5$ berbobot -4, $2 \to 4$ berbobot 1, $2 \to 5$ berbobot 7, $3 \to 2$ berbobot 4, $4 \to 1$ berbobot 2, $4 \to 3$ berbobot -5, dan $5 \to 4$ berbobot 6.



Gambar 15.2 Contoh berjalan: sebuah graf berarah bersimpul lima dengan sembilan sisi, dua di antaranya negatif ($1 \to 5$ berbobot -4 dan $4 \to 3$ berbobot -5). Ia tidak punya siklus berbobot negatif (§15.6.14 memeriksanya dengan mesin), sehingga setiap $\delta(i, j)$ berhingga.

Menerapkan Persamaan (15.1) simpul demi simpul memberikan

$$W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}.$$

Bacalah baris pertamanya melintang: dari simpul 1 Anda dapat berdiam di tempat dengan biaya 0; mencapai 2 langsung dengan biaya 3; mencapai 3 langsung dengan biaya 8; tidak ada sisi $1 \to 4$, sehingga ∞; dan mencapai 5 dengan biaya -4. Bacalah baris ketiganya: dari simpul 3 persis ada satu sisi keluar, menuju simpul 2, berbobot 4; selebihnya di baris itu, kecuali diagonalnya, adalah ∞. Perhatikan sekali lagi bahwa W **tidak simetris**: $w_{15} = -4$ tetapi $w_{51} = \infty$, sebab sisi $1 \to 5$ ada dan $5 \to 1$ tidak.

Menyimpan W memakan ruang $\Theta(n^2)$, entah grafnya punya sembilan sisi atau sembilan juta. Untuk graf-graf di bab ini itu justru pertukaran yang tepat, sebab *keluarannya* toh berukuran $\Theta(n^2)$; Anda tidak dapat melaporkan jarak semua pasangan dengan ruang yang lebih kecil daripada yang diperlukan untuk menuliskannya. Algoritma Johnson di §15.8 adalah satu-satunya perkecualian yang patut diketahui: ia membaca grafnya dari daftar ketetanggaan dan baru mewujudkan jawaban $n \times n$-nya di ujung.

15.2.2 Keluarannya: dua matriks, bukan satu

Kita harus berbicara dengan persis tentang apa yang harus kita hasilkan. Keluaran yang pertama adalah matriks jaraknya. Kita menyebutnya $D = (d_{ij})$, sebuah matriks $n \times n$ yang d_{ij}-nya memuat bobot lintasan terpendek dari simpul i ke simpul j, sehingga pada saat berakhir

$$d_{ij} = \delta(i, j) \quad \text{untuk semua } i, j \in V. \tag{15.2}$$

(Metode perkalian matriks di §15.4 membangun barisan matriks yang disebutnya $L^{(m)}$, bukan D, karena suatu alasan yang nanti akan jelas; L yang terakhir sama dengan D yang ini.)

Keluaran yang kedua adalah yang paling sering lupa diminta pemula, dan yang membuat jawabannya menjadi *dapat dipakai*. Mengetahui bahwa perjalanan termurah dari Keputih ke Darmo berongkos delapan belas menit tidaklah banyak menolong sopir yang ingin tahu jalan mana yang harus ditempuh. Maka berdampingan dengan D kita menghitung **matriks pendahulu**

> **DEFINISI. Matriks pendahulu** adalah matriks $n \times n$, $\Pi = (\pi_{ij})$, yang π_{ij}-nya bernilai NIL bila $i = j$ atau bila sama sekali tidak ada lintasan dari i ke j, dan selebihnya bernilai **pendahulu** j **pada suatu lintasan terpendek dari** i. Artinya: π_{ij} menyebutkan simpul tempat Anda berdiri persis sebelum Anda tiba di j, pada satu rute termurah tertentu yang berawal di i.

Bacalah definisi itu sekali lagi, pelan-pelan, sebab indeksnya mudah salah baca. π_{ij} *bukanlah* simpul kedua rutenya; ia simpul *kedua dari terakhir*. Baris i dari Π memerikan seluruh pohon rute termurah yang keluar dari i, disimpan dengan menunjukkan setiap simpul mundur ke arah i. Subgraf ini disebut **subgraf pendahulu** G **untuk** i, yang didefinisikan oleh

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\},$$

$$E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} - \{i\}\},$$

sehingga $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$. Dengan kata-kata: simpul $G_{\pi,i}$ adalah i sendiri bersama segala sesuatu yang terjangkau dari i, dan sisinya satu per simpul terjangkau, masing-masing menunjuk dari pendahulu tercatat simpul itu ke simpulnya sendiri. Bila algoritmanya benar maka $G_{\pi,i}$ adalah sebuah **pohon lintasan terpendek berakar di** i — persis benda yang dibangun Bab 13 untuk satu sumber, hanya saja kini kita membangun n buah sekaligus, satu per baris Π.

> **DENGAN BAHASA SEDERHANA.** D memberi tahu Anda *berapa*. Π memberi tahu Anda *bagaimana*. D adalah daftar harga; Π adalah setumpuk n peta jalan, satu per titik berangkat, masing-masing terlipat menjadi satu baris berisi n bilangan. Keduanya berukuran $\Theta(n^2)$, dan itulah sebabnya kita sanggup menyimpan keduanya.

15.2.3 Mencetak sebuah lintasan dari Π

Mengubah satu baris Π kembali menjadi rute adalah rekursi tiga baris, yang diperlihatkan pada **Gambar 15.3**.

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
jika i == j
    cetak i
jika tidak, jika π_ij == NIL
    cetak "tidak ada lintasan dari" i "ke" j
selain itu PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
    cetak j
```

Gambar 15.3 PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j) memulihkan rute yang sesungguhnya dari matriks pendahulunya, mencetaknya maju sekalipun Π menyimpannya mundur.

Kuncinya adalah bahwa panggilan rekursifnya datang *sebelum* pencetakannya. Untuk mencetak rute dari i ke j, cetaklah dahulu rute dari i ke pendahulu j, dan baru sesudah itu cetak j. Maka rekursinya melangkah mundur dari j ke i selagi ia menurun, dan mencetak maju dari i ke j selagi ia kembali. Ketiga kasusnya persis ketiga kasus dalam definisi π_{ij}:

1. $i = j$**.** Kita sudah tiba kembali di titik awalnya. Cetaklah i lalu berhenti. Inilah kasus dasarnya, dan hanya inilah yang mengakhiri rekursinya secara wajar.
2. $\pi_{ij} =$ NIL **dan** $i \neq j$**.** Tidak ada lintasan. Katakanlah demikian lalu berhenti.
3. **Selain itu.** Rekurslah pada (Π, i, π_{ij}) — yaitu, cetaklah rute dari i sampai simpul persis sebelum j — lalu cetak j.

Prosedurnya berjalan dalam waktu linear terhadap banyaknya simpul pada lintasan yang dicetaknya, yang paling banyak n, sehingga mencetak seluruh n^2 lintasannya berongkos $O(n^3)$: sama dengan menghitungnya. Kita akan melihatnya bekerja di §15.6.11, pada bilangan sungguhan yang dihasilkan program sungguhan.

15.2.4 Sebuah pemeriksaan kewarasan yang pertama atas ukurannya

Sebelum kita menulis satu algoritma pun, ada gunanya menancapkan skala persoalannya di benak, sebab skala itulah yang menjelaskan setiap keputusan rancangan yang menyusul.

Keluarannya punya n^2 entri. Membacanya kembali saja sudah berongkos $\Omega(n^2)$. Maka tidak ada algoritma semua pasangan yang dapat mengalahkan $\Omega(n^2)$, dan algoritma di bab ini mendarat di n^3 atau sedikit lebih baik — dalam jarak satu faktor n dari lantai mutlaknya. Untuk $n = 1000$ itu berarti semiliar operasi untuk jawabannya dan sejuta bilangan untuk disimpan: satu-dua detik, dan delapan megabita. Untuk $n = 10{,}000$ itu setriliun operasi dan delapan ratus megabita: beberapa menit dan sebuah mesin besar. Untuk $n = 100{,}000$ — sebuah situs web yang sedang-sedang saja — *keluarannya saja* sudah sepuluh miliar bilangan, delapan puluh gigabita, dan persoalan semua pasangan memang pertanyaan yang keliru. Mengetahui di mana tembok itu berdiri adalah bagian dari menguasai bahan ini.

15.3 Algoritma yang Sudah Kentara, dan Harganya

Inilah algoritma pertama yang terpikir oleh siapa pun, dan ia layak dinyatakan lebih dahulu justru supaya kita dapat mengukur segala sesuatu yang lain terhadapnya:

> **PENDEKATAN NAIFNYA.** Untuk menyelesaikan lintasan terpendek semua pasangan, jalankanlah sebuah algoritma lintasan terpendek sumber tunggal sebanyak $|V|$ kali, sekali dengan tiap simpul sebagai sumbernya, lalu tumpuklah $|V|$ vektor jawabannya menjadi sebuah matriks.

Ini bukan lelucon dan bukan pula boneka jerami. Ia benar, ia sederhana, dan untuk sebagian graf ia yang tercepat yang diketahui orang. Berapa ongkosnya sepenuhnya bergantung pada algoritma sumber tunggal mana yang boleh kita pakai, dan itu bergantung pada bobotnya.

15.3.1 Semua bobot tak negatif: $|V| \times$ Dijkstra

Bila tidak ada sisi yang negatif, Dijkstra berlaku, dan kita membayar ongkosnya sebanyak n kali. Ada tiga varian, satu per implementasi antrean prioritas, dan ketiganya patut dipisahkan dengan cermat sebab yang keliru rutin dikutip orang.

Larik linear sebagai antrean prioritas-min. Tiap EXTRACT-MIN memindai lariknya dalam waktu $O(V)$ dan ada V buah, sehingga pengeluarannya berongkos $O(V^2)$; tiap DECREASE-KEY hanyalah satu penugasan, $O(1)$, dan banyaknya paling banyak E. Maka satu kali jalan berongkos $O(V^2 + E)$, dan V kali jalan berongkos

$$O(V \cdot (V^2 + E)) = O(V^3 + VE) = O(V^3),$$

langkah terakhirnya berlaku karena $E \leq V^2$ selalu, sehingga $VE \leq V^3$ dan suku V^3-nya mendominasi. Jadi varian pertama berongkos $O(V^3 + VE) = O(V^3)$.

Tumpukan-min biner. Kini EXTRACT-MIN berongkos $O(\lg V)$ dan DECREASE-KEY berongkos $O(\lg V)$, sehingga satu kali jalan menjadi $O((V + E)\lg V) = O(E\lg V)$ untuk graf terhubung, dan V kali jalan menjadi

$$O(VE\lg V).$$

Varian kedua ini datang dengan syarat yang teramat penting: ini merupakan **perbaikan bila grafnya jarang**. Pada graf padat, tempat $E = \Theta(V^2)$, ia menjadi $O(V^3\lg V)$, yang *lebih buruk* daripada versi lariknya sebesar faktor $\lg V$. Tumpukannya tidak lebih baik secara semesta; ia lebih baik persis ketika $E = o(V^2/\lg V)$.

Sebuah *Fibonacci heap* [7]. DECREASE-KEY turun menjadi $O(1)$ teramortisasi sementara EXTRACT-MIN tetap $O(\lg V)$ teramortisasi, sehingga satu kali jalan berongkos $O(V\lg V + E)$ dan V kali jalan berongkos

$$O(V^2\lg V + VE).$$

Varian ketiga ini memberikan batas yang diwarisi algoritma Johnson di §15.8. Ia tak pernah lebih buruk daripada kedua yang lain, walaupun faktor konstanta yang tersembunyi di dalam *Fibonacci heap* cukup besar sehingga dalam praktik tumpukan biner biasanya menang di bawah beberapa ratus ribu simpul.

15.3.2 Ada sisi negatif: $|V| \times$ Bellman–Ford

Begitu satu sisi saja negatif, Dijkstra menjadi tidak sah: kita *tidak dapat* memakai algoritma Dijkstra dan harus menjalankan algoritma Bellman–Ford yang lebih lambat itu sekali dari tiap simpul. Satu kali jalan Bellman–Ford berongkos $O(VE)$, sehingga V kali jalan berongkos

$$O(V^2E),$$

yang pada graf padat, tempat $E = \Theta(V^2)$, menjadi

$$O(V^4).$$

Itulah angka yang harus diingat, dan ia sungguh-sungguh buruk. Pada $n = 500$ ia berarti 6×10^{10} relaksasi sisi — hitungan bermenit-menit di tempat Floyd–Warshall hanya memerlukan sepersekian detik. Pertanyaan yang wajar pun muncul: bagaimana membuatnya lebih baik?

15.3.3 Mengapa kita tidak bisa begitu saja tetap memakai Dijkstra

Godaan untuk tetap mencobanya memang besar. Kebenaran Dijkstra bersandar pada satu anggapan: ketika sebuah simpul u dikeluarkan dari antreannya dengan taksiran terkecil, taksiran itu sudah final, sebab rute lain mana pun menuju u harus melewati simpul yang masih berada di antreannya, yang taksirannya setidaknya sama besar, lalu berjalan lebih jauh lagi di sepanjang sisi berbobot tak negatif — sehingga ia hanya bisa lebih buruk. Sebuah sisi negatif meruntuhkan argumen itu dalam sekali tebas: berjalan *lebih jauh* kini dapat membuat sebuah rute menjadi *lebih murah*.

Ada gunanya menyaksikan kegagalannya alih-alih sekadar diberi tahu tentangnya, dan ada gunanya pula menyaksikan sesuatu yang lebih halus. **Kode 15.1** menjalankan Dijkstra baku dari setiap simpul dua buah graf — contoh berjalan bersimpul lima, dan sebuah graf bersimpul empat yang dibangun khusus sebagai perangkap — lalu mengaudit setiap jawabannya terhadap Floyd–Warshall.

Kode 15.1 Dijkstra diaudit terhadap Floyd–Warshall pada dua graf bersisi negatif tanpa siklus negatif. Kompilasilah dengan `g++ -std=c++17 -O2 -o dijkstra_fails dijkstra_fails.cpp`.

```
// Bab 15, Kode: mengapa |V| x Dijkstra bukan pilihan begitu ada sisi yang negatif.
// Dijkstra dijalankan dari setiap simpul dan jawabannya dibandingkan dengan jawaban
// yang dihasilkan Floyd-Warshall. Kedua graf tidak punya siklus berbobot negatif.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;
typedef vector<vector<long long>> Matrix;

// Dijkstra baku: sebuah simpul dikeluarkan sekali, dinyatakan final, lalu sisi keluarnya
// direlaksasi sekali. Anggapan tunggal itulah yang dipatahkan sisi negatif.
vector<long long> dijkstra(const Matrix& W, int s) {
    int n = (int)W.size();
    vector<long long> d(n, INF);
    vector<char> done(n, 0);
    d[s] = 0;
    for (int t = 0; t < n; ++t) {
        int u = -1;
        for (int v = 0; v < n; ++v)
            if (!done[v] && d[v] < INF && (u == -1 || d[v] < d[u])) u = v;
        if (u == -1) break;
        done[u] = 1;
        for (int v = 0; v < n; ++v)
            if (!done[v] && W[u][v] < INF && d[u] + W[u][v] < d[v])
                d[v] = d[u] + W[u][v];               // simpul yang sudah final tidak
dikunjungi ulang
    }
    return d;
}

Matrix floydWarshall(Matrix D) {
    int n = (int)D.size();
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (D[i][k] < INF && D[k][j] < INF)
                    D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
    return D;
}
```

Kode 15.1 (Lanjutan) *Dijkstra diaudit terhadap Floyd–Warshall pada dua graf bersisi negatif tanpa siklus negatif. Kompilasilah dengan g++ -std=c++17 -O2 -o dijkstra_fails dijkstra_fails.cpp.*

```
int audit(const string& title, const Matrix& W) {
    int n = (int)W.size();
    Matrix D = floydWarshall(W);
    cout << title << '\n';
    int wrong = 0;
    for (int s = 0; s < n; ++s) {
        vector<long long> d = dijkstra(W, s);
        for (int v = 0; v < n; ++v)
            if (d[v] != D[s][v]) {
                ++wrong;
                cout << "   delta(" << s + 1 << "," << v + 1 << "): Dijkstra says "
                     << d[v] << ", the truth is " << D[s][v] << "  <-- WRONG\n";
            }
    }
    cout << "   entries Dijkstra got wrong: " << wrong << " out of " << n * n << "\n\n";
    return wrong;
}

int main() {
    Matrix sample = {
        {  0,   3,   8, INF,  -4},
        {INF,   0, INF,   1,   7},
        {INF,   4,   0, INF, INF},
        {  2, INF,  -5,   0, INF},
        {INF, INF, INF,   6,   0}
    };
    // 1 -> 2 (1), 1 -> 3 (2), 3 -> 2 (-3), 2 -> 4 (1). Simpul 2 difinalkan pada
    // jarak 1 dan sisinya ke 4 direlaksasi, sebelum simpul 3 menurunkan 2 menjadi -1.
    Matrix trap = {
        {  0,   1,   2, INF},
        {INF,   0, INF,   1},
        {INF,  -3,   0, INF},
        {INF, INF, INF,   0}
    };
    audit("(a) the five-vertex running example:", sample);
    audit("(b) a four-vertex graph built to expose the flaw:", trap);
    cout << "Moral: on (a) Dijkstra happened to be right, and that is worse than\n"
            "being wrong, because it invites you to trust it on (b).\n";
    return 0;
}
```

Keluarannya:

```
(a) the five-vertex running example:
   entries Dijkstra got wrong: 0 out of 25

(b) a four-vertex graph built to expose the flaw:
   delta(1,2): Dijkstra says 1, the truth is -1  <-- WRONG
   delta(1,4): Dijkstra says 2, the truth is 0  <-- WRONG
   entries Dijkstra got wrong: 2 out of 16

Moral: on (a) Dijkstra happened to be right, and that is worse than
being wrong, because it invites you to trust it on (b).
```

Perhatikanlah baik-baik apa yang terjadi. Pada contoh berjalannya, Dijkstra menghasilkan kedua puluh lima jawaban yang benar. Ia tidak berhak atas itu; ia sekadar beruntung. Pada graf (b) — sisi $1 \to 2$ berbobot 1, $1 \to 3$ berbobot 2, $3 \to 2$ berbobot -3, $2 \to 4$ berbobot 1 — ia gagal. Berangkat dari simpul 1 ia menyetel $d[2] = 1$ dan $d[3] = 2$, mengeluarkan simpul 2 lebih dahulu sebab $1 < 2$, menyatakannya final, lalu merelaksasi $2 \to 4$ sehingga $d[4] = 2$. Barulah sesudah itu ia mengeluarkan simpul 3 dan mendapati bahwa $1 \to 3 \to 2$ berongkos $2 + (-3) = -1$, yang lebih baik daripada 1. Tetapi simpul 2 sudah telanjur difinalkan, sisi keluarnya sudah telanjur

direlaksasi, dan perbaikannya tak pernah sampai ke simpul 4. Jawaban sebenarnya adalah $\delta(1,4) = 1 \to 3 \to 2 \to 4 = 2 - 3 + 1 = 0$; Dijkstra melaporkan 2.

> **JEBAKAN.** Algoritma yang keliru *kadang-kadang* lebih berbahaya daripada yang keliru selalu, sebab pengujian tidak akan menemukannya. Janganlah memakai Dijkstra pada graf yang mungkin memuat sisi negatif, sebaik apa pun kelihatannya ia berperilaku pada data contoh Anda. §15.8 memperlihatkan cara yang sah untuk memperoleh kecepatan Dijkstra pada graf semacam itu: ubahlah bobotnya lebih dahulu, dan buktikan bahwa perubahannya tidak merusak apa-apa.

15.3.4 Apa yang sedang kita coba kalahkan

Tabel 15.1 mengumpulkan biaya naifnya. Segala sesuatu di sisa bab ini diukur terhadap tabel ini.

Tabel 15.1 Biaya menyelesaikan lintasan terpendek semua pasangan dengan menjalankan algoritma sumber tunggal sekali per simpul. Baris terakhirnyalah yang menyakitkan, dan baris itulah yang diserang sisa bab ini.

Keadaan	Algoritma sumber tunggal, dijalankan V kali	Biaya total	Pada graf padat ($E=\Theta(V^2)$)
semua bobot ≥ 0, antrean larik	Dijkstra	$O(V^3 + VE)$	$O(V^3)$
semua bobot ≥ 0, tumpukan biner	Dijkstra	$O(VE\lg V)$	$O(V^3\lg V)$
semua bobot ≥ 0, *Fibonacci heap*	Dijkstra	$O(V^2\lg V + VE)$	$O(V^3)$
ada bobot < 0	Bellman–Ford	$O(V^2E)$	$O(V^4)$

Dua pengamatan menyiapkan segala sesuatu yang menyusul. Pertama, pada graf padat berbobot tak negatif, $V \times$ Dijkstra sudah mencapai $O(V^3)$ — sehingga algoritma baru mana pun harus menyamainya sekaligus menangani sisi negatif, atau mengalahkannya pada graf jarang. Floyd–Warshall melakukan yang pertama; Johnson melakukan yang kedua. Kedua, bencananya ada di baris terakhir: $O(V^4)$ untuk kasus umumnya. Metode perkalian matriks di §15.4 akan menurunkannya menjadi $\Theta(n^3\lg n)$ dan Floyd–Warshall menjadi $\Theta(n^3)$, keduanya menangani sisi negatif tanpa mengeluh.

15.4 Lintasan Terpendek dan Perkalian Matriks

15.4.1 Resep pemrograman dinamis, diingat kembali

Ingatlah kembali resep empat langkah untuk sebuah algoritma pemrograman dinamis yang sudah digariskan Bab 7:

1. **Perikanlah struktur sebuah penyelesaian optimal.**
2. **Definisikanlah secara rekursif nilai sebuah penyelesaian optimal.**
3. **Hitunglah nilai penyelesaian optimalnya secara bawah-atas.**
4. **Susunlah sebuah penyelesaian optimal dari keterangan yang sudah dihitung.**

Kita akan mengikuti keempat langkah itu berurutan, lalu mengikutinya *sekali lagi* di §15.6 dengan pemerian yang berbeda pada langkah 1 — dan itulah seluruh alasan mengapa di bab ini ada dua algoritma yang berbeda, bukan satu. Pemrograman dinamis bukanlah satu algoritma; ia sebuah metode, dan metodenya berlapang dada bagi lebih daripada satu jawaban yang bagus.

15.4.2 Langkah 1: struktur sebuah lintasan terpendek

Pemerian di bagian ini memakai **banyaknya sisi**. **Gambar 15.4** menggelarnya.

- Setiap sublintasan dari lintasan terpendek adalah lintasan terpendek (Teorema 13.1).
- Misalkan p lintasan terpendek dari i ke j dengan paling banyak m sisi; tanpa siklus berbobot negatif, m semacam itu berhingga.
- Bila $i = j$, maka p berbobot 0 dan tidak memuat sisi.
- Bila $i \neq j$, pecah p menjadi $i \overset{p'}{\rightsquigarrow} k \to j$, dengan p' memuat paling banyak $m-1$ sisi. Menurut Teorema 13.1, p' lintasan terpendek dari i ke k, jadi $\delta(i,j) = \delta(i,k) + w_{kj}$.

Gambar 15.4 Struktur sebuah lintasan terpendek yang bersisi paling banyak m. Bila $i \neq j$, penggallah sisi terakhirnya dan sisanya haruslah lintasan terpendek pula — inilah Teorema 13.1 sedang bekerja.

Inilah argumennya selengkapnya. Misalkan p sebuah lintasan terpendek dari simpul i ke simpul j yang memuat **paling banyak** m **sisi**. Karena grafnya tidak punya siklus berbobot negatif, m berhingga: lintasan terpendek tak pernah perlu mengulang simpul, sehingga ia bersisi paling banyak $n-1$. Sekarang pecahlah menjadi dua kasus.

Kasus A: $i = j$**.** Maka lintasan kosongnya — tanpa sisi sama sekali — berbobot 0, dan karena tidak ada siklus berbobot negatif, tidak ada jalan tertutup dari i kembali ke i yang dapat mengalahkan 0. Jadi p berbobot 0 dan tanpa sisi.

Kasus B: $i \neq j$**.** Maka p sedikitnya bersisi satu. Tulislah k untuk simpul persis sebelum j pada p, lalu uraikanlah

$$i \overset{p'}{\rightsquigarrow} k \to j,$$

dengan p' adalah segala sesuatu kecuali sisi terakhirnya. Karena p bersisi paling banyak m dan kita membuang satu, p' memuat **paling banyak** $m-1$ **sisi**. Menurut Teorema 13.1, p' adalah lintasan terpendek dari i ke k. Dan karena itu

$$\delta(i,j) = \delta(i,k) + w_{kj}.$$

Persamaan terakhir itulah seluruh isi langkah 1. Ia berkata: *biaya rute terbaik dari i ke j adalah biaya rute terbaik menuju suatu tempat yang berjarak satu langkah sebelum j, ditambah langkah terakhir itu.* Kita tidak tahu simpul k itu yang mana — tetapi calonnya hanya n buah, dan kita dapat mencoba semuanya.

> **DENGAN BAHASA SEDERHANA.** Bayangkanlah Anda diberi cara termurah untuk pergi dari Keputih ke Darmo, dan diberi tahu bahwa ia menempuh paling banyak enam ruas jalan. Tutuplah ruas terakhirnya dengan ibu jari Anda. Apa pun yang tersisa haruslah cara termurah untuk pergi dari Keputih ke tempat ibu jari itu bermula — sebab seandainya bukan, Anda dapat menggantinya dengan yang lebih murah dan mengalahkan rute yang sudah dikatakan termurah kepada Anda. Itulah Teorema 13.1, dan hanya itulah gagasan di bagian ini.

15.4.3 Langkah 2: definisi rekursif $l_{ij}^{(m)}$

Kini kita menamai besaran yang akan dihitung rekursinya; **Gambar 15.5** memperkenalkannya.

- Misalkan $l_{ij}^{(m)}$ bobot minimum dari lintasan mana pun dari i ke j yang memuat **paling banyak** m sisi.
- Untuk $m = 0$ ada lintasan dari i ke j tanpa sisi jika dan hanya jika $i = j$, sehingga

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{jika } i = j \\ \infty & \text{jika } i \neq j \end{cases}$$

- Untuk $m \geq 1$, $l_{ij}^{(m)}$ adalah yang lebih kecil di antara $l_{ij}^{(m-1)}$ (lintasan terbaik dengan paling banyak $m - 1$ sisi) dan lintasan terbaik dengan tepat m sisi, yang ditemukan dengan mencoba setiap pendahulu k yang mungkin bagi j.

Gambar 15.5 Definisi $l_{ij}^{(m)}$ beserta kasus dasarnya $l_{ij}^{(0)}$. Perhatikanlah bahwa superskripnya berada di dalam tanda kurung — ia mencacah sisi, ia bukan pangkat.

> **DEFINISI.** Misalkan $l_{ij}^{(m)}$ adalah **bobot minimum sembarang lintasan dari simpul i ke simpul j yang memuat paling banyak m sisi**. Bila lintasan semacam itu tidak ada, $l_{ij}^{(m)} = \infty$.

Setiap bagian notasi itu penting, dan ia mudah salah ditulis, maka marilah kita berlaku persis. Huruf l berasal dari *length* (panjang) dalam arti bobot. **Subskrip** i dan j adalah kedua ujungnya: i lebih dahulu, sumbernya; j kemudian, tujuannya. **Superskrip** (m), yang ditulis di dalam tanda kurung, adalah *jatah sisinya* — bilangan terbesar sisi yang boleh dipakai lintasannya. Ia sebuah label, bukan pangkat: $l_{ij}^{(2)}$ tidak berarti l_{ij} dikuadratkan. Menuliskannya sebagai `l(m)ij` atau `lij^m` justru memusnahkan pembedaan yang membuat rekurensinya terbaca, dan superskrip berkurung itu adalah notasi baku di dalam pustakanya [1].

Kasus dasarnya adalah $m = 0$. Dengan jatah nol sisi Anda tidak dapat pergi ke mana-mana, sehingga satu-satunya perjalanan yang tersedia adalah perjalanan yang sudah selesai sebelum ia bermula:

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{jika } i = j, \\ \infty & \text{jika } i \neq j. \end{cases} \tag{15.3}$$

Dalam bentuk matriks, $L^{(0)}$ adalah matriks dengan 0 di sepanjang diagonalnya dan ∞ di semua tempat lain. Ingatlah matriks itu baik-baik: §15.4.9 akan memperlihatkan bahwa ia *unsur identitas* perkalian ganjil yang sebentar lagi kita definisikan, persis sebagaimana matriks identitas biasa — 1 di diagonalnya, 0 di tempat lain — adalah identitas untuk perkalian biasa.

15.4.4 Rekurensinya, suku demi suku

Untuk $m \geq 1$ kita menghitung $l_{ij}^{(m)}$ dari nilai $l_{ij}^{(m-1)}$ yang sudah diketahui. **Gambar 15.6** memberikan rekurensinya.

Secara rekursif:

$$l_{ij}^{(m)} = \min\Big(l_{ij}^{(m-1)},\ \min_{1\le k\le n}\{l_{ik}^{(m-1)} + w_{kj}\}\Big)$$
$$= \min_{1\le k\le n}\{l_{ik}^{(m-1)} + w_{kj}\} \qquad \text{sebab } w_{jj} = 0 \text{ untuk semua } j$$

Tanpa siklus berbobot negatif, setiap kali $\delta(i,j) < \infty$ ada lintasan terpendek dari i ke j yang sederhana, sehingga memuat paling banyak $n-1$ sisi; lintasan dengan sisi lebih banyak tidak mungkin lebih ringan. Karena itu

$$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots$$

Gambar 15.6 Rekurensi Persamaan (15.4), penyederhanaannya yang satu baris, dan fakta, Persamaan (15.6), bahwa barisan $l_{ij}^{(m)}$ berhenti berubah pada $m = n-1$.

Rekurensinya adalah

$$l_{ij}^{(m)} = \min\Big(l_{ij}^{(m-1)},\ \min_{1\le k\le n}\Big\{l_{ik}^{(m-1)} + w_{kj}\Big\}\Big). \tag{15.4}$$

Bongkarlah ia sepotong demi sepotong, sebab setiap lambang di dalamnya sedang mengerjakan sesuatu.

min **yang di luar** punya dua argumen, bersesuaian dengan dua hal yang mungkin merupakan rute-terbaik-dengan-paling-banyak-m-sisi.

Argumen pertama, $l_{ij}^{(m-1)}$, adalah bobot lintasan terpendek dari i ke j yang memakai paling banyak $m-1$ sisi. Mengapa ia seorang calon? Sebab "paling banyak m sisi" mencakup "paling banyak $m-1$ sisi". Menaikkan jatahnya tak pernah bisa memperburuk keadaan, dan rute terbaik di bawah jatah yang lebih kecil tetap sah di bawah jatah yang lebih besar. Argumen ini meliput kasus ketika sisi tambahan yang dianugerahkan kepada kita ternyata tidak diperlukan.

Argumen kedua, $\min_{1\le k\le n}\Big\{l_{ik}^{(m-1)} + w_{kj}\Big\}$, meliput kasus ketika kita memang memakai sisi tambahannya. Ia sendiri sebuah minimum, yang diambil atas seluruh n pilihan yang mungkin bagi simpul k yang kita lewati persis sebelum tiba di j. Untuk tiap calon k ongkosnya adalah $l_{ik}^{(m-1)}$ — cara terbaik mencapai k dengan paling banyak $m-1$ sisi — ditambah w_{kj}, bobot satu lompatan terakhir dari k ke j. Bila sisi $k \to j$ tidak ada maka $w_{kj} = \infty$ dan calon itu kalah dengan sendirinya; bila k tak terjangkau dari i dalam $m-1$ sisi maka $l_{ik}^{(m-1)} = \infty$ dan ia pun kalah. Notasi $\min_{1\le k\le n}$ berarti "ambillah yang terkecil dari n bilangan ini, selagi k berjalan dari 1 sampai n"; subskrip di bawah min menyebutkan peubah yang sedang disapu beserta jangkauannya, persis seperti subskrip di bawah $\sum$.

Inilah persis penguraian di §15.4.2, yang diubah menjadi rumus. Langkah 1 memberi tahu kita bahwa entah $i = j$ dan lintasannya kosong, atau lintasannya berakhir dengan suatu sisi $k \to j$ dan bagian sebelumnya optimal. Rekurensinya mencoba perluasan kosongnya dan seluruh n sisi terakhirnya, lalu menyimpan yang terbaik.

15.4.5 Mengapa argumen pertamanya dapat dibuang

Berikutnya datang sebuah penyederhanaan yang tampak seperti tipuan padahal sebenarnya sebuah pengamatan tentang diagonal W. Ambillah minimum yang di dalam lalu tinjaulah calon yang satu ini: $k = j$. Nilainya adalah

$$l_{ij}^{(m-1)} + w_{jj} = l_{ij}^{(m-1)} + 0 = l_{ij}^{(m-1)},$$

sebab $w_{jj} = 0$ untuk semua j, menurut baris pertama Persamaan (15.1). Tetapi $l_{ij}^{(m-1)}$ justru argumen pertama minimum yang di luar. Jadi argumen pertamanya *sudah menjadi salah satu calon* di minimum yang di dalam, dan menyertakannya secara terpisah tidak mengubah apa pun. Rekurensinya meringkas menjadi

$$l_{ij}^{(m)} = \min_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}, \qquad \text{perhatikan bahwa } w_{jj} = 0 \text{ untuk semua } j. \tag{15.5}$$

MENGAPA INI PENTING. Bentuk yang sudah disederhanakan itu persis punya satu minimum atas persis satu indeks. Itulah bentuk sebuah hasil kali matriks, dan mengenalinya adalah inti seluruh bagian ini. Seandainya kita mendefinisikan w_{jj} sebagai sesuatu yang bukan 0 — ∞, misalnya, yang akan dipilih pembaca yang terlalu berhati-hati untuk "tidak ada gelung-diri" — peringkasannya akan gagal, rekurensinya akan mempertahankan bentuk berbagian duanya, dan kaitannya dengan perkalian matriks akan tak terlihat. Definisi di Persamaan (15.1) dipilih justru untuk membuat Persamaan (15.5) benar.

Marilah kita periksa Persamaan (15.5) pada satu entri contoh berjalannya sebelum kita memercayainya. Ambillah $i = 2, j = 4, m = 2$. Dengan $L^{(1)} = W$, kelima calonnya adalah

$$\begin{aligned}
k = 1 &: l_{21}^{(1)} + w_{14} = \infty + \infty = \infty, \\
k = 2 &: l_{22}^{(1)} + w_{24} = 0 + 1 = 1, \\
k = 3 &: l_{23}^{(1)} + w_{34} = \infty + \infty = \infty, \\
k = 4 &: l_{24}^{(1)} + w_{44} = 1 + 0 = 1, \\
k = 5 &: l_{25}^{(1)} + w_{54} = 7 + 6 = 13,
\end{aligned}$$

sehingga $l_{24}^{(2)} = \min\{\infty, 1, \infty, 1{,}13\} = 1$. Dua calon seri pada 1: rute satu-sisi yang sejati $2 \to 4$ yang tercacah lewat $k = 2$ (nol sisi untuk mencapai 2, lalu sisi $2 \to 4$), dan rute yang sama yang tercacah lewat $k = 4$ (satu sisi untuk mencapai 4, lalu "berdiam di tempat" berbobot nol). Penggandaan itulah harga penyederhanaannya, dan harganya nol.

15.4.6 Kapan barisannya berhenti berubah?

Kini kita tahu cara berpindah dari $L^{(m-1)}$ ke $L^{(m)}$. Sejauh mana kita harus melangkah? Argumennya berparuh dua.

Paruh pertama: setiap lintasan terpendek bersisi paling banyak $n-1$. Inilah Teorema 13.4 di Bab 13; argumennya cukup pendek untuk diulang. Andaikanlah $\delta(i,j) < \infty$, sehingga ada suatu lintasan dari i ke j. Klaimnya: ada lintasan terpendek dari i ke j yang **sederhana**, yang berarti tidak mengulang simpul. Sebab andaikanlah sebuah lintasan terpendek p mengulang sebuah simpul v. Maka p memuat sebuah siklus yang menembus v. Siklus itu berbobot ≥ 0, sebab grafnya tidak punya siklus berbobot negatif. Menghapusnya karena itu menghasilkan sebuah lintasan dari i ke j yang bobotnya tidak lebih besar daripada $w(p)$, dengan sisi yang jelas lebih sedikit. Ulangilah sampai tidak ada simpul yang terulang. Yang tersisa adalah lintasan sederhana, yang bobotnya tetap minimum. Lintasan sederhana mengunjungi paling banyak n simpul berbeda dan karena itu memakai paling banyak $n-1$ sisi.

Paruh kedua: sisi tambahan tak pernah menolong. Lintasan dari i ke j dengan sisi lebih daripada $n-1$ tidak mungkin berbobot lebih rendah daripada lintasan terpendek dari i ke j — menurut definisi "terpendek", tidak ada yang bisa. Jadi menaikkan jatahnya melampaui $n-1$ tak pernah dapat menurunkan jawabannya.

Satukanlah kedua paruhnya. Pada jatah $n-1$ kita sudah sanggup membiayai lintasan sederhana terbaiknya, dan di luar $n-1$ tidak ada yang membaik. Karena itu

$$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots \tag{15.6}$$

untuk setiap pasangan i,j. Barisan $l_{ij}^{(0)}, l_{ij}^{(1)}, l_{ij}^{(2)}, \ldots$ tidak menaik, dan ia mencapai limitnya selambat-lambatnya pada langkah $n-1$ lalu tinggal di sana selamanya. Inilah syarat berhenti bagi segala sesuatu di §15.4 dan §15.5 (pada edisi ketiga buku Cormen dkk. ia adalah persamaan (25.3)).

> **DI MANA ANGGAPAN TANPA-SIKLUS-NEGATIF DIPAKAI.** Di paruh pertama, dan hanya di sana. Seandainya ada siklus berbobot negatif pada rute dari i ke j, menghapusnya justru akan *menaikkan* bobotnya, sehingga kita tidak dapat menghapusnya, lintasan sederhana tidak akan optimal, dan $l_{ij}^{(m)}$ akan terus turun selamanya — dan itu benar, sebab $\delta(i,j)$ memang akan $-\infty$. Persamaan (15.6) adalah titik persisnya "tanpa siklus negatif" membayar ongkosnya.

15.4.7 Langkah 3: menghitung secara bawah-atas — EXTEND-SHORTEST-PATHS

Rencananya begini: dari matriks masukan $W = (w_{ij})$, hitunglah deret matriks $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, dengan $L^{(m)} = (l_{ij}^{(m)})$, dan matriks terakhirnya $L^{(n-1)}$ memuat bobot lintasan terpendek yang sesungguhnya. Deretnya bermula dengan

$$l_{ij}^{(1)} = w_{ij} \quad \text{untuk semua } i,j \in V, \qquad \text{sehingga } L^{(1)} = W.$$

Mengapa $L^{(1)}$ sama dengan W dan bukan dengan sesuatu yang dihitung? Terapkanlah Persamaan (15.5) dengan $m = 1$: $l_{ij}^{(1)} = \min_k \left\{ l_{ik}^{(0)} + w_{kj} \right\}$. Tetapi $l_{ik}^{(0)}$ bernilai 0 ketika $k = i$ dan ∞ selebihnya, sehingga hanya calon $k = i$ yang selamat, dan memberikan $l_{ij}^{(1)} = 0 + w_{ij} = w_{ij}$. Rute terbaik yang memakai paling banyak satu sisi adalah sisi langsungnya bila ada, dan tidak ada apa-apa bila tidak ada. Persis sebagaimana seharusnya.

Langkah dari satu matriks ke matriks berikutnya adalah sebuah prosedur bernama EXTEND-SHORTEST-PATHS, yang diperlihatkan pada **Gambar 15.7**.

```
EXTEND-SHORTEST-PATHS(L, W)
1  n = L.rows
2  misalkan L' = (l'_ij) sebuah matriks n × n baru
3  untuk i = 1 sampai n
4      untuk j = 1 sampai n
5          l'_ij = ∞
6          untuk k = 1 sampai n
7              l'_ij = min(l'_ij, l_ik + w_kj)
8  kembalikan L'
```

Gambar 15.7 EXTEND-SHORTEST-PATHS(L, W), yang mengambil sebuah matriks berisi rute-terbaik-sejauh-ini lalu memperpanjang setiap satunya dengan satu sisi lagi. Tiga gelung bersarang, sehingga waktunya $\Theta(n^3)$.

Bacalah ia sebagai bahasa Indonesia. Baris 1 menanyakan kepada matriksnya berapa baris yang dipunyainya. Baris 2 menyediakan tempat bagi jawabannya, $L' = (l'_{ij})$. Baris 3 dan 4 melangkahi setiap sel jawabannya. Baris 5 menyetel sel itu menjadi ∞ — identitas bagi min, nilai yang kalah melawan apa pun. Baris 6 menyapu simpul antara k dari 1 sampai n, dan baris 7 menyimpan minimum berjalannya:

$$l'_{ij} = \min(l'_{ij},\ l_{ik} + w_{kj}).$$

Baris 8 mengembalikan L'. Tiga gelung bersarang yang masing-masing berputar n kali, dengan kerja $O(1)$ di dalamnya, memberikan waktu jalan $\Theta(n^3)$.

15.4.8 Keserupaan dengan perkalian matriks biasa

Kini tibalah pengamatan yang memberi bagian ini namanya. Taruhlah EXTEND-SHORTEST-PATHS berdampingan dengan prosedur baku untuk mengalikan dua matriks bujur sangkar, yang diperlihatkan pada **Gambar 15.8**.

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
misalkan C = (c_ij) sebuah matriks n × n baru
untuk i = 1 sampai n
    untuk j = 1 sampai n
        c_ij = 0
        untuk k = 1 sampai n
            c_ij = c_ij + a_ik · b_kj
kembalikan C
```

Gambar 15.8 SQUARE-MATRIX-MULTIPLY(A, B), algoritma sekolahan $\Theta(n^3)$ untuk $C = A \cdot B$. Taruhlah ia berdampingan dengan **Gambar 15.7** dan keduanya berbeda di persis dua tempat.

Keduanya prosedur yang sama. Baris demi baris:

Tabel 15.2 EXTEND-SHORTEST-PATHS adalah SQUARE-MATRIX-MULTIPLY dengan min menggantikan $+$, $+$ menggantikan $\times$, dan ∞ menggantikan 0 sebagai nilai awalnya. Selebihnya — struktur gelungnya, pola indeksnya, waktu jalannya — sama persis.

Baris	SQUARE-MATRIX-MULTIPLY	EXTEND-SHORTEST-PATHS
1	$n = A.rows$	$n = L.rows$
2	misalkan C matriks $n \times n$ yang baru	misalkan $L' = (l'_{ij})$ matriks $n \times n$ yang baru
3–4	**for** i, **for** j	**for** i, **for** j
5	$c_{ij} = 0$	$l'_{ij} = \infty$
6	**for** $k = 1$ **to** n	**for** $k = 1$ **to** n
7	$c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$	$l'_{ij} = \min(l'_{ij},\ l_{ik} + w_{kj})$
8	**return** C	**return** L'

Sebagaimana digelar **Tabel 15.2** baris demi baris, penggantiannya adalah:

$$+ \ \rightarrow \ \min, \qquad \times \rightarrow \ +, \qquad 0 \ \rightarrow \ \infty.$$

Hanya itu. Hasil kali biasa mengakumulasi $c_{ij} = \sum_k a_{ik}\, b_{kj}$; punya kita mengakumulasi $l'_{ij} = \min_k\{l_{ik} + w_{kj}\}$. Keduanya menyapu k melintasi baris i matriks kirinya dan kolom j matriks kanannya, menggabungkan pasangan ke-k-nya lalu melipat hasilnya menjadi satu.

Karena ia memang *sebuah* hasil kali, kita boleh menuliskannya sebagai hasil kali. Misalkan $A \odot B$ menyatakan matriks yang dikembalikan EXTEND-SHORTEST-PATHS(A, B). Maka seluruh rencananya menjadi sebuah barisan pangkat:

$$\begin{aligned} L^{(1)} &= L^{(0)} \odot W = W, \\ L^{(2)} &= L^{(1)} \odot W = W^2, \\ L^{(3)} &= L^{(2)} \odot W = W^3, \\ &\vdots \\ L^{(n-1)} &= L^{(n-2)} \odot W = W^{n-1}, \end{aligned}$$

dan menurut Persamaan (15.6) matriks $L^{(n-1)} = W^{n-1}$ memuat bobot lintasan terpendeknya. Superskrip pada W di sini adalah pangkat yang sejati — perkalian-$\odot$ berulang — sementara superskrip berkurung pada L adalah jatah sisi. Keduanya bernilai sama, dan justru itulah sebabnya notasinya sepadan dengan risiko kerancuannya.

15.4.9 Semiring (min, +), dan mengapa keserupaannya persis

Kemiripannya bukan kebetulan dan bukan pula jembatan keledai. Ia sebuah contoh dari fakta aljabar yang umum, dan mengetahuinya mengubah satu algoritma menjadi sebuah keluarga.

Hasil kali matriks sekolahan bekerja atas sembarang **semiring**: sebuah himpunan dengan dua operasi, sebuah "penjumlahan" $\oplus$ yang asosiatif dan komutatif dengan identitas $\bar{0}$, sebuah "perkalian" $\otimes$ yang asosiatif dengan identitas $\bar{1}$, dan sebuah hukum distributif $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$. Bilangan riil dengan $+$ dan $\times$ membentuk satu. Begitu pula bilangan $\mathbb{R} \cup \{\infty\}$ dengan $\oplus = \min$ dan $\otimes = +$ — **semiring** (min, +), yang sering disebut semiring *tropis*:

- min asosiatif dan komutatif, dan identitasnya $\bar{0} = \infty$, sebab $\min(x, \infty) = x$;
- $+$ asosiatif, dan identitasnya $\bar{1} = 0$, sebab $x + 0 = x$;

- distributivitasnya berlaku: $a + \min(b, c) = \min(a + b, a + c)$, yang benar untuk bilangan riil dan yang persisnya adalah pernyataan bahwa menambahkan sebuah konstanta kepada dua perjalanan tidak mengubah yang mana yang lebih pendek.

Tiga akibat menyusul seketika dan tanpa biaya.

Matriks identitasnya. Di dalam semiring $(\min, +)$ matriks identitasnya memuat $\bar{1} = 0$ di diagonalnya dan $\bar{0} = \infty$ di luarnya — dan itu persis matriks $L^{(0)}$ dari Persamaan (15.3). Jadi $L^{(0)} \odot A = A$ untuk setiap A, dan kasus dasar kita bukanlah pilihan sembarangan melainkan satu-satunya yang mungkin.

Asosiativitas. Perkalian matriks atas sebuah semiring bersifat asosiatif, sehingga $(A \odot B) \odot C = A \odot (B \odot C)$. §15.5 seluruhnya dibangun di atas ini.

Persoalan lain, gratis. Sisipkanlah semiring yang berbeda dan gelung rangkap tiga yang sama menyelesaikan persoalan yang berbeda. Dengan $\oplus =$ OR dan $\otimes =$ AND Anda memperoleh keterjangkauan — itulah §15.7. Dengan $\oplus = \max$ dan $\otimes = \min$ Anda memperoleh lintasan terlebar (*bottleneck* maksimum). Dengan $\oplus = +$ dan $\otimes = \times$ atas cacah lintasan Anda memperoleh banyaknya lintasan terpendek. Soal 15.14 beserta penyelesaiannya memperagakan ketiganya pada data sungguhan.

15.4.10 SLOW-ALL-PAIRS-SHORTEST-PATHS

Membungkus hasil kali berulangnya di dalam sebuah gelung memberikan prosedur pada **Gambar 15.9**.

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1  n = W.rows
2  L^(1) = W
3  untuk m = 2 sampai n − 1
4      misalkan L^(m) sebuah matriks n × n baru
5      L^(m) = EXTEND-SHORTEST-PATHS(L^(m−1), W)
6  kembalikan L^(n−1)
```

Gambar 15.9 SLOW-ALL-PAIRS-SHORTEST-PATHS(W). Ia melakukan $n - 2$ panggilan ke EXTEND-SHORTEST-PATHS, masing-masing berongkos $\Theta(n^3)$, sehingga totalnya $\Theta(n^4)$.

Baris 2 menyetel $L^{(1)} = W$, sebagaimana dibenarkan §15.4.7. Baris 3 menjalankan m dari 2 sampai $n - 1$, dan baris 5 menghitung $L^{(m)}$ dari $L^{(m-1)}$ dengan satu hasil kali bersama W. Baris 6 mengembalikan $L^{(n-1)}$, yang menurut Persamaan (15.6) memuat bobot lintasan terpendeknya.

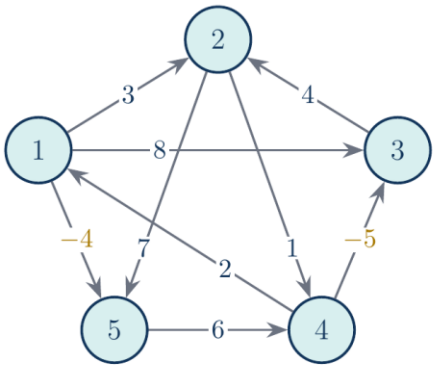
Waktu jalannya adalah $n - 2$ hasil kali, masing-masing $\Theta(n^3)$, sehingga

$$\Theta((n - 2) \cdot n^3) = \Theta(n^4).$$

Ia disebut SLOW bukan tanpa alasan: pada graf padat ini tidak lebih baik daripada $|V| \times$ Bellman–Ford di **Tabel 15.1**. Tujuannya bukanlah kecepatan melainkan *wawasan* — dan §15.5 akan mengubah wawasan itu menjadi penghematan yang sungguhan.

15.4.11 Contoh berjalan, setiap matriks antaranya

Gambar 15.10 memperlihatkan contoh berjalannya beserta setiap matriks yang dihasilkan SLOW-ALL-PAIRS-SHORTEST-PATHS atasnya.



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Gambar 15.10 Graf bersimpul lima itu (Gambar 25.1 pada edisi ketiga buku Cormen dkk.) beserta barisan $L^{(1)}, L^{(2)}, L^{(3)}, L^{(4)}$ yang dihitung SLOW-ALL-PAIRS-SHORTEST-PATHS. Karena $n = 5$, matriks terakhirnya $L^{(4)} = L^{(n-1)}$ memuat bobot lintasan terpendeknya, dan $L^{(5)} = L^{(4)} \odot W$ sama dengan $L^{(4)}$.

Program pada **Kode 15.2** menghasilkan ulang semuanya. Setiap matriks yang dicetak di sisa bab ini adalah keluaran yang sungguh-sungguh dihasilkan sebuah program di halaman ini; tidak ada satu pun yang diketik dengan tangan.

Kode 15.2 SLOW-ALL-PAIRS-SHORTEST-PATHS selengkapnya, sambil mencetak setiap matriks antaranya. Kompilasilah dengan `g++ -std=c++17 -O2 -o slow_apsp slow_apsp.cpp`.

```
// Bab 15, Kode: SLOW-ALL-PAIRS-SHORTEST-PATHS pada graf contoh berjalan.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;      // mewakili takhingga; tidak pernah dijumlahkan
dengan dirinya sendiri

typedef vector<vector<long long>> Matrix;

// EXTEND-SHORTEST-PATHS(L, W): "hasil kali matriks" L . W di dalam semiring (min, +).
// Perkalian biasa akan menulis c_ij = c_ij + a_ik * b_kj; di sini jumlahnya diganti
// dengan minimum dan hasil kalinya diganti dengan jumlah.
Matrix extendShortestPaths(const Matrix& L, const Matrix& W) {
    int n = (int)L.size();
    Matrix Lp(n, vector<long long>(n, INF));      // baris 2: sebuah matriks n x n yang
baru, L'
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            Lp[i][j] = INF;                       // baris 5: l'_ij = takhingga
            for (int k = 0; k < n; ++k)
                if (L[i][k] < INF && W[k][j] < INF)          // jaga-jaga terhadap INF +
INF
                    Lp[i][j] = min(Lp[i][j], L[i][k] + W[k][j]);   // baris 7
        }
    return Lp;
}

void printMatrix(const string& name, const Matrix& M) {
    int n = (int)M.size();
    cout << name << " =\n";
    for (int i = 0; i < n; ++i) {
        cout << "   ";
        for (int j = 0; j < n; ++j) {
            if (M[i][j] >= INF) cout << setw(5) << "inf";
            else                cout << setw(5) << M[i][j];
        }
        cout << '\n';
    }
    cout << '\n';
}
```

Kode 15.2 (Lanjutan) *SLOW-ALL-PAIRS-SHORTEST-PATHS selengkapnya, sambil mencetak setiap matriks antaranya. Kompilasilah dengan g++ -std=c++17 -O2 -o slow_apsp slow_apsp.cpp.*

```
int main() {
    const int n = 5;
    Matrix W = {
        {  0,   3,   8, INF,  -4},
        {INF,   0, INF,   1,   7},
        {INF,   4,   0, INF, INF},
        {  2, INF,  -5,   0, INF},
        {INF, INF, INF,   6,   0}
    };

    printMatrix("W", W);

    // SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
    Matrix L = W;                       // baris 2: L^(1) = W
    printMatrix("L^(1)", L);
    for (int m = 2; m <= n - 1; ++m) {  // baris 3: untuk m = 2 sampai n-1
        L = extendShortestPaths(L, W);  // baris 5
        printMatrix("L^(" + to_string(m) + ")", L);
    }

    // Satu hasil kali tambahan, untuk melihat bahwa tidak ada lagi yang berubah.
    Matrix L5 = extendShortestPaths(L, W);
    printMatrix("L^(5)", L5);
    cout << "L^(5) equals L^(4)? " << (L5 == L ? "yes" : "no") << '\n';
    return 0;
}
```

Keluarannya, dengan blok `W` dihilangkan karena ia mengulang `L^(1)`:

```
L^(1) =
       0    3    8  inf   -4
     inf    0  inf    1    7
     inf    4    0  inf  inf
       2  inf   -5    0  inf
     inf  inf  inf    6    0

L^(2) =
       0    3    8    2   -4
       3    0   -4    1    7
     inf    4    0    5   11
       2   -1   -5    0   -2
       8  inf    1    6    0

L^(3) =
       0    3   -3    2   -4
       3    0   -4    1   -1
       7    4    0    5   11
       2   -1   -5    0   -2
       8    5    1    6    0

L^(4) =
       0    1   -3    2   -4
       3    0   -4    1   -1
       7    4    0    5    3
       2   -1   -5    0   -2
       8    5    1    6    0

L^(5) =
       0    1   -3    2   -4
       3    0   -4    1   -1
       7    4    0    5    3
       2   -1   -5    0   -2
       8    5    1    6    0

L^(5) equals L^(4)? yes
```

Bandingkanlah matriks-matriks ini dengan yang ada di **Gambar 15.10**: keduanya cocok entri demi entri. Dan baris terakhirnya meneguhkan Persamaan (15.6) secara empiris — barisannya sudah berhenti bergerak pada $m = n - 1 = 4$, dan hasil kali kelimanya tidak mengubah apa pun.

15.4.12 Satu matriks dihitung sel demi sel

Keluaran di atas gampang diperiksa tetapi sukar *dirasakan*. Marilah kita menghitung satu baris utuh $L^{(2)}$ dengan tangan, supaya tidak ada lagi yang tersisa menjadi misteri. Kita memakai Persamaan (15.5) dengan $m = 2$, dan $L^{(1)} = W$ di sepanjangnya.

Baris 4 dari $L^{(2)}$. Untuk tiap tujuan j kita meminimumkan $l_{4k}^{(1)} + w_{kj}$ atas $k = 1,2,3,4,5$. Baris 4 dari W adalah $(2, \infty, -5, 0, \infty)$, sehingga hanya $k \in \{1,3,4\}$ yang dapat menyumbang suku berhingga.

$l_{41}^{(2)}$: calonnya adalah $l_{41}^{(1)} + w_{11} = 2 + 0 = 2$; $l_{42}^{(1)} + w_{21} = \infty + \infty = \infty$; $l_{43}^{(1)} + w_{31} = -5 + \infty = \infty$; $l_{44}^{(1)} + w_{41} = 0 + 2 = 2$; $l_{45}^{(1)} + w_{51} = \infty + \infty = \infty$. Minimumnya: **2**. Itulah sisi langsung $4 \to 1$.

$l_{42}^{(2)}$: calonnya $2 + 3 = 5$ (pergi $4 \to 1$ lalu $1 \to 2$); $\infty + 0 = \infty$; $-5 + 4 = \mathbf{-1}$ (pergi $4 \to 3$ lalu $3 \to 2$); $0 + \infty = \infty$; $\infty + \infty = \infty$. Minimumnya: $\mathbf{-1}$. Di sinilah sisi negatifnya membayar ongkosnya — rute dua sisi yang menembus simpul 3 berongkos -1, sementara rute satu sisi sama sekali tidak ada.

$l_{43}^{(2)}$: calonnya $2 + 8 = 10$; $\infty + \infty = \infty$; $-5 + 0 = -5$; $0 + (-5) = -5$; $\infty + \infty = \infty$. Minimumnya: $\mathbf{-5}$, yaitu sisi langsungnya, yang tercacah dua kali seperti di §15.4.5.

$l_{44}^{(2)}$: calonnya $2 + \infty = \infty$; $\infty + 1 = \infty$; $-5 + \infty = \infty$; $0 + 0 = 0$; $\infty + 6 = \infty$. Minimumnya: $\mathbf{0}$. Berdiam di tempat tetap gratis — dan perhatikanlah bahwa tidak ada siklus dua sisi menembus simpul 4 yang negatif, dan itulah bukti setempat bahwa graf ini tidak punya siklus berbobot negatif.

$l_{45}^{(2)}$: calonnya $2 + (-4) = \mathbf{-2}$ (pergi $4 \to 1$ lalu $1 \to 5$); $\infty + 7 = \infty$; $-5 + \infty = \infty$; $0 + \infty = \infty$; $\infty + 0 = \infty$. Minimumnya: $\mathbf{-2}$.

Jadi baris 4 dari $L^{(2)}$ adalah $(2,\ -1,\ -5,\ 0,\ -2)$ — persis baris keempat yang tercetak di atas, dan persis baris keempat $L^{(2)}$ pada **Gambar 15.10**.

Baris 2 dari $L^{(2)}$, yang dihitung dengan cara yang sama dari baris 2 dari W, yaitu $(\infty, 0, \infty, 1, 7)$:

- $l_{21}^{(2)} = \min\{\infty + 0,\ 0 + \infty,\ \infty + \infty,\ 1 + 2,\ 7 + \infty\} = \mathbf{3}$, lewat $2 \to 4 \to 1$.
- $l_{22}^{(2)} = \min\{\infty + 3,\ 0 + 0,\ \infty + 4,\ 1 + \infty,\ 7 + \infty\} = \mathbf{0}$.
- $l_{23}^{(2)} = \min\{\infty + 8,\ 0 + \infty,\ \infty + 0,\ 1 + (-5),\ 7 + \infty\} = \mathbf{-4}$, lewat $2 \to 4 \to 3$.
- $l_{24}^{(2)} = \min\{\infty + \infty,\ 0 + 1,\ \infty + \infty,\ 1 + 0,\ 7 + 6\} = \mathbf{1}$ (sudah dihitung di §15.4.5).
- $l_{25}^{(2)} = \min\{\infty + (-4),\ 0 + 7,\ \infty + \infty,\ 1 + \infty,\ 7 + 0\} = \mathbf{7}$, yaitu sisi langsungnya.

Jadi baris 2 adalah $(3,\ 0,\ -4,\ 1,\ 7)$: sekali lagi persis seperti yang dicetak programnya. Perhatikanlah bagaimana baris 2 berubah antara $L^{(1)}$ dan $L^{(2)}$: kedua entri ∞ di kolom 1 dan 3 menjadi 3 dan -4, sebab dengan jatah dua sisi simpul 2 kini dapat mencapai simpul 1 dan 3 dengan melompat lewat simpul 4. Itulah seluruh algoritmanya dalam satu kalimat: *setiap hasil kali membiarkan tiap rute tumbuh satu sisi lagi.*

15.4.13 Dari mana matriksnya berasal, dalam satu gambaran

Ada gunanya berhenti sejenak untuk melihat apa *arti* matriks yang berturut-turut itu, sebab artinyalah yang memungkinkan bagian berikutnya.

$L^{(1)}$ adalah peta perjalanan satu lompatan. $L^{(2)}$ adalah peta perjalanan paling banyak dua lompatan. $L^{(3)}$, paling banyak tiga. $L^{(4)}$, paling banyak empat — dan karena lintasan sederhana pada graf bersimpul lima bersisi paling banyak empat, $L^{(4)}$ adalah jawaban akhirnya. Bacalah entri l_{15}: ia bernilai -4 di setiap matriks, sebab sisi langsung $1 \to 5$ berbobot -4 sudah optimal dan tidak ada yang mengalahkannya. Bacalah l_{12}: ia bernilai 3 di $L^{(1)}$, $L^{(2)}$, dan $L^{(3)}$, lalu jatuh ke 1 di $L^{(4)}$, sebab rute termurah $1 \to 5 \to 4 \to 3 \to 2$ berbobot $-4 + 6 - 5 + 4 = 1$ memerlukan keempat sisinya dan tidak dapat muncul sebelum jatahnya mencapai empat. Entri tunggal itulah alasan mengapa algoritmanya harus berjalan sampai $m = n - 1$ dan tidak boleh berhenti lebih awal atas dasar firasat.

15.5 Memperbaiki Waktu Jalannya: Pemangkatan Berulang

15.5.1 Kita hanya menginginkan matriks yang terakhir

Satu pengamatan membuat segala sesuatu menjadi lebih murah: kita **tidak** berkepentingan menghitung *semua* matriks $L^{(m)}$. Kita hanya berkepentingan pada $L^{(n-1)}$. Matriks antaranya

adalah sarana, bukan tujuan, dan bila ada cara mencapai yang terakhir tanpa singgah di semua pendahulunya, kita sebaiknya mengambilnya.

Tanpa siklus berbobot negatif, Persamaan (15.6) mengatakan $L^{(m)} = L^{(n-1)}$ untuk **semua** bilangan bulat $m \geq n - 1$. Itu pernyataan yang lebih kuat daripada "barisannya menuju suatu limit": ia berkata barisannya *persis tetap* mulai dari indeks $n - 1$ dan seterusnya. Jadi tidaklah penting apakah kita menghitung $L^{(n-1)}$ atau $L^{(n)}$ atau $L^{(1000)}$ — asalkan indeksnya sedikitnya $n - 1$, kita memperoleh matriks yang sama, dan matriks itulah jawabannya. Kita bebas melampaui sasaran.

15.5.2 Asosiativitas itulah yang kita jual

Bahan yang kedua adalah asosiativitas hasil kali $\odot$, yang dijamin §15.4.9. Asosiativitas berkata bahwa urutan pengurungan sebuah rantai hasil kali tidak mengubah hasilnya. SLOW-ALL-PAIRS-SHORTEST-PATHS menghitung

$$W^4 = ((W \odot W) \odot W) \odot W$$

dalam tiga hasil kali. Tetapi asosiativitas mengizinkan

$$W^4 = (W \odot W) \odot (W \odot W),$$

dan kedua hasil kali yang di dalam itu *perhitungan yang sama*, sehingga kita cukup melakukannya sekali lalu memakainya ulang. Dua hasil kali, bukan tiga. Pada contoh yang lebih besar penghematannya dramatis: W^{1024} memerlukan 1023 hasil kali dengan cara lambatnya dan sepuluh dengan cara cerdiknya.

15.5.3 Barisan pangkat duanya

Gambar 15.11 menuliskan barisannya.

Yang kita perlukan hanya $L^{(n-1)}$, dan $L^{(m)} = L^{(n-1)}$ untuk setiap $m \geq n - 1$. Karena "perkalian" EXTEND-SHORTEST-PATHS bersifat asosiatif, kuadratkan berulang-ulang:

$$\begin{aligned}
L^{(1)} &= W, \\
L^{(2)} &= W^2 = W \cdot W, \\
L^{(4)} &= W^4 = W^2 \cdot W^2, \\
L^{(8)} &= W^8 = W^4 \cdot W^4, \\
&\vdots \\
L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}.
\end{aligned}$$

Karena $2^{\lceil \lg(n-1) \rceil} \geq n - 1$, hasil kali terakhir sama dengan $L^{(n-1)}$ sesudah hanya $\lceil \lg(n-1) \rceil$ perkalian.

Gambar 15.11 Hanya $\lceil \lg(n-1) \rceil$ hasil kali yang diperlukan, sebab masing-masing melipatduakan jatah sisinya. Hasil kali terakhirnya $L^{(2^{\lceil \lg(n-1) \rceil})}$ sama dengan $L^{(n-1)}$ menurut Persamaan (15.6).

Barisannya adalah

$$
\begin{aligned}
L^{(1)} &= W, \\
L^{(2)} &= W^2 = W \odot W, \\
L^{(4)} &= W^4 = W^2 \odot W^2, \\
L^{(8)} &= W^8 = W^4 \odot W^4, \\
&\vdots \\
L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \odot W^{2^{\lceil \lg(n-1) \rceil - 1}}.
\end{aligned}
$$

Setiap baris sesudah yang pertama mengkuadratkan baris di atasnya, sehingga jatah sisinya berlipat dua pada tiap langkah: 1,2,4,8,16, … Inilah persis teknik yang dipakai di Bab 3 untuk menghitung a^b dalam lg$b$ perkalian, yang diterapkan pada matriks di dalam semiring yang berbeda.

Lambang $\lceil x \rceil$ adalah **langit-langit** (*ceiling*) x: bilangan bulat terkecil yang sedikitnya x. Jadi $\lceil 3.2 \rceil = 4$, $\lceil 4 \rceil = 4$, $\lceil \lg 4 \rceil = 2$. Dan lg berarti $\log_2$, sebagaimana di seluruh buku ini.

15.5.4 Mengapa $\lceil \lg(n-1) \rceil$ hasil kali sudah cukup

Argumennya dua baris panjangnya. Inilah ia dengan setiap langkahnya ditampakkan.

Kita berhenti ketika pangkatnya pertama kali mencapai atau melewati $n-1$. Sesudah t kali pengkuadratan pangkatnya adalah 2^t. Maka kita menginginkan t terkecil dengan $2^t \geq n-1$. Mengambil lg kedua ruasnya — sah, sebab lg menaik — memberikan $t \geq \lg(n-1)$, dan karena t haruslah bilangan bulat, t terkecil semacam itu adalah

$$t = \lceil \lg(n-1) \rceil.$$

Pada titik itu pangkatnya adalah $2^{\lceil \lg(n-1) \rceil}$, yang memenuhi

$$2^{\lceil \lg(n-1) \rceil} \geq n-1$$

menurut konstruksinya. Menurut Persamaan (15.6), indeks mana pun yang berada pada atau melampaui $n-1$ memberikan matriks yang sama, sehingga

$$L^{(2^{\lceil \lg(n-1) \rceil})} = L^{(n-1)},$$

dan hasil kali terakhirnya adalah jawabannya. Dalam satu kalimat: *karena* $2^{\lceil \lg(n-1) \rceil} \geq n-1$, *hasil kali terakhir* $L^{(2^{\lceil \lg(n-1) \rceil})}$ *sama dengan* $L^{(n-1)}$.

Patut direnungkan: kita biasanya menghitung sebuah matriks dengan jatah sisi yang *lebih besar* daripada yang kita perlukan, dan itu tidak apa-apa. Melampaui sasaran tidak berongkos apa-apa sebab barisannya tetap melewati $n-1$. Bila seorang kolega berkeberatan bahwa W^{16} "mencacah lintasan bersisi enam belas" pada graf yang lintasan bergunanya tak satu pun bersisi lebih daripada sembilan, jawabannya adalah: benar, dan lintasan-lintasan itu tak pernah menang, sebab sembarang lintasan yang punya simpul terulang dapat dihapus siklusnya tanpa menaikkan bobotnya.

> **SEBUAH CACAH YANG KONKRET.** Untuk $n = 5$: $\lceil \lg 4 \rceil = 2$ hasil kali, berbanding $n-2 = 3$ untuk versi lambatnya. Hemat satu. Untuk $n = 1000$: $\lceil \lg 999 \rceil = 10$ hasil kali (sebab $2^9 = 512 < 999 \leq 1024 = 2^{10}$), berbanding 998. Hemat seratus kali lipat. Untuk $n = 10^6$: 20 hasil

> kali berbanding 999,998 — lima puluh ribu kali lipat. Makin besar grafnya, makin berharga tipuannya.

15.5.5 FASTER-ALL-PAIRS-SHORTEST-PATHS

Prosedurnya diperlihatkan pada **Gambar 15.12**.

FASTER-ALL-PAIRS-SHORTEST-PATHS(W)

1 $n = W.rows$
2 $L^{(1)} = W$
3 $m = 1$
4 **selama** $m < n - 1$
5 **misalkan** $L^{(2m)}$ sebuah matriks $n \times n$ baru
6 $L^{(2m)} =$ EXTEND-SHORTEST-PATHS($L^{(m)}, L^{(m)}$)
7 $m = 2m$
8 **kembalikan** $L^{(m)}$

Gambar 15.12 FASTER-ALL-PAIRS-SHORTEST-PATHS(W). Baris 6 mengkuadratkan matriks yang sedang berjalan alih-alih mengalikannya dengan W, dan itulah seluruh bedanya dari **Gambar 15.9**.

Bandingkanlah ia baris demi baris dengan SLOW-ALL-PAIRS-SHORTEST-PATHS dan Anda akan menemukan persis dua perubahan. Pertama, baris 6 memanggil EXTEND-SHORTEST-PATHS($L^{(m)}, L^{(m)}$) — pengkuadratan — di tempat versi lambatnya memanggil EXTEND-SHORTEST-PATHS($L^{(m-1)}, W$). Kedua, baris 7 melipatduakan m di tempat versi lambatnya menambahkan satu. Syarat gelung di baris 4, `while m < n - 1`, itulah yang membuat cacahnya keluar sebagai $\lceil \lg(n-1) \rceil$: badan gelungnya berjalan selama jatah yang sedang berlaku masih kurang daripada $n - 1$, dan tiap badan melipatduakannya.

Karena masing-masing dari $\lceil \lg(n-1) \rceil$ hasil kali matriksnya memakan waktu $\Theta(n^3)$, FASTER-ALL-PAIRS-SHORTEST-PATHS berjalan dalam

$$\Theta(n^3 \lceil \lg(n-1) \rceil) = \Theta(n^3 \lg n).$$

Itu lebih baik daripada $\Theta(n^4)$ sebesar faktor $n/\lg n$ — untuk $n = 1000$, kira-kira seratus kali lipat.

15.5.6 Contohnya lagi, dalam dua hasil kali

Kode 15.3 menjalankan versi pengkuadratannya pada contoh berjalannya.

Kode 15.3 FASTER-ALL-PAIRS-SHORTEST-PATHS, versi pemangkatan berulangnya. Kompilasilah dengan `g++ -std=c++17 -O2 -o faster_apsp faster_apsp.cpp`.

```
// Bab 15, Kode: FASTER-ALL-PAIRS-SHORTEST-PATHS -- pemangkatan berulang.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;
typedef vector<vector<long long>> Matrix;

Matrix extendShortestPaths(const Matrix& A, const Matrix& B) {
    int n = (int)A.size();
    Matrix C(n, vector<long long>(n, INF));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            C[i][j] = INF;
            for (int k = 0; k < n; ++k)
                if (A[i][k] < INF && B[k][j] < INF)
                    C[i][j] = min(C[i][j], A[i][k] + B[k][j]);
        }
    return C;
}

void printMatrix(const string& name, const Matrix& M) {
    int n = (int)M.size();
    cout << name << " =\n";
    for (int i = 0; i < n; ++i) {
        cout << "   ";
        for (int j = 0; j < n; ++j) {
            if (M[i][j] >= INF) cout << setw(5) << "inf";
            else                cout << setw(5) << M[i][j];
        }
        cout << '\n';
    }
    cout << '\n';
}
```

Kode 15.3 (Lanjutan) *FASTER-ALL-PAIRS-SHORTEST-PATHS, versi pemangkatan berulangnya. Kompilasilah dengan g++ -std=c++17 -O2 -o faster_apsp faster_apsp.cpp.*

```
int main() {
    const int n = 5;
    Matrix W = {
        {  0,   3,   8, INF,  -4},
        {INF,   0, INF,   1,   7},
        {INF,   4,   0, INF, INF},
        {  2, INF,  -5,   0, INF},
        {INF, INF, INF,   6,   0}
    };

    // FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
    Matrix L = W;            // baris 2: L^(1) = W
    int m = 1;               // baris 3
    int products = 0;
    printMatrix("L^(1)", L);
    while (m < n - 1) {                      // baris 4
        L = extendShortestPaths(L, L);       // baris 6: L^(2m) = L^(m) . L^(m)
        ++products;
        m = 2 * m;                           // baris 7
        printMatrix("L^(" + to_string(m) + ")", L);
    }
    cout << "n = " << n
         << ", matrix products performed = " << products
         << ", final exponent m = " << m << '\n';
    return 0;
}
```

Keluarannya:

```
L^(1) =
       0    3    8  inf   -4
     inf    0  inf    1    7
     inf    4    0  inf  inf
       2  inf   -5    0  inf
     inf  inf  inf    6    0

L^(2) =
       0    3    8    2   -4
       3    0   -4    1    7
     inf    4    0    5   11
       2   -1   -5    0   -2
       8  inf    1    6    0

L^(4) =
       0    1   -3    2   -4
       3    0   -4    1   -1
       7    4    0    5    3
       2   -1   -5    0   -2
       8    5    1    6    0

n = 5, matrix products performed = 2, final exponent m = 4
```

Tiga hal patut diperhatikan. Matriks terakhirnya $L^{(4)}$ **sama persis** dengan $L^{(4)}$ yang dihasilkan versi lambatnya di §15.4.11 — jawaban yang sama, hasil kali yang lebih sedikit. Matriks $L^{(3)}$ sama sekali tidak pernah dihitung; algoritmanya melompat dari jatah 2 ke jatah 4. Dan cacah di kakinya meneguhkan teorinya: dua hasil kali, persis [lg4].

Tabel 15.3 menggelar kedua jadwalnya berdampingan.

Tabel 15.3 Kedua jadwalnya pada contoh berjalan bersimpul lima. Versi lambatnya menambahkan satu pada jatah sisinya tiap hasil kali; versi cepatnya melipatduakannya.

Langkah	SLOW-APSP menghitung	Jatah	FASTER-APSP menghitung	Jatah
awal	$L^{(1)} = W$	1	$L^{(1)} = W$	1
1	$L^{(2)} = L^{(1)} \odot W$	2	$L^{(2)} = L^{(1)} \odot L^{(1)}$	2
2	$L^{(3)} = L^{(2)} \odot W$	3	$L^{(4)} = L^{(2)} \odot L^{(2)}$	4 — berhenti
3	$L^{(4)} = L^{(3)} \odot W$	4 — berhenti	—	—
hasil kali	$n - 2 = 3$		$\lceil \lg(n-1) \rceil = 2$	

15.5.7 Apa yang *tidak* diberikan pemangkatan berulang kepada kita

Dua peringatan yang jujur, sebab buku ajar yang hanya melaporkan kabar baik mengajar dengan buruk.

Ia tetap $\Theta(n^3 \lg n)$, dan Floyd–Warshall mengalahkannya. Bagian berikutnya menyajikan sebuah algoritma yang menyelesaikan persoalan yang sama dalam $\Theta(n^3)$ — tanpa logaritma sama sekali — dengan faktor konstanta yang lebih kecil dan memori yang lebih sedikit. Bila yang Anda inginkan hanyalah jawabannya, pakailah Floyd–Warshall. Nilai §15.4 dan §15.5 terletak pada *gagasannya*: pengamatan bahwa lintasan terpendek hidup di dalam sebuah semiring, yang menyeberang menjadi penutupan transitif di §15.7, menjadi lintasan terlebar, menjadi pencacahan lintasan, dan menjadi algoritma perkalian matriks cepat yang berada di luar jangkauan buku ini.

Strassen tidak menolong di sini. Pikiran yang wajar adalah: bila EXTEND-SHORTEST-PATHS itu perkalian matriks, mengapa tidak memakai algoritma perkalian matriks subkubik lalu memperoleh lintasan terpendek semua pasangan dalam $O(n^{2.81} \lg n)$? Jawabannya adalah bahwa algoritma Strassen beserta turunannya bersandar pada **pengurangan** — mereka menghitung jumlah dan selisih submatriks lalu mencoret suku-sukunya. Semiring $(\min, +)$ tidak punya pengurangan: tidak ada operasi yang membatalkan min. Jadi algoritma perkalian cepatnya sama sekali tidak berlaku. Ini bukan lubang yang menunggu diisi mahasiswa yang lebih cerdik; ia halangan struktural yang sejati, dan algoritma semua pasangan subkubik yang memang ada — punya Zwick, misalnya [8] — menempuh jalan yang cukup berbeda.

15.6 Algoritma Floyd–Warshall

Tiga fakta memperkenalkannya: ia menerima sebuah graf berarah berbobot $G = (V, E)$; sisi berbobot negatif diperbolehkan tetapi siklus berbobot negatif tidak; dan ia sebuah algoritma pemrograman dinamis yang berjalan dalam waktu $\Theta(V^3)$. Inilah algoritma yang paling sering diraih pemrogram yang bekerja ketika mereka memerlukan lintasan terpendek semua pasangan, dan alasannya adalah bahwa gelung terdalamnya lima baris panjangnya, sama sekali tidak memuat struktur data, dan berjalan secepat memori.

15.6.1 Cara lain memotong sebuah lintasan

Segala sesuatu di §15.4 bergantung pada satu pilihan: kita memerikan sebuah lintasan terpendek menurut **berapa banyak sisi** yang dipakainya, lalu membangun jawabannya dengan menaikkan jatah itu. Floyd dan Warshall membuat pilihan yang berbeda. Mereka memerikan sebuah lintasan terpendek menurut **simpul mana saja yang boleh dilewatinya**, lalu membangun jawabannya

dengan memperbesar himpunan yang diizinkan itu. Seluruh beda antara kedua algoritmanya adalah keputusan yang satu ini, dan patut disaksikan betapa banyak yang menyusul darinya.

Lebih dahulu sebuah definisi.

> **DEFINISI.** Sebuah **simpul antara** (*intermediate vertex*) pada sebuah lintasan sederhana $p = \langle v_1, v_2, \ldots, v_l \rangle$ adalah sembarang simpul p selain v_1 atau v_l — yaitu, sembarang simpul di dalam himpunan $\{v_2, v_3, \ldots, v_{l-1}\}$.

Kedua ujungnya bukan simpul antara. Lintasan satu sisi $\langle i, j \rangle$ sama sekali tidak punya simpul antara. Lintasan dua sisi $\langle i, k, j \rangle$ punya persis satu, yaitu k.

Sekarang nomorilah simpulnya $V = \{1,2, \ldots, n\}$ lalu tetapkanlah suatu k antara 0 dan n. Tinjaulah himpunan $\{1,2, \ldots, k\}$ — k simpul pertama dalam penomorannya. Untuk sebuah pasangan i, j, pandanglah **semua lintasan dari i ke j yang simpul antaranya seluruhnya diambil dari** $\{1,2, \ldots, k\}$. Kedua ujungnya, i dan j, tidak harus berada di himpunan itu; hanya apa yang terjadi di antaranya yang dibatasi. Misalkan p sebuah lintasan berbobot minimum di antara mereka. Amatilah bahwa p sederhana — simpul yang terulang akan memberikan siklus berbobot ≥ 0, yang dapat dihapus tanpa menaikkan bobotnya, persis seperti di §15.4.6.

Algoritmanya kini memanfaatkan sebuah hubungan antara p dan lintasan terpendek dari i ke j yang seluruh simpul antaranya berada di himpunan yang lebih kecil, $\{1,2, \ldots, k-1\}$.

15.6.2 Kedua kasusnya

Gambar 15.13 menyatakan hubungannya.

> Misalkan p lintasan terpendek dari i ke j yang semua simpul antaranya terletak di $\{1, 2, \ldots, k\}$. Hubungannya bergantung pada apakah k simpul antara bagi p:
>
> - k **tidak dilewati** p. Maka setiap simpul antara p ada di $\{1, \ldots, k-1\}$, dan p juga lintasan terpendek dari i ke j yang semua simpul antaranya di $\{1, \ldots, k-1\}$.
> - k **dilewati** p. Pecah p menjadi $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$; semua simpul antara p_1 maupun p_2 ada di $\{1, \ldots, k-1\}$.

Gambar 15.13 Struktur sebuah lintasan terpendek yang simpul antaranya terletak di $\{1, \ldots, k\}$, dipecah menurut apakah k sendiri terpakai.

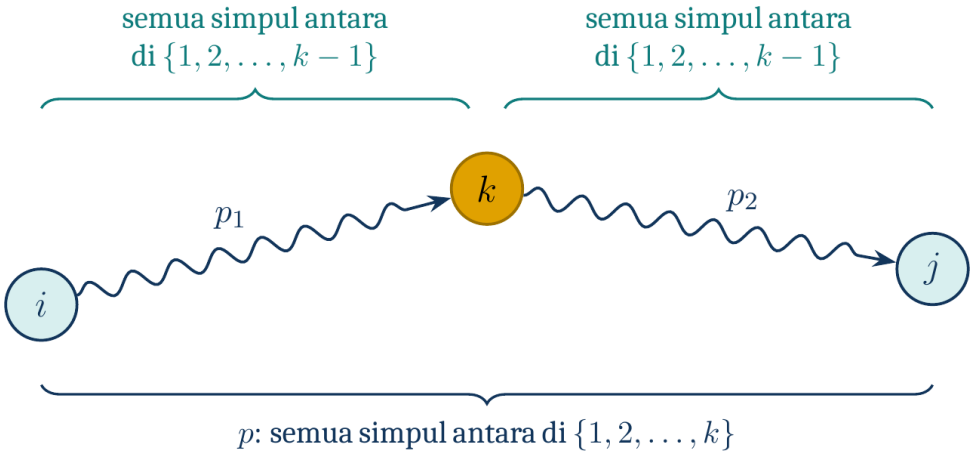
Hubungannya bergantung pada **apakah k merupakan simpul antara p atau bukan**, dan hanya kedua kemungkinan inilah yang ada.

Kasus 1: k bukan simpul antara p. Maka seluruh simpul antara p sudah terletak di $\{1,2, \ldots, k-1\}$. Jadi sebuah lintasan terpendek dari i ke j yang simpul antaranya di $\{1, \ldots, k-1\}$ *juga* merupakan lintasan terpendek dari i ke j yang simpul antaranya di $\{1, \ldots, k\}$ — izin tambahannya tidak terpakai, dan memberikan izin yang tidak Anda pakai tidak mengubah apa pun.

Kasus 2: k merupakan simpul antara p. Maka kita boleh menguraikan p di k:

$$i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j,$$

dengan p_1 bagian p sebelum k dan p_2 bagian sesudahnya. **Gambar 15.14** menggambarkan persis ini.



Gambar 15.14 Lintasan p (Gambar 25.3 pada edisi ketiga buku Cormen dkk.) adalah lintasan terpendek dari i ke j dan k adalah simpul antaranya yang bernomor tertinggi. Kedua paruhnya, p_1 dan p_2, lalu punya seluruh simpul antaranya di $\{1,2, \ldots, k-1\}$.

15.6.3 Mengapa kedua paruhnya dibatasi ke $\{1, \ldots, k-1\}$

Inilah langkah yang paling sering dilewati pemula. Bacalah pelan-pelan.

Menurut Teorema 13.1, p_1 adalah lintasan terpendek dari i ke k yang seluruh simpul antaranya berada di himpunan $\{1,2, \ldots, k\}$ — sebab ia sublintasan sebuah lintasan terpendek, dan simpul antaranya termasuk di antara simpul antara p.

Tetapi sekarang: **simpul k bukanlah simpul antara lintasan p_1.** Mengapa bukan? Sebab k adalah *ujung* p_1, dan menurut definisi di §15.6.1 sebuah ujung tak pernah menjadi simpul antara. Dan k tidak mungkin muncul untuk kedua kalinya di tengah p_1, sebab p sederhana sehingga tidak ada simpul yang terulang.

Karena itu seluruh simpul antara p_1 berada di himpunan $\{1,2, \ldots, k-1\}$, dan p_1 adalah lintasan terpendek dari i ke k yang seluruh simpul antaranya di $\{1,2, \ldots, k-1\}$. Serupa dengan itu, p_2 adalah lintasan terpendek dari k ke simpul j yang seluruh simpul antaranya di $\{1,2, \ldots, k-1\}$.

> **DENGAN BAHASA SEDERHANA.** Andaikanlah Anda diberi tahu rute termurah dari Keputih ke Darmo yang boleh melewati mana pun di antara tujuh simpul yang pertama, dan andaikanlah rute itu memang melewati simpul 7. Potonglah ia menjadi dua di simpul 7. Tiap paruhnya adalah rute termurah menurut haknya sendiri (Teorema 13.1) — dan tak satu pun paruhnya melewati *tengah* simpul 7, sebab simpul 7 kini menjadi ujung, dan rutenya tak pernah menyinggahi satu tempat dua kali. Jadi tiap paruhnya adalah rute termurah yang hanya memakai **enam** simpul pertama di tengahnya. Dua persoalan yang lebih kecil dengan bentuk yang persis sama. Itulah substruktur optimal, dan itulah sebabnya pemrograman dinamis bekerja di sini.

15.6.4 Rekurensinya

Kini kita dapat mendefinisikan besarannya lalu menuliskan rekurensinya, yang dirangkum pada **Gambar 15.15**.

Misalkan $d_{ij}^{(k)}$ bobot lintasan terpendek dari i ke j yang semua simpul antaranya ada di $\{1, 2, \ldots, k\}$.
Untuk $k = 0$ lintasannya sama sekali tidak memiliki simpul antara, sehingga memuat paling banyak satu sisi dan $d_{ij}^{(0)} = w_{ij}$. Secara rekursif,

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{jika } k = 0 \\ \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{jika } k \geq 1 \end{cases}$$

Setiap lintasan memiliki semua simpul antaranya di $\{1, \ldots, n\}$, sehingga jawabannya $D^{(n)} = \left(d_{ij}^{(n)}\right)$: $d_{ij}^{(n)} = \delta(i, j)$ untuk semua $i, j \in V$.

Gambar 15.15 Definisi $d_{ij}^{(k)}$, kasus dasarnya $d_{ij}^{(0)} = w_{ij}$, dan rekurensi pada Persamaan (15.7). Matriks terakhirnya $D^{(n)} = (d_{ij}^{(n)})$ adalah jawabannya.

DEFINISI. Misalkan $d_{ij}^{(k)}$ adalah bobot sebuah lintasan terpendek dari simpul i ke simpul j **yang seluruh simpul antaranya berada di himpunan** $\{1,2, \ldots, k\}$.

Sebagaimana pada $l_{ij}^{(m)}$, superskrip berkurungnya adalah label dan bukan pangkat. Tetapi kini ia melabeli sesuatu yang berbeda: (m) mencacah sisi, (k) mencacah *simpul antara yang diizinkan*. Kedua keluarga matriksnya berbeda, keduanya dihitung dengan cara yang berbeda, dan hanya anggota terakhirnya yang sama.

Kasus dasarnya. Ketika $k = 0$, sebuah lintasan dari simpul i ke simpul j yang tidak punya simpul antara bernomor lebih tinggi daripada 0 **sama sekali tidak punya simpul antara** — himpunan $\{1, \ldots, 0\}$ kosong. Lintasan semacam itu bersisi paling banyak satu, sehingga ia entah sisi langsung (i, j), atau tidak ada apa-apa. Maka

$$d_{ij}^{(0)} = w_{ij},$$

yang dalam bentuk matriks berarti $D^{(0)} = W$.

Kasus rekursifnya. Untuk $k \geq 1$ kedua kasus §15.6.2 menjadi kedua argumen sebuah minimum. Kasus 1 — jangan memakai k — menyumbang $d_{ij}^{(k-1)}$. Kasus 2 — pakailah k — menyumbang $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$, yaitu cara terbaik pergi dari i ke k dengan hanya memakai simpul antara bernomor rendah, ditambah cara terbaik pergi dari k ke j dengan hanya memakai simpul antara bernomor rendah. Persis satu di antara kedua kasusnya berlaku bagi lintasan optimalnya, dan karena kita mengambil minimum atas keduanya, kita tidak mungkin keliru ke arah mana pun. Jadi

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{jika } k = 0, \\ \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{jika } k \geq 1. \end{cases} \tag{15.7}$$

Mengapa jawabannya adalah $D^{(n)}$. Bagi **sembarang** lintasan, seluruh simpul antaranya berada di himpunan $\{1,2, \ldots, n\}$ — tidak ada simpul lain untuk ditempati. Jadi pembatasan pada $k = n$ sama sekali bukan pembatasan, dan

$$d_{ij}^{(n)} = \delta(i, j) \quad \text{untuk semua } i, j \in V. \tag{15.8}$$

Matriks $D^{(n)} = (d_{ij}^{(n)})$ adalah jawaban yang kita cari: ia memenuhi sasaran Persamaan (15.2) yang ditetapkan di §15.2.2.

> **SALAH BACA YANG SERING TERJADI.** Pada suku $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$, kedua superskripnya adalah $(k-1)$, bukan (k). Itulah seluruh isi §15.6.3: kedua paruhnya dibatasi ke himpunan yang *lebih kecil*. Menuliskan $d_{ik}^{(k)} + d_{kj}^{(k)}$ akan melingkar — ia akan mendefinisikan $D^{(k)}$ lewat dirinya sendiri — dan ia akan mengizinkan sebuah rute dari i ke k yang melewati k, yang tidak masuk akal.

Bandingkanlah Persamaan (15.7) dengan Persamaan (15.5) sejenak. Rekurensi perkalian matriksnya mengambil minimum atas n calon dan harus diterapkan $n-1$ kali; rekurensi Floyd–Warshall mengambil minimum atas hanya **dua** calon dan harus diterapkan n kali. Di situlah sumber seluruh beda kecepatannya: n putaran berisi n^2 pembaruan berwaktu tetap adalah $\Theta(n^3)$, sementara n putaran berisi n^2 minimum-atas-n akan menjadi $\Theta(n^4)$.

15.6.5 Algoritmanya

Gambar 15.16 memberikan prosedurnya.

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  untuk k = 1 sampai n
4      misalkan D^(k) = (d_ij^(k)) sebuah matriks n × n baru
5      untuk i = 1 sampai n
6          untuk j = 1 sampai n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  kembalikan D^(n)
```

Gambar 15.16 FLOYD-WARSHALL(W): lima kata keadaan dan tiga gelung bersarang. Baris 7 adalah rekurensi pada Persamaan (15.7) yang dituliskan lengkap.

Baris 2 menyetel $D^{(0)} = W$, kasus dasarnya. Baris 3 menjalankan k dari 1 sampai n: inilah gelung yang memperbesar himpunan yang diizinkan, satu simpul demi satu simpul. Baris 5 dan 6 melangkahi setiap pasangan (i, j). Baris 7 menerapkan rekurensinya. Baris 8 mengembalikan $D^{(n)}$, yang menurut Persamaan (15.8) adalah matriks bobot lintasan terpendeknya.

Nilai $d_{ij}^{(k)}$ dihitung **menurut nilai k yang menaik**. Subbagian berikutnya menjelaskan mengapa itu bukan kegemaran gaya melainkan tuntutan kebenaran.

15.6.6 Mengapa k harus menjadi gelung terluar

Inilah pertanyaan yang seharusnya diajukan setiap pembaca dan yang jarang diajukan: badan gelung rangkap tiganya kelihatan setangkup, jadi mengapa kita tidak boleh menuliskan gelungnya dalam urutan sesuka hati?

Jawabannya adalah struktur kebergantungannya. Untuk menghitung $d_{ij}^{(k)}$ kita memerlukan tiga nilai, semuanya bersuperskrip $(k-1)$: $d_{ij}^{(k-1)}$, $d_{ik}^{(k-1)}$, dan $d_{kj}^{(k-1)}$. Setiap nilai yang kita baca harus milik putaran **sebelumnya**. Bila k adalah gelung terluarnya, itu terjadi dengan sendirinya: ketika

kita memulai putaran k, setiap entri matriksnya bersuperskrip $(k-1)$, dan kita menyelesaikan putarannya sebelum beranjak. Bila k *bukan* yang terluar — bila, katakanlah, i yang terluar dan k yang terdalam — maka ketika kita menghitung entri d_{ij} kita mungkin meraih d_{kj} pada baris k yang belum disinggahi sapuan ini, dan kita memperoleh nilai dari tahap perhitungan yang lebih awal dan lebih lemah daripada yang dituntut rekurensinya.

Akibatnya tidaklah halus. **Kode 15.4** membangun dua graf bersimpul empat yang dirancang untuk menampakkannya. Graf (a) adalah lintasan berarah $1 \to 4 \to 3 \to 2$, tiap sisinya berbobot 1: rute dari 1 ke 2 memerlukan simpul 4 dan 3 sebagai simpul antara, keduanya bernomor *lebih tinggi* daripada tujuannya. Graf (b), yang sekaligus merupakan jawaban Soal 15.9, punya sisi $1 \to 4$ berbobot 6, $4 \to 3$ berbobot 3, $4 \to 2$ berbobot 6, dan $3 \to 2$ berbobot 1.

Kode 15.4 Gelung rangkap tiga yang sama dengan k terluar dan dengan k terdalam, pada dua graf yang dibangun untuk membedakan keduanya. Kompilasilah dengan `g++ -std=c++17 -O2 -o ex_loop_order ex_loop_order.cpp`.

```
// Bab 15, Kode: apa yang terjadi ketika k bukan gelung terluar.
// Dua graf: pada (a) urutan yang keliru kehilangan sebuah lintasan sama sekali, pada (b)
ia menemukan
// lintasan tetapi menagih terlalu mahal. Keduanya bersimpul empat dan berbobot tak
negatif.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;
typedef vector<vector<long long>> Matrix;

Matrix loopKOutermost(Matrix D) {
    int n = (int)D.size();
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (D[i][k] < INF && D[k][j] < INF)
                    D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
    return D;
}

Matrix loopKInnermost(Matrix D) {
    int n = (int)D.size();
    for (int i = 0; i < n; ++i)                        // KELIRU: i terluar
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                if (D[i][k] < INF && D[k][j] < INF)
                    D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
    return D;
}

void printMatrix(const string& name, const Matrix& M) {
    cout << name << '\n';
    for (size_t i = 0; i < M.size(); ++i) {
        cout << "   ";
        for (size_t j = 0; j < M.size(); ++j) {
            if (M[i][j] >= INF) cout << setw(6) << "inf";
            else                cout << setw(6) << M[i][j];
        }
        cout << '\n';
    }
}
```

Kode 15.4 (Lanjutan) *Gelung rangkap tiga yang sama dengan k terluar dan dengan k terdalam, pada dua graf yang dibangun untuk membedakan keduanya. Kompilasilah dengan g++ -std=c++17 -O2 -o ex_loop_order ex_loop_order.cpp.*

```
void compare(const string& title, const Matrix& W, int si, int sj) {
    Matrix right = loopKOutermost(W);
    Matrix wrong = loopKInnermost(W);
    cout << title << '\n';
    printMatrix("  k outermost (correct):", right);
    printMatrix("  i outermost, k innermost (wrong):", wrong);
    cout << "  the two agree: " << (right == wrong ? "yes" : "NO") << '\n';
    cout << "  delta(" << si + 1 << "," << sj + 1 << "): correct = " << right[si][sj]
         << ", wrong = ";
    if (wrong[si][sj] >= INF) cout << "inf (no path found at all)\n";
    else                      cout << wrong[si][sj] << '\n';

    // Mengulang urutan gelung yang keliru memang akhirnya menuju jawaban benar -- ia hanya
    // lebih lambat dan menuntut argumen kebenaran tambahan.
    Matrix repaired = W;
    int passes = 0;
    while (repaired != right && passes < 10) { repaired = loopKInnermost(repaired);
++passes; }
    cout << "  passes of the wrong order needed before it matches: " << passes << "\n\n";
}

int main() {
    // (a) Lintasan berarah 1 -> 4 -> 3 -> 2, tiap sisi berbobot 1. Simpul antara
    // yang dibutuhkan lintasan terpendek dari 1 ke 2 adalah 4 dan 3 -- keduanya
    // bernomor lebih tinggi daripada tujuannya, dan itulah yang mengalahkan urutan yang
keliru.
    Matrix a = {
        {  0, INF, INF,   1},
        {INF,   0, INF, INF},
        {INF,   1,   0, INF},
        {INF, INF,   1,   0}
    };
    // (b) Sisi 1->4 (6), 4->3 (3), 4->2 (6), 3->2 (1). Urutan yang keliru menemukan
    // 1 -> 4 -> 2 seharga 12 dan tidak pernah menemukan 1 -> 4 -> 3 -> 2 seharga 10.
    Matrix b = {
        {  0, INF, INF,   6},
        {INF,   0, INF, INF},
        {INF,   1,   0, INF},
        {INF,   6,   3,   0}
    };
    compare("(a) the wrong order loses the path entirely:", a, 0, 1);
    compare("(b) the wrong order finds a path, but overcharges:", b, 0, 1);
    return 0;
}
```

Keluarannya:

```
(a) the wrong order loses the path entirely:
  k outermost (correct):
        0     3     2     1
      inf     0   inf   inf
      inf     1     0   inf
      inf     2     1     0
  i outermost, k innermost (wrong):
        0   inf     2     1
      inf     0   inf   inf
      inf     1     0   inf
      inf     2     1     0
  the two agree: NO
  delta(1,2): correct = 3, wrong = inf (no path found at all)
  passes of the wrong order needed before it matches: 2

(b) the wrong order finds a path, but overcharges:
  k outermost (correct):
        0    10     9     6
      inf     0   inf   inf
      inf     1     0   inf
      inf     4     3     0
  i outermost, k innermost (wrong):
        0    12     9     6
      inf     0   inf   inf
      inf     1     0   inf
      inf     4     3     0
  the two agree: NO
  delta(1,2): correct = 10, wrong = 12
  passes of the wrong order needed before it matches: 2
```

Pada graf (a) urutan yang keliru bukan sekadar memberikan jawaban yang sedikit terlalu besar; ia melaporkan **tidak ada lintasan sama sekali** dari simpul 1 ke simpul 2, padahal nyatanya ada lintasan berbobot 3. Telusurilah. Dengan i terluar, hal pertama yang dikerjakan programnya adalah mengisi baris 1 selengkapnya. Ketika ia sampai di sel (1,2) ia menyapu k dan mencari $d_{1k} + d_{k2}$. Kombinasi yang berguna adalah $k = 4$: $d_{14} = 1$ dan d_{42} semestinya 2 — tetapi baris 4 belum disentuh, sehingga d_{42} masih ∞. Kombinasinya gagal. Pada saat baris 4 akhirnya terisi dengan $d_{42} = 2$, programnya sudah lama melewati sel (1,2) dan tak akan pernah menengoknya lagi.

Graf (b) memperlihatkan cara gagal yang lebih senyap. Urutan yang keliru memang menemukan sebuah rute dari 1 ke 2 — rute dua sisi $1 \to 4 \to 2$ seharga $6 + 6 = 12$ — tetapi ia tak pernah menemukan $1 \to 4 \to 3 \to 2$ seharga $6 + 3 + 1 = 10$, sebab ketika gelung-j mencapai kolom 2 entri d_{13} belum diperbaiki dari ∞ menjadi 9; itu baru terjadi satu kolom kemudian. Bilangan yang berhingga, masuk akal, dan keliru jauh lebih sukar diendus di lingkungan produksi daripada sebuah ∞.

Baris terakhir tiap blok adalah cerita rakyat yang sungguh berguna: menjalankan urutan gelung yang keliru **berulang kali** memang akhirnya menuju jawaban benar, di sini sesudah dua sapuan. Tetapi tiap sapuan berongkos $\Theta(n^3)$, dan banyaknya sapuan yang diperlukan bergantung pada penomoran simpulnya, sehingga biaya totalnya tidak terbatasi $\Theta(n^3)$ dan algoritmanya telah kehilangan jaminannya. Betulkanlah urutan gelungnya.

> **KUTU FLOYD–WARSHALL YANG PALING SERING TERJADI.** `for (i) for (j) for (k)` alih-alih `for (k) for (i) for (j)`. Ia terkompilasi, ia berjalan, ia menghasilkan matriks yang kelihatan masuk akal, dan ia keliru. Bila Anda mengingat satu hal saja dari bagian ini, ingatlah bahwa k yang pertama.

15.6.7 Matriks pendahulunya

Versi pada **Gambar 15.16** hanya mengembalikan jarak. Untuk memperoleh rutenya kita menghitung $\Pi^{(k)}$ berdampingan dengan $D^{(k)}$, memakai analisis kasus yang sama.

> **DEFINISI.** Misalkan $\pi_{ij}^{(k)}$ adalah pendahulu simpul j pada sebuah lintasan terpendek dari i yang seluruh simpul antaranya terletak di $\{1,2,\ldots,k\}$.

Kasus dasarnya mencerminkan $d_{ij}^{(0)} = w_{ij}$. Tanpa simpul antara yang diizinkan, satu-satunya rute dari i ke j adalah sisi langsungnya, yang pendahulu j-nya adalah i:

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{jika } i = j \text{ atau } w_{ij} = \infty, \\ i & \text{jika } i \neq j \text{ dan } w_{ij} < \infty. \end{cases} \tag{15.9}$$

Bermula dari Persamaan (15.9), kasus rekursifnya mengikuti kedua kasus rekurensinya. Bila minimum di Persamaan (15.7) dicapai oleh argumen pertamanya — kita *tidak* merutekan lewat k — maka rutenya tidak berubah dan begitu pula pendahulunya. Bila minimumnya dicapai oleh argumen kedua — kita *memang* merutekan lewat k — maka rute barunya berakhir dengan paruh ekor $k \rightsquigarrow j$, sehingga pendahulu j adalah apa pun yang menjadi pendahulunya pada rute terbaik **dari** k:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{jika } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{jika } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases} \tag{15.10}$$

Baris keduanya pantas ditatap, sebab terkaan yang wajar — $\pi_{ij}^{(k)} = k$ — justru keliru. Simpul k berada di suatu tempat di *tengah* rute barunya, bukan di ujungnya. Yang kita perlukan adalah simpul persis sebelum j, dan pada penggal $k \rightsquigarrow j$ simpul itu persis $\pi_{kj}^{(k-1)}$. Di dalam kode, barisnya adalah `P[i][j] = P[k][j];`, bukan `P[i][j] = k;`.

15.6.8 Contoh berjalan, setiap $D^{(k)}$ dan $\Pi^{(k)}$

Gambar 15.17 memperlihatkan penelusuran lengkap Floyd–Warshall pada contoh berjalannya: enam matriks jarak dan enam matriks pendahulu.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Gambar 15.17 Barisan lengkap matriks $D^{(k)}$ dan $\Pi^{(k)}$ yang dihitung Floyd–Warshall untuk graf pada **Gambar 15.2** (Gambar 25.4 pada edisi ketiga buku Cormen dkk.). Baris $k = 0$ adalah masukannya; baris $k = 5$ adalah jawabannya.

Kode 15.5 adalah program lengkap yang menghasilkannya, beserta rekonstruksi lintasan dan pendeteksian siklus negatif.

Kode 15.5 Floyd–Warshall beserta matriks pendahulunya, mencetak setiap $D^{(k)}$ dan $\Pi^{(k)}$ lalu setiap rutenya. Kompilasilah dengan `g++ -std=c++17 -O2 -o floyd_warshall floyd_warshall.cpp`.

```
// Bab 15, Kode: algoritma Floyd-Warshall, lengkap dengan matriks pendahulu
// dan rekonstruksi lintasan, ditelusuri pada graf contoh berjalan.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;
const int NIL = -1;

typedef vector<vector<long long>> Matrix;
typedef vector<vector<int>>       PiMatrix;

void printD(const string& name, const Matrix& D) {
    int n = (int)D.size();
    cout << name << " =\n";
    for (int i = 0; i < n; ++i) {
        cout << "   ";
        for (int j = 0; j < n; ++j) {
            if (D[i][j] >= INF) cout << setw(6) << "inf";
            else                cout << setw(6) << D[i][j];
        }
        cout << '\n';
    }
}

void printPi(const string& name, const PiMatrix& P) {
    int n = (int)P.size();
    cout << name << " =\n";
    for (int i = 0; i < n; ++i) {
        cout << "   ";
        for (int j = 0; j < n; ++j) {
            if (P[i][j] == NIL) cout << setw(6) << "NIL";
            else                cout << setw(6) << P[i][j] + 1;   // berbasis 1 untuk
pencetakan
        }
        cout << '\n';
    }
}
```

Kode 15.5 (Lanjutan) *Floyd–Warshall beserta matriks pendahulunya, mencetak setiap D^{(k)} dan \Pi^{(k)} lalu setiap rutenya. Kompilasilah dengan g++ -std=c++17 -O2 -o floyd_warshall floyd_warshall.cpp.*

```
// PRINT-ALL-PAIRS-SHORTEST-PATH(Pi, i, j), persis seperti gambar di Subbab 15.2.3.
void printAllPairsShortestPath(const PiMatrix& P, int i, int j) {
    if (i == j) {
        cout << i + 1;
    } else if (P[i][j] == NIL) {
        cout << "no path from " << i + 1 << " to " << j + 1 << " exists";
    } else {
        printAllPairsShortestPath(P, i, P[i][j]);
        cout << " -> " << j + 1;
    }
}

int main() {
    const int n = 5;
    Matrix W = {
        {  0,   3,   8, INF,  -4},
        {INF,   0, INF,   1,   7},
        {INF,   4,   0, INF, INF},
        {  2, INF,  -5,   0, INF},
        {INF, INF, INF,   6,   0}
    };

    // D^(0) = W, dan Pi^(0) dari kasus dasar Pi di Subbab 15.6.7.
    Matrix   D = W;
    PiMatrix P(n, vector<int>(n, NIL));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (i != j && W[i][j] < INF) P[i][j] = i;

    printD("D^(0)", D);   printPi("Pi^(0)", P);   cout << '\n';

    // FLOYD-WARSHALL(W), dikerjakan di tempat: baris 7 pseudokodenya.
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (D[i][k] < INF && D[k][j] < INF && D[i][k] + D[k][j] < D[i][j]) {
                    D[i][j] = D[i][k] + D[k][j];
                    P[i][j] = P[k][j];            // pendahulu j pada lintasan yang
menembus k
                }
        printD("D^(" + to_string(k + 1) + ")", D);
        printPi("Pi^(" + to_string(k + 1) + ")", P);
        cout << '\n';
    }
```

Kode 15.5 (Lanjutan) *Floyd–Warshall beserta matriks pendahulunya, mencetak setiap D^{(k)} dan \Pi^{(k)} lalu setiap rutenya. Kompilasilah dengan g++ -std=c++17 -O2 -o floyd_warshall floyd_warshall.cpp.*

```
    // Siklus berbobot negatif akan tampak sebagai entri negatif pada diagonalnya.
    bool negativeCycle = false;
    for (int i = 0; i < n; ++i) if (D[i][i] < 0) negativeCycle = true;
    cout << "negative-weight cycle detected? " << (negativeCycle ? "yes" : "no") << "\n\n";

    cout << "Shortest paths read out of Pi^(5):\n";
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            if (i == j) continue;
            cout << "  " << i + 1 << " to " << j + 1 << " (weight " << setw(3) << D[i][j]
<< "): ";
            printAllPairsShortestPath(P, i, j);
            cout << '\n';
        }
    return 0;
}
```

Keluarannya, selengkapnya:

```
D^(0) =
        0     3     8   inf    -4
      inf     0   inf     1     7
      inf     4     0   inf   inf
        2   inf    -5     0   inf
      inf   inf   inf     6     0
Pi^(0) =
      NIL     1     1   NIL     1
      NIL   NIL   NIL     2     2
      NIL     3   NIL   NIL   NIL
        4   NIL     4   NIL   NIL
      NIL   NIL   NIL     5   NIL

D^(1) =
        0     3     8   inf    -4
      inf     0   inf     1     7
      inf     4     0   inf   inf
        2     5    -5     0    -2
      inf   inf   inf     6     0
Pi^(1) =
      NIL     1     1   NIL     1
      NIL   NIL   NIL     2     2
      NIL     3   NIL   NIL   NIL
        4     1     4   NIL     1
      NIL   NIL   NIL     5   NIL

D^(2) =
        0     3     8     4    -4
      inf     0   inf     1     7
      inf     4     0     5    11
        2     5    -5     0    -2
      inf   inf   inf     6     0
Pi^(2) =
      NIL     1     1     2     1
      NIL   NIL   NIL     2     2
      NIL     3   NIL     2     2
        4     1     4   NIL     1
      NIL   NIL   NIL     5   NIL

D^(3) =
        0     3     8     4    -4
      inf     0   inf     1     7
      inf     4     0     5    11
        2    -1    -5     0    -2
      inf   inf   inf     6     0
Pi^(3) =
      NIL     1     1     2     1
      NIL   NIL   NIL     2     2
      NIL     3   NIL     2     2
        4     3     4   NIL     1
      NIL   NIL   NIL     5   NIL
```

```
D^(4) =
        0     3    -1     4    -4
        3     0    -4     1    -1
        7     4     0     5     3
        2    -1    -5     0    -2
        8     5     1     6     0
Pi^(4) =
      NIL     1     4     2     1
        4   NIL     4     2     1
        4     3   NIL     2     1
        4     3     4   NIL     1
        4     3     4     5   NIL

D^(5) =
        0     1    -3     2    -4
        3     0    -4     1    -1
        7     4     0     5     3
        2    -1    -5     0    -2
        8     5     1     6     0
Pi^(5) =
      NIL     3     4     5     1
        4   NIL     4     2     1
        4     3   NIL     2     1
        4     3     4   NIL     1
        4     3     4     5   NIL

negative-weight cycle detected? no
```

Setiap satu dari kedua belas matriks ini cocok, entri demi entri, dengan **Gambar 15.17**. Dan $D^{(5)}$ cocok dengan $L^{(4)}$ dari §15.4.11: dua algoritma yang sama sekali berbeda, satu jawaban.

15.6.9 Putaran $k = 1$, sel demi sel

Sekarang kita mengerjakan satu putaran utuh dengan tangan. Putaran $k = 1$ bertanya: *untuk setiap pasangan* (i, j), *apakah merutekan lewat simpul 1 mengalahkan apa yang sudah kita punya?* Calonnya adalah $d_{i1}^{(0)} + d_{1j}^{(0)}$; petahananya adalah $d_{ij}^{(0)}$; pemenangnya yang lebih kecil. Kedua puluh lima selnya, diambil menurut urutan kunjungan gelungnya, dengan memakai $D^{(0)} = W$:

Baris $i = 1$**.** Setiap calonnya adalah $d_{11} + d_{1j} = 0 + d_{1j} = d_{1j}$, seri dengan petahananya. Tidak ada yang dapat berubah: $(0,3,8,\infty,-4)$ tetap $(0,3,8,\infty,-4)$. Secara umum, **baris** k **dan kolom** k **tak pernah berubah selama putaran** k, sebab calonnya merosot menjadi petahananya ditambah nol. Itu fakta yang berguna dan §15.6.13 mengubahnya menjadi sebuah teorema.

Baris $i = 2$**.** Di sini $d_{21}^{(0)} = \infty$: simpul 2 tidak punya rute ke simpul 1 tanpa simpul antara, sehingga setiap calon $\infty + d_{1j}$ bernilai ∞ dan kalah. Baris 2 tetap $(\infty, 0, \infty, 1,7)$. Secara umum, **bila** $d_{ik} = \infty$ **maka baris** i **tidak dapat berubah selama putaran** k — sebuah uji yang patut dipasang di kodenya, dan **Kode 15.11** memasangnya.

Baris $i = 3$**.** Ceritanya sama: $d_{31}^{(0)} = \infty$, sehingga baris 3 tetap $(\infty, 4,0, \infty, \infty)$.

Baris $i = 4$**.** Kini ada sesuatu yang terjadi, sebab $d_{41}^{(0)} = 2$ berhingga — simpul 4 punya sisi langsung ke simpul 1.

- $j = 1$: $\min(d_{41}^{(0)} = 2,\ d_{41}^{(0)} + d_{11}^{(0)} = 2 + 0 = 2) = 2$. Tidak berubah.
- $j = 2$: $\min(d_{42}^{(0)} = \infty,\ d_{41}^{(0)} + d_{12}^{(0)} = 2 + 3 = 5) = \mathbf{5}$. **Berubah** dari ∞ menjadi 5: rute $4 \to 1 \to 2$ adalah rute pertama dari 4 ke 2 yang kita temukan. Karena perbaikannya datang dari merutekan lewat $k = 1$, Persamaan (15.10) menyetel $\pi_{42}^{(1)} = \pi_{12}^{(0)} = 1$.

- $j = 3$: $\min(-5,\ 2 + 8 = 10) = -5$. Tidak berubah; sisi langsung $4 \to 3$ berbobot -5 lebih baik daripada memutar lewat 1.
- $j = 4$: $\min(0,\ 2 + \infty = \infty) = 0$. Tidak berubah.
- $j = 5$: $\min(d_{45}^{(0)} = \infty,\ 2 + (-4) = -2) = \mathbf{-2}$. **Berubah**: rute $4 \to 1 \to 5$ berongkos $2 - 4 = -2$. Dan $\pi_{45}^{(1)} = \pi_{15}^{(0)} = 1$.

Baris $i = 5$**.** $d_{51}^{(0)} = \infty$, sehingga tidak ada yang berubah: $(\infty, \infty, \infty, 6, 0)$.

Dua sel berubah dalam seluruh putarannya, keduanya di baris 4, dan keduanya tampak di $D^{(1)}$ yang tercetak di atas: angka 5 di posisi (4,2) dan angka -2 di posisi (4,5). Kedua entri $\Pi^{(1)}$ yang bersesuaian keduanya menjadi 1. Selebihnya tidak tersentuh. Itulah satu putaran utuh Floyd–Warshall, tanpa satu langkah pun dihilangkan.

15.6.10 Putaran $k = 4$, tempat sebagian besar kerjanya terjadi

Putaran $k = 1$ sepi sebab simpul 1 hanya punya satu sisi masuk. Putaran $k = 4$ adalah yang ramai, sebab pada saat itu $D^{(3)}$ sudah punya banyak entri berhingga dan simpul 4 duduk di tengah grafnya. Bekerja dari $D^{(3)}$, yang barisnya adalah $(0,3,8,4,-4)$, $(\infty, 0, \infty, 1,7)$, $(\infty, 4,0,5,11)$, $(2, -1, -5,0, -2)$, dan $(\infty, \infty, \infty, 6,0)$:

Baris 1 ($d_{14}^{(3)} = 4$): $j = 1$: $\min(0,4 + 2 = 6) = 0$. $j = 2$: $\min(3,4 + (-1) = 3) = 3$, seri, sehingga menurut Persamaan (15.10) petahananya mempertahankan selnya dan π tidak berubah. $j = 3$: $\min(8,\ 4 + (-5) = -1) = \mathbf{-1}$, **berubah**, dan $\pi_{13}^{(4)} = \pi_{43}^{(3)} = 4$. $j = 4$: $\min(4,4 + 0) = 4$. $j = 5$: $\min(-4,\ 4 + (-2) = 2) = -4$.

Baris 2 ($d_{24}^{(3)} = 1$): $j = 1$: $\min(\infty,\ 1 + 2 = 3) = \mathbf{3}$, **berubah**, $\pi_{21}^{(4)} = \pi_{41}^{(3)} = 4$. $j = 2$: $\min(0,\ 1 + (-1) = 0) = 0$. $j = 3$: $\min(\infty,\ 1 + (-5) = -4) = \mathbf{-4}$, **berubah**, $\pi_{23}^{(4)} = \pi_{43}^{(3)} = 4$. $j = 4$: $\min(1,1 + 0) = 1$. $j = 5$: $\min(7,\ 1 + (-2) = -1) = \mathbf{-1}$, **berubah**, $\pi_{25}^{(4)} = \pi_{45}^{(3)} = 1$.

Baris 3 ($d_{34}^{(3)} = 5$): $j = 1$: $\min(\infty,\ 5 + 2 = 7) = \mathbf{7}$, **berubah**, $\pi_{31}^{(4)} = 4$. $j = 2$: $\min(4,\ 5 + (-1) = 4) = 4$, seri. $j = 3$: $\min(0,\ 5 + (-5) = 0) = 0$, seri — dan perhatikanlah bahwa ini diagonalnya, yang berarti siklus $3 \to 2 \to 4 \to 3$ berbobot persis 0, bukan negatif. $j = 4$: $\min(5,5 + 0) = 5$. $j = 5$: $\min(11,\ 5 + (-2) = 3) = \mathbf{3}$, **berubah**, $\pi_{35}^{(4)} = \pi_{45}^{(3)} = 1$.

Baris 4 adalah baris k, sehingga tidak ada yang berubah, persis seperti diramalkan §15.6.9.

Baris 5 ($d_{54}^{(3)} = 6$): $j = 1$: $\min(\infty, 6 + 2 = 8) = \mathbf{8}$, **berubah**, $\pi_{51}^{(4)} = 4$. $j = 2$: $\min(\infty, 6 + (-1) = 5) = \mathbf{5}$, **berubah**, $\pi_{52}^{(4)} = \pi_{42}^{(3)} = 3$. $j = 3$: $\min(\infty, 6 + (-5) = 1) = \mathbf{1}$, **berubah**, $\pi_{53}^{(4)} = 4$. $j = 4$: $\min(6,6 + 0) = 6$. $j = 5$: $\min(0,6 + (-2) = 4) = 0$.

Sembilan sel berubah di putaran 4 — bandingkanlah $D^{(3)}$ dan $D^{(4)}$ yang tercetak dan Anda akan mencacah persis sembilan beda. Sesudah putaran ini setiap entri D berhingga: grafnya terhubung kuat, dan simpul 4 adalah simpul induk yang menyatukan kepingan-kepingannya.

15.6.11 Membaca rute dari Π

Dengan $\Pi^{(5)}$ di tangan, PRINT-ALL-PAIRS-SHORTEST-PATH dari §15.2.3 mengubah entri mana pun menjadi rute. Sisa keluaran **Kode 15.5** mengerjakan kedua puluhnya:

```
Shortest paths read out of Pi^(5):
  1 to 2 (weight   1): 1 -> 5 -> 4 -> 3 -> 2
  1 to 3 (weight  -3): 1 -> 5 -> 4 -> 3
  1 to 4 (weight   2): 1 -> 5 -> 4
  1 to 5 (weight  -4): 1 -> 5
  2 to 1 (weight   3): 2 -> 4 -> 1
  2 to 3 (weight  -4): 2 -> 4 -> 3
  2 to 4 (weight   1): 2 -> 4
  2 to 5 (weight  -1): 2 -> 4 -> 1 -> 5
  3 to 1 (weight   7): 3 -> 2 -> 4 -> 1
  3 to 2 (weight   4): 3 -> 2
  3 to 4 (weight   5): 3 -> 2 -> 4
  3 to 5 (weight   3): 3 -> 2 -> 4 -> 1 -> 5
  4 to 1 (weight   2): 4 -> 1
  4 to 2 (weight  -1): 4 -> 3 -> 2
  4 to 3 (weight  -5): 4 -> 3
  4 to 5 (weight  -2): 4 -> 1 -> 5
  5 to 1 (weight   8): 5 -> 4 -> 1
  5 to 2 (weight   5): 5 -> 4 -> 3 -> 2
  5 to 3 (weight   1): 5 -> 4 -> 3
  5 to 4 (weight   6): 5 -> 4
```

Ambillah baris pertamanya lalu ikuti rekursinya dengan tangan. Kita menginginkan rute dari 1 ke 2. $\pi_{12}^{(5)} = 3$, sehingga cetaklah rute dari 1 ke 3, lalu cetak "2". Rute dari 1 ke 3: $\pi_{13}^{(5)} = 4$, sehingga cetaklah rute dari 1 ke 4, lalu "3". Rute dari 1 ke 4: $\pi_{14}^{(5)} = 5$, sehingga cetaklah rute dari 1 ke 5, lalu "4". Rute dari 1 ke 5: $\pi_{15}^{(5)} = 1$, sehingga cetaklah rute dari 1 ke 1 — dan itu kasus dasarnya, yang mencetak "1" — lalu "5". Membuka gulungannya: `1`, `5`, `4`, `3`, `2`. Periksalah bobotnya: $w_{15} + w_{54} + w_{43} + w_{32} = -4 + 6 - 5 + 4 = 1$, yaitu $d_{12}^{(5)}$. Benar.

15.6.12 Waktu jalannya

Waktu jalannya ditentukan oleh gelung **for** rangkap tiga di baris 3–7. Tiap eksekusi baris 7 memakan waktu $O(1)$ — satu penjumlahan, satu pembandingan, satu penugasan, tanpa struktur data — dan baris 7 dieksekusi persis n^3 kali. Jadi algoritmanya berjalan dalam waktu

$$\Theta(n^3).$$

Konstantanya kecil dan pola akses memorinya nyaris ideal: pada putaran k, gelung-j yang terdalam menyusuri baris i dari D dan baris k dari D, keduanya bersebelahan di memori, sementara d_{ik} adalah satu nilai tunggal yang dipegang di dalam register. Inilah sebabnya Floyd–Warshall rutin mengalahkan algoritma yang batas asimtotiknya lebih baik pada graf bersimpul beberapa ribu.

Ruangnya adalah $\Theta(n^2)$ untuk D dan $\Theta(n^2)$ lagi untuk Π. Pembacaan harfiah **Gambar 15.16** akan menyarankan $\Theta(n^3)$, sebab baris 4 mengalokasikan sebuah matriks $D^{(k)}$ yang *baru* pada tiap putaran, dan menyimpan seluruh $n + 1$ buahnya. Subbagian berikutnya memperlihatkan bahwa kita tidak perlu berbuat demikian.

15.6.13 Mengerjakannya di tempat, dan mengapa itu aman

Tengoklah lagi **Kode 15.5**: ia memelihara satu matriks `D` dan menimpanya. Itulah yang dilakukan setiap implementasi sungguhan, dan itu menuntut pembuktian, sebab rekurensinya membaca $d_{ik}^{(k-1)}$ dan $d_{kj}^{(k-1)}$ selagi kita sibuk menimpa matriksnya dengan nilai putaran k. Mungkinkah kita membaca sebuah nilai yang sudah diperbarui, lalu memperoleh jawaban yang keliru?

TEOREMA 15.1 (optimasi di tempat itu aman). Menjalankan gelung rangkap tiganya pada satu matriks tunggal, sambil menimpa d_{ij} di tempat, menghasilkan persis matriks $D^{(n)}$ yang dihasilkan versi dengan $n + 1$ matriks terpisah.

Bukti. Satu-satunya nilai yang dibaca selama putaran k yang mungkin sudah berubah selama putaran k adalah nilai di baris k dan kolom k — yaitu, entri berbentuk d_{kj} atau d_{ik}. Maka cukuplah memperlihatkan bahwa **putaran k tak pernah mengubah baris k atau kolom k**.

Ambillah sebuah entri di baris k, katakanlah d_{kj}. Selama putaran k calon penggantinya adalah $d_{kk} + d_{kj}$. Nah, d_{kk} adalah bobot rute termurah dari k kembali ke k; ia bernilai 0 bila tidak ada siklus menembus k, dan bernilai bobot siklus termurah semacam itu bila ada — yang ≥ 0, sebab grafnya tidak punya siklus berbobot negatif. Jadi $d_{kk} \geq 0$, sehingga $d_{kk} + d_{kj} \geq d_{kj}$, dan calonnya tak pernah mengalahkan petahananya. Baris k tak tersentuh. Argumen untuk kolom k sama persis: calon bagi d_{ik} adalah $d_{ik} + d_{kk} \geq d_{ik}$.

Karena itu setiap nilai yang dibaca selama putaran k masih membawa nilai putaran ke-$(k-1)$-nya pada saat ia dibaca, entah kita mengalokasikan matriks baru atau tidak. Kedua versinya menghitung benda yang sama. ■

Perhatikanlah di mana hipotesisnya masuk: $d_{kk} \geq 0$, yang persis berarti "tidak ada siklus berbobot negatif". Seandainya grafnya memang memuat satu, versi di tempatnya dapat berbeda dari versi buku ajarnya — walaupun dalam hal itu tak satu pun dari keduanya sedang menghitung sesuatu yang bermakna, sebab jawaban sebenarnya adalah $-\infty$.

Penghematannya sungguhan: memori $\Theta(n^2)$ alih-alih $\Theta(n^3)$. Pada $n = 2000$ itu berarti 32 megabita alih-alih 64 gigabita. Itulah beda antara algoritma yang dapat Anda jalankan dan yang tidak.

15.6.14 Mendeteksi siklus berbobot negatif tanpa biaya

Segala sesuatu sejauh ini menganggap tidak ada siklus berbobot negatif. Bagaimana bila kita tidak yakin? Floyd–Warshall menjawab pertanyaannya tanpa biaya tambahan, dan ujinya satu baris.

UJINYA. Sesudah gelung rangkap tiganya, periksalah diagonalnya. Grafnya memuat siklus berbobot negatif **jika dan hanya jika** $d_{ii} < 0$ untuk suatu simpul i.

Mengapa ia berhasil. Entri d_{ii} adalah bobot jalan tertutup termurah dari i kembali ke i. Bila tidak ada siklus negatif, tidak ada jalan tertutup yang berbobot negatif, sehingga $d_{ii} = 0$ (jalan kosongnya) untuk setiap i. Sebaliknya, bila $d_{ii} < 0$ maka ada jalan tertutup menembus i yang berbobot negatif, dan jalan tertutup berbobot negatif pastilah memuat sebuah siklus negatif — uraikanlah ia menjadi siklus sederhana; seandainya semuanya berbobot ≥ 0 totalnya tidak mungkin negatif. Akhirnya, setiap siklus negatif menembus suatu simpul, dan pada saat k mencapai simpul bernomor tertinggi pada siklus itu, seluruh siklusnya tersedia bagi rekurensinya, sehingga entri diagonal yang bersesuaian sudah menjadi negatif. **Kode 15.6** memperagakan semuanya.

Kode 15.6 Pendeteksian siklus berbobot negatif dari diagonalnya, pada tiga graf. Kompilasilah dengan `g++ -std=c++17 -O2 -o negcycle negcycle.cpp`.

```
// Bab 15, Kode: mendeteksi siklus berbobot negatif dari diagonal D.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;
typedef vector<vector<long long>> Matrix;

// Mengembalikan true ketika G memuat siklus berbobot negatif. D ditinggalkan memuat
// hasil gelung rangkap tiganya, yang diagonalnya adalah buktinya.
bool floydWarshallDetect(Matrix& D) {
    int n = (int)D.size();
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (D[i][k] < INF && D[k][j] < INF && D[i][k] + D[k][j] < D[i][j])
                    D[i][j] = D[i][k] + D[k][j];
    for (int i = 0; i < n; ++i) if (D[i][i] < 0) return true;
    return false;
}

void report(const string& title, Matrix W) {
    cout << title << '\n';
    bool bad = floydWarshallDetect(W);
    cout << "   diagonal after the triple loop: ";
    for (size_t i = 0; i < W.size(); ++i) cout << setw(4) << W[i][i];
    cout << "\n   negative-weight cycle? " << (bad ? "YES" : "no") << "\n\n";
}

int main() {
    // (a) Contoh berjalan -- tanpa siklus negatif, sekalipun ada dua sisi negatif.
    report("(a) the running example (edges of weight -4 and -5, but no negative cycle):", {
        {  0,   3,   8, INF,  -4},
        {INF,   0, INF,   1,   7},
        {INF,   4,   0, INF, INF},
        {  2, INF,  -5,   0, INF},
        {INF, INF, INF,   6,   0}
    });
```

Kode 15.6 (Lanjutan) *Pendeteksian siklus berbobot negatif dari diagonalnya, pada tiga graf. Kompilasilah dengan g++ -std=c++17 -O2 -o negcycle negcycle.cpp.*

```
    // (b) Graf bersimpul tiga yang siklusnya 1 -> 2 -> 3 -> 1 berbobot total -1.
    report("(b) a graph whose cycle 1 -> 2 -> 3 -> 1 has weight 2 + 3 + (-6) = -1:", {
        {  0,   2, INF},
        {INF,   0,   3},
        { -6, INF,   0}
    });

    // (c) Graf bersimpul dua dengan siklus-dua yang negatif.
    report("(c) a graph whose cycle 1 -> 2 -> 1 has weight 1 + (-4) = -3:", {
        {  0,   1},
        { -4,   0}
    });
    return 0;
}
```

Keluarannya:

```
(a) the running example (edges of weight -4 and -5, but no negative cycle):
   diagonal after the triple loop:    0   0   0   0   0
   negative-weight cycle? no

(b) a graph whose cycle 1 -> 2 -> 3 -> 1 has weight 2 + 3 + (-6) = -1:
   diagonal after the triple loop:   -1  -1  -2
   negative-weight cycle? YES

(c) a graph whose cycle 1 -> 2 -> 1 has weight 1 + (-4) = -3:
   diagonal after the triple loop:   -3  -6
   negative-weight cycle? YES
```

Graf (a) adalah contoh berjalannya: dua dari sembilan sisinya negatif, namun diagonalnya nol semua, dan itulah bukti terperiksa mesin bahwa setiap hasil di §15.6.8 bermakna. Pada (b) siklus $1 \to 2 \to 3 \to 1$ berbobot $2 + 3 - 6 = -1$, dan ketiga entri diagonalnya menjadi negatif. Pada (c) siklus-duanya berbobot -3, dan diagonalnya melaporkan -3 dan -6 — perhatikanlah bahwa bilangan di luar diagonal pada graf semacam itu tidak bermakna, sebab jawaban sebenarnya adalah $-\infty$; hanya *uji tandanya* yang dapat dipercaya.

15.6.15 Keempat metodenya sepakat

Sebelum beranjak, ada gunanya memeriksa seluruh §15.3 sampai §15.6 terhadap dirinya sendiri. **Kode 15.7** menjalankan $|V| \times$ Bellman–Ford, SLOW-APSP, pemangkatan berulang, Floyd–Warshall, dan (sambil menengok ke depan) Johnson pada contoh berjalannya lalu membandingkan kelima matriksnya.

Kode 15.7 Lima algoritma, satu graf, satu jawaban. Kompilasilah dengan `g++ -std=c++17 -O2 -o crosscheck crosscheck.cpp`.

```
// Bab 15, Kode: lima jalan menuju jawaban yang sama. Menjalankan |V| x Bellman-Ford,
// SLOW-APSP, pemangkatan berulang, Floyd-Warshall, dan Johnson pada contoh berjalan
// lalu memeriksa bahwa setiap pasang metode menghasilkan matriks yang persis sama.
#include <iostream>
#include <iomanip>
#include <vector>
#include <queue>
#include <string>
using namespace std;

const long long INF = 1000000000LL;
typedef vector<vector<long long>> Matrix;
struct Edge { int u, v; long long w; };

static const int n = 5;
static Matrix W = {
    {  0,   3,   8, INF,  -4},
    {INF,   0, INF,   1,   7},
    {INF,   4,   0, INF, INF},
    {  2, INF,  -5,   0, INF},
    {INF, INF, INF,   6,   0}
};
static vector<Edge> E = {
    {0,1,3}, {0,2,8}, {0,4,-4}, {1,3,1}, {1,4,7},
    {2,1,4}, {3,0,2}, {3,2,-5}, {4,3,6}
};

Matrix extendShortestPaths(const Matrix& A, const Matrix& B) {
    Matrix C(n, vector<long long>(n, INF));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                if (A[i][k] < INF && B[k][j] < INF)
                    C[i][j] = min(C[i][j], A[i][k] + B[k][j]);
    return C;
}

Matrix methodSlow() {
    Matrix L = W;
    for (int m = 2; m <= n - 1; ++m) L = extendShortestPaths(L, W);
    return L;
}
```

Kode 15.7 (Lanjutan) *Lima algoritma, satu graf, satu jawaban. Kompilasilah dengan g++ -std=c++17 -O2 -o crosscheck crosscheck.cpp.*

```
Matrix methodSquaring() {
    Matrix L = W;
    for (int m = 1; m < n - 1; m *= 2) L = extendShortestPaths(L, L);
    return L;
}

Matrix methodFloydWarshall() {
    Matrix D = W;
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (D[i][k] < INF && D[k][j] < INF)
                    D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
    return D;
}

Matrix methodRepeatedBellmanFord() {
    Matrix D(n, vector<long long>(n, INF));
    for (int s = 0; s < n; ++s) {
        vector<long long> d(n, INF);
        d[s] = 0;
        for (int pass = 1; pass <= n - 1; ++pass)
            for (const Edge& e : E)
                if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) d[e.v] = d[e.u] + e.w;
        D[s] = d;
    }
    return D;
}

Matrix methodJohnson() {
    int s = n;
    vector<Edge> Ep = E;
    for (int v = 0; v < n; ++v) Ep.push_back({s, v, 0});
    vector<long long> h(n + 1, INF);
    h[s] = 0;
    for (int pass = 1; pass <= n; ++pass)
        for (const Edge& e : Ep)
            if (h[e.u] < INF && h[e.u] + e.w < h[e.v]) h[e.v] = h[e.u] + e.w;

    vector<vector<pair<int,long long>>> adj(n);
    for (const Edge& e : E) adj[e.u].push_back({e.v, e.w + h[e.u] - h[e.v]});
```

Kode 15.7 (Lanjutan) *Lima algoritma, satu graf, satu jawaban. Kompilasilah dengan g++ -std=c++17 -O2 -o crosscheck crosscheck.cpp.*

```
    Matrix D(n, vector<long long>(n, INF));
    for (int u = 0; u < n; ++u) {
        vector<long long> d(n, INF);
        d[u] = 0;
        priority_queue<pair<long long,int>, vector<pair<long long,int>>, greater<>> Q;
        Q.push({0, u});
        while (!Q.empty()) {
            auto [du, x] = Q.top(); Q.pop();
            if (du > d[x]) continue;
            for (auto [y, wt] : adj[x])
                if (d[x] + wt < d[y]) { d[y] = d[x] + wt; Q.push({d[y], y}); }
        }
        for (int v = 0; v < n; ++v) if (d[v] < INF) D[u][v] = d[v] + h[v] - h[u];
    }
    return D;
}

void printMatrix(const string& name, const Matrix& M) {
    cout << name << ":\n";
    for (int i = 0; i < n; ++i) {
        cout << "   ";
        for (int j = 0; j < n; ++j) {
            if (M[i][j] >= INF) cout << setw(6) << "inf";
            else                cout << setw(6) << M[i][j];
        }
        cout << '\n';
    }
}

int main() {
    vector<pair<string, Matrix>> results = {
        {"|V| x Bellman-Ford", methodRepeatedBellmanFord()},
        {"SLOW-APSP",          methodSlow()},
        {"repeated squaring",  methodSquaring()},
        {"Floyd-Warshall",     methodFloydWarshall()},
        {"Johnson",            methodJohnson()}
    };
    printMatrix("the agreed answer", results[0].second);
    cout << '\n';
    bool allAgree = true;
    for (size_t t = 1; t < results.size(); ++t) {
        bool same = (results[t].second == results[0].second);
        allAgree = allAgree && same;
        cout << setw(20) << left << results[t].first
             << " agrees with " << results[0].first << ": "
             << (same ? "YES" : "NO") << '\n';
```

Kode 15.7 (Lanjutan) *Lima algoritma, satu graf, satu jawaban. Kompilasilah dengan g++ -std=c++17 -O2 -o crosscheck crosscheck.cpp.*

```
    }
    cout << "\nall five methods agree: " << (allAgree ? "YES" : "NO") << '\n';
    return 0;
}
```

Keluarannya:

```
the agreed answer:
        0     1    -3     2    -4
        3     0    -4     1    -1
        7     4     0     5     3
        2    -1    -5     0    -2
        8     5     1     6     0

SLOW-APSP            agrees with |V| x Bellman-Ford: YES
repeated squaring    agrees with |V| x Bellman-Ford: YES
Floyd-Warshall       agrees with |V| x Bellman-Ford: YES
Johnson              agrees with |V| x Bellman-Ford: YES

all five methods agree: YES
```

Itulah jenis pemeriksaan yang patut dibangun ke dalam pekerjaan Anda sendiri. Lima implementasi mandiri atas spesifikasi yang sama, yang saling divalidasi silang, adalah bukti kebenaran yang jauh lebih kuat daripada salah satunya diuji sendirian.

15.7 Penutupan Transitif Sebuah Graf Berarah

15.7.1 Pertanyaannya, dan jawaban yang malas

Kadang-kadang kita tidak peduli seberapa jauh jarak dua simpul, melainkan hanya **apakah yang satu dapat dicapai dari yang lain sama sekali**. Dengan persis: diberikan sebuah graf berarah $G = (V, E)$ dengan $V = \{1,2,\dots,n\}$, tentukanlah apakah G memuat sebuah lintasan dari i ke j, untuk semua pasangan simpul $i, j \in V$.

DEFINISI. Penutupan transitif G adalah graf $G^* = (V, E^*)$ dengan

$$E^* = \{(i,j)\text{:ada lintasan dari simpul } i \text{ ke simpul } j \text{ di } G\}.$$

Kata *transitif* berasal dari sifat relasi yang bersesuaian: sebuah relasi R disebut transitif bila aRb dan bRc bersama-sama memaksa aRc. Penutupan transitif adalah relasi transitif terkecil yang memuat relasi aslinya — dalam bahasa graf, tambahkanlah sebuah sisi pintas di mana pun ada lintasan.

Jawaban pertama memakan waktu sepuluh detik untuk diimplementasikan: **berilah bobot 1 pada tiap sisi E lalu jalankan algoritma Floyd–Warshall.** Bila ada lintasan dari simpul i ke simpul j kita memperoleh $d_{ij} < n$, sebab lintasan sederhana bersisi paling banyak $n - 1$ dan kini tiap sisi berongkos 1. Selebihnya $d_{ij} = \infty$. Ongkosnya $\Theta(n^3)$, dan kodenya adalah kode yang sudah Anda punya.

15.7.2 Jawaban yang lebih baik: tukarlah operatornya

Metode kedua dapat menghemat waktu dan ruang dalam praktik. Ia **mengganti operasi logika $\vee$ (OR logika) dan $\wedge$ (AND logika) untuk operasi aritmetika** min **dan** $+$ di dalam algoritma Floyd–Warshall.

Inilah semiring $(\min, +)$ dari §15.4.9 yang diganti dengan semiring boolean $(\vee,\wedge)$, yang $\bar{0} =$ FALSE dan $\bar{1} =$ TRUE. Penggantiannya terpaksa terjadi begitu Anda melihat apa *arti* operatornya. Dalam versi jaraknya, min menjawab "rute mana di antara kedua ini yang lebih baik?" dan $+$ menjawab "berapa biaya melakukan ini lalu itu?". Dalam versi keterjangkauannya, $\vee$ menjawab "adakah salah satu dari kedua rute ini yang tersedia?" dan $\wedge$ menjawab "dapatkah saya melakukan ini **dan lalu** itu?". Struktur yang sama, aritmetika yang berbeda.

Definisikanlah, untuk $i, j, k = 1,2, \ldots, n$, nilai $t_{ij}^{(k)}$ sebagai 1 bila ada lintasan di dalam graf G dari simpul i ke simpul j yang seluruh simpul antaranya berada di himpunan $\{1,2, \ldots, k\}$, dan 0 selebihnya. Kita menyusun penutupan transitif $G^* = (V, E^*)$ dengan memasukkan sisi (i, j) ke dalam E^* jika dan hanya jika $t_{ij}^{(n)} = 1$. **Gambar 15.18** memberikan definisi rekursifnya.

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{jika } i \neq j \text{ dan } (i,j) \notin E \\ 1 & \text{jika } i = j \text{ atau } (i,j) \in E \end{cases}$$

dan untuk $k \geq 1$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right).$$

Gambar 15.18 Kasus dasar $t_{ij}^{(0)}$ dan rekurensi bagi $t_{ij}^{(k)}$, yang merupakan rekurensi Floyd–Warshall dengan $\vee$ menggantikan min dan $\wedge$ menggantikan $+$.

Ditulis lengkap:

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{jika } i \neq j \text{ dan } (i,j) \notin E, \\ 1 & \text{jika } i = j \text{ atau } (i,j) \in E, \end{cases}$$

(15.11)

dan untuk $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}). \tag{15.12}$$

Bacalah Persamaan (15.12) dengan suara keras: *j terjangkau dari i dengan hanya memakai k simpul pertama di tengahnya bila entah ia sudah terjangkau dengan hanya memakai $k-1$ yang pertama, atau Anda dapat pergi dari i ke k dan dari k ke j, keduanya hanya memakai $k-1$ yang pertama.* Ia persis analisis kasus §15.6.2 dengan "termurah" diganti "mungkin".

Perhatikanlah bahwa kasus dasar Persamaan (15.11) memberikan $t_{ii}^{(0)} = 1$ untuk setiap i, sebab sebuah simpul mencapai dirinya sendiri lewat lintasan kosong. Itulah padanan boolean bagi $w_{ii} = 0$.

15.7.3 Algoritmanya dan contohnya

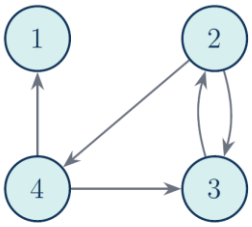
Gambar 15.19 memberikan prosedurnya.

TRANSITIVE-CLOSURE(G)

```
1  n = |G.V|
2  misalkan T^(0) = (t_ij^(0)) sebuah matriks n × n baru
3  untuk i = 1 sampai n
4      untuk j = 1 sampai n
5          jika i == j atau (i, j) ∈ G.E
6              t_ij^(0) = 1
7          selain itu t_ij^(0) = 0
8  untuk k = 1 sampai n
9      misalkan T^(k) = (t_ij^(k)) sebuah matriks n × n baru
10     untuk i = 1 sampai n
11         untuk j = 1 sampai n
12             t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
13 kembalikan T^(n)
```

Gambar 15.19 TRANSITIVE-CLOSURE(G). Baris 3–7 membangun $T^{(0)}$ dari himpunan sisinya; baris 8–12 adalah gelung rangkap tiga Floyd–Warshall dengan operator boolean. Ia berjalan dalam waktu $\Theta(n^3)$.

Gambar 15.20 memperlihatkan sebuah graf bersimpul empat beserta kelima matriksnya.



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Gambar 15.20 Sebuah graf berarah dengan sisi $2 \to 3$, $2 \to 4$, $3 \to 2$, $4 \to 1$, $4 \to 3$, beserta matriks $T^{(k)}$ yang dihitung algoritma penutupan transitifnya (Gambar 25.5 pada edisi ketiga buku Cormen dkk.).

Kode 15.8 menghasilkannya ulang.

Kode 15.8 Algoritma penutupan transitif selengkapnya. Kompilasilah dengan `g++ -std=c++17 -O2 -o transitive_closure transitive_closure.cpp`.

```
// Bab 15, Kode: TRANSITIVE-CLOSURE pada graf contoh bersimpul empat.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

typedef vector<vector<int>> BoolMatrix;   // entri 0/1, sehingga | dan & dapat dipakai
langsung

void printT(const string& name, const BoolMatrix& T) {
    int n = (int)T.size();
    cout << name << " =\n";
    for (int i = 0; i < n; ++i) {
        cout << "   ";
        for (int j = 0; j < n; ++j) cout << setw(3) << T[i][j];
        cout << '\n';
    }
    cout << '\n';
}

int main() {
    const int n = 4;
    // Graf contoh: sisi 2->3, 2->4, 3->2, 4->1, 4->3 (simpul bernomor 1..4).
    vector<pair<int,int>> edges = {{2,3},{2,4},{3,2},{4,1},{4,3}};

    // Baris 2-7: bangunlah T^(0).
    BoolMatrix T(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i) T[i][i] = 1;               // i == j
    for (auto& e : edges) T[e.first - 1][e.second - 1] = 1; // (i,j) di dalam G.E
    printT("T^(0)", T);

    // Baris 8-12: gelung rangkap tiga yang sama seperti Floyd-Warshall, dengan OR untuk
min dan AND untuk +.
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                T[i][j] = T[i][j] | (T[i][k] & T[k][j]);
        printT("T^(" + to_string(k + 1) + ")", T);
    }

    cout << "Edges of the transitive closure G* (excluding self-loops):\n   ";
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (i != j && T[i][j]) cout << "(" << i + 1 << "," << j + 1 << ") ";
    cout << '\n';
    return 0;
}
```

Keluarannya:

```
T^(0) =
     1  0  0  0
     0  1  1  1
     0  1  1  0
     1  0  1  1

T^(1) =
     1  0  0  0
     0  1  1  1
     0  1  1  0
     1  0  1  1

T^(2) =
     1  0  0  0
     0  1  1  1
     0  1  1  1
     1  0  1  1

T^(3) =
     1  0  0  0
     0  1  1  1
     0  1  1  1
     1  1  1  1

T^(4) =
     1  0  0  0
     1  1  1  1
     1  1  1  1
     1  1  1  1

Edges of the transitive closure G* (excluding self-loops):
   (2,1) (2,3) (2,4) (3,1) (3,2) (3,4) (4,1) (4,2) (4,3)
```

Kelima matriksnya cocok dengan **Gambar 15.20**. Susurilah putaran-putaran yang menarik. Putaran $k = 1$ tidak mengubah apa pun, sebab simpul 1 **tidak punya sisi keluar** — baris 1 dari $T^{(0)}$ adalah (1,0,0,0) — sehingga $t_{1j}^{(0)} = 0$ untuk setiap $j \neq 1$ dan suku $t_{i1} \wedge t_{1j}$ selalu 0. Putaran $k = 2$ menyetel $t_{34}^{(2)} = 1$: simpul 3 mencapai simpul 2, dan simpul 2 mencapai simpul 4, sehingga $3 \to 2 \to 4$ ada. Putaran $k = 3$ menyetel $t_{42}^{(3)} = 1$ lewat $4 \to 3 \to 2$. Putaran $k = 4$ adalah yang besar: ia mengisi seluruh baris 2 dan 3, sebab keduanya kini mencapai simpul 4, dan simpul 4 mencapai simpul 1. Baris 1 tetap (1,0,0,0) selamanya, dan itu benar: simpul 1 adalah muara dan tidak mencapai apa pun kecuali dirinya sendiri.

15.7.4 Mengapa boolean jauh lebih cepat dalam praktik

Kedua metodenya $\Theta(n^3)$, jadi mengapa versi booleannya menghemat waktu dan ruang? Dua alasan.

Ruang. Sebuah jarak memerlukan satu kata 32- atau 64-bita. Sebuah boolean memerlukan satu **bita tunggal**. Memampatkan matriksnya ke dalam kata mesin memangkas memorinya sebesar faktor 32 atau 64, dan itulah yang menentukan apakah sebuah graf besar muat di dalam tembolok.

Waktu. Dan begitu bitnya termampatkan, gelung-j terdalam di baris 12 menjadi sebuah OR bitwise atas dua baris utuh — dan prosesor 64-bita mengerjakan 64 putaran gelung itu dalam satu instruksi. Di dalam C++ ini dua baris dengan `std::bitset`. **Kode 15.9** mengukurnya.

Kode 15.9 Gelung rangkap tiga booleannya, satu bita per entri dan satu bit per entri, diukur. Kompilasilah dengan `g++ -std=c++17 -O2 -o ex_bitset_tc ex_bitset_tc.cpp`.

```
// Bab 15, penyelesaian soal: penutupan transitif pada 64 kolom tiap instruksi.
// Gelung-j terdalam TRANSITIVE-CLOSURE meng-OR-kan baris utuh, sehingga satu kata mesin
dapat
// mengerjakan 64 putarannya sekaligus. Kerja Theta(n^3) yang sama, konstanta yang jauh
lebih kecil.
#include <iostream>
#include <iomanip>
#include <vector>
#include <bitset>
#include <random>
#include <chrono>
using namespace std;
using namespace std::chrono;

const int N = 1024;                     // ditetapkan pada saat kompilasi, sebagaimana
dituntut std::bitset

int main() {
    mt19937 rng(20250915);
    uniform_int_distribution<int> pick(0, N - 1);

    vector<vector<char>>   plain(N, vector<char>(N, 0));
    vector<bitset<N>>      fast(N);

    for (int i = 0; i < N; ++i) { plain[i][i] = 1; fast[i][i] = 1; }
    for (int e = 0; e < 4 * N; ++e) {   // sebuah digraf acak yang jarang
        int u = pick(rng), v = pick(rng);
        plain[u][v] = 1; fast[u][v] = 1;
    }

    auto t0 = steady_clock::now();
    for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
            if (plain[i][k])
                for (int j = 0; j < N; ++j)
                    plain[i][j] = plain[i][j] | plain[k][j];
    auto t1 = steady_clock::now();

    for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
            if (fast[i][k]) fast[i] |= fast[k];        // 1024 kolom dalam 16 kata
    auto t2 = steady_clock::now();
```

Kode 15.9 (Lanjutan) *Gelung rangkap tiga booleannya, satu bita per entri dan satu bit per entri, diukur. Kompilasilah dengan g++ -std=c++17 -O2 -o ex_bitset_tc ex_bitset_tc.cpp.*

```
    bool same = true;
    long long reachable = 0;
    for (int i = 0; i < N && same; ++i)
        for (int j = 0; j < N; ++j) {
            if ((bool)plain[i][j] != fast[i][j]) { same = false; break; }
            reachable += plain[i][j];
        }

    cout << "n = " << N << ", edges = " << 4 * N << '\n';
    cout << "byte-per-entry version: " << fixed << setprecision(1)
         << duration<double, milli>(t1 - t0).count() << " ms\n";
    cout << "bitset version:         "
         << duration<double, milli>(t2 - t1).count() << " ms\n";
    cout << "the two closures agree: " << (same ? "yes" : "NO") << '\n';
    cout << "reachable pairs found:  " << reachable << " out of " << (long long)N * N <<
'\n';
    return 0;
}
```

Keluarannya pada mesin yang dipakai menyiapkan bab ini:

```
n = 1024, edges = 4096
byte-per-entry version: 231.7 ms
bitset version:         3.2 ms
the two closures agree: yes
reachable pairs found:  1015046 out of 1048576
```

Faktor lebih daripada **tujuh puluh** untuk perubahan dua baris, dengan keluaran yang sama persis. Ini patut diresapi sebagai pelajaran umum: analisis asimtotik memberi tahu Anda algoritma mana yang harus dipilih, dan ia sama sekali tidak mengatakan apa-apa tentang faktor konstantanya, yang di sini bernilai hampir dua orde besaran. Keduanya penting; tak satu pun menggantikan yang lain.

15.8 Algoritma Johnson untuk Graf Jarang

15.8.1 Yang diinginkan

Floyd–Warshall berongkos $\Theta(V^3)$ entah grafnya punya sejuta sisi atau sembilan. Pada graf **jarang** — yaitu graf yang E-nya jauh lebih kecil daripada V^2, dan itu memerikan hampir setiap graf yang muncul dari jaringan jalan, perayapan web, jejaring sosial, atau sebuah rangkaian — itu boros. Sasarannya adalah sebuah algoritma yang mencari lintasan terpendek antara semua pasangan dalam waktu

$$O(V^2 \lg V + VE)$$

yang untuk graf jarang **secara asimtotik lebih cepat daripada pemangkatan matriks berulang maupun algoritma Floyd–Warshall**. Ia mengembalikan matriks bobot lintasan terpendek untuk semua pasangan simpul, atau melaporkan bahwa graf masukannya memuat siklus berbobot negatif. Ia memakai algoritma Dijkstra sekaligus algoritma Bellman–Ford sebagai subrutin. Dan ia memakai teknik **pembobotan ulang** (*reweighting*).

Dari mana batas itu berasal? Ia persis biaya V kali jalannya Dijkstra dengan *Fibonacci heap*, dari **Tabel 15.1**. Jadi algoritma Johnson pada dasarnya adalah jawaban atas satu pertanyaan: *bagaimana kita membuat $V \times$ Dijkstra menjadi sah pada graf bersisi negatif?*

15.8.2 Apa yang harus dicapai sebuah pembobotan ulang

Gambar 15.21 menggelar rencananya beserta dua kewajiban yang menyertainya.

- Bila setiap bobot sisi tidak negatif, menjalankan Dijkstra sekali dari tiap simpul menyelesaikan semua pasangan; dengan *Fibonacci heap* waktunya $O(V^2 \lg V + VE)$.
- Bila G memiliki sisi berbobot negatif tetapi tanpa siklus berbobot negatif, hitunglah bobot baru $\widehat{w}$ yang tidak negatif agar metode yang sama dapat dipakai.
- Bobot barunya harus memenuhi dua sifat:

  1. Untuk semua $u, v \in V$, sebuah lintasan p adalah lintasan terpendek dari u ke v menurut w **jika dan hanya jika** ia lintasan terpendek dari u ke v menurut $\widehat{w}$.
  2. Untuk setiap sisi (u, v), bobot baru $\widehat{w}(u, v)$ tidak negatif.

Gambar 15.21 Bila sebuah fungsi bobot baru $\widehat{w}$ melestarikan lintasan terpendek dan tak negatif di mana-mana, maka $V \times$ Dijkstra atas $\widehat{w}$ menjawab pertanyaan aslinya.

Himpunan bobot sisi yang baru, $\widehat{w}$, harus memenuhi dua sifat:

1. **Lintasan terpendeknya lestari.** Untuk semua pasangan simpul $u, v \in V$, sebuah lintasan p adalah lintasan terpendek dari u ke v memakai fungsi bobot w **jika dan hanya jika** p juga lintasan terpendek dari u ke v memakai fungsi bobot $\widehat{w}$.
2. **Ketaknegatifan.** Untuk semua sisi (u, v), bobot barunya $\widehat{w}(u, v)$ tak negatif.

Sifat 2-lah yang membuat Dijkstra menjadi sah. Sifat 1-lah yang membuat jawabannya *bermakna*: seandainya pembobotan ulangnya mengubah rute mana yang terbaik, kita akan sedang menyelesaikan persoalan yang lain lalu melaporkan jawaban atas pertanyaan yang keliru. Perhatikanlah bahwa sifat 1 berbicara tentang lestarinya *lintasan*, bukan *bobot* — bobotnya jelas berubah, dan §15.8.5 memperlihatkan cara mengubahnya kembali.

PEMBOBOTAN ULANG YANG GAGAL. Gagasan yang kentara adalah mencari bobot paling negatif $-c$ lalu menambahkan c pada setiap sisi. Sifat 2 terpenuhi. Sifat 1 gagal parah, sebab lintasan bersisi banyak memperoleh tambahan c sekali per sisi sementara lintasan bersisi sedikit hanya memperoleh c beberapa kali, sehingga rute panjang terhukum. Ambillah $1 \to 2$ berbobot -2, $2 \to 3$ berbobot -2, dan $1 \to 3$ berbobot -3: rute termurah dari 1 ke 3 adalah yang dua sisi, seharga -4 lawan -3 untuk sisi langsungnya. Bobot paling negatifnya -3, jadi $c = 3$; menambahkannya memberikan bobot 1, 1, dan 0, dan kini rute dua sisinya berongkos 2 sementara sisi langsungnya berongkos 0. Lintasan terpendeknya telah berubah. Pembobotan ulang yang benar harus menambahkan sejumlah yang bergantung pada **kedua ujungnya**, bukan pada cacah sisinya.

15.8.3 Pembobotan ulang yang berhasil

Inilah gagasannya, dan inilah jantung algoritmanya. Pilihlah sembarang fungsi $h: V \to \mathbb{R}$ yang memberikan sebuah bilangan kepada setiap **simpul**, lalu definisikan

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \tag{15.13}$$

untuk setiap sisi $(u, v) \in E$. Pikirkanlah $h(v)$ sebagai *ketinggian* yang diberikan kepada simpul v: biaya baru sebuah sisi adalah biaya lamanya ditambah turunnya ketinggian sepanjang sisi itu.

LEMA 15.2 (pembobotan ulang melestarikan lintasan terpendek). Misalkan $h: V \to \mathbb{R}$ sembarang fungsi dan misalkan $\widehat{w}$ didefinisikan oleh Persamaan (15.13). Maka untuk sembarang lintasan $p = \langle v_0, v_1, \ldots, v_k \rangle$,

$$\widehat{w}(p) = w(p) + h(v_0) - h(v_k).$$

Akibatnya p adalah lintasan terpendek dari v_0 ke v_k di bawah w jika dan hanya jika ia lintasan terpendek di bawah $\widehat{w}$; dan G punya siklus berbobot negatif di bawah w jika dan hanya jika ia punya satu di bawah $\widehat{w}$.

Bukti. Jumlahkanlah Persamaan (15.13) di sepanjang lintasannya:

$$\begin{aligned}\widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) = \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\ &= \sum_{i=1}^{k} w(v_{i-1}, v_i) + \sum_{i=1}^{k} (h(v_{i-1}) - h(v_i)) \\ &= w(p) + h(v_0) - h(v_k).\end{aligned}$$

Jumlah yang kedua **teleskopis**: menuliskannya lengkap memberikan $(h(v_0) - h(v_1)) + (h(v_1) - h(v_2)) + \cdots + (h(v_{k-1}) - h(v_k))$, dan setiap suku di bagian dalamnya muncul sekali secara positif dan sekali secara negatif, sehingga semuanya saling meniadakan dan hanya $h(v_0) - h(v_k)$ yang bertahan.

Sekarang ambillah sembarang dua lintasan p dan q dari u ke v. Keduanya memperoleh konstanta yang *sama*, $h(u) - h(v)$, yang hanya bergantung pada kedua ujungnya dan bukan pada rutenya. Jadi $\widehat{w}(p) - \widehat{w}(q) = w(p) - w(q)$: *urutan* rute antara sepasang ujung yang tetap tidak tersentuh. Yang satu lebih pendek daripada yang lain di bawah $\widehat{w}$ persis ketika ia lebih pendek di bawah w.

Untuk klaim siklusnya, ambillah p sebagai sebuah siklus, sehingga $v_0 = v_k$. Maka $h(v_0) - h(v_k) = 0$ dan $\widehat{w}(p) = w(p)$ persis: siklus mempertahankan bobotnya di bawah pembobotan ulang mana pun, sehingga siklus negatif tidak dapat diciptakan ataupun dimusnahkan. ■

Ini membuktikan sifat 1 bagi *setiap* pilihan h. Seluruh seninya terletak pada memilih h yang sekaligus memenuhi sifat 2.

15.8.4 Memilih h: simpul bantunya

Sifat 2 menuntut $\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$ untuk setiap sisi, yang disusun ulang menjadi

$$h(v) \leq h(u) + w(u, v) \quad \text{untuk setiap sisi } (u, v) \in E.$$

Tetapi itu syarat yang sudah akrab! Ia persis pernyataan bahwa tidak ada sisi yang dapat direlaksasi — bentuk sisi **ketaksamaan segitiga**, yang dibuktikan di Bab 13 sebagai Akibat 13.3: $\delta(s, v) \leq \delta(s, u) + w(u, v)$ untuk setiap sisi (u, v) dan setiap sumber s. Jadi bila kita menyetel $h(v) = \delta(s, v)$ untuk suatu sumber s, sifat 2 datang dengan cuma-cuma.

Ada satu kesulitan: tidak ada satu pun simpul G yang harus mencapai semua simpul lainnya, dan $\delta(s, v) = \infty$ untuk v yang tak terjangkau membuat Persamaan (15.13) tak bermakna. Penambalnya adalah baris 1 JOHNSON di §15.8.5: **ciptakanlah sebuah simpul baru.**

Susunlah $G' $ dengan

$$G'.V = G.V \cup \{s\}, \qquad G'.E = G.E \cup \{(s, v) : v \in G.V\}, \qquad w(s, v) = 0 \text{ untuk semua } v \in G.V,$$

dengan s adalah simpul yang benar-benar baru dan bukan anggota $G.V$. Kini s mencapai segala sesuatu dalam satu lompatan, sehingga setiap $\delta(s,v)$ berhingga (kecuali bila ada siklus negatif, yang kita deteksi). Dan s **tidak punya sisi masuk**, sehingga tidak ada lintasan terpendek antara dua simpul asli yang dapat melewatinya: untuk meninggalkan s Anda harus telah memasukinya, dan tidak ada yang memasukinya. Maka menambahkan s tidak mengubah satu pun jarak di antara simpul aslinya. Setelah itu setellah

$$h(v) = \delta(s,v) \quad \text{untuk semua } v \in G'.V,$$

yang dihitung dengan sekali jalannya Bellman–Ford dari s — yang sekaligus persis merupakan uji bagi siklus berbobot negatif.

MENGAPA h INI MEMBUAT SETIAP BOBOT MENJADI TAK NEGATIF. Untuk sembarang sisi (u,v) dari G, ketaksamaan segitiga pada G' memberikan $\delta(s,v) \le \delta(s,u) + w(u,v)$, yaitu $h(v) \le h(u) + w(u,v)$, yaitu $0 \le w(u,v) + h(u) - h(v) = \widehat{w}(u,v)$. Itulah seluruh argumennya, dan panjangnya satu baris begitu Anda melihat bahwa tuntutannya *adalah* ketaksamaan segitiga.

15.8.5 Algoritmanya

Gambar 15.22 memberikan JOHNSON(G,w).

```
JOHNSON(G, w)
1  bentuk G', dengan G'.V = G.V ∪ {s},
       G'.E = G.E ∪ {(s, v) : v ∈ G.V}, dan w(s, v) = 0 untuk semua v ∈ G.V
2  jika BELLMAN-FORD(G', w, s) == FALSE
3      cetak "graf masukan memuat siklus berbobot negatif"
4  selain itu untuk tiap simpul v ∈ G'.V
5          tetapkan h(v) sebagai nilai δ(s, v) yang dihitung BELLMAN-FORD
6      untuk tiap sisi (u, v) ∈ G'.E
7          ŵ(u, v) = w(u, v) + h(u) − h(v)
8      misalkan D = (d_uv) sebuah matriks n × n baru
9      untuk tiap simpul u ∈ G.V
10         jalankan DIJKSTRA(G, ŵ, u) untuk menghitung δ̂(u, v) bagi semua v ∈ G.V
11         untuk tiap simpul v ∈ G.V
12             d_uv = δ̂(u, v) + h(v) − h(u)
13     kembalikan D
```

Gambar 15.22 JOHNSON(G,w). Baris 1 menambahkan simpul bantunya, baris 2 menjalankan Bellman–Ford sekali, baris 6–7 membobot ulang, baris 9–10 menjalankan Dijkstra $|V|$ kali, dan baris 12 mengubah tiap jawabannya kembali.

Baris 12 adalah langkah yang membatalkan pembobotan ulangnya. Dijkstra atas $\widehat{w}$ mengembalikan $\hat{\delta}(u,v)$, yaitu bobot lintasan terpendek di bawah bobot barunya. Menurut Lema 15.2 yang diterapkan pada sebuah lintasan terpendek p dari u ke v,

$$\hat{\delta}(u,v) = \delta(u,v) + h(u) - h(v), \qquad \text{sehingga} \qquad d_{uv} = \delta(u,v) = \hat{\delta}(u,v) + h(v) - h(u).$$

(15.14)

Perhatikanlah tanda yang terbalik: untuk memulihkan yang asli kita mengurangkan apa yang tadi kita tambahkan.

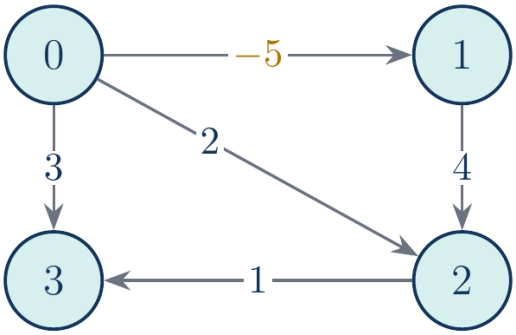
Waktu jalannya. Baris 1 membangun G' dalam waktu $O(V)$. Baris 2 adalah sekali jalannya Bellman–Ford pada G', yang bersimpul $V+1$ dan bersisi $E+V$, berongkos $O(VE)$. Baris 4–7 menyentuh setiap sisi sekali, $O(E)$. Baris 9–11 menjalankan Dijkstra V kali; dengan *Fibonacci heap* tiap kali jalan berongkos $O(V\lg V+E)$, sehingga gelungnya berongkos $O(V^2\lg V+VE)$, dan suku itulah yang mendominasi. Totalnya:

$$O(V^2\lg V+VE).$$

Dengan tumpukan-min biner sebagai gantinya, tiap Dijkstra berongkos $O(E\lg V)$ dan totalnya $O(VE\lg V)$ — biasanya pilihan yang lebih baik dalam praktik, dan itulah yang dipakai **Kode 15.10**.

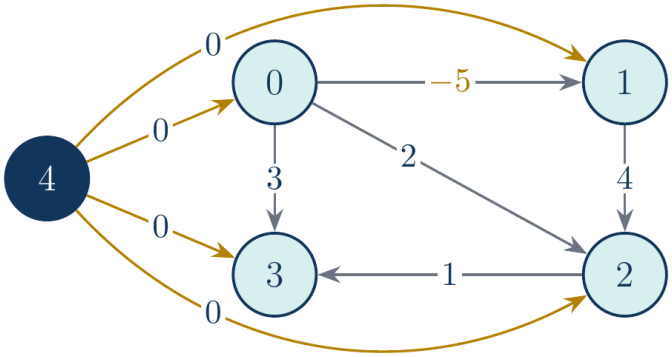
15.8.6 Sebuah pembobotan ulang kecil yang dikerjakan tuntas

Sebelum graf bersimpul lima itu, sebuah graf bersimpul empat memberikan ilustrasi yang paling gamblang yang mungkin ada. Grafnya, **Gambar 15.23**, bersimpul bernomor 0,1,2,3 dan bersisi 0 → 1 berbobot −5, 0 → 2 berbobot 2, 0 → 3 berbobot 3, 1 → 2 berbobot 4, dan 2 → 3 berbobot 1.



Gambar 15.23 Sebuah graf bersimpul empat dengan satu sisi negatif, 0 → 1 berbobot −5. Dijkstra tidak diperbolehkan di sini.

Langkah satu: tambahkanlah sebuah sumber s dan tambahkanlah sisi dari s ke semua simpul graf aslinya. Di sini s adalah simpul baru bernomor 4. Hasilnya adalah **Gambar 15.24**, dengan empat sisi berbobot nol yang baru digambar dengan warna yang berbeda.



Gambar 15.24 Graf yang sama dengan simpul bantu 4 ditambahkan, tersambung ke tiap simpul asli lewat sebuah sisi berbobot 0. Simpul 4 tidak punya sisi masuk, sehingga ia tidak dapat muncul di dalam lintasan terpendek mana pun antara simpul-simpul asli.

Langkah dua: jalankanlah Bellman–Ford dari simpul 4 untuk memperoleh h:

- Sisi (4,0): $h[0]=0$.
- Sisi (4,1): $0+(-5)=-5$, sehingga $h[1]=-5$ — cara termurah mencapai simpul 1 adalah 4 → 0 → 1, bukan sisi langsung berbobot nolnya.

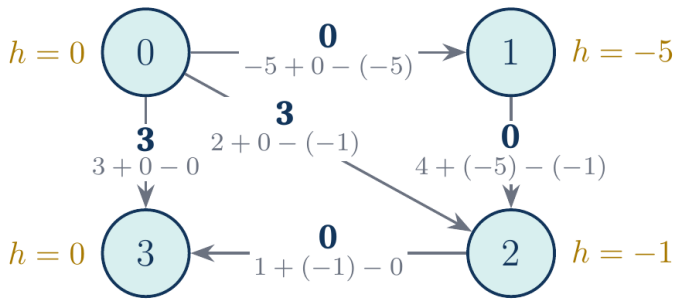
- Sisi (4,2): $0 + (-5) + 4 = -1$, sehingga $h[2] = -1$, lewat $4 \to 0 \to 1 \to 2$.
- Sisi (4,3): $h[3] = 0$.

yang memberikan $h[\,] = \{0, -5, -1, 0\}$.

Langkah tiga: buanglah sumbernya lalu bobotilah ulang dengan $\widehat{w}(u, v) = w(u, v) + h[u] - h[v]$. Sisi demi sisi, hitungannya adalah:

- (0,1): $-5 + 0 - (-5) = 0$.
- (0,2): $2 + 0 - (-1) = 3$.
- (0,3): $3 + 0 - 0 = 3$.
- (1,2): $4 + (-5) - (-1) = 0$.
- (2,3): $1 + (-1) - 0 = 0$.

Gambar 15.25 menggambarkan hasilnya, dengan tiap bobot baru berwarna biru di atas hitungan yang menghasilkannya berwarna merah.



Gambar 15.25 Graf bersimpul empat itu sesudah dibobot ulang. Setiap bobotnya kini tak negatif — tiga dari kelimanya persis nol — sehingga algoritma Dijkstra dapat dipakai dari tiap simpul sebagai sumbernya.

Dua rincian patut diperhatikan, sebab keduanya berulang di setiap contoh. Pertama, sisi negatif $0 \to 1$ menjadi persis 0, bukan sekadar tak negatif. Itu bukan keberuntungan: sisinya terletak *pada* sebuah lintasan terpendek dari sumbernya, sehingga ketaksamaan segitiganya berlaku dengan kesamaan, sehingga $\widehat{w}$-nya persis nol. Secara umum, **setiap sisi pada suatu lintasan terpendek dari s dibobot ulang menjadi nol**, dan sebaliknya pula — sehingga sisi berbobot nol pada $\widehat{w}$ persis merupakan sisi yang dapat dipakai dalam rute terpendek yang keluar dari s. (Sisi-sisi itu tidak harus membentuk sebuah *pohon*; Soal 15.17 memperlihatkan mengapa tidak.) Kedua, bobotnya tidak semuanya menyusut: (0,2) naik dari 2 menjadi 3. Pembobotan ulang bukanlah potongan harga yang seragam; ia sebuah penataan ulang, dan Lema 15.2 adalah jaminan bahwa penataan ulangnya adil.

15.8.7 Contoh berjalan bersimpul lima, dibobot ulang

Kini contoh berjalannya. **Kode 15.10** adalah implementasi lengkapnya.

Kode 15.10 Algoritma Johnson selengkapnya: Bellman–Ford dari sebuah simpul bantu, pembobotan ulang, $|V|$ kali jalannya Dijkstra, dan pengubahannya kembali. Kompilasilah dengan `g++ -std=c++17 -O2 -o johnson johnson.cpp`.

```
// Bab 15, Kode: algoritma Johnson untuk graf jarang, dijalankan pada graf contoh
// berjalan. Bellman-Ford memasok h, lalu Dijkstra dijalankan sekali tiap simpul.
#include <iostream>
#include <iomanip>
#include <vector>
#include <queue>
#include <string>
using namespace std;

const long long INF = 1000000000LL;

struct Edge { int u, v; long long w; };

// BELLMAN-FORD pada G', mengembalikan false ketika ada siklus berbobot negatif yang
terjangkau.
bool bellmanFord(int n, const vector<Edge>& E, int s, vector<long long>& d) {
    d.assign(n, INF);
    d[s] = 0;
    for (int pass = 1; pass <= n - 1; ++pass)          // |V'| - 1 sapuan
        for (const Edge& e : E)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) d[e.v] = d[e.u] + e.w;
    for (const Edge& e : E)                            // satu sapuan lagi: masih adakah
perbaikan?
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) return false;
    return true;
}

// DIJKSTRA dengan tumpukan-min biner, hanya untuk bobot tak negatif.
vector<long long> dijkstra(int n, const vector<vector<pair<int,long long>>>& adj, int s) {
    vector<long long> d(n, INF);
    d[s] = 0;
    priority_queue<pair<long long,int>, vector<pair<long long,int>>, greater<>> Q;
    Q.push({0, s});
    while (!Q.empty()) {
        auto [du, u] = Q.top(); Q.pop();
        if (du > d[u]) continue;                       // entri tumpukan yang sudah basi
        for (auto [v, w] : adj[u])
            if (d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
    }
    return d;
}
```

Kode 15.10 (Lanjutan) *Algoritma Johnson selengkapnya: Bellman–Ford dari sebuah simpul bantu, pembobotan ulang, |V| kali jalannya Dijkstra, dan pengubahannya kembali. Kompilasilah dengan g++ -std=c++17 -O2 -o johnson johnson.cpp.*

```
int main() {
    const int n = 5;                                    // simpul 1..5, disimpan 0..4
    vector<Edge> E = {
        {0,1,3}, {0,2,8}, {0,4,-4},
        {1,3,1}, {1,4,7},
        {2,1,4},
        {3,0,2}, {3,2,-5},
        {4,3,6}
    };

    // Baris 1: G' = G ditambah sumber baru s dengan sisi berbobot 0 ke setiap simpul.
    int s = n;                                          // simpul bantu, indeks 5
    vector<Edge> Eprime = E;
    for (int v = 0; v < n; ++v) Eprime.push_back({s, v, 0});

    // Baris 2-5: Bellman-Ford dari s memberikan h(v) = delta(s, v).
    vector<long long> h;
    if (!bellmanFord(n + 1, Eprime, s, h)) {
        cout << "the input graph contains a negative-weight cycle\n";
        return 0;
    }
    cout << "h(v) = delta(s,v) for v = 1..5:  ";
    for (int v = 0; v < n; ++v) cout << h[v] << ' ';
    cout << "\n\n";

    // Baris 6-7: bobotilah ulang. w-hat(u,v) = w(u,v) + h(u) - h(v).
    vector<vector<pair<int,long long>>> adj(n);
    cout << "reweighted edges:\n";
    for (const Edge& e : E) {
        long long wh = e.w + h[e.u] - h[e.v];
        adj[e.u].push_back({e.v, wh});
        cout << "   w-hat(" << e.u + 1 << "," << e.v + 1 << ") = "
             << setw(3) << e.w << " + (" << h[e.u] << ") - (" << h[e.v] << ") = " << wh <<
'\n';
    }
    cout << '\n';

    // Baris 8-12: satu Dijkstra tiap simpul, lalu batalkan pembobotan ulangnya.
    vector<vector<long long>> D(n, vector<long long>(n, INF));
    for (int u = 0; u < n; ++u) {
        vector<long long> dhat = dijkstra(n, adj, u);
        for (int v = 0; v < n; ++v)
            if (dhat[v] < INF) D[u][v] = dhat[v] + h[v] - h[u];
    }
```

Kode 15.10 (Lanjutan) *Algoritma Johnson selengkapnya: Bellman–Ford dari sebuah simpul bantu, pembobotan ulang, |V| kali jalannya Dijkstra, dan pengubahannya kembali. Kompilasilah dengan g++ -std=c++17 -O2 -o johnson johnson.cpp.*

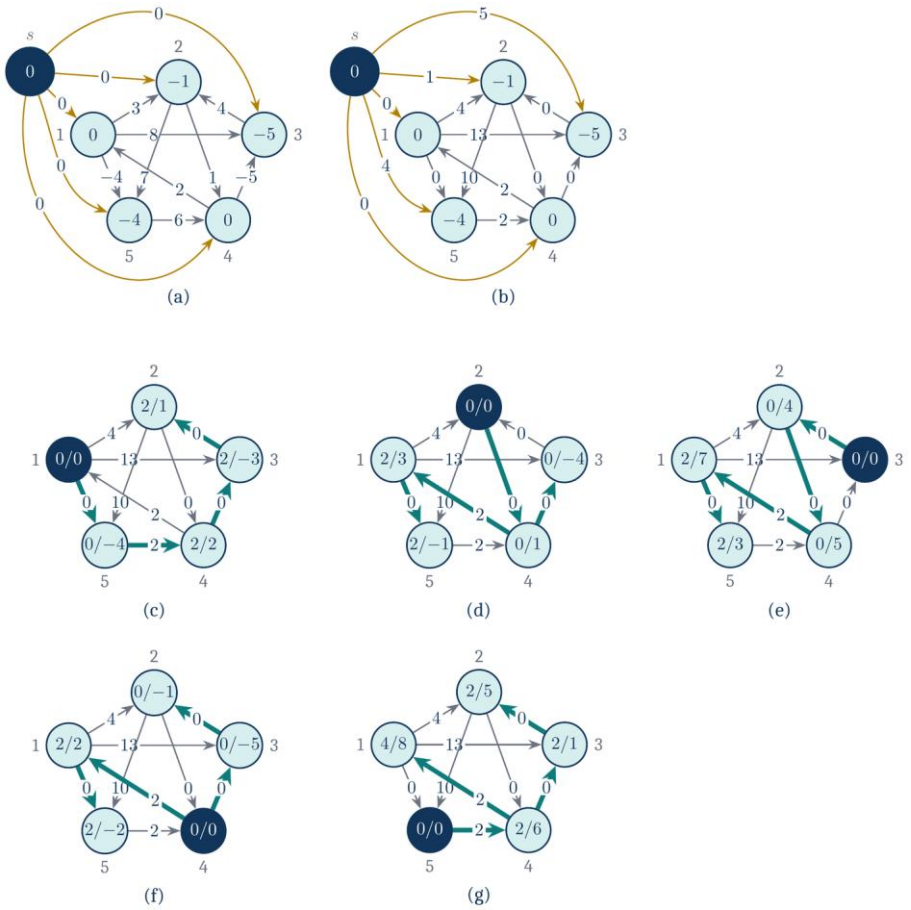
```
    cout << "D = matrix of shortest-path weights:\n";
    for (int i = 0; i < n; ++i) {
        cout << "   ";
        for (int j = 0; j < n; ++j) {
            if (D[i][j] >= INF) cout << setw(6) << "inf";
            else                cout << setw(6) << D[i][j];
        }
        cout << '\n';
    }
    return 0;
}
```

Keluarannya:

```
h(v) = delta(s,v) for v = 1..5:  0 -1 -5 0 -4

reweighted edges:
   w-hat(1,2) =   3 + (0) - (-1) = 4
   w-hat(1,3) =   8 + (0) - (-5) = 13
   w-hat(1,5) =  -4 + (0) - (-4) = 0
   w-hat(2,4) =   1 + (-1) - (0) = 0
   w-hat(2,5) =   7 + (-1) - (-4) = 10
   w-hat(3,2) =   4 + (-5) - (-1) = 0
   w-hat(4,1) =   2 + (0) - (0) = 2
   w-hat(4,3) =  -5 + (0) - (-5) = 0
   w-hat(5,4) =   6 + (-4) - (0) = 2

D = matrix of shortest-path weights:
        0     1    -3     2    -4
        3     0    -4     1    -1
        7     4     0     5     3
        2    -1    -5     0    -2
        8     5     1     6     0
```

Setiap bilangan di sini dapat diperiksa dengan tangan: $h[\,] = \{0, -1, -5, 0, -4\}$ dan kesembilan nilai yang dibobot ulang, 4,13,0,0,10,0,2,0,2. Kesembilannya tak negatif, sebagaimana dijanjikan Lema 15.2 dan §15.8.4. Empat di antaranya nol, dan keempat sisi itu — $1 \to 5$, $2 \to 4$, $3 \to 2$, $4 \to 3$ — persis merupakan sisi yang dapat dipakai dalam rute terpendek yang keluar dari sumber bantunya, tepat seperti diramalkan §15.8.6. Soal 15.17 meminta Anda memeriksanya satu per satu lalu memperhatikan bahwa, karena graf ini punya siklus berbobot nol, keempatnya tidak sepenuhnya membentuk sebuah pohon. Dan matriks akhir D-nya adalah matriks yang sama yang dihasilkan Floyd–Warshall di §15.6.8 dan yang dihasilkan metode matriks lambatnya di §15.4.11.

Gambar 15.26 memperlihatkan seluruh algoritmanya dalam satu gambar, dari G' dengan bobot aslinya, lewat graf yang sudah dibobot ulang, sampai kelima kali jalannya Dijkstra.



Gambar 15.26 Algoritma Johnson pada graf **Gambar 15.2** (Gambar 25.6 pada edisi ketiga buku Cormen dkk.). **(a)** G' dengan bobot aslinya; simpul baru s berwarna biru tua dan tiap simpul v memperlihatkan $h(v) = \delta(s, v)$. **(b)** Sesudah dibobot ulang dengan $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$; setiap bobotnya kini tak negatif. **(c)–(g)** Dijkstra dijalankan dari tiap simpul bergantian, sumbernya berwarna biru tua dan pohon lintasan terpendeknya digambar tebal berwarna hijau-biru; tiap simpul memperlihatkan $\hat{\delta}(u, v)$ dan $\delta(u, v)$ yang dipisahkan garis miring, yang berhubungan lewat $d_{uv} = \delta(u, v) = \hat{\delta}(u, v) + h(v) - h(u)$.

Periksalah satu entri bagian (c) terhadap Persamaan (15.14). Sumbernya adalah simpul 1, sehingga $h(u) = h(1) = 0$. Simpul 3 memperlihatkan $2/-3$: Dijkstra atas $\widehat{w}$ menemukan $\hat{\delta}(1,3) = 2$, dan $h(3) = -5$, sehingga $\delta(1,3) = 2 + (-5) - 0 = -3$. Itulah entri di baris 1, kolom 3 matriks yang tercetak. Jawaban 2 dari graf yang dibobot ulang itu bukanlah jawaban atas pertanyaan siapa pun; Persamaan (15.14)-lah yang mengubahnya kembali menjadi jawaban.

15.9 Algoritma yang Mana, dan Kapan

15.9.1 Keempat ongkosnya berdampingan

Tabel 15.4 mengumpulkan semuanya.

Tabel 15.4 Setiap algoritma di bab ini, beserta ongkosnya dan kasus yang diperuntukkan baginya. Kedua yang dicetak tebal adalah kedua yang sungguh akan Anda pakai.

Algoritma	Menangani sisi negatif?	Waktu jalan	Ruang	Catatan
$V \times$ Dijkstra (larik)	tidak	$O(V^3 + VE)$	$\Theta(V^2)$	pilihan benar paling sederhana pada graf padat tak negatif
$V \times$ Dijkstra (tumpukan biner)	tidak	$O(VE\lg V)$	$\Theta(V + E)$	bagus pada graf jarang tak negatif
$V \times$ Bellman–Ford	ya	$O(V^2E)$, padat $O(V^4)$	$\Theta(V + E)$	pakailah hanya bila tidak ada pilihan lain
SLOW-APSP ($\odot W$ berulang)	ya	$\Theta(V^4)$	$\Theta(V^2)$	bersifat pengajaran; tak pernah pilihan produksi yang tepat
Pemangkatan berulang	ya	$\Theta(V^3\lg V)$	$\Theta(V^2)$	gagasan semiringnya, dibuat efisien
Floyd–Warshall	ya	$\Theta(V^3)$	$\Theta(V^2)$	bawaannya; konstanta mungil, sepele dikodekan
Penutupan transitif (boolean)	–	$\Theta(V^3)$, $\Theta(V^3/64)$ termampat	$\Theta(V^2)$ bit	keterjangkauan saja
Johnson	ya	$O(V^2\lg V + VE)$	$\Theta(V + E)$	bawaannya untuk graf jarang

Perbandingan asimtotik antara kedua baris tebalnya menjadi sederhana begitu dituliskan. Floyd–Warshall adalah $\Theta(V^3)$ tanpa memedulikan E; Johnson adalah $O(V^2\lg V + VE)$. Johnson lebih baik secara asimtotik persis ketika $VE = o(V^3)$, yaitu ketika

$$E = o(V^2).$$

Graf mana pun yang cacah sisinya sungguh-sungguh subkuadratik — jaringan jalan, yang tiap simpangnya punya segelintir jalan keluar; sebuah situs web, yang tiap halamannya menaut ke beberapa puluh halaman lain; sebuah graf sosial, yang derajatnya terbatasi kesabaran manusia — memihak Johnson. Graf yang nyaris tiap pasangannya tersambung memihak Floyd–Warshall.

15.9.2 Titik silangnya, diukur

Asimtotik tidak menetapkan konstantanya, sehingga cara yang jujur untuk menemukan titik silangnya adalah mengukurnya. **Kode 15.11** menyapu kerapatannya pada $V = 600$ yang tetap.

Kode 15.11 Floyd–Warshall melawan $V \times$ Dijkstra pada sembilan kerapatan, $V = 600$ di sepanjangnya. Kompilasilah dengan `g++ -std=c++17 -O2 -o crossover crossover.cpp`.

```
// Bab 15, Kode: mengukur di mana Johnson menyalip Floyd-Warshall.
// Graf acak dengan bobot tak negatif, n tetap, kerapatannya disapu.
#include <iostream>
#include <iomanip>
#include <vector>
#include <queue>
#include <random>
#include <chrono>
using namespace std;
using namespace std::chrono;

const long long INF = 1000000000LL;
struct Edge { int u, v; long long w; };

double timeFloydWarshall(int n, const vector<Edge>& E) {
    vector<vector<long long>> D(n, vector<long long>(n, INF));
    for (int i = 0; i < n; ++i) D[i][i] = 0;
    for (const Edge& e : E) D[e.u][e.v] = min(D[e.u][e.v], e.w);
    auto t0 = steady_clock::now();
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i) {
            if (D[i][k] >= INF) continue;              // tidak ada yang dapat diperbaiki
lewat k
            for (int j = 0; j < n; ++j)
                if (D[k][j] < INF && D[i][k] + D[k][j] < D[i][j])
                    D[i][j] = D[i][k] + D[k][j];
        }
    auto t1 = steady_clock::now();
    return duration<double, milli>(t1 - t0).count();
}

double timeJohnson(int n, const vector<Edge>& E) {
    // Bobotnya di sini sudah tak negatif, sehingga h bernilai 0 seluruhnya dan sapuan
    // Bellman-Ford hanyalah formalitas; yang kita ukur adalah |V| kali jalannya Dijkstra.
    vector<vector<pair<int,long long>>> adj(n);
    for (const Edge& e : E) adj[e.u].push_back({e.v, e.w});
    auto t0 = steady_clock::now();
    vector<long long> d(n);
    long long checksum = 0;
    for (int s = 0; s < n; ++s) {
        fill(d.begin(), d.end(), INF);
        d[s] = 0;
        priority_queue<pair<long long,int>, vector<pair<long long,int>>, greater<>> Q;
        Q.push({0, s});
        while (!Q.empty()) {
            auto [du, u] = Q.top(); Q.pop();
            if (du > d[u]) continue;
```

Kode 15.11 (Lanjutan) *Floyd–Warshall melawan V\times Dijkstra pada sembilan kerapatan, V = 600 di sepanjangnya. Kompilasilah dengan g++ -std=c++17 -O2 -o crossover crossover.cpp.*

```
            for (auto [v, w] : adj[u])
                if (d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
        }
        for (int v = 0; v < n; ++v) if (d[v] < INF) checksum += d[v];
    }
    auto t1 = steady_clock::now();
    if (checksum == -1) cout << "";                    // jaga agar kerjanya tidak dibuang
kompilator
    return duration<double, milli>(t1 - t0).count();
}

int main() {
    const int n = 600;
    mt19937 rng(20250915);
    uniform_int_distribution<int> weight(1, 1000);
    uniform_int_distribution<int> vertex(0, n - 1);

    cout << "n = " << n << ", weights in [1,1000], one random graph per row\n\n";
    cout << setw(10) << "|E|" << setw(12) << "|E|/V"
         << setw(16) << "Floyd (ms)" << setw(16) << "Johnson (ms)"
         << setw(10) << "winner" << '\n';

    for (long long m : {1200LL, 3000LL, 6000LL, 12000LL, 24000LL, 48000LL,
                        96000LL, 180000LL, 360000LL}) {
        vector<Edge> E;
        E.reserve(m);
        for (long long t = 0; t < m; ++t) {
            int u = vertex(rng), v = vertex(rng);
            if (u == v) { --t; continue; }
            E.push_back({u, v, weight(rng)});
        }
        double fw = timeFloydWarshall(n, E);
        double jo = timeJohnson(n, E);
        cout << setw(10) << m << setw(12) << fixed << setprecision(1) << (double)m / n
             << setw(16) << setprecision(1) << fw
             << setw(16) << setprecision(1) << jo
             << setw(10) << (jo < fw ? "Johnson" : "Floyd") << '\n';
    }
    return 0;
}
```

Keluarannya pada mesin yang dipakai menyiapkan bab ini:

```
n = 600, weights in [1,1000], one random graph per row

       |E|       |E|/V      Floyd (ms)    Johnson (ms)    winner
      1200         2.0            21.6            12.9   Johnson
      3000         5.0            86.1            38.2   Johnson
      6000        10.0            99.3            56.7   Johnson
     12000        20.0           103.5            78.3   Johnson
     24000        40.0           112.5           103.1   Johnson
     48000        80.0           113.6           133.2     Floyd
     96000       160.0           115.7           171.7     Floyd
    180000       300.0           116.4           217.6     Floyd
    360000       600.0           118.6           301.5     Floyd
```

Titik silangnya duduk di sekitar $|E|/|V| \approx 80$, yaitu $|E| \approx 48{,}000$ berbanding maksimum yang mungkin $600 \times 599 = 359{,}400$ — kerapatan kira-kira **13 persen**. Di bawah itu, jalankanlah Johnson; di atasnya, jalankanlah Floyd–Warshall.

Dua ciri tabelnya pantas dikomentari. Mulai $|E|/|V| = 10$ ke atas, waktu Floyd–Warshall pada dasarnya mendatar — 99 md sampai 119 md melintasi perubahan cacah sisi enam puluh kali

lipat — persis seperti diramalkan $\Theta(V^3)$. Kedua baris terjarangnya lebih cepat hanya karena `continue` di dalam `timeFloydWarshall`, yang melewati satu gelung dalam utuh setiap kali $d_{ik} = \infty$, dan pada graf yang nyaris tak terhubung lompatan itu tersedia hampir sepanjang waktu. Waktu Johnson, sebaliknya, tumbuh mantap bersama E, persis seperti diramalkan $O(VE\lg V)$. Dan di baris terjarangnya Johnson kira-kira 1.7 kali lebih cepat — persis rezim tempat sebuah jaringan jalan yang sungguhan bermukim.

15.9.3 Sebuah prosedur pengambilan keputusan

Disatukan, **Tabel 15.5** adalah apa yang harus dilakukan ketika seseorang menyerahkan sebuah graf kepada Anda.

Tabel 15.5 Sebuah prosedur pengambilan keputusan bagi persoalan lintasan terpendek semua pasangan. Bekerjalah menurun; "ya" yang pertama adalah jawaban Anda.

Pertanyaan	Bila ya	Bila tidak
Apakah Anda hanya memerlukan keterjangkauan, bukan jarak?	penutupan transitif boolean, dimampatkan ke dalam bit (§15.7)	lanjutkan
Apakah n kecil (katakanlah $n \leq 400$), atau apakah kodenya akan dibaca orang?	Floyd–Warshall — lima baris, tanpa struktur data (§15.6)	lanjutkan
Apakah grafnya padat, $E \gtrsim 0.13\,V^2$ menurut pengukuran §15.9.2?	Floyd–Warshall	lanjutkan
Apakah semua bobotnya tak negatif?	$V \times$ Dijkstra dengan tumpukan biner	Johnson: satu Bellman–Ford, bobotilah ulang, lalu $V \times$ Dijkstra (§15.8)
Apakah Anda memerlukan rutenya dan bukan hanya ongkosnya?	peliharalah Π berdampingan dengan D (§15.6.7); ongkosnya ruang $\Theta(n^2)$ dan nol waktu	—
Mungkinkah ada siklus berbobot negatif?	periksalah diagonalnya sesudah Floyd–Warshall (§15.6.14), atau biarlah Bellman–Ford di baris 2 Johnson melaporkannya	—

15.10 Profitina dalam Praktik

15.10.1 Di mana jarak semua pasangan muncul di sebuah platform GEO

Profitina adalah sebuah platform visibilitas AI: ia memindai sebuah situs web, menilai seberapa siap situs itu untuk **dikutip oleh sebuah mesin jawaban AI**, lalu menjelaskan setiap sinyal yang diukurnya [9]. Pekerjaan itu adalah persoalan graf lebih sering daripada yang tampak.

Sebuah situs yang dirayapi *adalah* sebuah graf berarah. Halaman adalah simpul; tautan adalah sisi. Banyaknya klik dari halaman muka ke suatu halaman — **kedalaman kliknya** — adalah sebuah jarak lintasan terpendek dengan tiap sisi berbobot 1, dan itu penting sebab halaman yang terkubur enam klik dalam adalah halaman yang kecil kemungkinannya pernah dicapai sebuah mesin jawaban, sebaik apa pun isinya. Ragam berbobotnya lebih banyak memberi tahu lagi: berikanlah bobot kecil pada sebuah sisi ketika tautannya menonjol dan bobot yang lebih besar ketika ia terkubur di kaki halaman, maka jarak lintasan terpendeknya menjadi ukuran seberapa sukar sebuah halaman *ditemukan*, bukan sekadar seberapa jauh ia.

Dua pertanyaan yang harus dijawab analitik platformnya sungguh-sungguh merupakan pertanyaan semua pasangan, bukan pertanyaan sumber tunggal.

Diameter graf tautan internalnya. Yang terbesar di antara $\delta(u, v)$ atas semua pasangan adalah perjalanan kasus terburuk di dalam situsnya. Diameter yang besar adalah peringatan struktural: ada bagian situs yang hanya terjangkau lewat rantai yang panjang, dan menata ulang navigasinya akan lebih menolong daripada menulis ulang naskahnya. Menemukan unsur terbesar sebuah himpunan menuntut penghitungan himpunannya, dan itu persoalan semua pasangan. Untuk satu bagian situs berisi beberapa ratus halaman, Floyd–Warshall menjawabnya dalam hitungan milidetik.

Keterjangkauan, dan laporan halaman yatimnya. Halaman mana saja yang dapat dicapai dari halaman muka? Halaman mana saja yang dapat mencapai peta situsnya? Itulah penutupan transitif §15.7, dan versi termampat-bit pada **Kode 15.9** menangani puluhan ribu halaman dengan nyaman. Halaman yang tidak muncul di baris penutupannya mana pun kecuali barisnya sendiri adalah **halaman yatim** — ada di peladen, tak terlihat oleh perayap, dan karena itu tak terlihat oleh mesin jawaban. Inilah salah satu pemeriksaan yang paling murah sekaligus paling berharga yang dapat dijalankan sebuah pemindai.

Dua peringatan, keduanya mencerminkan aturan Profitina sendiri. Pertama, platformnya **memberi saran; penggunanyalah yang memutuskan** — diameter yang besar adalah temuan untuk dijelaskan, tak pernah jaminan atas apa pun. Kedua, situs yang sungguhan cepat tumbuh melampaui titik saat sebuah matriks $n \times n$ masih terjangkau: pada 100.000 halaman keluarannya saja sudah delapan puluh gigabita (§15.2.4). Karena itu sistem produksi menjalankan algoritma semua pasangan pada *bagian-bagian* situsnya — sebuah subgraf berisi beberapa ratus atau beberapa ribu halaman — lalu memakai algoritma sumber tunggal dari segelintir akar yang ditunjuk untuk seluruh situsnya. Mengetahui Anda sedang berada di rezim yang mana adalah paruh praktis bab ini.

15.10.2 Studi kasus yang dikerjakan: Profitina Laundry

Studi kasus yang dapat dijalankan di buku ini selalu **Profitina Laundry**, sebuah contoh pengajaran dan bukan produk yang sesungguhnya.

Profitina Laundry menjemput dan mengantar di delapan bagian Surabaya: Keputih, Gebang, Klampis, Manyar, Ngagel, Darmo, Wonokromo, dan Rungkut. Ruas jalan di antaranya **satu arah di beberapa tempat**, sehingga matriks waktu berkendaranya tidak simetris — perjalanan dari Keputih ke Manyar memakan sembilan menit sementara kembalinya memakan dua belas. Pemiliknya punya dua pertanyaan, dan keduanya bukan pertanyaan yang sama.

1. *Di mana depotnya sebaiknya ditaruh, bila janji kepada pelanggan adalah "kami akan sampai kepada Anda dalam X menit"?* Itu menanyakan lokasi yang perjalanan keluar **terburuknya** paling kecil — **pusat** grafnya, simpul yang **eksentrisitasnya** terkecil, dengan eksentrisitas v adalah $\max_j \delta(v, j)$.
2. *Di mana depotnya sebaiknya ditaruh, bila tujuannya meminimumkan total berkendara?* Itu menanyakan lokasi yang **jumlah** waktu pergi-dan-kembalinya terkecil.

Keduanya memerlukan matriks semua pasangan yang lengkap. **Kode 15.12** menghitungnya dengan Floyd–Warshall, sambil membawa Π supaya rutenya dapat dibaca kembali.

Kode 15.12 Profitina Laundry: waktu berkendara semua pasangan, eksentrisitas, pusat grafnya, diameternya, dan rute di balik perjalanan terburuknya. Kompilasilah dengan `g++ -std=c++17 -O2 -o laundry laundry.cpp`.

```
// Bab 15, Kode: Profitina Laundry -- memilih depot dari matriks jarak
// semua pasangan. Floyd-Warshall pada jaringan jalan satu arah, lalu eksentrisitas,
// pusat grafnya, dan diameternya.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;

const long long INF = 1000000000LL;
const int NIL = -1;

int main() {
    // Delapan tempat di dalam jaringan antarannya. Waktunya adalah menit berkendara pada
ruas
    // satu arah, sehingga matriksnya TIDAK simetris: 0 -> 1 bisa lebih mahal daripada 1 ->
0.
    vector<string> name = {
        "Keputih", "Gebang", "Klampis", "Manyar", "Ngagel", "Darmo", "Wonokromo", "Rungkut"
    };
    const int n = 8;

    // Ruas berarah (dari, ke, menit).
    vector<vector<long long>> road = {
        {0, 1,  4}, {1, 0,  5},
        {0, 2,  9}, {2, 0,  8},
        {1, 2,  3}, {2, 1,  3},
        {1, 3,  7},
        {2, 3,  2}, {3, 2,  4},
        {3, 4,  6}, {4, 3,  6},
        {2, 5, 14},
        {4, 5,  3}, {5, 4,  4},
        {5, 6,  5}, {6, 5,  5},
        {4, 6,  9},
        {6, 0, 20},
        {6, 7,  6}, {7, 6,  6},
        {0, 7,  9}, {7, 0,  9}
    };

    vector<vector<long long>> d(n, vector<long long>(n, INF));
    vector<vector<int>>       pi(n, vector<int>(n, NIL));
    for (int i = 0; i < n; ++i) d[i][i] = 0;
    for (auto& r : road) {
        int u = (int)r[0], v = (int)r[1];
        if (r[2] < d[u][v]) { d[u][v] = r[2]; pi[u][v] = u; }
    }
```

Kode 15.12 (Lanjutan) *Profitina Laundry: waktu berkendara semua pasangan, eksentrisitas, pusat grafnya, diameternya, dan rute di balik perjalanan terburuknya. Kompilasilah dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
    // FLOYD-WARSHALL, di tempat, sambil membawa serta matriks pendahulunya.
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (d[i][k] < INF && d[k][j] < INF && d[i][k] + d[k][j] < d[i][j]) {
                    d[i][j] = d[i][k] + d[k][j];
                    pi[i][j] = pi[k][j];
                }

    cout << "All-pairs driving times (minutes)\n\n" << setw(11) << " ";
    for (int j = 0; j < n; ++j) cout << setw(11) << name[j];
    cout << '\n';
    for (int i = 0; i < n; ++i) {
        cout << setw(11) << name[i];
        for (int j = 0; j < n; ++j) {
            if (d[i][j] >= INF) cout << setw(11) << "-";
            else                cout << setw(11) << d[i][j];
        }
        cout << '\n';
    }

    // Eksentrisitas v adalah jarak lintasan terpendek terbesar dari v ke mana pun.
    // Pusatnya adalah simpul yang eksentrisitasnya terkecil: depot terbaik di bawah
    // janji layanan kasus terburuk. Diameternya adalah eksentrisitas terbesar.
    cout << "\nEccentricity of each candidate depot (worst outbound run):\n";
    long long best = INF, diameter = 0;
    int center = NIL;
    for (int v = 0; v < n; ++v) {
        long long ecc = 0;
        for (int j = 0; j < n; ++j) ecc = max(ecc, d[v][j]);
        cout << "   " << setw(11) << name[v] << setw(5) << ecc << " min\n";
        if (ecc < best) { best = ecc; center = v; }
        diameter = max(diameter, ecc);
    }
    cout << "\ncenter  = " << name[center] << " (eccentricity " << best << " min)\n";
    cout << "diameter = " << diameter << " min\n";
```

Kode 15.12 (Lanjutan) *Profitina Laundry: waktu berkendara semua pasangan, eksentrisitas, pusat grafnya, diameternya, dan rute di balik perjalanan terburuknya. Kompilasilah dengan g++ -std=c++17 -O2 -o laundry laundry.cpp.*

```
    // Biaya pulang-pergi total untuk melayani setiap titik dari sebuah depot tetap: jumlah
    // waktu pergi-dan-kembali. Pertanyaan yang berbeda, dan sering kali jawaban yang
berbeda.
    cout << "\nSum of out-and-back times from each candidate depot:\n";
    long long bestSum = INF; int bestHub = NIL;
    for (int v = 0; v < n; ++v) {
        long long total = 0;
        for (int j = 0; j < n; ++j) total += d[v][j] + d[j][v];
        cout << "   " << setw(11) << name[v] << setw(5) << total << " min\n";
        if (total < bestSum) { bestSum = total; bestHub = v; }
    }
    cout << "\nlowest total round-trip: " << name[bestHub] << " (" << bestSum << " min)\n";

    // Matriks pendahulunya masih memuat rutenya, bukan hanya waktunya.
    cout << "\nThe worst run out of the center, read back from Pi:\n   ";
    int worst = 0;
    for (int j = 0; j < n; ++j) if (d[center][j] > d[center][worst]) worst = j;
    vector<string> route;
    for (int v = worst; v != NIL; v = pi[center][v]) {
        route.push_back(name[v]);
        if (v == center) break;
    }
    for (int t = (int)route.size() - 1; t >= 0; --t)
        cout << route[t] << (t ? " -> " : "");
    cout << "  (" << d[center][worst] << " min)\n";
    return 0;
}
```

Keluarannya:

```
All-pairs driving times (minutes)

               Keputih     Gebang    Klampis     Manyar     Ngagel      Darmo  Wonokromo
Rungkut
    Keputih          0          4          7          9         15         18         15
9
     Gebang          5          0          3          5         11         14         19
14
    Klampis          8          3          0          2          8         11         16
17
     Manyar         12          7          4          0          6          9         14
20
     Ngagel         18         13         10          6          0          3          8
14
      Darmo         20         17         14         10          4          0          5
11
  Wonokromo         15         19         19         15          9          5          0
6
    Rungkut          9         13         16         18         15         11          6
0

Eccentricity of each candidate depot (worst outbound run):
       Keputih   18 min
        Gebang   19 min
       Klampis   17 min
        Manyar   20 min
        Ngagel   18 min
         Darmo   20 min
     Wonokromo   19 min
       Rungkut   18 min

center  = Klampis (eccentricity 17 min)
diameter = 20 min

Sum of out-and-back times from each candidate depot:
       Keputih  164 min
        Gebang  147 min
       Klampis  138 min
        Manyar  137 min
        Ngagel  140 min
         Darmo  152 min
     Wonokromo  171 min
       Rungkut  179 min

lowest total round-trip: Manyar (137 min)

The worst run out of the center, read back from Pi:
   Klampis -> Keputih -> Rungkut  (17 min)
```

Kedua pertanyaannya punya dua jawaban yang berbeda. Taruhlah depotnya di **Klampis** dan tidak ada pelanggan yang berjarak lebih daripada tujuh belas menit — janji terkuat yang tersedia. Taruhlah ia di **Manyar** dan total berkendara hariannya turun menjadi 137 menit, satu menit lebih sedikit daripada Klampis, tetapi perjalanan terburuknya melar menjadi dua puluh menit. Mana yang benar bergantung pada apakah usahanya sedang menjual sebuah jaminan atau sedang meminimumkan bahan bakar, dan algoritmanya tidak dapat membuat pilihan itu. Yang dapat dilakukan algoritmanya — dan yang sudah dilakukannya di sini — adalah menggantikan sebuah perdebatan dengan sebuah tabel.

Perhatikanlah pula ketidaksetangkupan yang dihasilkan ruas satu arahnya. Keputih mencapai Manyar dalam sembilan menit tetapi Manyar memerlukan dua belas untuk kembali; Klampis mencapai Wonokromo dalam enam belas tetapi kembalinya sembilan belas; Manyar mencapai Rungkut dalam dua puluh sementara Rungkut mencapai Manyar dalam delapan belas. Sebagian

pasangan kebetulan sama — Keputih dan Wonokromo berjarak lima belas menit ke arah mana pun — tetapi bahkan di situ rutenya berbeda: ruas satu arah dari Wonokromo ke Keputih memakan dua puluh menit, dan perjalanan lima belas menit itu memutar lewat Rungkut. Model tak berarah akan diam-diam merata-ratakan semua itu lalu menghasilkan jawaban yang masuk akal dan keliru. Matriks semua pasangan yang berarah menjaganya tetap terpisah.

Akhirnya, blok terakhirnya memperlihatkan Π membayar ongkosnya: perjalanan terburuk yang keluar dari Klampis bukanlah satu ruas jalan melainkan rute dua kaki Klampis → Keputih → Rungkut. Tanpa matriks pendahulunya kita akan tahu tujuh belas menitnya dan tidak tahu jalannya.

15.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Menuliskan gelungnya dalam urutan yang keliru. `for (i) for (j) for (k)` alih-alih `for (k) for (i) for (j)`. Ia terkompilasi, ia berjalan, dan ia keliru; **Kode 15.4** memperlihatkannya melaporkan ∞ di tempat jawabannya 3. Ingatlah: k **yang pertama**, sebab rekurensinya membaca nilai putaran ke-$(k-1)$ dan hanya posisi terluar yang menjaminnya.

Meratakan superskripnya. Menuliskan $l_{ij}^{(m)}$ sebagai `l(m)ij` atau `lij^m`, atau $d_{ij}^{(k)}$ sebagai `dij^k`. Superskrip berkurungnya adalah *label tahap*, bukan pangkat, dan mendempetkannya dengan subskripnya memusnahkan satu-satunya hal yang membuat notasinya ada: matriks yang mana di dalam barisannya yang sedang Anda bicarakan. Di bab ini, $l_{ij}^{(m)}$ punya jatah m **sisi** sementara $d_{ij}^{(k)}$ punya himpunan simpul antara yang diizinkan sebanyak k **simpul**; mencampuradukkan keduanya membuat §15.4 dan §15.6 tampak sebagai algoritma yang sama, padahal bukan.

Menghitung `INF + INF`. Bila ∞ diwakili sebagai `INT_MAX`, maka `INF + INF` meluap menjadi bilangan negatif dan algoritmanya dengan riang melaporkan sebuah rute berbobot −2 yang menembus dua sisi yang tidak ada. Entah jagalah penjumlahannya — sebagaimana dilakukan setiap kode di bab ini — atau pilihlah sentinel yang cukup kecil sehingga dua kali lipatnya masih muat, dan itulah sebabnya `1000000000LL` dipakai di sini dan bukan `LLONG_MAX`.

Menyetel $\pi_{ij} = k$ alih-alih $\pi_{ij} = \pi_{kj}$. Simpul k berada di tengah rute yang diperbaiki, bukan di ujungnya. Pendahulu j adalah simpul sebelum j, yang pada penggal ekor $k \rightsquigarrow j$ adalah π_{kj}. Keliru di sini menghasilkan sebuah Π yang "rute" cetakannya bukan lintasan di dalam grafnya. Ujilah rekonstruksi lintasannya dengan menjumlahkan bobot sisi di sepanjang rute yang tercetak lalu membandingkannya dengan d_{ij} — §15.6.11 mengerjakan persis itu.

Memakai Dijkstra sebab graf ujinya kebetulan berhasil. **Kode 15.1** menegaskannya: pada contoh berjalannya, Dijkstra baku menghasilkan dua puluh lima jawaban benar dari dua puluh lima, dan pada sebuah graf bersimpul empat ia menghasilkan dua jawaban keliru dari enam belas. Sebuah sisi negatif membatalkan bukti kebenaran Dijkstra, entah ia kebetulan merusak contoh Anda atau tidak. Bobotilah ulang lebih dahulu (§15.8), atau pakailah Floyd–Warshall.

Lupa bahwa pembobotan ulangnya harus dibatalkan. Dijkstra atas $\widehat{w}$ mengembalikan $\hat{\delta}(u,v)$, yang *bukan* jawabannya. Persamaan (15.14), $d_{uv} = \hat{\delta}(u,v) + h(v) - h(u)$, mengubahnya kembali. Menghilangkan baris 12 **Gambar 15.22** memberikan matriks berisi bilangan yang seluruhnya tak negatif, yang pada graf bersisi negatif semestinya seketika terasa mencurigakan.

Menambahkan sebuah konstanta pada setiap sisi untuk membuang bobot negatifnya. Jebakan yang disebut di §15.8.2: ia menghukum lintasan sebanding dengan cacah sisinya lalu mengubah lintasan mana yang terpendek. Pembobotan ulang yang benar bergantung pada kedua

ujungnya, lewat sebuah fungsi simpul h, dan justru itulah sebabnya Lema 15.2 teleskopis sementara tambalan naifnya tidak.

Menganggap D setangkup. Pada graf berarah, $\delta(u,v)$ dan $\delta(v,u)$ tidak berkaitan; jaringan laundry §15.10.2 punya pasangan yang berbeda tiga menit. Hanya pada graf tak berarah, yang tiap sisinya dapat dipakai kedua arah dengan biaya yang sama, D bersifat setangkup.

Menjalankan algoritma semua pasangan padahal yang Anda perlukan satu sumber. Bila pertanyaannya "seberapa jauh segala sesuatu dari sini?", sekali jalannya Dijkstra atau Bellman–Ford menjawabnya dalam sepersekian waktu dan sepersekian memori. Semua pasangan diperuntukkan bagi saat Anda sungguh-sungguh memerlukan seluruh tabelnya.

Memercayai entri di luar diagonal ketika ada siklus negatif. Begitu $d_{ii} < 0$ untuk suatu i, jawaban sebenarnya yang melibatkan komponen itu adalah $-\infty$ dan bilangan berhingga di dalam matriksnya hanyalah artefak dari berapa putaran yang kebetulan berjalan. Periksalah diagonalnya (§15.6.14) *sebelum* membaca apa pun yang lain.

15.12 Ringkasan Bab

Persoalan lintasan terpendek semua pasangan meminta $\delta(i,j)$ untuk setiap pasangan terurut simpul, beserta sebuah rute yang mencapainya. Masukannya adalah matriks ketetanggaan $W = (w_{ij})$ dari Persamaan (15.1), dengan 0 di diagonalnya dan ∞ untuk sisi yang tidak ada. Keluarannya adalah sebuah matriks jarak dan sebuah **matriks pendahulu** $\Pi = (\pi_{ij})$, yang darinya PRINT-ALL-PAIRS-SHORTEST-PATH memulihkan rute mana pun dalam waktu linear terhadap panjangnya.

Penyelesaian **naifnya** menjalankan sebuah algoritma sumber tunggal n kali: $O(V^3)$ dengan Dijkstra ketika setiap bobotnya tak negatif, $O(V^2E)$ — yaitu $O(V^4)$ pada graf padat — dengan Bellman–Ford ketika tidak.

Metode **perkalian matriks** memerikan sebuah lintasan terpendek menurut banyaknya sisi yang dipakainya. Dengan $l_{ij}^{(m)}$ adalah bobot terkecil sebuah lintasan dari i ke j yang memakai paling banyak m sisi, dan $l_{ij}^{(0)} = 0$ untuk $i = j$ serta ∞ selebihnya, rekurensinya adalah $l_{ij}^{(m)} = \min(l_{ij}^{(m-1)}, \min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\})$, yang meringkas menjadi $\min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}$ sebab $w_{jj} = 0$. Karena lintasan terpendek bersifat sederhana sehingga bersisi paling banyak $n-1$, berlakulah $\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots$. Satu langkah rekurensinya adalah EXTEND-SHORTEST-PATHS, yang merupakan hasil kali matriks sekolahan dengan min menggantikan $+$ dan $+$ menggantikan $\times$ — semiring $(\min, +)$ — sehingga $L^{(m)} = W^m$ dan SLOW-APSP berongkos $\Theta(n^4)$. Karena $\odot$ bersifat asosiatif dan barisannya tetap melewati $n-1$, **pemangkatan berulang** mencapai $L^{(2^{\lceil \lg(n-1)\rceil})} = W^{2^{\lceil \lg(n-1)\rceil}}$ hanya dalam $\lceil \lg(n-1)\rceil$ hasil kali, sehingga $\Theta(n^3 \lg n)$.

Algoritma **Floyd–Warshall** justru memerikan sebuah lintasan terpendek menurut simpul mana saja yang boleh dilewatinya. Dengan $d_{ij}^{(k)}$ adalah bobot terkecil sebuah lintasan dari i ke j yang simpul antaranya terletak di $\{1, \ldots, k\}$, rekurensinya adalah $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ dengan $d_{ij}^{(0)} = w_{ij}$, dan $d_{ij}^{(n)} = \delta(i,j)$. Karena calonnya hanya dua, gelung rangkap tiganya $\Theta(n^3)$ — dan k **harus** menjadi gelung terluarnya, sebab setiap nilai yang dibaca harus milik putaran sebelumnya. Optimasi menjadi satu matriks tunggal di tempat itu aman, sebab putaran k tak

pernah mengubah baris k atau kolom k. Sebuah entri negatif pada diagonalnya sesudah itu persis merupakan syarat adanya siklus berbobot negatif.

Penutupan transitif adalah gelung rangkap tiga yang sama atas semiring boolean: $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$, dengan $t_{ij}^{(0)} = 1$ ketika $i = j$ atau $(i,j) \in E$. Ia $\Theta(n^3)$, dan memampatkan barisnya ke dalam kata mesin memberikan percepatan terukur lebih daripada tujuh puluh kali.

Algoritma Johnson membuat $V \times$ Dijkstra menjadi sah pada graf bersisi negatif. Dengan menambahkan sumber bantu s yang tersambung ke tiap simpul lewat sisi berbobot nol, sekali jalannya Bellman–Ford memberikan $h(v) = \delta(s,v)$; pembobotan ulang dengan $\widehat{w}(u,v) = w(u,v) + h(u) - h(v)$ membiarkan tiap lintasan terpendek tetap menjadi lintasan terpendek, sebab $\widehat{w}(p) = w(p) + h(v_0) - h(v_k)$ bersifat teleskopis, sekaligus membuat setiap bobotnya tak negatif, sebab tuntutannya persis ketaksamaan segitiga. Lalu V kali jalannya Dijkstra beserta pengubahan $d_{uv} = \hat{\delta}(u,v) + h(v) - h(u)$ memberikan jawabannya dalam $O(V^2 \lg V + VE)$.

Yang mana yang dipakai. Floyd–Warshall ketika grafnya padat atau kecil atau kodenya harus terbaca; Johnson ketika ia jarang. Titik silang terukurnya, pada $V = 600$, adalah kerapatan sekitar tiga belas persen.

15.12.1 Menatap ke depan

Bab 16, Evaluasi dan Pembahasan IV, membahas soal-soal tentang Bab 13 sampai 15 dan tentang buku ini secara keseluruhan. Setiap algoritma di bab ini muncul di sana, dan dua jenis soal patut dilatih sebab bab ini menjawabnya dengan program alih-alih dengan pernyataan: *telusurilah Floyd–Warshall pada graf ini, sambil memperlihatkan setiap* $D^{(k)}$, dan *mengapa* k *harus menjadi gelung terluar?*

15.13 Soal Latihan

15.1 Tuliskanlah matriks W dari Persamaan (15.1) untuk graf berarah atas simpul $\{1,2,3\}$ dengan sisi $1 \to 2$ berbobot 4, $2 \to 3$ berbobot -2, dan $3 \to 1$ berbobot 5. Lalu hitunglah $L^{(2)}$ dan $L^{(3)}$ dengan tangan dan nyatakanlah $\delta(i,j)$ untuk kesembilan pasangannya.

15.2 Hitunglah baris 3 dari $L^{(2)}$ untuk contoh berjalan bersimpul lima dengan tangan, sambil memperlihatkan kelima calon bagi masing-masing dari kelima entrinya, lalu periksalah jawaban Anda terhadap §15.4.11.

15.3 Jelaskanlah, dalam dua kalimat, apa yang akan salah pada penyederhanaan dari Persamaan (15.4) menjadi Persamaan (15.5) seandainya definisi W menaruh ∞ dan bukan 0 di diagonalnya. Apakah SLOW-APSP tetap benar seandainya kita mempertahankan rekurensi pada Persamaan (15.4) yang belum disederhanakan beserta ∞ di diagonalnya?

15.4 Buktikanlah bahwa hasil kali $\odot$ EXTEND-SHORTEST-PATHS bersifat asosiatif, yaitu $(A \odot B) \odot C = A \odot (B \odot C)$ untuk semua matriks $n \times n$ atas $\mathbb{R} \cup \{\infty\}$. Bagian mana bab ini yang bergantung pada hasil itu?

15.5 Perlihatkanlah bahwa matriks $L^{(0)}$ dari Persamaan (15.3) — nol di diagonalnya, ∞ di tempat lain — adalah identitas dua sisi bagi $\odot$. Mengapa ia padanan yang tepat bagi matriks identitas biasa?

15.6 Berapa banyak panggilan ke EXTEND-SHORTEST-PATHS yang dilakukan SLOW-ALL-PAIRS-SHORTEST-PATHS, dan berapa banyak yang dilakukan FASTER-ALL-PAIRS-SHORTEST-PATHS,

untuk $n = 5$, $n = 100$, $n = 1000$, dan $n = 1026$? Berikanlah rumus umumnya dan komentarilah kasus yang terakhir.

15.7 Untuk contoh berjalan bersimpul lima, kenalilah satu entri matriks jawabannya yang *tidak* mencapai nilai akhirnya sampai $m = 4$, sebutkanlah rute yang bertanggung jawab, dan jelaskanlah mengapa entri tunggal ini melarang SLOW-APSP berhenti lebih awal.

15.8 Kerjakanlah putaran $k = 2$ Floyd–Warshall pada contoh berjalannya dengan tangan, bermula dari $D^{(1)}$ yang tercetak di §15.6.8. Daftarlah setiap sel yang berubah beserta rute yang bertanggung jawab, dan nyatakanlah perubahan yang bersesuaian pada Π.

15.9 Telusurilah graf (b) dari **Kode 15.4** dengan tangan di bawah urutan gelung `for (i) for (j) for (k)`, dan jelaskanlah dengan persis mengapa ia melaporkan nilai yang *berhingga tetapi terlalu besar* bagi $\delta(1,2)$ dan bukan ∞ yang dihasilkan graf (a). Mengapa jawaban keliru yang berhingga itu lebih berbahaya di antara keduanya?

15.10 Buktikanlah bahwa pada putaran k Floyd–Warshall, tidak ada entri baris k atau kolom k yang berubah. Di bagian mana bab ini hal itu dipakai, dan hipotesis apa tentang grafnya yang dituntut buktinya?

15.11 Jelaskanlah mengapa pembaruan pendahulu pada Persamaan (15.10) adalah $\pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$ dan bukan $\pi_{ij}^{(k)} = k$. Susunlah graf terkecil yang padanya versi kelirunya menghasilkan "rute" yang bukan lintasan di dalam grafnya.

15.12 Hitunglah **eksentrisitas** setiap simpul contoh berjalan bersimpul lima dari matriks $D^{(5)}$ di §15.6.8, lalu dari situ **pusat** dan **diameter**nya. Apakah pusatnya tunggal?

15.13 Jalankanlah TRANSITIVE-CLOSURE pada contoh berjalan bersimpul lima lalu berikanlah $T^{(0)}$ sampai $T^{(5)}$. Pada putaran mana matriksnya pertama kali menjadi satu semua, dan simpul mana yang bertanggung jawab?

15.14 Gelung rangkap tiga Floyd–Warshall bekerja atas sembarang semiring. Tulislah satu program yang menjalankannya atas tiga semiring: $(\min, +)$ untuk bobot lintasan terpendek, (max,min) untuk **lintasan terlebar** (rute yang sisi tersempitnya selebar mungkin), dan sebuah matriks pendamping $(+,\times)$ yang mencacah berapa banyak lintasan terpendek berbeda yang menghubungkan tiap pasangan. Peragakanlah masing-masing pada sebuah graf kecil lalu periksalah cacahnya dengan tangan.

15.15 Buktikanlah bahwa pembobotan ulang Johnson tidak dapat menciptakan ataupun memusnahkan siklus berbobot negatif, dan jelaskanlah mengapa fakta itu diperlukan agar baris 2 **Gambar 15.22** menjadi uji yang sah.

15.16 Mengapa simpul bantu s pada algoritma Johnson tak pernah dapat muncul sebagai simpul antara pada sebuah lintasan terpendek antara dua simpul asli? Berikanlah alasan satu barisnya, dan sebutkanlah apa yang akan rusak seandainya kita memilih $h(v) = \delta(u_0, v)$ untuk suatu simpul u_0 yang sudah ada.

15.17 Pada graf yang sudah dibobot ulang di §15.8.7, persis empat sisinya berbobot 0. Kenalilah keempatnya, buktikanlah pernyataan umum bahwa $\widehat{w}(u, v) = 0$ jika dan hanya jika $\delta(s, v) = \delta(s, u) + w(u, v)$, lalu putuskanlah apakah sisi berbobot nol graf yang satu ini membentuk sebuah pohon. Berikanlah alasan jawaban Anda.

15.18 Dengan memakai **Tabel 15.4**, turunkanlah secara aljabar kerapatan yang membuat $O(V^2 \lg V + VE)$ milik Johnson sama dengan $\Theta(V^3)$ milik Floyd–Warshall, dengan mengabaikan

konstantanya, lalu bandingkanlah ramalan Anda dengan titik silang terukur di §15.9.2. Mengapa keduanya berbeda?

15.14 Penyelesaian Soal Latihan

15.1 Menurut Persamaan (15.1), dengan ∞ di tempat yang tidak bersisi:

$$W = \begin{pmatrix} 0 & 4 & \infty \\ \infty & 0 & -2 \\ 5 & \infty & 0 \end{pmatrix}.$$

$L^{(1)} = W$. Untuk $L^{(2)}$, terapkanlah $l_{ij}^{(2)} = \min_k\{w_{ik} + w_{kj}\}$. Baris 1: $l_{11}^{(2)} = \min(0+0,\ 4+\infty,\ \infty+5) = 0$; $l_{12}^{(2)} = \min(0+4,\ 4+0,\ \infty+\infty) = 4$; $l_{13}^{(2)} = \min(0+\infty,\ 4+(-2),\ \infty+0) = 2$. Baris 2: $l_{21}^{(2)} = \min(\infty+0,\ 0+\infty,\ -2+5) = 3$; $l_{22}^{(2)} = \min(\infty+4,\ 0+0,\ -2+\infty) = 0$; $l_{23}^{(2)} = -2$. Baris 3: $l_{31}^{(2)} = 5$; $l_{32}^{(2)} = \min(5+4,\ \infty+0,\ 0+\infty) = 9$; $l_{33}^{(2)} = \min(5+\infty,\ \infty+(-2),\ 0+0) = 0$. Jadi

$$L^{(2)} = \begin{pmatrix} 0 & 4 & 2 \\ 3 & 0 & -2 \\ 5 & 9 & 0 \end{pmatrix}.$$

Karena $n = 3$, $L^{(n-1)} = L^{(2)}$ sudah menjadi jawabannya, dan memang $L^{(3)} = L^{(2)} \odot W = L^{(2)}$ (pembaca sebaiknya memeriksa satu entri, katakanlah $l_{13}^{(3)} = \min(0+\infty,\ 4+(-2),\ 2+0) = 2$). Jadi δ adalah matriks di atas. Pemeriksaan kewarasan: satu-satunya siklusnya adalah $1 \to 2 \to 3 \to 1$ berbobot $4 - 2 + 5 = 7 > 0$, sehingga tidak ada siklus negatif dan diagonalnya nol semua, sebagaimana seharusnya.

15.2 Baris 3 dari W adalah $(\infty, 4, 0, \infty, \infty)$, sehingga hanya $k = 2$ dan $k = 3$ yang memberikan faktor kiri yang berhingga. $l_{31}^{(2)} = \min\{\infty+0,\ 4+\infty,\ 0+\infty,\ \infty+2,\ \infty+\infty\} = \infty$. $l_{32}^{(2)} = \min\{\infty+3,\ 4+0,\ 0+4,\ \infty+\infty,\ \infty+\infty\} = 4$ — seri antara "capailah 2 dalam satu langkah lalu berdiamlah" dan "berdiamlah di 3 lalu ambil sisinya". $l_{33}^{(2)} = \min\{\infty+8,\ 4+\infty,\ 0+0,\ \infty+\infty,\ \infty+\infty\} = 0$. $l_{34}^{(2)} = \min\{\infty+\infty,\ 4+1,\ 0+\infty,\ \infty+0,\ \infty+6\} = 5$, lewat $3 \to 2 \to 4$. $l_{35}^{(2)} = \min\{\infty+(-4),\ 4+7,\ 0+\infty,\ \infty+\infty,\ \infty+0\} = 11$, lewat $3 \to 2 \to 5$. Baris 3 dari $L^{(2)}$ adalah $(\infty, 4, 0, 5, 11)$, cocok dengan §15.4.11.

15.3 Dengan ∞ di diagonalnya calon $k = j$ akan menyumbang $l_{ij}^{(m-1)} + \infty = \infty$ dan bukan $l_{ij}^{(m-1)}$, sehingga argumen pertama minimum yang di luar tidak lagi termasuk di antara calon yang di dalam dan peringkasan menjadi Persamaan (15.5) menjadi tidak sah. Ya, SLOW-APSP akan tetap benar dengan rekurensi pada Persamaan (15.4) yang *belum disederhanakan* — minimum berargumen dua itu tetap mencacah kedua kasusnya — tetapi EXTEND-SHORTEST-PATHS akan memerlukan suku tambahan, kemiripannya dengan perkalian matriks akan lenyap, dan bersamanya lenyap pula tipuan pemangkatan berulangnya, sebab $L^{(0)}$ tidak lagi menjadi identitas bagi $\odot$.

15.4 Ambillah sembarang i, j. Menuliskan kedua ruasnya lengkap,

$$\begin{aligned}((A \odot B) \odot C)_{ij} &= \min_q \left\{\min_k\{a_{ik} + b_{kq}\} + c_{qj}\right\} \\ &= \min_q \min_k \{a_{ik} + b_{kq} + c_{qj}\},\end{aligned}$$

dengan memakai distributivitas ($x + \min\{y, z\} = \min\{x + y, x + z\}$) untuk mendorong c_{qj} masuk. Serupa dengan itu $(A \odot (B \odot C))_{ij} = \min_k \min_q \{a_{ik} + b_{kq} + c_{qj}\}$. Keduanya adalah minimum himpunan berhingga yang sama berisi n^2 bilangan $\{a_{ik} + b_{kq} + c_{qj}\}$, yang diambil dalam urutan yang berbeda, dan minimum sebuah himpunan berhingga tidak bergantung pada urutan pemindaiannya. Karena itu keduanya sama. §15.5 bergantung pada ini: pemangkatan berulang hanya sah karena pengurungannya boleh ditata ulang.

15.5 $(L^{(0)} \odot A)_{ij} = \min_k \left\{l_{ik}^{(0)} + a_{kj}\right\}$. Setiap suku dengan $k \neq i$ bernilai $\infty + a_{kj} = \infty$; suku dengan $k = i$ bernilai $0 + a_{ij} = a_{ij}$. Jadi minimumnya adalah a_{ij}. Argumen di sisi yang lain setangkup. Ia padanan yang tepat sebab pada semiring mana pun matriks identitasnya membawa identitas perkaliannya $\bar{1}$ di diagonalnya dan identitas penjumlahannya $\bar{0}$ di tempat lain; di sini $\bar{1} = 0$ (sebab $x + 0 = x$) dan $\bar{0} = \infty$ (sebab $\min(x, \infty) = x$), dan itu persis $L^{(0)}$.

15.6 SLOW-APSP melakukan $n - 2$ panggilan (gelungnya berjalan $m = 2, \ldots, n - 1$); FASTER-APSP melakukan $\lceil \lg(n - 1) \rceil$. Jadi: $n = 5$, 3 berbanding 2; $n = 100$, 98 berbanding 7; $n = 1000$, 998 berbanding 10; $n = 1026$, 1024 berbanding 11. Kasus terakhirnya mendidik: $n = 1025$ juga memerlukan 10 (sebab $2^{10} = 1024 \geq 1024$) tetapi $n = 1026$ memerlukan 11, sehingga satu simpul tambahan saja dapat menambah satu hasil kali matriks utuh — ongkosnya fungsi tangga terhadap n, bukan fungsi mulus.

15.7 Entrinya adalah l_{12}, yang bernilai 3 di $L^{(1)}$, $L^{(2)}$, dan $L^{(3)}$ lalu jatuh ke 1 hanya di $L^{(4)}$. Rute yang bertanggung jawab adalah $1 \to 5 \to 4 \to 3 \to 2$, berbobot $-4 + 6 - 5 + 4 = 1$, yang memakai empat sisi dan karena itu tidak dapat ditemukan dengan jatah tiga. Ia melarang berhenti lebih awal atas dasar firasat. Bila diamati sendirian, entri l_{12} bertahan pada 3 selama tiga matriks berturut-turut dan tampak sudah mapan — lalu berubah. Satu entri yang sudah berhenti bergerak tidak membuktikan apa-apa, begitu pula jatah sisi kecil yang tetap, yang dipilih karena "rute biasanya pendek". (Ada satu uji yang *aman*: bila seluruh matriks memenuhi $L^{(m)} = L^{(m-1)}$, maka $L^{(m+1)} = L^{(m)} \odot W = L^{(m-1)} \odot W = L^{(m)}$, sehingga tidak ada lagi yang akan pernah berubah. Di sini uji itu baru menyala pada $m = 5$, sebab $L^{(4)}$ masih berbeda dari $L^{(3)}$.) Tanpa pemeriksaan semacam itu, berhenti yang aman adalah $m = n - 1$.

15.8 Bermula dari $D^{(1)}$ yang barisnya $(0,3,8,\infty,-4)$, $(\infty,0,\infty,1,7)$, $(\infty,4,0,\infty,\infty)$, $(2,5,-5,0,-2)$, $(\infty,\infty,\infty,6,0)$, putaran $k = 2$ menguji $d_{i2} + d_{2j}$ terhadap d_{ij}.

- $(1,4)$: $\min(\infty,\ d_{12} + d_{24} = 3 + 1 = 4) = 4$. **Berubah**; rutenya $1 \to 2 \to 4$; $\pi_{14}^{(2)} = \pi_{24}^{(1)} = 2$.
- $(3,4)$: $\min(\infty,\ 4 + 1 = 5) = 5$. **Berubah**; rutenya $3 \to 2 \to 4$; $\pi_{34}^{(2)} = 2$.
- $(3,5)$: $\min(\infty,\ 4 + 7 = 11) = 11$. **Berubah**; rutenya $3 \to 2 \to 5$; $\pi_{35}^{(2)} = \pi_{25}^{(1)} = 2$.

Selebihnya tidak bergerak: baris 2 adalah baris k; baris 5 punya $d_{52} = \infty$; calon baris 4, yaitu $5 + d_{2j}$, semuanya kalah ($5 + 1 = 6 > 0$ pada $j = 4$, $5 + 7 = 12 > -2$ pada $j = 5$). Tiga sel, persis ketiga beda antara $D^{(1)}$ dan $D^{(2)}$ yang tercetak.

15.9 $\delta(1,2)$ yang sebenarnya adalah $6 + 3 + 1 = 10$, di sepanjang $1 \to 4 \to 3 \to 2$. Dengan i terluar, baris 1 diisi dari kiri ke kanan. Pada sel (1,2) sapuan-k-nya hanya dapat memakai $k = 4$, sebab d_{13} masih ∞ pada saat itu; ia menemukan $d_{14} + d_{42} = 6 + 6 = 12$ lalu menuliskan 12. Satu kolom kemudian, pada sel (1,3), sapuan yang sama menyetel $d_{13} = d_{14} + d_{43} = 6 + 3 = 9$ — tetapi sel (1,2) sudah dilewati dan tak pernah ditengok lagi, sehingga perbaikan $d_{13} + d_{32} = 9 + 1 = 10$ tak pernah dipertimbangkan. Jawaban yang dilaporkan adalah 12, sebagaimana dikukuhkan programnya. Ia kegagalan yang lebih berbahaya sebab sebuah ∞ di tempat yang jelas

ada rutenya kentara pada pengujian apa pun, sementara 12 melawan 10 yang sebenarnya kelihatan benar-benar masuk akal: ia rute yang nyata, berpanjang yang masuk akal, dan hanya implementasi kedua atau pemeriksaan tangan yang menyingkapnya.

15.10 Selama putaran k calon bagi d_{kj} adalah $d_{kk} + d_{kj}$ dan calon bagi d_{ik} adalah $d_{ik} + d_{kk}$. Karena grafnya tidak punya siklus berbobot negatif, $d_{kk} \geq 0$: jalan tertutup termurah menembus k adalah yang kosong, berbobot 0. Jadi kedua calonnya ≥ petahananya, dan tidak ada entri yang berubah. Hal itu dipakai di §15.6.13 untuk membuktikan bahwa optimasi di tempatnya aman, dan hipotesis yang dituntut persis **tidak ada siklus berbobot negatif**.

15.11 Bila minimumnya datang dari merutekan lewat k, lintasan yang diperbaiki itu berakhir dengan penggal $k \rightsquigarrow j$, dan simpul persis sebelum j adalah pendahulu j *pada rute terbaik yang keluar dari* k, yaitu $\pi_{kj}^{(k-1)}$. Simpul k duduk di tengah rute barunya, bukan di ujungnya. Kedua aturannya berbeda persis ketika penggal $k \rightsquigarrow j$ bersisi lebih daripada satu, yaitu ketika $\pi_{kj} \neq k$. Graf terkecil semacam itu bersimpul empat dan bersisi tiga: $1 \to 3$, $3 \to 2$, dan $2 \to 4$, masing-masing berbobot 1. Putaran $k = 2$ menyetel $d_{34} = d_{32} + d_{24} = 2$ dengan $\pi_{34} = \pi_{24} = 2$. Putaran $k = 3$ lalu memperbaiki d_{14} dari ∞ menjadi $d_{13} + d_{34} = 1 + 2 = 3$. Aturan yang benar mencatat $\pi_{14} = \pi_{34} = 2$, dan PRINT-ALL-PAIRS-SHORTEST-PATH menghasilkan $1 \to 3 \to 2 \to 4$, yang jumlah bobot sisinya $3 = d_{14}$. Aturan yang keliru mencatat $\pi_{14} = k = 3$, dan prosedur yang sama menghasilkan $1 \to 3 \to 4$ — sama sekali bukan lintasan di dalam grafnya, sebab sisi $3 \to 4$ tidak ada. Perhatikanlah bahwa matriks jaraknya sama saja di kedua jalan: hanya rutenya yang rusak, dan itulah sebabnya kutu ini selamat dari pengujian apa pun yang memeriksa bilangan dan bukan lintasan.

15.12 Dari $D^{(5)}$, eksentrisitasnya (maksimum tiap baris) adalah: simpul 1, $\max(0,1,-3,2,-4) = 2$; simpul 2, $\max(3,0,-4,1,-1) = 3$; simpul 3, $\max(7,4,0,5,3) = 7$; simpul 4, $\max(2,-1,-5,0,-2) = 2$; simpul 5, $\max(8,5,1,6,0) = 8$. **Diameter**nya adalah yang terbesar, 8, yang dicapai oleh $\delta(5,1)$ di sepanjang $5 \to 4 \to 1$. **Pusat**nya tidak tunggal: simpul 1 maupun simpul 4 sama-sama bereksentrisitas 2. Seri lazim terjadi pada graf kecil, dan sebuah aturan pemecah seri — indeks terkecil, atau jumlah baris terkecil — harus dinyatakan, bukan diserahkan kepada implementasinya.

15.13 Dengan $t_{ij}^{(0)} = 1$ persis di tempat $i = j$ atau $w_{ij} < \infty$:

$$T^{(0)} = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}, \quad T^{(1)} = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}, \quad T^{(2)} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix},$$

$T^{(3)} = T^{(2)}$ (satu-satunya sisi keluar simpul 3 menuju simpul 2, yang tidak menambahkan apa pun yang baru), dan $T^{(4)} = T^{(5)} = $ matriks satu semua. Matriksnya pertama kali menjadi satu semua pada putaran $k = 4$, dan **simpul 4-lah yang bertanggung jawab**: ia satu-satunya simpul yang punya sisi masuk ke simpul 1, sehingga sampai merutekan lewat 4 diizinkan, kolom 1 bernilai nol di baris 2, 3, dan 5.

15.14 Kode 15.13 adalah satu program yang mengerjakan ketiganya.

Kode 15.13 Gelung rangkap tiga yang sama atas tiga semiring: lintasan terpendek, lintasan terlebar, dan cacah lintasan terpendek. Kompilasilah dengan `g++ -std=c++17 -O2 -o ex_semirings ex_semirings.cpp`.

```
// Bab 15, penyelesaian soal: satu gelung rangkap tiga, tiga persoalan yang berbeda.
// Kerangka Floyd-Warshall tidak peduli apa arti "min" dan "+", asalkan
// pasangan itu menaati hukum semiring. Ubah pasangannya, berubahlah persoalannya.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;

void banner(const string& s) { cout << "\n" << s << "\n"; }

int main() {
    // ------------------------------------------------------------ (min, +)
    // Bobot lintasan terpendek, pada contoh berjalan bersimpul lima.
    {
        const int n = 5;
        vector<vector<long long>> D = {
            {  0,   3,   8, INF,  -4},
            {INF,   0, INF,   1,   7},
            {INF,   4,   0, INF, INF},
            {  2, INF,  -5,   0, INF},
            {INF, INF, INF,   6,   0}
        };
        for (int k = 0; k < n; ++k)
            for (int i = 0; i < n; ++i)
                for (int j = 0; j < n; ++j)
                    if (D[i][k] < INF && D[k][j] < INF)
                        D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
        banner("(min,+) on the running example -- shortest-path weight:");
        for (int i = 0; i < n; ++i) {
            cout << "   ";
            for (int j = 0; j < n; ++j) cout << setw(5) << D[i][j];
            cout << '\n';
        }
    }
```

Kode 15.13 (Lanjutan) *Gelung rangkap tiga yang sama atas tiga semiring: lintasan terpendek, lintasan terlebar, dan cacah lintasan terpendek. Kompilasilah dengan g++ -std=c++17 -O2 -o ex_semirings ex_semirings.cpp.*

```
    // ------------------------------------------------------------------ (max, min)
    // Lintasan terlebar (bottleneck maksimum). Nilai sisinya adalah kapasitas dalam liter per
    // menit; b[i][j] adalah aliran terbesar yang dapat diangkut satu rute dari i ke j.
    {
        const int n = 5;
        const long long NONE = -1;
        // Kapasitas: 1->2 = 10, 2->4 = 5, 1->3 = 4, 3->4 = 8, 4->5 = 7.
        vector<vector<long long>> B(n, vector<long long>(n, NONE));
        vector<vector<long long>> cap = {{0,1,10},{1,3,5},{0,2,4},{2,3,8},{3,4,7}};
        for (int i = 0; i < n; ++i) B[i][i] = INF;      // rute tanpa sisi tidak terbatas
        for (auto& c : cap) B[c[0]][c[1]] = c[2];
        for (int k = 0; k < n; ++k)
            for (int i = 0; i < n; ++i)
                for (int j = 0; j < n; ++j)
                    if (B[i][k] != NONE && B[k][j] != NONE)
                        B[i][j] = max(B[i][j], min(B[i][k], B[k][j]));
        banner("(max,min) -- widest path, bottleneck capacity:");
        for (int i = 0; i < n; ++i) {
            cout << "   ";
            for (int j = 0; j < n; ++j) {
                if (B[i][j] == NONE)     cout << setw(6) << "-";
                else if (B[i][j] == INF) cout << setw(6) << "inf";
                else                     cout << setw(6) << B[i][j];
            }
            cout << '\n';
        }
        cout << "   widest route from 1 to 4 carries " << B[0][3] << " l/min"
             << " (via 2, whose 5 beats the lower branch's 4)\n";
    }

    // ------------------------------------------------------------------ (+, x)
    // Berapa banyak lintasan terpendek berbeda yang menghubungkan tiap pasangan. Dua matriks berjalan
    // beriringan: bobot D, dan cacah C lintasan berbobot minimum.
    {
        const int n = 5;
        // 1->2, 1->3, 2->4, 3->4, 4->5 masing-masing berbobot 1; ditambah 1->5 langsung berbobot 3.
        vector<vector<long long>> E = {{0,1,1},{0,2,1},{1,3,1},{2,3,1},{3,4,1},{0,4,3}};
        vector<vector<long long>> D(n, vector<long long>(n, INF));
        vector<vector<long long>> C(n, vector<long long>(n, 0));
        for (int i = 0; i < n; ++i) { D[i][i] = 0; C[i][i] = 1; }
        for (auto& e : E) { D[e[0]][e[1]] = e[2]; C[e[0]][e[1]] = 1; }
        for (int k = 0; k < n; ++k)
            for (int i = 0; i < n; ++i) {
                if (i == k || D[i][k] >= INF) continue;
                for (int j = 0; j < n; ++j) {
                    if (j == k || j == i || D[k][j] >= INF) continue;
```

Kode 15.13 (Lanjutan) *Gelung rangkap tiga yang sama atas tiga semiring: lintasan terpendek, lintasan terlebar, dan cacah lintasan terpendek. Kompilasilah dengan g++ -std=c++17 -O2 -o ex_semirings ex_semirings.cpp.*

```
                    long long cand = D[i][k] + D[k][j];
                    if (cand < D[i][j])       { D[i][j] = cand; C[i][j] = C[i][k] *
C[k][j]; }
                    else if (cand == D[i][j]) { C[i][j] += C[i][k] * C[k][j]; }
                }
            }
        banner("(+,x) -- number of distinct shortest paths:");
        cout << "   weights:\n";
        for (int i = 0; i < n; ++i) {
            cout << "   ";
            for (int j = 0; j < n; ++j) {
                if (D[i][j] >= INF) cout << setw(5) << "inf"; else cout << setw(5) <<
D[i][j];
            }
            cout << '\n';
        }
        cout << "   counts:\n";
        for (int i = 0; i < n; ++i) {
            cout << "   ";
            for (int j = 0; j < n; ++j) cout << setw(5) << C[i][j];
            cout << '\n';
        }
        cout << "   delta(1,5) = " << D[0][4] << ", achieved by " << C[0][4]
             << " different paths\n";
    }
    return 0;
}
```

Keluarannya:

```
(min,+) on the running example -- shortest-path weight:
       0    1   -3    2   -4
       3    0   -4    1   -1
       7    4    0    5    3
       2   -1   -5    0   -2
       8    5    1    6    0

(max,min) -- widest path, bottleneck capacity:
      inf   10    4    5    5
        -  inf    -    5    5
        -    -  inf    8    7
        -    -    -  inf    7
        -    -    -    -  inf
   widest route from 1 to 4 carries 5 l/min (via 2, whose 5 beats the lower branch's 4)

(+,x) -- number of distinct shortest paths:
   weights:
       0    1    1    2    3
     inf    0  inf    1    2
     inf  inf    0    1    2
     inf  inf  inf    0    1
     inf  inf  inf  inf    0
   counts:
       1    1    1    2    3
       0    1    0    1    1
       0    0    1    1    1
       0    0    0    1    1
       0    0    0    0    1
   delta(1,5) = 3, achieved by 3 different paths
```

Memeriksanya dengan tangan: rute terlebar dari 1 ke 4 menempuh entah $1 \to 2 \to 4$, yang sisi tersempitnya $\min(10,5) = 5$, atau $1 \to 3 \to 4$, yang tersempitnya $\min(4,8) = 4$; maksimumnya 5,

sebagaimana tercetak. Cacah $C_{15} = 3$ bersesuaian dengan $1 \to 2 \to 4 \to 5$, $1 \to 3 \to 4 \to 5$, dan $1 \to 5$ yang langsung, semuanya berbobot 3; dan $C_{14} = 2$ bersesuaian dengan $1 \to 2 \to 4$ dan $1 \to 3 \to 4$. Perhatikanlah kedua penjaga `i == k` dan `j == k` di gelung pencacahnya: tanpa keduanya calon merosot $d_{ii} + d_{ij} = d_{ij}$ akan tercacah sebagai lintasan kedua yang berbeda dan cacahnya akan berlipat dua.

15.15 Menurut Lema 15.2 yang diterapkan pada sebuah siklus p dengan $v_0 = v_k$, kita memperoleh $\widehat{w}(p) = w(p) + h(v_0) - h(v_0) = w(p)$. Siklus mempertahankan bobotnya persis. Jadi sebuah siklus negatif di bawah $\widehat{w}$ jika dan hanya jika ia negatif di bawah w, dan pembobotan ulang tidak dapat menciptakan ataupun memusnahkannya. Ini penting bagi baris 2 sebab Bellman–Ford dijalankan pada G' dengan bobot **asli** w, sebelum pembobotan ulang apa pun — dan G' punya siklus negatif jika dan hanya jika G punya, sebab sisi yang ditambahkan semuanya meninggalkan s dan tidak ada yang memasukinya, sehingga tidak ada siklus baru yang tercipta. Seandainya pembobotan ulang dapat memusnahkan siklus negatif, uji yang dilakukan pada satu tahap tidak akan mengatakan apa-apa tentang tahap yang lain, dan algoritmanya dapat melanjutkan menjalankan Dijkstra pada graf yang lintasan terpendeknya tidak ada.

15.16 Sebab s **tidak punya sisi masuk**: untuk muncul sebagai simpul antara sebuah lintasan Anda harus dimasuki sekaligus ditinggalkan, dan tidak ada yang memasuki s. Memilih $h(v) = \delta(u_0, v)$ untuk simpul u_0 yang sudah ada rusak sebab u_0 mungkin tidak mencapai setiap simpul, sehingga $h(v) = \infty$ untuk sebagian v dan Persamaan (15.13) menjadi $\infty - \infty$, yang tak terdefinisi. (Bila u_0 kebetulan mencapai setiap simpul, pilihan itu sebenarnya berhasil: Lema 15.2 berlaku untuk h *apa pun*, dan ketaksamaan segitiga tetap membuat setiap bobot baru tak negatif. Bahwa u_0 punya sisi masuk dan dapat terletak di dalam lintasan terpendek lain tidaklah berbahaya.) Simpul bantu sekadar cara yang murah dan seragam untuk menjamin setiap $h(v)$ berhingga.

15.17 Dari keluaran **Kode 15.10**, keempat sisi berbobot nolnya adalah (1,5), (2,4), (3,2), dan (4,3). **Pernyataan umumnya.** $\widehat{w}(u, v) = w(u, v) + h(u) - h(v) = 0$ persis ketika $h(v) = h(u) + w(u, v)$, yaitu ketika $\delta(s, v) = \delta(s, u) + w(u, v)$. Sisi yang memenuhi ini disebut **ketat** (*tight*): mencapai v dengan lebih dahulu menempuh rute termurah ke u lalu menyeberangi (u, v) semurah mencapai v sama sekali, sehingga (u, v) adalah sisi terakhir suatu rute termurah menuju v. Sebaliknya, bila (u, v) terletak pada sebuah rute termurah dari s, maka menurut Teorema 13.1 awalan yang berakhir di u sendiri termurah, sehingga $\delta(s, v) = \delta(s, u) + w(u, v)$ dan sisinya ketat. **Memeriksa keempatnya**, dengan $h = (0, -1, -5, 0, -4)$ untuk simpul 1 sampai 5: (1,5) sebab $h(5) = -4 = 0 + (-4)$; (4,3) sebab $h(3) = -5 = 0 + (-5)$; (3,2) sebab $h(2) = -1 = -5 + 4$; (2,4) sebab $h(4) = 0 = -1 + 1$. Keempatnya ketat. **Apakah keempatnya membentuk pohon?** Tidak. Bersama sisi berbobot nol $s \to 1$ dan $s \to 4$ dari G', sisi-sisi ketatnya memuat siklus $4 \to 3 \to 2 \to 4$, berbobot $-5 + 4 + 1 = 0$. Itu sepenuhnya sah — yang dilarang adalah siklus *negatif*, bukan siklus nol — dan itu berarti subgraf ketatnya di sini adalah sebuah DAG lintasan terpendek yang memuat siklus berbobot nol, bukan sebuah pohon. Pohon lintasan terpendek yang sejati selalu dapat diambil dengan memelihara persis satu sisi ketat masuk per simpul, tetapi kode yang menganggap sisi berbobot nolnya sudah membentuk pohon, lalu menyusuri penunjuk induknya tanpa pemeriksaan kunjungan, akan berputar selamanya pada graf ini.

15.18 Setelah $V^2 \lg V + VE = V^3$ lalu selesaikanlah untuk E: membagi dengan V memberikan $V \lg V + E = V^2$, sehingga $E = V^2 - V \lg V$. Pada $V = 600$ itu meramalkan $E \approx 360{,}000 - 600 \times 9.2 \approx 354{,}500$ — yaitu kerapatan sekitar 98 persen, sementara pengukuran di §15.9.2 menaruh titik silangnya dekat 13 persen. Keduanya berbeda sebab rumus asimtotiknya membuang faktor konstanta, dan konstantanya di sini sangat berbeda: pernyataan terdalam Floyd–Warshall adalah satu penjumlahan dan satu pembandingan pada memori yang

bersebelahan, sementara pernyataan Dijkstra melibatkan sebuah dorongan ke antrean prioritas, sebuah pengambilan, sebuah penelusuran daftar ketetanggaan yang meleset dari tembolok, dan sebuah pemeriksaan entri basi. Bila selisihnya kita modelkan sebagai sebuah konstanta tunggal c yang mengalikan kerja per sisi Johnson, titik silangnya bergeser ke $E \approx (V^2 - V\lg V)/c$; menyamakannya dengan 48,000 hasil pengukuran memberikan $c \approx 354{,}500/48{,}000 \approx 7.4$. Biaya per sisi kira-kira tujuh kali biaya Floyd–Warshall adalah harga yang sangat masuk akal bagi sebuah dorongan tumpukan, sebuah pengambilan tumpukan, sebuah kelesetan tembolok pada daftar ketetanggaannya, dan sebuah uji entri basi. Pelajarannya sama dengan yang ditarik di §15.7.4: asimtotik memilih keluarga algoritmanya, pengukuran memilih anggotanya.

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] R. W. Floyd, “Algorithm 97: Shortest path,” *Communications of the ACM*, vol. 5, no. 6, p. 345, Jun. 1962.

[3] S. Warshall, “A theorem on Boolean matrices,” *Journal of the ACM*, vol. 9, no. 1, pp. 11–12, Jan. 1962.

[4] D. B. Johnson, “Efficient algorithms for shortest paths in sparse networks,” *Journal of the ACM*, vol. 24, no. 1, pp. 1–13, Jan. 1977.

[5] J. Kleinberg and É. Tardos, *Algorithm Design*, 1st ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[6] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2011.

[7] M. L. Fredman and R. E. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[8] U. Zwick, “All pairs shortest paths using bridging sets and rectangular matrix multiplication,” *Journal of the ACM*, vol. 49, no. 3, pp. 289–317, May 2002.

[9] Profitina, “AI Visibility (GEO) platform — product overview,” Profitina Group, Surabaya, Indonesia, 2026. [Online]. Available: https://profitina.com

Bab 16
Evaluasi dan Pembahasan IV: Lintasan Terpendek, Analisis Amortisasi, dan Keseluruhan Buku

Bab terakhir ini tidak menambahkan teori baru. Ia mengonsolidasikan Bab 13 sampai 15 — lintasan terpendek sumber tunggal, analisis amortisasi, dan lintasan terpendek semua pasangan — lalu mundur selangkah untuk memadukan keseluruhan buku, dengan cara mengerjakan persoalan. Jumlahnya dua puluh satu, dan setiap persoalan diselesaikan dari nol: setiap bilangan antara dituliskan, dan setiap klaim numerik diperiksa oleh program C++17 lengkap yang sudah dikompilasi dengan `g++ -std=c++17 -O2 -Wall` lalu dijalankan. Lima persoalan pemadu membuka bab ini: lima belas pernyataan benar-atau-salah yang merentang seluruh buku, subbarisan menaik terpanjang, sebuah bukti induksi tentang pohon biner diperluas, hampiran-2 untuk persoalan pedagang keliling, dan perbandingan Bellman–Ford dengan Dijkstra. Tujuh persoalan kemudian melatih Bab 13 sampai 15 satu alat demi satu alat: penelusuran Dijkstra dengan seluruh tabel jarak-dan-pendahulu sesudah setiap iterasi, Bellman–Ford putaran demi putaran dengan sebuah siklus negatif yang dideteksi dan diperlihatkan, lintasan terpendek pada graf berarah taksiklik, pencacah biner dan tabel dinamis yang masing-masing dianalisis dengan metode agregat, akuntansi, dan potensial dengan seluruh hitungannya ditampilkan, Floyd–Warshall dengan setiap matriks $D^{(0)}$ sampai $D^{(4)}$ dihitung sel demi sel, dan pembobotan ulang Johnson yang dikerjakan selengkapnya. Sembilan persoalan lanjutan meninjau kembali bagian buku selebihnya. Setiap persoalan diakhiri dengan varian *uji diri* berikut jawabannya. Bab ini ditutup dengan empat puluh latihan uji diri singkat, sebuah tabel pemilihan algoritma yang merupakan satu-satunya halaman dari buku ini yang layak disimpan, tamasya berpemandu ke enam wilayah besar yang tidak dimasuki buku ini, dan sepatah kata penutup.

16.1 Posisi Kita Saat Ini

16.1.1 Apa yang diberikan Bab 13 sampai 15

Tiga bab telah memberi kita perangkat kedua, dan setiap persoalan di bagian tengah bab ini meraih salah satu alatnya. Ada baiknya kita menyebutkan alat-alat itu sebelum mulai.

Dari **Bab 13** kita memiliki persoalan lintasan terpendek sumber tunggal (*single-source shortest paths*): sebuah graf berarah berbobot $G = (V, E)$ dengan fungsi bobot w, sebuah sumber s, dan tugas menghitung, untuk setiap simpul v, bobot lintasan terpendek $\delta(s, v)$ beserta sebuah lintasan yang mencapainya. Kita memiliki dua larik yang dipelihara setiap algoritmanya — taksiran $d[v]$ dan pendahulu $\pi[v]$ — dan satu-satunya operasi dasar yang mengubah keduanya, yaitu **relaksasi** (*relaxation*) sebuah sisi (u, v):

$$\text{if } d[v] > d[u] + w(u, v) \text{ then } d[v] \leftarrow d[u] + w(u, v), \quad \pi[v] \leftarrow u. \tag{16.1}$$

Kita memiliki lima sifat relaksasi (batas atas, tanpa lintasan, konvergensi, relaksasi lintasan, subgraf pendahulu), ketaksamaan segitiga $\delta(s, v) \leq \delta(s, u) + w(u, v)$, dan tiga algoritma yang

hanya berbeda dalam *urutan* merelaksasi sisinya: Dijkstra (simpul belum-selesai terdekat lebih dahulu, hanya untuk bobot tak negatif), Bellman–Ford (setiap sisi, $|V| - 1$ kali, bobot negatif diizinkan dan siklus negatif dideteksi), dan algoritma DAG (simpul dalam urutan topologis, satu putaran, bobot apa pun).

Dari **Bab 14** kita memiliki **analisis amortisasi** (*amortized analysis*): membatasi biaya total sebuah *barisan* n operasi, sehingga satu operasi yang sesekali mahal dibayar oleh banyak operasi murah. Kita memiliki tiga metodenya — metode **agregat** (hitung totalnya, lalu bagi dengan n), metode **akuntansi** (tarifkan biaya tetap pada setiap operasi dan simpan saldo kredit yang tidak boleh negatif), dan metode **potensial** (definisikan potensial Φ pada struktur datanya dan bebankan pada setiap operasi biaya sebenarnya ditambah perubahan potensialnya) — serta tiga struktur data tempat ketiganya dilatih: tumpukan dengan `MULTIPOP`, pencacah biner k-bit, dan tabel dinamis yang berlipat dua ketika penuh.

Dari **Bab 15** kita memiliki persoalan semua pasangan (*all-pairs*) dan tiga penyelesaiannya: "perkalian" berulang matriks jarak, algoritma **Floyd–Warshall** dengan rekurensinya atas *simpul mana yang boleh menjadi simpul antara sebuah lintasan*, dan algoritma **Johnson**, yang membobot ulang setiap sisi dengan potensial h hasil satu kali Bellman–Ford sehingga $|V|$ kali Dijkstra menjadi sah. Ketiga bab itu mengikuti Cormen dkk. [1].

16.1.2 Seluruh buku pada satu halaman

Bab ini juga memadukan seluruh buku, jadi seluruh perangkatnya diletakkan di atas meja. **Tabel 16.1** adalah petanya. Untuk setiap bab, tabel itu menyebut satu gagasan yang menjadi alasan keberadaan bab itu — satu kalimat yang, jika Anda kehilangan semua hal lain, memungkinkan Anda membangun kembali sisanya — dan menunjukkan di mana dalam bab ini gagasan itu bekerja.

Tabel 16.1 Satu gagasan per bab, dan di mana bab ini memakainya. Jika Anda dapat menyebutkan keenam belasnya di luar kepala, seluruh buku ada di tangan Anda.

Bab	Satu gagasannya	Dipakai di sini
1	Sebuah persoalan diselesaikan dalam lima fase: memahami, mengabstraksi, merancang, menganalisis, mengimplementasikan. Menulis kode adalah fase terakhir, bukan yang pertama.	Subbab 16.2
2	Invarian gelung adalah cara Anda mengetahui sebuah gelung benar: inisialisasi, pemeliharaan, terminasi.	Subbab 16.4.4
3	Bandingkan laju pertumbuhan, bukan stopwatch. $T(n) = aT(n/b) + f(n)$ diselesaikan dengan membandingkan $f(n)$ dengan $n^{\log_b a}$.	Persoalan 16.1, pernyataan (11)
4	Kendala adalah anggaran waktu. $n \leq 10^5$ berarti $O(n\lg n)$ atau lebih baik; $n \leq 500$ berarti $O(n^3)$ sudah memadai.	Subbab 16.4.7
5	Heapsort menukar pemilihan murah dari *heap* dengan $\Theta(n\lg n)$ yang selalu berlaku; quicksort menukar partisi murah dengan $\Theta(n\lg n)$ yang biasanya berlaku. $\Omega(n\lg n)$ adalah lantai pengurutan berbasis perbandingan.	Persoalan 16.1, pernyataan (1) dan (5)
6	Invarian keseimbangan mengubah $O(n)$ menjadi $O(\lg n)$, dan rotasi adalah satu-satunya alat yang dibutuhkan untuk memeliharanya.	Persoalan 16.3
7	Ketika subpersoalan bertumpang tindih dan optimumnya dibangun dari potongan-potongan optimal, ingatlah alih-alih menghitung ulang.	Persoalan 16.2 dan 16.19
8	Jawabannya adalah penurunannya. Kode tanpa invarian adalah tebakan yang kebetulan lolos sebuah uji.	Subbab 16.2
9	Pilihan yang terbaik secara lokal hanya aman secara global jika sifat pilihan rakus (*greedy-choice property*) sudah dibuktikan.	Persoalan 16.4 dan 16.16
10	Satu kerangka penelusuran, dua wadah: antrean memberi jumlah sisi paling sedikit, tumpukan memberi struktur.	Persoalan 16.1 (pernyataan 13) dan 16.8
11	Sifat potongan: sisi teringan yang menyeberangi potongan mana pun yang dihormati hutan saat ini adalah sisi yang aman.	Persoalan 16.4 dan 16.21
12	Sebuah algoritma baru menjadi artefak rekayasa ketika ia diimplementasikan, diukur, dan dipertahankan.	Subbab 16.16
13	Relaksasi ditambah sebuah urutan. Urutan Dijkstra menurut jarak, urutan Bellman–Ford "semua sisi, $\lvert V\rvert - 1$ kali", urutan DAG topologis.	Persoalan 16.5 sampai 16.8
14	Biaya sebuah barisan operasi bukanlah langkah tunggal terburuk dikalikan banyaknya langkah.	Persoalan 16.9 dan 16.10
15	Rekurensi yang sama dua kali: atas panjang lintasan, dan atas simpul mana yang boleh menjadi simpul antara. Pembobotan ulang membuat Dijkstra sah pada sisi negatif.	Persoalan 16.11, 16.12, dan 16.17
16	Memilih algoritma adalah keterampilannya. Sisanya pembukuan.	Subbab 16.14

16.1.3 Bagaimana bab ini disusun

Bab ini dibangun dalam empat lapis.

1. **Sebuah metode** (Subbab 16.2): cara menghadapi persoalan yang terdiri atas beberapa bagian, atau bukti yang tidak kunjung selesai pada percobaan pertama.
2. **Lima persoalan pemadu** (Subbab 16.3 sampai 16.7), masing-masing dikerjakan selengkapnya karena masing-masing menarik dari beberapa bab sekaligus. Persoalan 16.4, hampiran untuk pedagang keliling, adalah permata utamanya: ia menyangkut pokok bahasan yang tidak pernah diajarkan buku ini, namun setiap bahan penyelesaiannya sudah ada di tangan Anda.
3. **Tujuh persoalan tentang Bab 13 sampai 15** (Subbab 16.8 sampai 16.10), masing-masing mengarah pada satu alat, dan sembilan **persoalan terkerjakan lanjutan** (Subbab 16.11) yang

meninjau kembali bagian buku selebihnya. Semuanya berbentuk sama, sehingga Anda selalu tahu di mana Anda berada:

- **Persoalan** — pernyataannya.
- **Apa yang diuji** — alatnya, dan di mana buku ini memperkenalkannya.
- **Penyelesaian** — langkah demi langkah, dengan setiap langkah hitungnya ditampilkan.
- **Program** — program C++17 lengkap, berikut keluaran yang benar-benar dicetaknya.
- **Uji diri** — sebuah varian untuk Anda kerjakan sendiri, berikut jawabannya.

4. **Konsolidasi** (Subbab 16.12 sampai 16.15): persoalan-persoalan itu dibandingkan berdampingan, empat puluh latihan uji diri, tabel pemilihan algoritma, dan ke mana melangkah selanjutnya.

Subbab 16.16 memperlihatkan di mana gagasan seluruh buku muncul dalam sebuah produk nyata, dan bab ini ditutup dengan kesalahan yang sering terjadi, ringkasan dan sepatah kata penutup, empat belas soal latihan, dan penyelesaian lengkapnya.

> **WAWASAN KUNCI: CARA MEMAKAI BAB INI.** Bacalah pernyataan setiap persoalan, lalu tutuplah penyelesaiannya dengan selembar kertas dan cobalah sendiri selama sepuluh menit sebelum membaca lebih lanjut. Penyelesaian yang sudah Anda gumuli selama sepuluh menit dipahami dengan cara yang tidak pernah terjadi pada penyelesaian yang hanya Anda baca. Varian *uji diri* ada agar Anda dapat menguji pemahaman itu pada sesuatu yang belum pernah Anda lihat dikerjakan.

16.2 Cara Menghadapi Persoalan Berbagian dan Bukti

Beberapa persoalan dalam bab ini punya bagian-bagian — *rancang, lalu analisis, lalu buktikan*; *uraikan, lalu tulis pseudokode, lalu ilustrasikan* — dan beberapa meminta bukti. Kedua jenis itu mengganjar disiplin yang sama, dan disiplin itu layak dinyatakan sekali, sebelum persoalan-persoalannya, alih-alih diulang di dalam masing-masing.

16.2.1 Tulislah jawaban dalam urutan yang tetap

Di dalam bagian mana pun, tulislah dengan urutan ini.

1. **Jawabannya, dalam satu kalimat.** "Algoritma ini berjalan dalam waktu $\Theta(n^2)$." "Pernyataan itu salah." Seorang pembaca — rekan kerja, penelaah, atau Anda sendiri bulan depan — mencari kalimat ini lebih dahulu. Jika ia terkubur di paragraf keempat, ia mungkin tidak pernah ditemukan.
2. **Asumsinya.** Apa persisnya masukannya? Apa arti setiap kata dalam persoalannya? Persoalan 16.2 sepenuhnya bertumpu pada arti satu kata, *subbarisan*, dan jawaban yang tidak menyatakannya mungkin menyelesaikan persoalan yang keliru.
3. **Penurunan atau buktinya.**
4. **Sebuah contoh terkerjakan.** Jangan pernah dihilangkan. Sebuah kasus kecil yang dibawa dengan tangan sampai tuntas adalah bukti termurah yang ada bahwa Anda memahami apa yang Anda tulis, dan juga cara tercepat untuk menemukan bahwa Anda belum memahaminya.

16.2.2 Ketika sebuah bukti tidak kunjung selesai

Bukti bukanlah satu kesatuan yang tak terbagi. Ia adalah barisan klaim, dan setiap klaim yang benar, dinyatakan dengan jelas, dan relevan adalah kemajuan. Yang merusak kemajuan bukanlah ketidaklengkapan, melainkan *ketidakpaduan*: tiga perempat halaman manipulasi aljabar tanpa

pernyataan tentang apa yang sedang dibuktikan. Jadi ketika sebuah bukti macet, kerjakan ini, berurutan.

1. **Nyatakan klaimnya secara formal.** Tulislah "Klaim: $d(L) \leq 2\,d(L^*)$." Biayanya delapan detik, dan ia memberi tahu Anda, dan setiap pembaca sesudahnya, ke mana Anda menuju.
2. **Sebutkan tekniknya dan besarannya.** "Dengan induksi kuat atas n, banyaknya simpul dalam." "Dengan fungsi potensial $= $ banyaknya bit 1." Ini menetapkan *jenis* objek apa bukti itu.
3. **Kerjakan kasus dasarnya.** Kasus dasar hampir selalu mudah, dan sering kali di sanalah sebuah klaim yang keliru pertama kali menampakkan diri.
4. **Nyatakan hipotesisnya dengan tepat.** Bukan "anggaplah berlaku untuk pohon yang lebih kecil", melainkan "anggaplah $X(n') = n' + 1$ untuk setiap $0 \leq n' < n$."
5. **Tuliskan langkah-langkah yang *dapat* Anda benarkan, dan beri label pada celahnya.** Tulis secara harfiah: "Tinggal menunjukkan bahwa $d(W) \leq 2\,d(T)$." Celah yang jujur dan berlabel adalah tindakan profesional: ia memberi tahu orang berikutnya — sering kali Anda sendiri — persis di mana harus mendorong. Celah yang disamarkan adalah perangkap bagi setiap orang yang kelak membaca bukti itu.
6. **Periksalah klaimnya pada sebuah contoh,** dan katakan bahwa itu ilustrasi, bukan bukti. Ia memperlihatkan bahwa Anda tahu apa yang *dikatakan* klaim itu, dan sangat sering ia mengisyaratkan langkah yang hilang.

> **WAWASAN KUNCI: BUKTI YANG MACET DIPERBAIKI DENGAN STRUKTUR, BUKAN DENGAN PANJANG.** Enam baris berlabel jelas — klaim, teknik, dasar, hipotesis, apa yang menyusul, apa yang masih hilang — lebih berharga daripada tiga halaman aljabar tak berlabel yang berhenti di tengah jalan. Keenam baris itu memperlihatkan bentuk argumennya bahkan sebelum argumennya lengkap, dan bentuk itulah yang biasanya mengisyaratkan langkah yang hilang.

16.2.3 Lima kebiasaan yang layak dipelihara

- **Jangan pernah menulis "jelas" atau "tentu saja" di dalam bukti.** Jika memang jelas, ia hanya butuh satu baris: tulislah baris itu.
- **Jangan pernah menyatakan batas O tanpa menyebut kasus mana yang dibatasinya.** "Quicksort adalah $O(n^2)$" dan "quicksort adalah $O(n\lg n)$" keduanya pernyataan yang benar tentang kasus yang berbeda, dan kalimat yang tidak menyebut kasus yang mana tidak bernilai apa-apa. Pernyataan (1) Persoalan 16.1 bertumpu tepat pada hal ini.
- **Berikan pseudokode sebelum kode.** Persoalan "rancanglah sebuah algoritma" meminta sebuah gagasan yang dapat diperiksa pembaca. Daftar kode C++ adalah bukti bahwa gagasan itu berjalan; ia bukan gagasannya.
- **Ilustrasikan algoritmanya, bukan grafnya.** Gambar sebuah algoritma lintasan terpendek yang tidak memperlihatkan taksiran $d[v]$ berubah hanyalah gambar sebuah graf.
- **Periksalah setiap bilangan dua kali** — sekali dengan tangan dan sekali dengan program. Setiap tabel dalam bab ini dihasilkan dengan kedua cara itu, dan hasilnya sepakat.

16.3 Persoalan 16.1 — Lima Belas Pernyataan, Benar atau Salah

16.3.1 Persoalannya

Persoalan 16.1. Tandailah setiap pernyataan berikut Benar (B) atau Salah (S), dan berikan alasan untuk setiap jawaban.

1. Quicksort mempunyai kompleksitas asimtotik yang lebih buruk daripada merge sort.
2. Pencarian biner adalah $O(\log n)$.
3. Pencarian linear pada larik tak terurut adalah $O(n)$, tetapi pada larik terurut adalah $O(\log n)$.
4. Pencarian linear pada senarai berantai terurut adalah $O(n)$.
5. Jika Anda hanya perlu mencari satu nilai, pencarian linear pada larik tak terurut secara asimtotik lebih baik daripada mengurutkan lalu melakukan pencarian biner.
6. Jika semua bobot busur berbeda, pohon merentang minimumnya tunggal.
7. Pencarian biner pada sebuah larik menuntut larik itu terurut.
8. Penyisipan ke dalam senarai terurut dapat dikerjakan dalam $O(\log n)$.
9. Fungsi *hash* yang baik menyebarkan unsur-unsur secara seragam ke seluruh tabel *hash*.
10. Dengan fungsi *hash* yang baik dan data yang wajar, pencarian dalam tabel *hash* dapat dikerjakan dalam $O(1)$.
11. Jika sebuah kode mempunyai kompleksitas asimtotik $O(g(n))$, maka *paling sedikit* $g(n)$ operasi akan dijalankan ketika kode itu dijalankan dengan parameter n.
12. Jika tersedia implementasi yang baik dari berbagai algoritma untuk suatu proses, seperti mengurutkan, mencari, atau menemukan pohon merentang minimum, Anda *selalu* sebaiknya memilih algoritma dengan kompleksitas asimtotik yang lebih baik.
13. *Tidak mungkin* penelusuran mendalam dan penelusuran melebar sebuah graf mengunjungi simpul-simpulnya dalam urutan yang sama jika graf itu memuat lebih dari dua simpul.
14. Banyaknya simpul maksimum pada pohon biner bertinggi H ($H = 0$ untuk simpul daun) adalah $2^{H+1} - 1$.
15. Pada pohon biner lengkap, hanya simpul daun yang tidak mempunyai anak.

16.3.2 Apa yang diuji

Bukan ingatan. Sembilan dari kelima belas pernyataan itu benar dan enam salah, dan setiap yang salah itu salah karena *alasan yang diajarkan buku ini*, bukan karena salah cetak. Persoalan ini menguji apakah Anda dapat membedakan empat pasang gagasan yang lazim dicampuradukkan:

- batas **atas** dan cacahan **persis** (pernyataan 11, 12);
- kasus **terburuk** sebuah algoritma dan kasus **rata-ratanya** (pernyataan 1);
- apa yang diizinkan **bentuk** sebuah struktur data dan apa yang dikerjakan sebuah **algoritma** tertentu di atasnya (pernyataan 3, 4, 8);
- apa yang **dijamin** dan apa yang **lazim terjadi** (pernyataan 9, 10, 12).

Kerjakan kelima belasnya dengan keempat pasangan itu dalam benak, dan persoalan ini memakan lima belas menit. Kerjakan dengan hafalan, dan ia memakan satu jam — dan pernyataan (11) tetap akan keliru.

16.3.3 Jawabannya

Tabel 16.2 memberikan kelima belas jawabannya berikut alasan satu barisnya masing-masing. Subbab-subbab sesudahnya menurunkan jawaban yang layak diturunkan.

Tabel 16.2 Kelima belas jawaban Persoalan 16.1, berikut pembenaran satu baris untuk masing-masingnya. Sembilan benar dan enam salah.

#	Pernyataan, disingkat	Jawaban	Mengapa
1	Quicksort berkompleksitas asimtotik lebih buruk daripada merge sort	**B**	Kasus terburuk $\Theta(n^2)$ berhadapan dengan $\Theta(n\lg n)$
2	Pencarian biner $O(\log n)$	**B**	$T(n) = T(n/2) + \Theta(1)$
3	Pencarian linear: $O(n)$ tak terurut, $O(\log n)$ terurut	**S**	Pencarian linear $\Theta(n)$ apa pun urutannya
4	Pencarian linear pada senarai berantai terurut $O(n)$	**B**	Tanpa akses acak, jadi tanpa pembagian dua
5	Untuk satu pencarian, linear mengalahkan urutkan-lalu-cari-biner	**B**	$O(n)$ berhadapan dengan $\Omega(n\lg n)$
6	Bobot busur berbeda semua ⇒ MST-nya tunggal	**B**	Argumen pertukaran, Subbab 16.3.9
7	Pencarian biner menuntut lariknya terurut	**B**	Separuh yang dibuang harus dipastikan keliru
8	Penyisipan ke senarai terurut dapat dikerjakan dalam $O(\log n)$	**S**	Mencari tempatnya $O(\lg n)$; membuat ruang $\Theta(n)$
9	Fungsi *hash* yang baik menyebarkan kunci secara seragam	**B**	Itulah definisi *hashing* seragam sederhana
10	Fungsi *hash* baik + data wajar ⇒ pencarian $O(1)$	**B**	Harapan $\Theta(1+\alpha)$ dengan $\alpha = O(1)$
11	$O(g(n))$ berarti paling sedikit $g(n)$ operasi dijalankan	**S**	O batas atas; Ω yang batas bawah
12	Selalu pilih kompleksitas asimtotik yang lebih baik	**S**	Konstanta, memori, ukuran masukan, kestabilan
13	DFS dan BFS tak pernah sama pada graf >2 simpul	**S**	Sebuah lintasan atau bintang; 26 dari 38 graf bersimpul empat, Subbab 16.3.11
14	Pohon biner bertinggi H punya paling banyak $2^{H+1} - 1$ simpul	**B**	$\sum_{i=0}^{H} 2^i = 2^{H+1} - 1$
15	Pada pohon biner lengkap hanya daun yang tak beranak	**B**	Benar menurut definisi *daun*

16.3.4 Pernyataan (1) — yang membutuhkan satu kalimat syarat

"Quicksort mempunyai kompleksitas asimtotik yang lebih buruk daripada merge sort."

Jawabannya **Benar**, tetapi hanya di bawah satu pembacaan yang sebaiknya Anda nyatakan secara tersurat.

Merge sort berjalan dalam $\Theta(n\lg n)$ pada kasus terbaik, kasus rata-rata, dan kasus terburuk; pembelahannya menurut posisi, jadi datanya tidak dapat mengubah rekurensinya. Rekurensi quicksort bergantung pada porosnya. Dengan pembelahan yang seimbang, $T(n) = 2T(n/2) + \Theta(n) = \Theta(n\lg n)$. Dengan pembelahan seburuk-buruknya — satu unsur di satu sisi, $n-1$ di sisi lain, yang dihasilkan poros unsur-terakhir pada larik yang sudah terurut — rekurensinya menjadi

$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2).$$

Jadi:

- **kasus terburuk:** quicksort $\Theta(n^2)$, merge sort $\Theta(n\lg n)$. Quicksort lebih buruk. **Benar.**
- **kasus rata-rata:** keduanya $\Theta(n\lg n)$. Tak ada yang lebih buruk. Pernyataannya akan **salah**.

Ketika sebuah pernyataan benar di bawah satu pembacaan baku dan salah di bawah pembacaan yang lain, isinya terletak pada kalimat syaratnya. Tulislah: **"Benar, dengan membaca 'kompleksitas asimtotik' sebagai kasus terburuk: $\Theta(n^2)$ berhadapan dengan $\Theta(n\lg n)$. Pada kasus rata-rata keduanya $\Theta(n\lg n)$ dan pernyataannya akan salah."** Satu anak kalimat tambahan mengubah tebakan menjadi peragaan.

> **PERINGATAN: KESALAHAN PALING LAZIM PADA PERNYATAAN (1).** Banyak pembaca menjawab *Salah* karena mereka ingat bahwa quicksort lebih cepat daripada merge sort dalam praktik. Memang benar — konstantanya lebih kecil, lokalitasnya lebih baik, dan ia mengurutkan di tempat. Tetapi "lebih cepat dalam praktik" dan "berkompleksitas asimtotik lebih buruk" keduanya benar sekaligus, dan pernyataannya menanyakan yang kedua. Praktik adalah urusan pernyataan (12), tempat ia justru menjadi seluruh pokok persoalannya.

16.3.5 Pernyataan (3), (4), dan (7) — keterurutan dan algoritma yang memakainya

Ketiganya adalah satu gagasan yang dilihat dari tiga sisi: **mengurutkan data mengubah apa yang *dapat* dikerjakan sebuah algoritma; ia tidak mengubah apa yang *dikerjakan* algoritma tertentu.**

(3) "Pencarian linear pada larik tak terurut adalah $O(n)$, tetapi pada larik terurut adalah $O(\log n)$." **Salah.** Pencarian linear memeriksa $A[1], A[2], \ldots$ berurutan sampai ia menemukan kuncinya atau kehabisan unsur. Pada larik terurut ia boleh berhenti lebih awal ketika melewati kuncinya, tetapi kasus terburuknya — kuncinya unsur terakhir, atau lebih besar daripada setiap unsur — tetap berbiaya n perbandingan. Pencarian linear $\Theta(n)$ pada masukan terurut maupun tak terurut. Yang *memang* $O(\lg n)$ pada larik terurut adalah pencarian biner, dan itu algoritma yang berbeda.

(4) "Pencarian linear pada senarai berantai terurut adalah $O(n)$." **Benar**, dan karena alasan yang lebih kuat daripada di (3). Di dalam senarai berantai Anda tidak dapat mencapai unsur tengahnya tanpa berjalan ke sana. Pencarian biner di sini bukan sekadar tidak menolong; ia tidak tersedia, sebab langkah pembagian duanya membutuhkan akses $\Theta(1)$ ke posisi $\lfloor (lo + hi)/2 \rfloor$, sedangkan senarai memberikan $\Theta(n)$.

(7) "Pencarian biner pada sebuah larik menuntut larik itu terurut." **Benar.** Seluruh pembenaran algoritmanya terletak pada langkah pembuangan: sesudah membandingkan kuncinya dengan $A[\text{mid}]$ ia membuang separuh lariknya. Pembuangan itu hanya sahih jika keterurutan menjamin kuncinya tak mungkin berada di separuh yang dibuang. Cabutlah keterurutannya, dan pembuangan itu menjadi tebakan.

16.3.6 Pernyataan (5) — kapan tidak perlu mengurutkan

"Jika Anda hanya perlu mencari satu nilai, pencarian linear pada larik tak terurut secara asimtotik lebih baik daripada mengurutkan lalu melakukan pencarian biner." **Benar.**

Hitungannya pendek, dan hitungan itulah yang membuat jawabannya meyakinkan. Pencarian linear berbiaya $\Theta(n)$. Alternatifnya berbiaya

$$\underbrace{\Omega(n \lg n)}_{\text{pengurutannya}} + \underbrace{O(\lg n)}_{\text{pencariannya}} = \Omega(n \lg n),$$

dan n tumbuh lebih lambat daripada $n\lg n$, jadi pencarian linear menang secara asimtotik. Batas bawah $\Omega(n\lg n)$ pada pengurutannya bukan sifat satu algoritma pengurutan; ia batas bawah pengurutan berbasis perbandingan dari Bab 5, jadi tak ada kepintaran yang dapat memulihkan selisihnya.

Pertanyaan susulan yang wajar: *pada berapa kali pencarian pengurutan itu mulai membayar dirinya sendiri?* Dengan k kali pencarian, linear berbiaya $\Theta(kn)$ dan urutkan-lalu-cari berbiaya $\Theta(n\lg n + k\lg n)$. Keduanya berpotongan ketika $kn \approx n\lg n$, yaitu pada $k \approx \lg n$. Untuk sejuta unsur, $\lg n \approx 20$: urutkan jika Anda memperkirakan lebih dari sekitar dua puluh kali pencarian, pindai jika kurang.

16.3.7 Pernyataan (8) — jebakan dengan dua separuh

"Penyisipan ke dalam senarai terurut dapat dikerjakan dalam $O(\log n)$." **Salah**, dan inilah pernyataan dari kelima belasnya yang paling sering dijawab keliru.

Penyisipan punya dua bagian.

- **Mencari tempat unsur barunya.** Pada *larik* terurut, pencarian biner mengerjakannya dalam $O(\lg n)$.
- **Meletakkannya di sana.** Pada larik terurut, setiap unsur sesudah titik sisipnya harus bergeser satu tempat ke kanan: $\Theta(n)$ pada kasus terburuk.

$O(\lg n) + \Theta(n) = \Theta(n)$. Separuh yang murah tidak menyelamatkan separuh yang mahal.

Pada *senarai berantai* terurut kedua separuh itu bertukar peran: menyisipkan sebuah simpul begitu Anda memegang penunjuknya berbiaya $\Theta(1)$, tetapi mencari posisinya berbiaya $\Theta(n)$ sebab Anda harus berjalan. Bagaimanapun juga totalnya $\Theta(n)$. Pernyataannya salah untuk kedua wakilan senarai itu, dan itulah sebabnya ia salah tanpa syarat apa pun.

Lalu apa yang *memang* $O(\lg n)$? Penyisipan ke dalam pohon pencarian seimbang — pohon AVL atau pohon merah-hitam, Bab 6 — atau ke dalam *skip list*. Struktur-struktur itu ada justru karena larik terurut dan senarai terurut tidak sanggup mengerjakannya.

16.3.8 Pernyataan (9), (10), dan (12) — yang dijamin berhadapan dengan yang lazim

(9) "Fungsi hash yang baik menyebarkan unsur-unsur secara seragam ke seluruh tabel hash." **Benar** — itulah arti *baik*. Asumsi *hashing seragam sederhana* (*simple uniform hashing*) mengatakan tiap kunci sama mungkinnya mendarat di mana pun dari m slotnya, saling bebas satu sama lain.

(10) "Dengan fungsi hash yang baik dan data yang wajar, pencarian dalam tabel hash dapat dikerjakan dalam $O(1)$." **Benar.** Di bawah *hashing* seragam sederhana dengan perantaian, waktu pencarian yang diharapkan adalah $\Theta(1 + \alpha)$, dengan $\alpha = n/m$ faktor muatnya. Jika tabelnya diubah ukurannya sehingga α tetap di bawah sebuah konstanta — tabel dinamis Persoalan 16.10, diterapkan pada tabel *hash* — biaya harapannya $\Theta(1)$. Perhatikan kata-kata yang cermat itu: *diharapkan*, dan *dapat*. Kasus terburuknya, ketika setiap kunci bertabrakan, adalah $\Theta(n)$ — dan pernyataannya, yang berbunyi "dapat dikerjakan", sejalan dengan itu.

(12) "Anda selalu sebaiknya memilih algoritma dengan kompleksitas asimtotik yang lebih baik." **Salah**, dan kata yang bekerja di sini adalah *selalu*. Empat contoh sanggahan baku, dan salah satunya saja sudah cukup:

1. **Konstanta.** Pengurutan sisip $\Theta(n^2)$ dan merge sort $\Theta(n\lg n)$, namun setiap rutin pengurutan industri beralih ke pengurutan sisip di bawah sekitar enam belas unsur, sebab konstantanya lebih kecil dan datanya muat di *cache*. Bab 5.
2. **Memori.** Merge sort butuh ruang tambahan $\Theta(n)$; heapsort butuh $\Theta(1)$. Pada peranti dengan RAM 64 KB, heapsort menang sekalipun asimtotiknya seri.

3. **Ukuran masukan.** Asimtotik berarti "untuk n yang cukup besar". Jika n selalu 30, asimtotiknya sama sekali bukan tentang persoalan Anda.
4. **Sifat lain.** Merge sort stabil; quicksort dan heapsort tidak. Jika spesifikasinya menuntut kestabilan, asimtotiknya tidak punya hak suara.

Untuk kasus pohon merentang minimum yang disebut pernyataan itu secara tersurat: Kruskal $O(E\lg E)$ dan Prim dengan *Fibonacci heap* $O(E + V\lg V)$, yang secara asimtotik lebih baik — namun Kruskal sering lebih cepat pada graf jarang dan jauh lebih sederhana untuk diimplementasikan dengan benar, sedangkan konstanta *Fibonacci heap* terkenal buruk. Batas yang lebih baik bukanlah pilihan yang lebih baik.

16.3.9 Pernyataan (6) — mengapa bobot yang berbeda semua memaksa MST tunggal

"Jika semua bobot busur berbeda, pohon merentang minimumnya tunggal." **Benar.** Yang ini pantas mendapat bukti sungguhan; ia satu-satunya pernyataan dari kelima belasnya yang memang membutuhkannya.

Andaikan semua bobot sisinya berbeda dan, untuk kontradiksi, bahwa T_1 dan T_2 adalah dua pohon merentang minimum yang berbeda. Karena keduanya berbeda, ada sisi yang terletak di salah satunya dan tidak di yang lain. Di antara semua sisi dalam selisih simetris $(T_1 \setminus T_2) \cup (T_2 \setminus T_1)$, misalkan e adalah yang **berbobot terkecil**; bobotnya berbeda semua, jadi e tunggal. Tanpa mengurangi keumuman, $e \in T_1$ dan $e \notin T_2$.

Tambahkan e ke T_2. Sebuah pohon ditambah satu sisi memuat tepat satu siklus, sebutlah C, dan C melewati e. Tidak setiap sisi C dapat berada di T_1, sebab T_1 taksiklik dan memuat e. Jadi ada sisi $f \in C$ dengan $f \in T_2$ dan $f \notin T_1$ — artinya f juga berada di dalam selisih simetrisnya. Karena e adalah sisi teringan di selisih simetris itu dan bobotnya berbeda semua, $w(e) < w(f)$.

Kini bentuklah $T_3 = T_2 \cup \{e\} \setminus \{f\}$. Membuang sebuah sisi dari siklus tunggalnya menyisakan pohon merentang, jadi T_3 adalah pohon merentang, dan

$$w(T_3) = w(T_2) + w(e) - w(f) < w(T_2),$$

yang bertentangan dengan keminimuman T_2. Karena itu tidak ada dua pohon merentang minimum yang berbeda. ■

> **WAWASAN KUNCI: KEBALIKANNYA SALAH, DAN MENGETAHUI ITU TERMASUK BAGIAN DARI JAWABANNYA.** Bobot yang berbeda semua *memaksa* ketunggalan, tetapi bobot yang kembar tidak memaksa kejamakan. Graf berenam titik pada Subbab 16.6.7 punya tiga sisi berbobot 5 dan tetap punya tepat satu pohon merentang minimum — pencacahan gaya kasar atas seluruh $\binom{15}{5}$ himpunan bagian berlima sisi menemukan tepat satu yang berbobot minimum. Bobot kembar membuat MST kedua *mungkin*, bukan *pasti*.

16.3.10 Pernyataan (11), (14), dan (15) — batas, cacahan, dan definisi

(11) "Jika sebuah kode mempunyai kompleksitas asimtotik $O(g(n))$, maka paling sedikit $g(n)$ operasi akan dijalankan." **Salah**, dan inilah pernyataan yang paling sering dijawab keliru oleh pembaca yang sebenarnya fasih.

$f(n) = O(g(n))$ berarti: terdapat $c > 0$ dan n_0 sedemikian sehingga $0 \le f(n) \le c\,g(n)$ untuk semua $n \ge n_0$. Ia batas **atas**. Ia mengatakan kodenya mengerjakan *paling banyak* sekitar $g(n)$ operasi, bukan paling sedikit. Secara konkret, program yang mencetak "hello" sekali lalu berhenti

adalah $O(n^2)$ — definisinya terpenuhi dengan $c = 1$ — dan ia menjalankan sejumlah konstan operasi, bukan n^2 operasi. Pernyataan yang boleh Anda buat tentang *paling sedikit* adalah Ω: $f(n) = \Omega(g(n))$ berarti $f(n) \geq c\, g(n)$ pada akhirnya.

(14) "Banyaknya simpul maksimum pada pohon biner bertinggi H (dengan $H = 0$ untuk daun) adalah $2^{H+1} - 1$." **Benar.** Aras ke-i sebuah pohon biner memuat paling banyak 2^i simpul — aras 0 memuat akarnya, dan tiap simpul punya paling banyak dua anak, jadi tiap aras paling banyak berlipat dua. Pohon bertinggi H punya aras 0 sampai H, jadi paling banyak

$$\sum_{i=0}^{H} 2^i = 2^{H+1} - 1$$

simpul, menurut jumlah geometri Bab 3. Periksalah kasus-kasus kecilnya: $H = 0$ memberi $2^1 - 1 = 1$, satu simpul tunggal; $H = 1$ memberi $2^2 - 1 = 3$, satu akar dan dua anak; $H = 2$ memberi 7. Benar.

Perhatikan konvensinya. Buku ini menetapkan tinggi sebuah daun sebagai 0; sebagian pustaka menetapkannya 1, dan batasnya lalu berbunyi $2^H - 1$. Jika sebuah persoalan tidak menyatakan konvensinya, nyatakanlah yang Anda pakai. Di sini pernyataannya memberikannya, di dalam kurung, dan kurung itulah seluruh alasan mengapa pangkatnya $H + 1$.

(15) "Pada pohon biner lengkap, hanya simpul daun yang tidak mempunyai anak." **Benar**, dan ia benar dengan jauh lebih murah daripada dugaan kebanyakan pembaca: sebuah *daun* **didefinisikan** sebagai simpul yang tak beranak, jadi "hanya daun yang tak beranak" berlaku di setiap pohon apa pun, lengkap maupun tidak. Sebagian pembaca mencari jebakan di sini lalu mengarang satu. Tidak ada jebakannya. Jawab B lalu lanjut — itu tiga puluh detik yang dihemat untuk pernyataan (11), yang memang membutuhkannya.

16.3.11 Pernyataan (13) — yang dapat Anda sanggah dalam lima detik

"Tidak mungkin penelusuran mendalam dan penelusuran melebar sebuah graf mengunjungi simpul-simpulnya dalam urutan yang sama jika graf itu memuat lebih dari dua simpul." **Salah.**

Dua sanggahan, dan salah satunya saja sudah cukup:

- **Sebuah lintasan.** Ambillah simpul $1 - 2 - 3$ lalu mulailah di 1. Penelusuran melebar mengunjungi 1, lalu 2, lalu 3. Penelusuran mendalam mengunjungi 1, menyelam ke 2, menyelam ke 3. Urutan yang sama, tiga simpul.
- **Sebuah bintang.** Ambillah pusat c yang tersambung ke a, b, d, lalu mulailah di c. Melebar: c, a, b, d. Mendalam: c, turun ke a, naik lagi, turun ke b, naik lagi, turun ke d — c, a, b, d. Urutan yang sama, empat simpul.

Alih-alih mustahil, kesamaan itu justru *lazim*. **Kode 16.1** mencacah setiap graf pada empat simpul berlabel yang simpul 0-nya dapat menjangkau semua simpul lain, lalu menghitung seberapa sering kedua urutan itu berimpit.

Kode 16.1 Sanggahan menyeluruh atas pernyataan (13): setiap graf pada empat simpul berlabel dibangun, lalu urutan mendalam dan urutan melebar dari simpul 0 dibandingkan.

```
// Persoalan 16.1, pernyataan (13), mengaku bahwa urutan pencarian mendalam dan
// pencarian melebar tak pernah dapat sama pada graf bersimpul lebih dari dua.
// Program ini membangun setiap graf pada empat simpul berlabel lalu mencacah
// yang urutannya sama, mulai dari simpul 0 dan selalu memindai tetangga dalam
// urutan menaik.
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

const int n = 4;
bool adj[n][n];

void dfs(int u, vector<bool>& seen, vector<int>& out) {
    seen[u] = true; out.push_back(u);
    for (int v = 0; v < n; ++v) if (adj[u][v] && !seen[v]) dfs(v, seen, out);
}
vector<int> bfs(int s) {
    vector<bool> seen(n, false); vector<int> out;
    queue<int> Q; Q.push(s); seen[s] = true;
    while (!Q.empty()) {
        int u = Q.front(); Q.pop(); out.push_back(u);
        for (int v = 0; v < n; ++v) if (adj[u][v] && !seen[v]) { seen[v] = true; Q.push(v);
}
    }
    return out;
}

int main() {
    vector<pair<int,int>> pairs;
    for (int i = 0; i < n; ++i) for (int j = i + 1; j < n; ++j) pairs.push_back({i, j});
    int m = (int)pairs.size(), agree = 0, connected = 0, shown = 0;

    for (int mask = 0; mask < (1 << m); ++mask) {
        for (int i = 0; i < n; ++i) for (int j = 0; j < n; ++j) adj[i][j] = false;
        for (int k = 0; k < m; ++k) if (mask >> k & 1)
            adj[pairs[k].first][pairs[k].second] = adj[pairs[k].second][pairs[k].first] =
true;
```

Kode 16.1 (Lanjutan) *Sanggahan menyeluruh atas pernyataan (13): setiap graf pada empat simpul berlabel dibangun, lalu urutan mendalam dan urutan melebar dari simpul 0 dibandingkan.*

```
        vector<bool> seen(n, false); vector<int> D;
        dfs(0, seen, D);
        if ((int)D.size() != n) continue;                // capai keempatnya, atau lewati
        ++connected;
        vector<int> B = bfs(0);
        if (D == B) {
            ++agree;
            if (shown < 3) {
                ++shown;
                cout << "edges:";
                for (int k = 0; k < m; ++k) if (mask >> k & 1)
                    cout << " " << pairs[k].first << "-" << pairs[k].second;
                cout << "   order:";
                for (int v : D) cout << " " << v;
                cout << "\n";
            }
        }
    }
    cout << "\nconnected graphs on 4 labeled vertices reachable from 0: " << connected
         << "\nof those, DFS order = BFS order in " << agree << " of them\n";
    return 0;
}
```

Dikompilasi dengan `g++ -std=c++17 -O2 -Wall` lalu dijalankan, ia mencetak:

```
edges: 0-1 0-2 0-3   order: 0 1 2 3
edges: 0-1 0-2 0-3 1-2   order: 0 1 2 3
edges: 0-2 0-3 1-3   order: 0 2 3 1

connected graphs on 4 labeled vertices reachable from 0: 38
of those, DFS order = BFS order in 26 of them
```

Dua puluh enam dari tiga puluh delapan. Pernyataannya bukan sekadar salah; ia salah pada sebagian besar kasusnya. Satu kalimat dan satu lintasan bersimpul tiga sudah merupakan jawaban yang lengkap — tetapi memuaskan juga mengetahui seberapa keliru klaim itu.

16.3.12 Kesalahan yang paling sering terjadi pada persoalan ini

Menjawab pernyataan yang diduga alih-alih pernyataan yang tertulis. Pernyataan (3) dibaca sebagai "pencarian biner pada larik terurut adalah $O(\log n)$" — yang benar — alih-alih apa yang tertulis, yang berbicara tentang pencarian *linear*. Pernyataan (11) dibaca sebagai "$O(g(n))$ berarti kira-kira $g(n)$ operasi". Pernyataan (12) dibaca tanpa kata *selalu*.

Pertahanannya berbiaya tiga puluh detik: **garis bawahilah kata kerja dan kata pembatas pada setiap pernyataan sebelum Anda menjawabnya.** Pada (3) frasa kerjanya *pencarian linear*. Pada (11) pembatasnya *paling sedikit*. Pada (12) pembatasnya *selalu*. Pada (13) pembatasnya *tidak mungkin*. Empat garis bawah, dan empat jawaban yang jika tidak demikian hanyalah lemparan koin.

Uji diri. Putuskan, dengan alasan satu baris: (a) "Dijkstra berjalan dalam $O(E\lg V)$ dengan *binary heap*, jadi ia selalu lebih cepat daripada Bellman–Ford." (b) "Merelaksasi sebuah sisi tidak pernah dapat menaikkan $d[v]$." (c) "Biaya amortisasi sebuah operasi tidak pernah lebih kecil daripada biaya sebenarnya yang termurah."

Jawaban. (a) **Salah** — batasnya benar, tetapi Dijkstra sama sekali tidak dapat dipakai ketika ada bobot negatif, dan kata *selalu* mengabaikan hal itu; pernyataan (12) lagi. (b) **Benar** — relaksasi hanya memberikan $d[u] + w(u, v)$ ketika nilai itu *lebih kecil* daripada $d[v]$ (Persamaan (16.1)). (c) **Salah** — dalam analisis potensial tumpukan, `POP` berbiaya sebenarnya 1 dan berbiaya amortisasi 0; biaya amortisasi sebuah operasi murah dapat berada di bawah biaya sebenarnya karena operasi-operasi sebelumnya sudah membayarnya di muka.

16.4 Persoalan 16.2 — Subbarisan Menaik Monoton Terpanjang

16.4.1 Persoalannya

Persoalan 16.2. Rancanglah algoritma berwaktu polinomial untuk menemukan subbarisan menaik monoton terpanjang (*longest monotonically increasing subsequence*) dalam barisan n bilangan bulat yang berbeda semua; setiap bilangan muncul sekali dalam masukan. Berikan asumsi Anda, pseudokodenya, dan analisis kompleksitas waktunya.

16.4.2 Apa yang diuji

Tiga hal, dalam urutan yang disebutkan persoalannya.

Asumsi menguji apakah Anda dapat menyatakan apa itu *subbarisan* tanpa diberi tahu. Inilah seluruh kesulitan persoalan ini, bersembunyi dalam satu kata. Sebuah **subbarisan** (*subsequence*) diperoleh dengan menghapus nol unsur atau lebih dan mempertahankan sisanya dalam urutan

aslinya; ia tidak harus bersebelahan. Sebuah **substring** atau **sublarik** (*subarray*) bersebelahan. Selesaikan persoalan yang bersebelahan, dan Anda akan menghasilkan algoritma $O(n)$ yang benar untuk persoalan yang tidak ditanyakan siapa pun.

Pseudokode menguji apakah Anda dapat menyatakan sebuah program dinamis. Bab 7 memberi empat langkah: mencirikan struktur penyelesaian optimal, menuliskan rekurensi untuk nilainya, menghitung nilainya dari bawah ke atas, dan menyusun penyelesaiannya dari pilihan-pilihan yang dicatat. Langkah keempat — rekonstruksi — adalah yang paling sering terlupa, karena mudah sekali menghitung *panjangnya* lalu lupa bahwa persoalannya meminta *subbarisannya*.

Analisis kompleksitas waktu menguji apakah Anda dapat menghitung dua gelung bersarang dengan jujur, dan apakah Anda memperhatikan bahwa rekonstruksinya gratis.

Kata *polinomial* dalam persoalan itu adalah lantai, bukan sasaran. $O(n^2)$ adalah polinomial dan merupakan jawaban yang wajar. Subbab 16.4.9 memberikan $O(n\lg n)$, yang lebih baik, dan yang memperlihatkan apa artinya melangkah melewati jawaban benar yang pertama.

16.4.3 Asumsi, dinyatakan dengan benar

Tuliskan lima baris ini; masing-masing menutup satu pintu yang dapat dimasuki jawaban keliru.

1. Masukannya adalah $S[1..n]$, sebuah larik berisi n bilangan bulat.
2. Kesemua n bilangan itu **berbeda** — persoalannya mengatakan demikian. Ini penting: dengan unsur kembar Anda harus memilih antara menaik *tegas* dan *tak menurun*, dan keduanya memberi jawaban yang berbeda. Keberbedaan membuat persoalannya tidak ambigu.
3. *Menaik* berarti menaik **tegas**: $S[i_1] < S[i_2] < \cdots < S[i_k]$.
4. Sebuah **subbarisan** mempertahankan urutan relatif tetapi tidak harus bersebelahan: indeks $1 \le i_1 < i_2 < \cdots < i_k \le n$.
5. Barisan kosong dan setiap unsur tunggal menaik secara trivial, jadi jawabannya berpanjang paling sedikit 1 setiap kali $n \ge 1$.

16.4.4 Menurunkan rekurensinya

Penghalang bagi rekursi langsung adalah bahwa "subbarisan menaik terpanjang dari $S[1..i]$" bukan besaran yang dapat Anda perpanjang. Mengetahui bahwa $S[1..5]$ punya subbarisan menaik terpanjang berpanjang 3 tidak memberi tahu apa pun tentang apakah $S[6]$ dapat ditambahkan, sebab Anda tidak tahu subbarisan itu *berakhir dengan* apa.

Perbaikannya adalah perbaikan baku untuk keluarga persoalan ini: **perkuat subpersoalannya sampai ia dapat disusun.** Definisikan

$$L[i] = \text{panjang subbarisan menaik terpanjang dari } S[1..i] \text{ yang berakhir tepat}$$

Kata-kata *berakhir tepat di* $S[i]$ adalah seluruh wawasannya. Dengan kata-kata itu, penyusunan bekerja.

Substruktur optimal. Misalkan T adalah subbarisan menaik terpanjang yang berakhir di $S[i]$, dan andaikan panjangnya ≥ 2. Misalkan $S[j]$ adalah unsur tepat sebelum $S[i]$ di dalam T. Maka $j < i$ dan $S[j] < S[i]$, dan bagian T sampai $S[j]$ adalah subbarisan menaik dari $S[1..j]$ yang berakhir di $S[j]$. Lebih dari itu, ia harus merupakan subbarisan *terpanjang* semacam itu: jika ada yang lebih panjang yang berakhir di $S[j]$, kita dapat menukarnya masuk dan memperoleh subbarisan berakhir di $S[i]$ yang lebih panjang, bertentangan dengan pilihan T. Itulah argumen potong-dan-tempel (*cut-and-paste*) Bab 7, diterapkan di sini.

Jadi panjang T adalah $L[j] + 1$, dan karena j dapat berupa indeks mana pun dengan $j < i$ dan $S[j] < S[i]$, kita ambil yang terbaik:

$$L[i] = \begin{cases} \max\limits_{j<i \,\wedge\, S[j]<S[i]} (L[j]+1) & \text{jika } j \text{ seperti itu ada,} \\ 1 & \text{selainnya.} \end{cases} \tag{16.2}$$

Jawabannya. Subbarisan menaik terpanjang dari seluruh larik berakhir *di suatu tempat*, jadi panjangnya adalah $\max_{1\leq i\leq n} L[i]$.

Rekonstruksi. Simpanlah $P[i]$, indeks j yang mencapai maksimum untuk $L[i]$, atau 0 jika $L[i] = 1$. Mengikuti P dari k yang memaksimumkan kembali ke 0 mendaftar subbarisannya secara terbalik.

Invarian gelung yang membuat perhitungan dari bawah ke atas atas Persamaan (16.2) benar: *pada awal iterasi untuk indeks i, $L[1..i-1]$ dan $P[1..i-1]$ berisi nilai yang benar untuk subpersoalannya.* Inisialisasi berlaku secara hampa pada $i = 1$; pemeliharaan adalah rekurensi di atas, yang hanya membaca entri $j < i$; terminasi pada $i = n + 1$ memberikan seluruh tabelnya. Itulah kewajiban tiga bagian Bab 2, dilunasi dalam tiga baris.

16.4.5 Pseudokodenya, dibaca baris demi baris

Pseudokode 16.1 adalah algoritmanya, berikut penelusuran balik yang mengubah tabel panjang menjadi subbarisan yang sesungguhnya.

Pseudokode 16.1 Program dinamis $O(n^2)$ untuk subbarisan menaik terpanjang, disusul penelusuran balik yang memulihkan subbarisannya sendiri.

```
LIS(S, n)
 1  for i = 1 to n
 2      L[i] = 1
 3      P[i] = 0
 4      for j = 1 to i - 1
 5          if S[j] < S[i] and L[j] >= L[i]
 6              L[i] = L[j] + 1
 7              P[i] = j
 8  let k be an index with the largest value L[k] among L[1..n]
 9  i = 1                                        // penelusuran balik
10  j = k
11  repeat
12      T[i] = S[j]
13      i = i + 1;  j = P[j]
14  until j == 0
15  return L[k] and the reverse of T[1..L[k]]
```

Dua rincian di baris 5 layak mendapat satu kalimat masing-masing, sebab keduanya tampak seperti kekeliruan padahal bukan.

Mengapa `L[j] >= L[i]` dan bukan `L[j] + 1 > L[i]`? Keduanya uji yang sama. Menambahkan satu pada kedua ruas $L[j] \geq L[i]$ memberi $L[j] + 1 \geq L[i] + 1 > L[i]$, dan sebaliknya $L[j] + 1 > L[i]$ atas bilangan bulat berarti $L[j] \geq L[i]$. Bentuk yang pertama menghemat satu penjumlahan. Keduanya benar; ada baiknya Anda memeriksa sendiri bahwa keduanya sepakat.

Mengapa ia menyimpan j memenuhi syarat yang *terakhir*, bukan yang pertama? Karena ujinya $\geq$ dan bukan $>$, sebuah j yang lebih belakang dengan $L[j]$ yang sama menimpa yang lebih depan. Ini mengubah subbarisan terpanjang *mana* yang dilaporkan, bukan panjangnya. Keduanya jawaban yang sah.

Sebuah kekeliruan yang menggoda, dan mengapa ia tak berbahaya di sini tetapi tetap harus diperbaiki. Menggoda untuk menulis rekurensinya dengan rentang $1 \leq j \leq i$, yang mengizinkan $j = i$. Tidak ada kerusakan — syarat $S[j] < S[i]$ salah ketika $j = i$, sebab tak ada bilangan yang lebih kecil daripada dirinya sendiri — tetapi rentang yang benar adalah $1 \leq j < i$, dan itulah yang sungguh diimplementasikan baris 4 **Pseudokode 16.1**. Sebuah rekurensi dan kodenya seharusnya mengatakan hal yang sama; ketika keduanya berbeda tanpa bahaya hari ini, keduanya cenderung berbeda dengan berbahaya sesudah suntingan berikutnya.

16.4.6 Sebuah contoh terkerjakan yang lengkap, dengan tangan

Ambillah

$$S = \langle 9,\ 2,\ 5,\ 3,\ 7,\ 11,\ 8,\ 10,\ 13,\ 6 \rangle, \qquad n = 10.$$

Kerjakan gelung luarnya satu indeks demi satu indeks. Pada tiap langkah kita memindai $j = 1, \ldots, i - 1$ dan menyimpan yang terbaik.

- $i = 1, S[1] = 9$. Tak ada j. $L[1] = 1, P[1] = 0$.
- $i = 2, S[2] = 2$. $j = 1$: apakah $9 < 2$? Tidak. $L[2] = 1, P[2] = 0$.
- $i = 3$, $S[3] = 5$. $j = 1$: $9 < 5$? Tidak. $j = 2$: $2 < 5$ ya, dan $L[2] = 1 \geq L[3] = 1$, jadi $L[3] \leftarrow 1 + 1 = 2, P[3] \leftarrow 2$.
- $i = 4$, $S[4] = 3$. $j = 2$: $2 < 3$ ya, $L[2] = 1 \geq 1$, jadi $L[4] \leftarrow 2$, $P[4] \leftarrow 2$. $j = 3$: $5 < 3$? Tidak. Hasilnya $L[4] = 2, P[4] = 2$.
- $i = 5$, $S[5] = 7$. $j = 2$: $2 < 7$, $L[2] = 1 \geq 1$, jadi $L[5] \leftarrow 2$, $P[5] \leftarrow 2$. $j = 3$: $5 < 7$, $L[3] = 2 \geq 2$, jadi $L[5] \leftarrow 3, P[5] \leftarrow 3$. $j = 4$: $3 < 7$, tetapi $L[4] = 2 \geq 3$ salah. Hasilnya $L[5] = 3, P[5] = 3$.
- $i = 6$, $S[6] = 11$. $j = 1$: $9 < 11$, $L[1] = 1 \geq 1$, $L[6] \leftarrow 2$, $P[6] \leftarrow 1$. $j = 3$: $5 < 11$, $L[3] = 2 \geq 2$, $L[6] \leftarrow 3, P[6] \leftarrow 3$. $j = 5$: $7 < 11, L[5] = 3 \geq 3, L[6] \leftarrow 4, P[6] \leftarrow 5$. Hasilnya $L[6] = 4, P[6] = 5$.
- $i = 7$, $S[7] = 8$. j yang memenuhi syarat adalah 2,3,4,5; yang terbaik $j = 5$ dengan $L[5] = 3$. Hasilnya $L[7] = 4, P[7] = 5$.
- $i = 8, S[8] = 10$. j yang memenuhi syarat adalah 1,2,3,4,5,7; yang terakhir memperbaiki adalah $j = 7$ dengan $L[7] = 4$. Hasilnya $L[8] = 5, P[8] = 7$.
- $i = 9, S[9] = 13$. Perbaikan terakhir adalah $j = 8$ dengan $L[8] = 5$. Hasilnya $L[9] = 6, P[9] = 8$.
- $i = 10$, $S[10] = 6$. j yang memenuhi syarat adalah 2,3,4; yang terbaik $j = 3$ dengan $L[3] = 2$. Hasilnya $L[10] = 3, P[10] = 3$.

Tabel 16.3 menghimpun kedua lariknya.

Tabel 16.3 Tabel pemrograman dinamis untuk $S = \langle 9,2,5,3,7,11,8,10,13,6 \rangle$. $L[i]$ adalah panjang subbarisan menaik terpanjang yang berakhir di $S[i]$; $P[i]$ adalah indeks unsur sebelumnya.

i	1	2	3	4	5	6	7	8	9	10
$S[i]$	9	2	5	3	7	11	8	10	13	6
$L[i]$	1	1	2	2	3	4	4	5	6	3
$P[i]$	0	0	2	2	3	5	5	7	8	3

Entri terbesar adalah $L[9] = 6$, jadi $k = 9$. Telusuri balik:

$$j = 9 \to S[9] = 13; \quad j = P[9] = 8 \to 10; \quad j = P[8] = 7 \to 8; \quad j = P[7] = 5 \to 7; \quad j = P[5] = 3 \to 5; \quad j = P[3] = 2 \to 2; \quad j = P[2] = 0 \text{ berhenti.}$$

Membaca nilai-nilai yang terkumpul dari belakang:

$$\langle\, 2,\ 5,\ 7,\ 8,\ 10,\ 13 \,\rangle, \qquad \text{panjang } 6.$$

Periksalah: nilainya menaik, dan semuanya muncul di posisi $2 < 3 < 5 < 7 < 8 < 9$, yang juga menaik. Ia subbarisan menaik yang sah berpanjang 6.

16.4.7 Kompleksitas waktunya, dihitung dengan jujur

Baris 1–7 **Pseudokode 16.1** adalah dua gelung bersarang. Gelung luarnya berjalan n kali; untuk i tertentu gelung dalamnya berjalan $i - 1$ kali. Badannya $\Theta(1)$. Banyaknya seluruh iterasi dalam adalah

$$\sum_{i=1}^{n} (i-1) = \sum_{m=0}^{n-1} m = \frac{(n-1)n}{2} = \frac{n^2 - n}{2},$$

yang $\Theta(n^2)$. Baris 8 dan penelusuran balik baris 9–14 masing-masing membuat satu lintasan — penelusuran baliknya mengunjungi paling banyak n indeks sebab $P[j] < j$ secara tegas, jadi rantainya tidak dapat mengunjungi ulang sebuah indeks — memberi $\Theta(n)$ masing-masing. Karena itu

$$T(n) = \Theta(n^2) + \Theta(n) + \Theta(n) = \Theta(n^2).$$

Ruangnya $\Theta(n)$ untuk L, P, dan T bersama-sama. Menyebut $O(n^2)$ itu benar; $\Theta(n^2)$ lebih tajam dan sama benarnya, sebab gelung dalamnya selalu berjalan sampai tuntas — tidak ada jalan keluar awal, jadi tidak ada kasus terbaik yang lebih cepat.

Apakah n^2 cukup cepat? Aturan Bab 4 mengatakan: bacalah kendalanya sebagai anggaran. Pada $n = 10^4$ kita mengerjakan sekitar 5×10^7 iterasi dalam, jauh di bawah satu detik. Pada $n = 10^6$ kita akan mengerjakan 5×10^{11} — berjam-jam. Itulah alasan praktis untuk mengenal Subbab 16.4.9.

16.4.8 Programnya

Kode 16.2 adalah **Pseudokode 16.1** sebagai program C++ yang lengkap, mencetak seluruh tabelnya agar dapat dicocokkan dengan **Tabel 16.3** entri demi entri.

Kode 16.2 Algoritma $O(n^2)$ Persoalan 16.2 dalam C++, mencetak tabel L dan P.

```
// Persoalan 16.2: subbarisan menaik monoton terpanjang, dengan pemrograman
// dinamis O(n^2) dari Subbab 16.4.5, dan seluruh tabelnya diperlihatkan.
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

int main() {
    vector<int> S = {9, 2, 5, 3, 7, 11, 8, 10, 13, 6};
    int n = (int)S.size();

    vector<int> L(n + 1, 1), P(n + 1, 0);          // berbasis 1, seperti pseudokodenya

    for (int i = 1; i <= n; ++i) {
        L[i] = 1;
        P[i] = 0;
        for (int j = 1; j <= i - 1; ++j)
            // "L[j] >= L[i]" adalah uji yang sama dengan "L[j]+1 > L[i]"
            if (S[j - 1] < S[i - 1] && L[j] >= L[i]) {
                L[i] = L[j] + 1;
                P[i] = j;
            }
    }

    cout << " i   : ";
    for (int i = 1; i <= n; ++i) cout << setw(4) << i;
    cout << "\n S[i]: ";
    for (int i = 1; i <= n; ++i) cout << setw(4) << S[i - 1];
    cout << "\n L[i]: ";
    for (int i = 1; i <= n; ++i) cout << setw(4) << L[i];
    cout << "\n P[i]: ";
    for (int i = 1; i <= n; ++i) cout << setw(4) << P[i];
    cout << "\n";

    int k = 1;
    for (int i = 2; i <= n; ++i) if (L[i] > L[k]) k = i;   // L[k] terbesar yang pertama

    vector<int> T;
    for (int j = k; j != 0; j = P[j]) T.push_back(S[j - 1]);

    cout << "\nlength = " << L[k] << "\nsubsequence =";
    for (int i = (int)T.size() - 1; i >= 0; --i) cout << " " << T[i];
    cout << "\n";
    return 0;
}
```

Dikompilasi dengan `g++ -std=c++17 -O2 -Wall` lalu dijalankan:

```
 i   :    1   2   3   4   5   6   7   8   9  10
 S[i]:    9   2   5   3   7  11   8  10  13   6
 L[i]:    1   1   2   2   3   4   4   5   6   3
 P[i]:    0   0   2   2   3   5   5   7   8   3

length = 6
subsequence = 2 5 7 8 10 13
```

Setiap entrinya sepakat dengan perhitungan tangan pada **Tabel 16.3**.

16.4.9 Cara kedua: $O(n\lg n)$

Ada algoritma yang lebih baik, dan ia layak mendapat setengah halaman, sebab ia memperlihatkan bagaimana gagasan kedua — sebuah larik bantu yang terurut — menghapus seluruh satu faktor n.

Peliharalah larik `tail`, dengan `tail[k]` adalah **nilai terkecil yang dapat mengakhiri subbarisan menaik berpanjang $k+1$ di antara unsur-unsur yang sudah terlihat.** Dua fakta membuat cara ini bekerja:

1. **`tail` terurut.** Jika subbarisan berpanjang $k+1$ dapat berakhir di nilai v, maka k unsur pertamanya membentuk subbarisan berpanjang k yang berakhir di sesuatu yang lebih kecil daripada v; jadi `tail[k-1] < tail[k]`.
2. **Setiap unsur baru menggantikan tepat satu entri.** Diberikan $S[i]$, carilah posisi pertama k dengan `tail[k] >= S[i]` — sebuah pencarian biner, sebab `tail` terurut. Jika posisi seperti itu tidak ada, $S[i]$ memperpanjang deretan terpanjang dan ditambahkan di ujung. Jika ada, `tail[k]` digantikan oleh $S[i]$, yang lebih kecil dan karena itu paling sedikit sama bergunanya untuk masa depan, dan tidak pernah merugikan entri yang lebih pendek maupun yang lebih panjang.

Panjang `tail` pada akhirnya adalah panjang subbarisan menaik terpanjang. Untuk memulihkan subbarisannya sendiri, catatlah untuk setiap unsur indeks unsur yang mendahuluinya — yaitu `tailIdx[k-1]` pada saat penyisipan — lalu ikutilah rantainya ke belakang.

Biaya: n pencarian biner pada larik berpanjang paling banyak n, jadi $O(n\lg n)$, dengan ruang $\Theta(n)$. **Kode 16.3** mengimplementasikannya.

Kode 16.3 Cara $O(n\lg n)$ untuk Persoalan 16.2, mencetak larik `tail` sesudah setiap unsur.

```
// Persoalan 16.2, cara kedua: subbarisan menaik terpanjang yang sama, dalam
// O(n lg n), dengan patience sorting berikut pencarian biner.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> S = {9, 2, 5, 3, 7, 11, 8, 10, 13, 6};
    int n = (int)S.size();

    vector<int> tail;            // tail[k] = ekor terkecil yang mungkin bagi subbarisan
                                 // menaik berpanjang k+1 yang terlihat sejauh ini
    vector<int> tailIdx, parent(n, -1);

    for (int i = 0; i < n; ++i) {
        // posisi pertama yang ekornya >= S[i]; menaik tegas, jadi lower_bound
        int k = (int)(lower_bound(tail.begin(), tail.end(), S[i]) - tail.begin());
        if (k > 0) parent[i] = tailIdx[k - 1];
        if (k == (int)tail.size()) { tail.push_back(S[i]); tailIdx.push_back(i); }
        else                       { tail[k] = S[i];       tailIdx[k] = i;       }
        cout << "after S[" << i + 1 << "]=" << S[i] << "  tail =";
        for (int t : tail) cout << " " << t;
        cout << "\n";
    }

    vector<int> out;
    for (int i = tailIdx.back(); i != -1; i = parent[i]) out.push_back(S[i]);
    reverse(out.begin(), out.end());

    cout << "\nlength = " << tail.size() << "\nsubsequence =";
    for (int v : out) cout << " " << v;
    cout << "\n";
    return 0;
}
```

Keluarannya:

```
after S[1]=9  tail = 9
after S[2]=2  tail = 2
after S[3]=5  tail = 2 5
after S[4]=3  tail = 2 3
after S[5]=7  tail = 2 3 7
after S[6]=11  tail = 2 3 7 11
after S[7]=8  tail = 2 3 7 8
after S[8]=10  tail = 2 3 7 8 10
after S[9]=13  tail = 2 3 7 8 10 13
after S[10]=6  tail = 2 3 6 8 10 13

length = 6
subsequence = 2 3 7 8 10 13
```

Dua hal yang patut diperhatikan.

Pertama, **jawabannya sama panjang tetapi subbarisannya berbeda**: ⟨2,3,7,8,10,13⟩ berhadapan dengan ⟨2,5,7,8,10,13⟩ milik program dinamisnya. Keduanya subbarisan menaik terpanjang. Subbarisan menaik terpanjang tidak tunggal, dan persoalan yang meminta "subbarisan" terpanjang itu sebenarnya meminta *sebuah* subbarisan terpanjang. Menyatakan hal ini memperlihatkan bahwa Anda memahami apa yang Anda hitung.

Kedua, `tail` **sendiri bukan subbarisan dari** S**.** Sesudah unsur terakhir, `tail` berbunyi 2,3,6,8,10,13 — tetapi 6 muncul di posisi 10, sesudah 8, 10, dan 13. Larik itu adalah tabel akhiran terbaik yang diketahui, bukan jawaban. Mencetak `tail` sebagai jawaban adalah kekeliruan klasik dengan cara ini; rekonstruksi lewat `parent`-lah yang menghasilkan subbarisan yang sesungguhnya.

16.4.10 Kesalahan yang paling sering terjadi pada persoalan ini

Yang pertama adalah menyelesaikan persoalan yang **bersebelahan**. "Deretan menaik terpanjang" untuk masukan yang sama adalah ⟨2,5⟩ atau ⟨3,7,11⟩ — panjang 3, bukan 6 — dan ia dapat dihitung dalam satu lintasan. Algoritma $O(n)$ yang rapi dengan analisis yang benar tetap keliru jika ia menjawab persoalan yang lain.

Pertahanannya adalah asumsi keempat dari lima asumsi di Subbab 16.4.3. Menuliskan "subbarisan tidak harus bersebelahan" di puncak jawaban memaksa Anda menyadari, sebelum merancang apa pun, persoalan mana yang sedang Anda selesaikan.

Yang kedua adalah menghitung $\max_i L[i]$ lalu berhenti. Persoalannya berbunyi *temukan subbarisan menaik monoton terpanjang*, bukan *temukan panjangnya*. Tanpa larik P dan penelusuran baliknya, Anda menjawab persoalan yang lebih kecil.

Uji diri. Jalankan **Pseudokode 16.1** dengan tangan pada $S = \langle 3,1,8,2,5,9,4 \rangle$. Berikan L, P, dan subbarisannya.

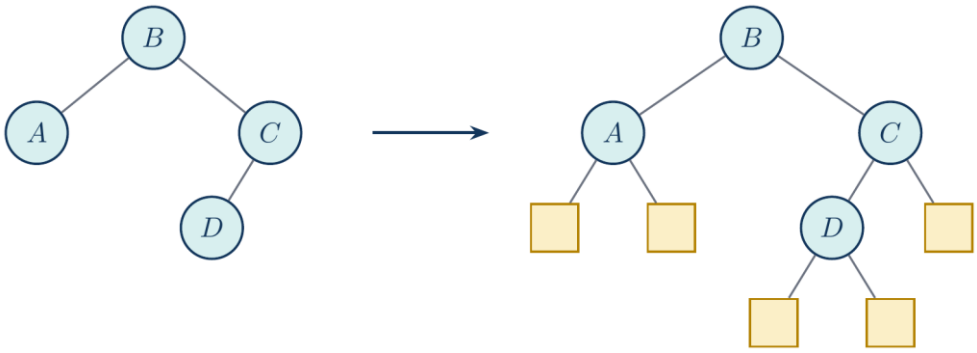
Jawaban. $L = \langle 1,1,2,2,3,4,3 \rangle$ dan $P = \langle 0,0,1,2,4,5,4 \rangle$. (Pada $i = 3$, $j = 1$ menetapkan $L[3] = 2$; lalu $j = 2$ mempunyai $L[2] = 1 \geq 2$ yang salah, jadi $P[3]$ tetap 1 — uji $\geq$ hanya mengutamakan j yang lebih belakang jika ia paling sedikit sama baiknya. Pada $i = 5$, $S[5] = 5$: $j = 4$ dengan $S[4] = 2$ dan $L[4] = 2$ yang menang.) Maksimumnya $L[6] = 4$; penelusuran balik $6 \to 5 \to 4 \to 2$ memberi ⟨1,2,5,9⟩.

16.5 Persoalan 16.3 — Pohon Biner Diperluas, Dibuktikan dengan Induksi

16.5.1 Persoalannya

Pohon biner dipakai untuk pencarian: pencarian dimulai di akar lalu berlanjut ke anak kiri atau anak kanan bergantung pada sebuah perbandingan. Kadang pencarian berhenti di sebuah simpul pohon. Kadang ia mencoba menyeberangi tautan kosong menuju anak yang tidak ada; pencarian itu lalu dikatakan *jatuh keluar* dari pohon. Jatuh keluar menghilangkan informasi — Anda tidak lagi tahu *di mana* hal itu terjadi — padahal pengetahuan itu sering berguna. Maka, untuk pohon biner apa pun, bentuklah **pohon biner diperluas** (*extended binary tree*) dengan mengganti setiap anak yang hilang (setiap penunjuk kosong) dengan sebuah daun khusus, yaitu **simpul luar** (*external node*). Simpul-simpul aslinya lalu disebut **simpul dalam** (*internal node*). Jika pohon aslinya kosong, pohon diperluasnya terdiri atas satu simpul luar. Dalam pohon biner diperluas, setiap simpul dalam mempunyai tepat dua anak dan setiap simpul luar tidak mempunyai anak.

Gambar 16.1 memperlihatkan sebuah contoh, dengan simpul luar digambar sebagai persegi dan simpul dalam sebagai lingkaran.



Gambar 16.1 Sebuah pohon biner biasa dengan empat simpul (kiri) dan pohon biner diperluas yang diperoleh darinya (kanan). Lingkaran adalah simpul dalam; persegi adalah simpul luar yang menggantikan penunjuk kosong. Empat simpul dalam, lima simpul luar.

Persoalan 16.3. Misalkan n menyatakan banyaknya simpul dalam pada sebuah pohon biner diperluas. Buktikan dengan induksi bahwa pohon biner diperluas dengan n simpul dalam mempunyai $n + 1$ simpul luar.

Hitunglah persegi di sisi kanan **Gambar 16.1**: simpul A menyumbang dua, simpul C satu, simpul D dua — lima simpul luar untuk empat simpul dalam. Itulah klaimnya, pada satu contoh.

16.5.2 Apa yang diuji

Apakah Anda dapat menuliskan induksi **atas besaran yang tepat**, dengan hipotesis yang dinyatakan secara cermat.

Jebakannya adalah berinduksi atas tinggi pohon. Induksi atas tinggi gagal di sini, atau paling tidak menjadi menyakitkan, sebab dua subpohon sebuah simpul tidak harus sama tinggi dan hipotesis induksinya harus cukup kuat untuk menampung keduanya. Besaran yang tepat adalah **banyaknya simpul dalam**, dan bentuk yang tepat adalah induksi **kuat**: hipotesisnya harus mencakup *setiap* ukuran yang lebih kecil, sebab pohon dengan n simpul dalam terbelah menjadi subpohon berukuran k dan $n - 1 - k$, dan k dapat berupa apa saja dari 0 sampai $n - 1$.

Ini struktur bukti yang layak dikuasai sepenuhnya. Setiap kali Anda diminta membuktikan sebuah pernyataan tentang pohon biner dengan induksi, struktur umum yang sama berlaku — belah di akar, terapkan hipotesis pada kedua subpohon, jumlahkan — dan Subbab 16.11 memakai kerangka yang sama sekali lagi.

16.5.3 Buktinya

Misalkan $X(n)$ menyatakan banyaknya simpul luar dalam pohon biner diperluas yang banyak simpul **dalam**-nya n. Kita buktikan $X(n) = n + 1$ untuk setiap bilangan bulat $n \geq 0$, dengan induksi kuat atas n.

Kasus dasar, $n = 0$. Pohon biner tanpa simpul adalah pohon kosong. Menurut definisi, memperluas pohon kosong menghasilkan satu simpul luar. Karena itu $X(0) = 1 = 0 + 1$, sebagaimana diminta.

Hipotesis induksi. Misalkan $n \geq 1$ dan anggaplah $X(n') = n' + 1$ untuk setiap n' dengan $0 \leq n' < n$.

Langkah induksi. Tinjaulah pohon biner diperluas dengan $n \geq 1$ simpul dalam. Karena $n \geq 1$, pohon aslinya tidak kosong dan karena itu mempunyai akar, yang merupakan simpul dalam. Buanglah akarnya; $n - 1$ simpul dalam yang tersisa terbagi antara subpohon kiri dan subpohon kanan. Misalkan k adalah banyaknya simpul dalam di subpohon kiri, sehingga subpohon kanan mempunyai $(n - 1) - k$ simpul dalam. Kedua bilangan itu terletak di $[0, n - 1]$, dan keduanya lebih kecil tegas daripada n, jadi hipotesis induksi berlaku untuk masing-masing.

Setiap subpohon itu sendiri adalah pohon biner diperluas, dan simpul luar seluruh pohon tepat merupakan simpul luar kedua subpohonnya — akarnya simpul dalam, dan memperluas seluruh pohon berarti memperluas setiap subpohon dengan cara yang sama. Karena itu

$$X(n) \;=\; X(k) \;+\; X((n-1)-k). \tag{16.3}$$

Terapkan hipotesisnya pada kedua suku Persamaan (16.3):

$$X(n) \;=\; (k+1) \;+\; ((n-1)-k+1) \;=\; (k+1)+(n-k) \;=\; n+1.$$

Perhatikan bahwa k saling meniadakan: hasilnya tidak bergantung pada bagaimana simpul-simpul dalam dibagi, dan justru itulah yang membuat klaimnya benar untuk *setiap* bentuk pohon. Dengan induksi kuat, $X(n) = n + 1$ untuk semua $n \geq 0$. ■

Periksalah langkahnya pada **Gambar 16.1**. Di sana $n = 4$ dengan akar B. Subpohon kirinya adalah simpul tunggal A, jadi $k = 1$ dan $X(1) = 2$. Subpohon kanannya adalah C dengan anak D, jadi $(n - 1) - k = 2$ dan $X(2) = 3$. Jumlahnya $2 + 3 = 5 = n + 1$. Benar.

16.5.4 Tulisan yang menggoda tetapi ceroboh, dan mengapa ia merugi

Bukti di atas pendek, dan justru karena itu mudah ditulis dengan ceroboh. Dua kekeliruan lazim terjadi, dan masing-masing mengubah argumen yang benar menjadi argumen yang tidak dapat dipercaya pembacanya.

- **Mendefinisikan lambangnya secara longgar.** Tulisan yang dibuka dengan *"Misalkan $X(n)$ menyatakan banyaknya simpul luar dalam pohon biner dengan n simpul"* telah mendefinisikan hal yang keliru: klaimnya tentang n simpul **dalam**, dan pohon biner diperluas dengan n simpul dalam mempunyai seluruhnya $2n + 1$ simpul. Jika sisa buktinya diam-diam memakai arti yang

benar, argumennya selamat, tetapi pembacanyalah yang harus memperbaikinya. Definisikan setiap lambang tepat sekali, dan tepat benar.

- **Notasi yang tidak seimbang.** Baris tampilan seperti $X((n) = X(k) + X((n-1) - k)$, dengan kurung yang tidak berpasangan, memaksa pembaca menebak apa maksudnya. Tebakan adalah jalan kesalahan menyebar. Perbaikilah kurungnya.

Tak satu pun kekeliruan itu memengaruhi matematikanya. Keduanya disebutkan karena kebiasaan memeriksa lambang Anda sendiri itulah yang membedakan bukti yang dapat dipercaya dari bukti yang harus diturunkan ulang oleh setiap pembacanya.

16.5.5 Dua bukti alternatif

Bukti kedua layak dimiliki, baik karena ia mungkin lebih cepat maupun karena ia merupakan pemeriksaan yang mandiri atas bukti pertama.

Alternatif 1 — induksi bertahap. Berinduksilah atas n dengan *membangun* pohonnya alih-alih membelahnya. Untuk $n = 0$ pohon diperluasnya adalah satu simpul luar, jadi $X(0) = 1$. Kini andaikan klaimnya berlaku untuk $n - 1$, dan ambillah pohon diperluas mana pun dengan n simpul dalam. Pilihlah sebuah simpul dalam yang kedua anaknya simpul luar — simpul seperti itu selalu ada, sebab mengikuti lintasan mana pun ke bawah pada akhirnya harus mencapai simpul yang tidak mempunyai anak dalam. Gantilah simpul dalam itu dengan satu simpul luar. Hasilnya adalah pohon biner diperluas dengan $n - 1$ simpul dalam; kita menghancurkan dua simpul luar dan menciptakan satu, jadi kita kehilangan tepat satu. Karena itu

$$X(n) - 1 = X(n-1) = n, \qquad \text{jadi} \qquad X(n) = n + 1.$$

Alternatif 2 — menghitung sisi, yang sama sekali bukan induksi. Setiap simpul kecuali akar mempunyai tepat satu sisi ke induknya, jadi pohon dengan N simpul mempunyai $N - 1$ sisi. Dalam pohon biner diperluas setiap simpul dalam mempunyai tepat dua anak dan setiap simpul luar tidak mempunyai anak, jadi banyaknya sisi juga $2n$ — hitunglah setiap sisi dari atas. Dengan $N = n + X(n)$ simpul seluruhnya,

$$n + X(n) - 1 = 2n \quad \Rightarrow \quad X(n) = n + 1.$$

Dua baris. Seandainya persoalannya hanya berbunyi "buktikan", ini sudah jawaban yang lengkap. Karena ia berbunyi "buktikan dengan induksi", berikan induksinya dan tawarkan ini sebagai pemeriksaan — kalimat seperti *"sebagai pemeriksaan, menghitung sisi dari atas memberi* $2n$ *dan dari bawah memberi* $n + X(n) - 1$*, dan keduanya sepakat"*.

16.5.6 Percobaannya

Bukti tidak membutuhkan percobaan, tetapi membangun percobaannya adalah cara Anda mengetahui apakah Anda sudah memahami definisi-definisinya. **Kode 16.4** membangun pohon biner acak dengan banyak simpul dalam yang ditentukan lalu menghitung kedua besarannya.

Kode 16.4 Percobaan di balik Persoalan 16.3: pohon biner acak dengan ukuran dalam tertentu, dengan kedua cacahannya diambil. Perhatikan bahwa `externals` mengembalikan 1 untuk penunjuk kosong — itulah definisi simpul luar, ditulis sebagai kode.

```
// Persoalan 16.3: pohon biner diperluas dengan n simpul dalam punya n+1 simpul luar.
// Buktinya dengan induksi; program ini eksperimen yang dijelaskan bukti itu.
#include <iostream>
#include <iomanip>
#include <vector>
#include <random>
using namespace std;

struct Node { Node *l = nullptr, *r = nullptr; };

// Membangun pohon biner acak dengan tepat n simpul dalam.
Node* build(int n, mt19937& rng) {
    if (n == 0) return nullptr;                 // anak null adalah simpul luar
    int k = uniform_int_distribution<int>(0, n - 1)(rng);   // ukuran subpohon kiri
    Node* t = new Node;
    t->l = build(k, rng);
    t->r = build(n - 1 - k, rng);
    return t;
}
int internals(Node* t) { return t ? 1 + internals(t->l) + internals(t->r) : 0; }
int externals(Node* t) { return t ? externals(t->l) + externals(t->r) : 1; }
void freeTree(Node* t) { if (t) { freeTree(t->l); freeTree(t->r); delete t; } }

int main() {
    mt19937 rng(20251214u);                     // benih tetap: hasil dapat diulang
    cout << "   n   internals   externals   externals - internals\n";
    for (int n : {0, 1, 2, 3, 4, 5, 10, 50, 1000}) {
        Node* t = build(n, rng);
        int in = internals(t), ex = externals(t);
        cout << setw(5) << n << setw(12) << in << setw(12) << ex
             << setw(24) << ex - in << "\n";
        freeTree(t);
    }
    bool ok = true;
    for (int trial = 0; trial < 20000 && ok; ++trial) {
        int n = uniform_int_distribution<int>(0, 60)(rng);
        Node* t = build(n, rng);
        if (externals(t) != internals(t) + 1) ok = false;
        freeTree(t);
    }
    cout << "\n20000 random trees checked: X(n) = n + 1 holds in "
         << (ok ? "every" : "NOT every") << " case\n";
    return 0;
}
```

Keluarannya:

```
   n   internals   externals   externals - internals
   0           0           1                       1
   1           1           2                       1
   2           2           3                       1
   3           3           4                       1
   4           4           5                       1
   5           5           6                       1
  10          10          11                       1
  50          50          51                       1
1000        1000        1001                       1

20000 random trees checked: X(n) = n + 1 holds in every case
```

Baris paling bermakna dalam program itu adalah `int externals(Node* t) { return t ? ... : 1; }`. Kasus dasar rekursinya mengembalikan 1 untuk penunjuk kosong — sebab penunjuk kosong *adalah* simpul luar. Satu karakter itu adalah definisi yang di Subbab 16.5.1 membutuhkan satu paragraf.

16.5.7 Mengapa hasil ini penting

Motivasi persoalannya sendiri menjelaskannya: simpul luar adalah tempat menyimpan informasi yang kalau tidak demikian akan hilang ketika sebuah pencarian jatuh keluar dari pohon. Cacahan $n + 1$ lalu memberi tahu Anda biaya penyimpanannya — satu simpul tambahan per simpul dalam, ditambah satu — jadi pohon diperluas berukuran kurang dari dua kali pohon aslinya.

Ia juga memberi, tanpa biaya, salah satu fakta yang paling sering dipakai tentang pohon biner: karena setiap simpul dalam mempunyai dua anak dalam pohon diperluas, banyaknya seluruh simpul adalah $n + (n + 1) = 2n + 1$, selalu ganjil. Dan ia adalah inti kombinatorik dari pernyataan bahwa pohon biner dengan n simpul dalam mempunyai tepat $n + 1$ *tautan kosong*, yang memungkinkan adanya pohon biner berbenang (*threaded binary tree*) — setiap satu dari $n + 1$ tautan kosong itu dapat dimanfaatkan ulang sebagai benang menuju penerus.

16.5.8 Kesalahan yang paling sering terjadi pada persoalan ini

Memakai induksi **lemah** — menganggap klaimnya hanya untuk $n - 1$ — lalu mencoba menerapkannya pada dua subpohon berukuran k dan $n - 1 - k$. Kecuali k kebetulan $n - 1$ atau 0, tak satu pun subpohon mempunyai $n - 1$ simpul dalam, jadi hipotesisnya tidak berlaku dan buktinya berlubang tepat di tempat yang seharusnya kokoh.

Perbaikannya satu kalimat: *"anggaplah* $X(n') = n' + 1$ *untuk semua* $0 \leq n' < n$*"*. Tuliskan sebagai baris tersendiri, berlabel **Hipotesis induksi**, dan buktinya tidak dapat berlubang di situ.

Yang kedua adalah lupa bahwa kasus dasarnya $n = 0$, bukan $n = 1$. Definisinya memberi tahu Anda menjadi apa pohon kosong jika diperluas — satu simpul luar — justru agar Anda dapat mulai dari nol. Bukti yang dimulai pada $n = 1$ tidak keliru, tetapi ia lalu harus menangani kasus subpohon kosong di dalam langkahnya, yang lebih banyak pekerjaan tanpa keuntungan.

Uji diri. Sebuah pohon biner *penuh* (*full*) adalah pohon yang setiap simpulnya mempunyai nol atau dua anak. Buktikan, dengan kerangka induksi kuat yang sama, bahwa pohon biner penuh dengan ℓ daun mempunyai tepat $\ell - 1$ simpul beranak dua.

Jawaban. Misalkan $I(\ell)$ adalah banyaknya simpul beranak dua. Dasar, $\ell = 1$: satu simpul, $I(1) = 0 = 1 - 1$. Langkah, $\ell \geq 2$: akarnya beranak dua, dan subpohon-subpohonnya penuh dengan $\ell_1, \ell_2 \geq 1$ daun, $\ell_1 + \ell_2 = \ell$, keduanya $< \ell$. Dengan induksi kuat $I(\ell) = 1 + (\ell_1 - 1) + (\ell_2 - 1) = \ell - 1$. Ini Persoalan 16.3 lagi dalam penyamaran: daun berperan sebagai simpul luar dan simpul beranak dua sebagai simpul dalam.

16.6 Persoalan 16.4 — Hampiran-2 untuk Persoalan Pedagang Keliling

16.6.1 Persoalannya

Persoalan 16.4. Misalkan $G = (V, E)$ adalah graf lengkap dengan himpunan simpul $V = \{1,2, \dots, n\}$ dan bobot $w(i, j) = d_{ij}$, dengan d_{ij} adalah jarak antara titik i dan j dalam sebuah contoh *Persoalan Pedagang Keliling* (*Traveling Salesman Problem*, TSP), dan anggaplah jarak-jarak itu memenuhi ketaksamaan segitiga. Tinjaulah algoritma berikut. (1) Pilihlah sebuah simpul $r \in V$ sebagai akar sebuah pohon. (2) Bangunlah pohon merentang minimum T untuk G berakar di r, memakai algoritma pohon merentang minimum berwaktu polinomial sebagai subrutin. (3) Susunlah daftar L simpul-simpul yang dikunjungi oleh *penelusuran preorder* (*preorder walk*) atas T. (4) Kembalikan L sebagai tur hampiran untuk G. Buktikan:

1. algoritma itu benar, yaitu ia menghasilkan sebuah tur;
2. waktu jalannya polinomial;
3. ia mempunyai batas rasio $\rho(n) = 2$.

Ketiga bagian itu tidak sama beratnya. Bagian 1 dan 2 masing-masing satu paragraf; bagian 3 adalah jantung persoalannya.

16.6.2 Apa yang diuji, dan kosakata yang Anda butuhkan dari nol

Algoritma hampiran (*approximation algorithms*) bukan salah satu bab buku ini — ia salah satu dari enam wilayah yang di Subbab 16.15 disebut berada di luarnya. Persoalan ini tetap adil, dengan alasan yang tepat: **setiap bahan buktinya sudah ada di tangan Anda.** Ia membutuhkan pohon merentang minimum (Bab 11), penelusuran *preorder* sebuah pohon (Bab 10), ketaksamaan segitiga (Bab 13), dan pengamatan bahwa menghapus satu sisi dari sebuah tur menyisakan pohon merentang. Yang hilang hanya kosakata, dan itu cukup empat definisi, masing-masing sepanjang satu kalimat. Itulah artinya menguasai sebuah perangkat: bukan mengingat jawaban, melainkan mengarahkan mesinnya pada sesuatu yang belum pernah Anda lihat.

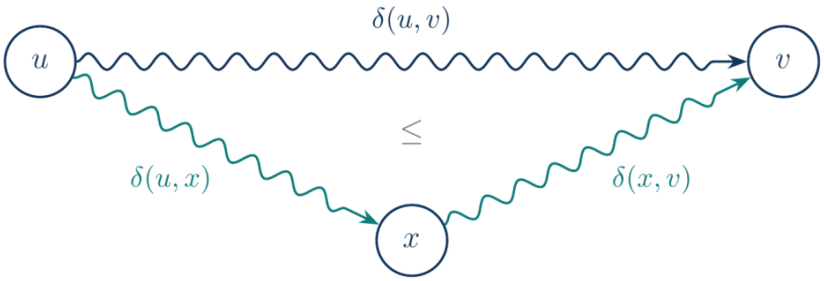
DEFINISI. Sebuah **tur** (*tour*) pada graf $G = (V, E)$ adalah siklus yang mengunjungi setiap simpul tepat sekali lalu kembali ke titik awalnya — sebuah *siklus Hamilton*. **Panjang**-nya adalah jumlah bobot $|V|$ sisinya. **Persoalan Pedagang Keliling** meminta tur yang panjangnya minimum.

DEFINISI. Fungsi bobot w memenuhi **ketaksamaan segitiga** (*triangle inequality*) jika, untuk setiap tiga simpul u, v, x,

$$w(u, v) \leq w(u, x) + w(x, v). \tag{16.4}$$

Dengan kata lain: jalan memutar lewat x tidak pernah menjadi jalan pintas. Contoh persoalan yang bobotnya memenuhi ini disebut contoh **metrik**. Jarak garis lurus, jarak berkendara, dan jarak Manhattan semuanya memenuhinya; tabel tarif dengan penerbangan sambungan yang didiskon mungkin tidak.

Gambar 16.2 adalah gambar ketaksamaan itu dari Bab 13. Di sana ia dipakai untuk $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$; gambarnya sama, dan maknanya pun sama.



Gambar 16.2 Ketaksamaan segitiga: pergi dari u ke v secara langsung tidak pernah lebih buruk daripada pergi dari u ke x lalu dari x ke v. Fakta tunggal inilah yang membuat langkah pemintasan pada Persoalan 16.4 sah.

DEFINISI. Algoritma untuk persoalan peminimuman mempunyai **batas rasio** (*ratio bound*) $\rho(n)$ jika, pada setiap masukan berukuran n, biaya C yang dikembalikannya dan biaya optimal C^* memenuhi $C/C^* \leq \rho(n)$. Algoritma seperti itu disebut **algoritma hampiran-$\rho(n)$**. Persoalan 16.4 meminta Anda membuktikan $\rho(n) = 2$ — rasio yang *konstan*, tidak bergantung pada n.

> **DEFINISI. Penelusuran penuh** (*full walk*) atas pohon berakar T mendaftar setiap simpul setiap kali simpul itu dijumpai: turun ke sebuah anak, dan sekali lagi dalam perjalanan naik. **Penelusuran *preorder*** mendaftar setiap simpul pada kali *pertama* ia dijumpai. Secara setara, penelusuran *preorder* adalah penelusuran penuh dengan kemunculan-kemunculan ulangnya dihapus.

Kesetaraan terakhir itu adalah kalimat yang menyelesaikan bagian 1, jadi ada baiknya memilikinya persis dalam bentuk itu.

16.6.3 Bagian 1 — algoritmanya menghasilkan sebuah tur

Klaim. Daftar L yang dikembalikan algoritma adalah tur pada G: ia memuat setiap simpul V tepat sekali.

Bukti. T adalah pohon merentang G, jadi himpunan simpulnya adalah V — itulah arti *merentang*. Penelusuran penuh W atas T yang dimulai di r melintasi setiap sisi T dua kali, sekali ke tiap arah, dan karena itu mengunjungi setiap simpul T paling sedikit sekali. Karena $V(T) = V$, penelusuran W memuat setiap simpul G paling sedikit sekali.

Penelusuran *preorder* L diperoleh dari W dengan hanya mempertahankan kemunculan pertama setiap simpul. Mempertahankan kemunculan pertama setiap simpul dari sebuah daftar yang memuat setiap simpul paling sedikit sekali menghasilkan daftar yang memuat setiap simpul **tepat** sekali. Karena itu L adalah permutasi dari V.

Akhirnya, G adalah graf **lengkap**, jadi untuk setiap dua simpul L_i dan L_{i+1} sisi (L_i, L_{i+1}) ada, demikian pula sisi penutup $(L_{|V|}, L_1)$. Karena itu barisan siklis $L_1, L_2, \ldots, L_{|V|}, L_1$ adalah siklus Hamilton — sebuah tur. ■

Dua catatan yang layak dinyatakan secara tersurat. Pertama, kelengkapan G dipakai, dan dipakai tepat sekali, di sini: tanpanya simpul-simpul berurutan dalam L mungkin tidak dihubungkan oleh sisi. Kedua, ketaksamaan segitiga **tidak** dipakai di bagian 1. Metrik ataupun tidak bobotnya, algoritma itu mengembalikan sebuah tur; kemetrikan hanya dibutuhkan untuk bagian 3. Menyatakannya secara tersurat memperlihatkan bahwa Anda tahu hipotesis mana yang mengerjakan tugas mana.

16.6.4 Bagian 2 — waktu jalannya polinomial

Bagian ini lugas, tetapi paragrafnya sebaiknya berisi bilangan, bukan kata sifat.

Misalkan $n = |V|$. Karena G lengkap, $|E| = \binom{n}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$.

Tabel 16.4 Biaya setiap langkah algoritma hampiran pada graf lengkap dengan n simpul. Pohon merentang minimumnya mendominasi.

Langkah	Apa yang dikerjakannya	Biaya
1	Memilih akar r	$\Theta(1)$
2	Membangun pohon merentang minimum	$\Theta(n^2)$ dengan Prim dan larik; $O(E\lg V) = O(n^2 \lg n)$ dengan Kruskal atau *binary heap*
3	Penelusuran *preorder* atas T	$\Theta(n)$ — pohonnya punya n simpul dan $n-1$ sisi
4	Mengembalikan L	$\Theta(n)$

Tabel 16.4 berjumlah $\Theta(n^2)$ dengan Prim berlarik, yang merupakan pilihan tepat di sini sebab graf lengkap sepadat-padatnya graf, dan versi larik Prim yang $\Theta(n^2)$ mengalahkan versi *heap* yang $O(E\lg V) = O(n^2 \lg n)$ pada masukan padat. (Itulah pernyataan (12) Persoalan 16.1, hadir

sebagai keputusan kerja alih-alih sebagai pendapat.) Bagaimanapun juga, $\Theta(n^2)$ ataupun $O(n^2 \lg n)$ adalah polinomial dalam n, dan itulah yang diminta bagian 2. ■

Satu kalimat tambahan layak ditulis: penyelesaian TSP yang *eksak* tidak diketahui dapat dihitung dalam waktu polinomial — versi keputusannya NP-lengkap, dan pencarian gaya kasar atas $(n-1)!/2$ tur jauh lebih buruk secara astronomis. Pada $n = 20$ itu sekitar 6×10^{16} tur. Seluruh tujuan algoritma ini adalah menukar jawaban eksak dengan jawaban polinomial, dan bagian 2-lah yang membuat pertukaran itu bernilai.

16.6.5 Bagian 3 — batas rasionya 2

Tuliskan sebagai rantai empat pernyataan, masing-masing pada barisnya sendiri, masing-masing dibenarkan. Notasi: T adalah pohon merentang minimum, W penelusuran penuhnya, L penelusuran *preorder* yang dikembalikan algoritma, L^* tur optimal, dan $d(\cdot)$ bobot total sebuah pohon, penelusuran, atau tur.

Langkah 1: $d(T) \leq d(L^*)$ — pohon merentang minimum tidak lebih berat daripada tur optimal.

Ambillah tur optimal L^* lalu hapuslah salah satu sisinya. Tur adalah siklus melalui semua n simpul; menghapus satu sisi dari sebuah siklus menyisakan lintasan melalui semua n simpul, yang terhubung, taksiklik, dan menyentuh setiap simpul — yaitu pohon merentang G. Sebutlah T'. Karena semua bobot tak negatif,

$$d(T') \;\leq\; d(L^*).$$

Dan karena T adalah pohon merentang *minimum*, ia tidak lebih berat daripada pohon merentang T':

$$d(T) \;\leq\; d(T') \;\leq\; d(L^*).$$

Langkah 2: $d(W) = 2\,d(T)$ — penelusuran penuh berbiaya tepat dua kali pohonnya.

Penelusuran penuh memasuki setiap subpohon sekali dan meninggalkannya sekali, jadi ia melintasi setiap sisi T tepat dua kali, sekali ke bawah dan sekali ke atas. Menjumlahkannya,

$$d(W) \;=\; \sum_{e \in T} 2\; w(e) \;=\; 2\,d(T).$$

Digabungkan dengan Langkah 1,

$$d(W) \;=\; 2\,d(T) \;\leq\; 2\,d(L^*).$$

Langkah 3: $d(L) \leq d(W)$ — membuang ulangan tidak pernah membuat perjalanan lebih panjang.

Di sinilah ketaksamaan segitiga membuktikan kegunaannya, dan inilah langkah yang paling sering hanya dinyatakan alih-alih dibuktikan. Daftar *preorder* L diperoleh dari W dengan menghapus simpul-simpul ulangan. Menghapus satu simpul ulangan berarti mengganti sepotong lintasan $u \to x \to v$ dari W dengan satu sisi $u \to v$. Menurut ketaksamaan segitiga Persamaan (16.4),

$$w(u,v) \;\le\; w(u,x) + w(x,v),$$

jadi penggantian itu tidak menaikkan totalnya. L diperoleh dari W dengan berhingga banyaknya penggantian seperti itu — satu per kemunculan yang dihapus — dan masing-masing tidak menaikkan total, jadi dengan induksi atas banyaknya penghapusan

$$d(L) \;\le\; d(W).$$

Langkah 4: gabungkan semuanya.

$$d(L) \;\le\; d(W) \;=\; 2\,d(T) \;\le\; 2\,d(L^*), \tag{16.5}$$

sehingga

$$\frac{d(L)}{d(L^*)} \;\le\; 2, \qquad \text{yaitu} \qquad \rho(n) = 2. \qquad \blacksquare$$

> **WAWASAN KUNCI: DI MANA SETIAP HIPOTESIS DIPAKAI.** Kelengkapan G dipakai di bagian 1, untuk menjamin sisi-sisi penutupnya ada. Ketaknegatifan bobot dipakai di Langkah 1, untuk mengatakan bahwa menghapus sebuah sisi tidak menaikkan total. Ketaksamaan segitiga dipakai di Langkah 3, dan tidak di tempat lain. Keminimuman T dipakai di Langkah 1. Jika Anda ditanya satu hal tentang bukti ini, pertanyaannya adalah ini.

16.6.6 Apa yang terjadi tanpa ketaksamaan segitiga

Buanglah ketaksamaan itu dan batasnya runtuh — bukan sedikit, melainkan sepenuhnya. Andaikan ada algoritma polinomial dengan rasio konstan ρ *apa pun* untuk TSP umum. Maka Anda dapat memutuskan apakah sebuah graf sembarang H pada n simpul mempunyai siklus Hamilton: bangunlah graf lengkap pada simpul-simpul yang sama, berilah bobot 1 pada setiap sisi H dan bobot $\rho n + 1$ pada setiap sisi yang tidak ada di H, lalu jalankan hampirannya. Jika H mempunyai siklus Hamilton, optimumnya n dan algoritmanya harus mengembalikan paling banyak ρn, jadi ia tidak dapat memakai sisi berat mana pun. Jika H tidak mempunyainya, setiap tur memakai sisi berat dan berbiaya lebih dari ρn. Jadi algoritma itu memutuskan siklus Hamilton, yang NP-lengkap — mustahil dalam waktu polinomial kecuali $\mathrm{P} = \mathrm{NP}$.

Dalam satu kalimat: *"Tanpa ketaksamaan segitiga, Langkah 3 gagal, dan bahkan tidak ada hampiran polinomial berasio konstan kecuali* $\mathrm{P} = \mathrm{NP}$*."*

16.6.7 Sebuah contoh terkerjakan yang lengkap

Bukti abstrak diperiksa dengan bilangan konkret. Ambillah enam titik jemput Profitina Laundry pada kisi jalan Surabaya dan ukurlah dalam blok dengan metrik Manhattan $d(i,j) = |x_i - x_j| + |y_i - y_j|$, yang memenuhi ketaksamaan segitiga:

$$A = (0,0), \quad B = (4,0), \quad C = (8,1), \quad D = (6,5), \quad E = (2,6), \quad F = (0,3).$$

Tabel 16.5 adalah matriks jarak yang dihasilkannya. Periksalah satu entri dengan tangan: $d(A,D) = |0-6| + |0-5| = 6 + 5 = 11$.

Tabel 16.5 Jarak Manhattan dalam blok antara enam titik jemput Profitina Laundry. Matriksnya simetris, diagonalnya nol, dan memenuhi ketaksamaan segitiga — tiga syarat yang menjadikannya metrik.

	A	B	C	D	E	F
A	0	4	9	11	8	3
B	4	0	5	7	8	7
C	9	5	0	6	11	10
D	11	7	6	0	5	8
E	8	8	11	5	0	5
F	3	7	10	8	5	0

Langkah 2 algoritmanya. Pohon merentang minimum berakar di A, dengan Prim: sisi termurah yang keluar dari $\{A\}$ adalah $A - F$ seharga 3; lalu $A - B$ seharga 4; lalu $B - C$ seharga 5; lalu $F - E$ seharga 5; lalu $E - D$ seharga 5. Jumlahnya

$$d(T) = 3 + 4 + 5 + 5 + 5 = 22.$$

Langkah 3. Berakar di A, pohonnya adalah $A \to \{B, F\}$, $B \to \{C\}$, $F \to \{E\}$, $E \to \{D\}$. Penelusuran penuhnya, dengan anak diambil menurut abjad, adalah

$$W = A,\ B,\ C,\ B,\ A,\ F,\ E,\ D,\ E,\ F,\ A,$$

yang panjangnya $4 + 5 + 5 + 4 + 3 + 5 + 5 + 5 + 5 + 3 = 44$, dan memang $44 = 2 \times 22 = 2\,d(T)$, tepat seperti diramalkan Langkah 2 buktinya.

Menghapus ulangan memberi daftar *preorder*

$$L = A,\ B,\ C,\ F,\ E,\ D,$$

dan tur $A \to B \to C \to F \to E \to D \to A$ berbiaya

$$4 + 5 + 10 + 5 + 5 + 11 = 40.$$

Optimumnya. Dengan enam simpul ada $5!/2 = 60$ tur yang berbeda, jadi gaya kasar masih terjangkau. Yang terbaik adalah $A \to B \to C \to D \to E \to F \to A$ seharga

$$4 + 5 + 6 + 5 + 5 + 3 = 28.$$

Keempat ketaksamaan Persamaan (16.5)**, diperiksa:**

$$d(T) = 22 \ \le\ d(L^*) = 28 \quad \checkmark$$

$$d(W) = 44 \ =\ 2\,d(T) = 44 \quad \checkmark$$

$$d(L) = 40 \ \le\ d(W) = 44 \quad \checkmark$$

$$\frac{d(L)}{d(L^*)} = \frac{40}{28} = 1.4286 \ \le\ 2 \quad \checkmark$$

16.6.8 Programnya

Kode 16.5 mengerjakan semua hal di atas dan mencetak setiap besaran antara, sehingga tak satu pun bilangan di Subbab 16.6.7 harus diterima begitu saja.

Kode 16.5 Algoritma hampiran Persoalan 16.4 selengkapnya, dengan ketaksamaan segitiga diverifikasi, pohon merentang minimum dibangun dengan Prim, penelusuran penuh dan penelusuran *preorder* dicetak, dan optimumnya ditemukan dengan gaya kasar sebagai pembanding.

```
// Persoalan 16.4: hampiran 2 bagi TSP metrik lewat pohon merentang minimum ditambah
// penelusuran preorder, dijalankan pada enam titik jemput Profitina Laundry
// dengan metrik Manhattan, lalu dibandingkan dengan tur optimal hasil gaya kasar.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <numeric>
#include <climits>
using namespace std;

const int N = 6;
const string name[N]  = {"A","B","C","D","E","F"};
const int  x[N] = {0, 4, 8, 6, 2, 0};
const int  y[N] = {0, 0, 1, 5, 6, 3};

int d[N][N];

vector<int> child[N];                        // pohon merentang minimumnya, berakar
vector<int> preorderList, fullWalk;

void preorder(int u) {
    preorderList.push_back(u);
    fullWalk.push_back(u);
    for (int v : child[u]) { preorder(v); fullWalk.push_back(u); }
}

int tourCost(const vector<int>& t) {
    int s = 0;
    for (size_t i = 0; i < t.size(); ++i) s += d[t[i]][t[(i + 1) % t.size()]];
    return s;
}
int walkCost(const vector<int>& w) {
    int s = 0;
    for (size_t i = 0; i + 1 < w.size(); ++i) s += d[w[i]][w[i + 1]];
    return s;
}

int main() {
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            d[i][j] = abs(x[i] - x[j]) + abs(y[i] - y[j]);   // Manhattan: sebuah metrik
```

Kode 16.5 (Lanjutan) *Algoritma hampiran Persoalan 16.4 selengkapnya, dengan ketaksamaan segitiga diverifikasi, pohon merentang minimum dibangun dengan Prim, penelusuran penuh dan penelusuran preorder dicetak, dan optimumnya ditemukan dengan gaya kasar sebagai pembanding.*

```
    cout << "distance matrix (Manhattan)\n    ";
    for (int j = 0; j < N; ++j) cout << setw(3) << name[j];
    cout << "\n";
    for (int i = 0; i < N; ++i) {
        cout << setw(4) << name[i];
        for (int j = 0; j < N; ++j) cout << setw(3) << d[i][j];
        cout << "\n";
    }

    bool tri = true;                      // hipotesis yang dibutuhkan buktinya
    for (int i = 0; i < N; ++i) for (int j = 0; j < N; ++j) for (int k = 0; k < N; ++k)
        if (d[i][j] > d[i][k] + d[k][j]) tri = false;
    cout << "\ntriangle inequality holds: " << (tri ? "yes" : "no") << "\n";

    // Langkah 2: pohon merentang minimum berakar di r = A, dengan Prim.
    const int r = 0;
    vector<int> key(N, INT_MAX), par(N, -1);
    vector<bool> in(N, false);
    key[r] = 0;
    int mstWeight = 0;
    for (int it = 0; it < N; ++it) {
        int u = -1;
        for (int v = 0; v < N; ++v) if (!in[v] && (u == -1 || key[v] < key[u])) u = v;
        in[u] = true;
        mstWeight += key[u];
        if (par[u] != -1) cout << "MST edge " << name[par[u]] << "-" << name[u]
                               << "  w = " << key[u] << "\n";
        for (int v = 0; v < N; ++v)
            if (!in[v] && d[u][v] < key[v]) { key[v] = d[u][v]; par[v] = u; }
    }
    cout << "MST weight d(T) = " << mstWeight << "\n";

    for (int v = 0; v < N; ++v) if (par[v] != -1) child[par[v]].push_back(v);
    for (int v = 0; v < N; ++v) sort(child[v].begin(), child[v].end());

    // Langkah 3 dan 4: penelusuran penuh, lalu preorder, lalu turnya.
    preorder(r);
    cout << "\nfull walk W    :";
    for (int v : fullWalk) cout << " " << name[v];
    cout << "\n  d(W) = " << walkCost(fullWalk) << "  (= 2 d(T) = " << 2 * mstWeight <<
")\n";
    cout << "preorder list L:";
    for (int v : preorderList) cout << " " << name[v];
    cout << "\n  d(L) = " << tourCost(preorderList) << "\n";
```

Kode 16.5 (Lanjutan) *Algoritma hampiran Persoalan 16.4 selengkapnya, dengan ketaksamaan segitiga diverifikasi, pohon merentang minimum dibangun dengan Prim, penelusuran penuh dan penelusuran preorder dicetak, dan optimumnya ditemukan dengan gaya kasar sebagai pembanding.*

```
    // Optimumnya, gaya kasar atas semua (N-1)! rotasi yang dipatok di A.
    vector<int> perm(N);
    iota(perm.begin(), perm.end(), 0);
    int best = INT_MAX; vector<int> bestTour;
    do {
        if (perm[0] != 0) continue;
        int c = tourCost(perm);
        if (c < best) { best = c; bestTour = perm; }
    } while (next_permutation(perm.begin(), perm.end()));

    cout << "\noptimal tour L*:";
    for (int v : bestTour) cout << " " << name[v];
    cout << "\n  d(L*) = " << best << "\n";
    cout << fixed << setprecision(4)
         << "\nratio d(L)/d(L*) = " << (double)tourCost(preorderList) / best
         << "   (the theorem guarantees at most 2)\n"
         << "check d(T) <= d(L*):    " << mstWeight << " <= " << best << "\n"
         << "check d(L) <= 2 d(L*):  " << tourCost(preorderList) << " <= " << 2 * best <<
"\n";
    return 0;
}
```

Dikompilasi dan dijalankan, ia mencetak:

```
distance matrix (Manhattan)
      A  B  C  D  E  F
   A  0  4  9 11  8  3
   B  4  0  5  7  8  7
   C  9  5  0  6 11 10
   D 11  7  6  0  5  8
   E  8  8 11  5  0  5
   F  3  7 10  8  5  0

triangle inequality holds: yes
MST edge A-F  w = 3
MST edge A-B  w = 4
MST edge B-C  w = 5
MST edge F-E  w = 5
MST edge E-D  w = 5
MST weight d(T) = 22

full walk W    : A B C B A F E D E F A
  d(W) = 44  (= 2 d(T) = 44)
preorder list L: A B C F E D
  d(L) = 40

optimal tour L*: A B C D E F
  d(L*) = 28

ratio d(L)/d(L*) = 1.4286   (the theorem guarantees at most 2)
check d(T) <= d(L*):    22 <= 28
check d(L) <= 2 d(L*):  40 <= 56
```

16.6.9 Seberapa ketat angka 2?

Batasnya *dibuktikan*, jadi rasionya tidak pernah dapat melebihi 2. Ia juga hampir *ketat*: ada contoh metrik yang rasionya mendekati 2 ketika n membesar. Enam titik kita memberi 1.4286; enam titik pada Soal Latihan 4 di Subbab 16.19 memberi 1.5556; dan dikenal keluarga contoh yang mendekati $2 - \varepsilon$ untuk setiap $\varepsilon > 0$. Jadi angka 2 bukan artefak bukti yang malas. Ringkasan

yang jujur adalah: *"batasnya 2, ia ketat secara asimtotik untuk algoritma ini, dan pada contoh yang lazim rasio yang teramati jauh di bawahnya."*

Jika Anda menginginkan jaminan yang lebih baik, algoritma Christofides [2] mengganti pelipatduaan pohon dengan pemasangan sempurna berbobot minimum pada simpul-simpul berderajat ganjil di T, dan mencapai $\rho = 3/2$ dalam waktu polinomial. Ia menjadi konstanta terbaik yang diketahui untuk TSP metrik sejak 1976 sampai 2021, ketika Karlin, Klein, dan Oveis Gharan [3] mengalahkannya dengan selisih yang lebih kecil daripada 10^{-36}; karya berikutnya oleh Gurvits, Klein, dan Leake [4] hanya sedikit memperbesar selisih itu, sehingga per September 2026 jaminan terbaik untuk TSP metrik praktis tetap 3/2. Satu kalimat yang menyebutnya menempatkan algoritma ini di dalam bidangnya.

16.6.10 Kesalahan yang paling sering terjadi pada persoalan ini

Yang pertama adalah membuktikan $d(L) \leq d(W)$ dengan berkata "membuang simpul membuat penelusurannya lebih pendek". Secara umum tidak demikian — membuang satu simpul dari sebuah penelusuran *mengganti dua sisi dengan satu*, dan tanpa ketaksamaan segitiga satu sisi pengganti itu dapat jauh lebih berat daripada dua sisi yang digantikannya. Pernyataan itu benar di sini **karena contohnya metrik**, dan buktinya harus mengatakan demikian, di tempat hal itu dipakai.

Yang kedua adalah membuktikan $d(T) \leq d(L^*)$ dengan menyatakan "pohon lebih murah daripada tur". Itu bukan teorema; itu kesimpulannya. Argumennya adalah: *hapuslah satu sisi dari tur optimal untuk memperoleh pohon merentang, yang bobotnya paling banyak bobot tur karena bobot tak negatif, dan T minimum di antara pohon merentang, jadi $d(T)$ paling banyak sebesar itu.* Tiga anak kalimat. Tuliskan ketiganya.

Uji diri. Pada contoh Subbab 16.6.7, andaikan penelusuran *preorder* mengambil anak-anak A dalam urutan F sebelum B. Tur apa yang dihasilkan, berapa biayanya, dan apakah Persamaan (16.5) masih berlaku?

Jawaban. Penelusuran penuhnya menjadi $A, F, E, D, E, F, A, B, C, B, A$, tetap sepanjang 44; daftar *preorder*-nya A, F, E, D, B, C, dan turnya berbiaya $3 + 5 + 5 + 7 + 5 + 9 = 34$. Maka $34 \leq 44 = 2 \cdot 22 \leq 56$, jadi rantainya berlaku — sebagaimana mestinya, sebab buktinya tidak pernah memakai urutan anak. Rasio $34/28 = 1.2143$ memperlihatkan bahwa pilihan urutan anak, yang dibiarkan bebas oleh algoritmanya, dapat sangat berpengaruh dalam praktik.

16.7 Persoalan 16.5 — Bellman–Ford Berhadapan dengan Dijkstra

16.7.1 Persoalannya

Persoalan 16.5. Bellman–Ford dan Dijkstra adalah dua algoritma yang menemukan lintasan terpendek dari satu simpul ke semua simpul lain pada sebuah graf berbobot.

1. Uraikan perbedaan keduanya dan penerapan masing-masing dalam kehidupan nyata.
2. Tuliskan pseudokode untuk masing-masing algoritma dan tunjukkan di mana keduanya berbeda.
3. Berikan ilustrasi singkat tentang cara kerja masing-masing algoritma.

16.7.2 Apa yang diuji

Apakah Anda memahami bahwa kedua algoritma itu adalah **gagasan yang sama dengan dua urutan yang berbeda**. Keduanya menginisialisasi $d[s] = 0$ dan $d[v] = \infty$, dan keduanya memperbaiki taksiran dengan relaksasi, Persamaan (16.1). Segala sesuatu sesudah itu adalah pilihan **sisi mana yang direlaksasi berikutnya, dan berapa kali**. Dijkstra merelaksasi sisi-sisi yang keluar dari simpul belum-selesai terdekat, masing-masing sekali. Bellman–Ford merelaksasi setiap sisi, $|V| - 1$ kali. Jawaban yang berbunyi "yang satu memakai antrean prioritas dan yang lain memakai gelung" menguraikan kodenya; jawaban yang berbunyi "keduanya hanya berbeda dalam urutan dan kelipatan relaksasinya, dan perbedaan itulah yang dirusak bobot negatif" menguraikan algoritmanya.

16.7.3 Bagian 1 — perbedaannya, dan penerapannya

Satu perbedaan yang tidak boleh dilewatkan jawaban mana pun adalah bahwa Bellman–Ford menangani sisi berbobot negatif dan Dijkstra tidak. **Tabel 16.6** memperlihatkan wujud jawaban yang lengkap.

Tabel 16.6 Dijkstra berhadapan dengan Bellman–Ford. Dua baris terakhir menjelaskan pembagian peran di dunia nyata: *router* memakai Bellman–Ford karena ia lokal; aplikasi peta memakai Dijkstra karena ia cepat dan bobot jalan tidak pernah negatif.

	Dijkstra	Bellman–Ford
Paradigma perancangan	Rakus (Bab 9)	Pemrograman dinamis atas panjang lintasan (Bab 7)
Bobot sisi negatif	**Tidak diizinkan**	Diizinkan
Siklus negatif	Perilaku tak terdefinisi	**Dideteksi**; mengembalikan `false`
Urutan relaksasi	Sisi keluar dari simpul belum-selesai terdekat	Semua sisi, $|V| - 1$ kali
Setiap simpul dituntaskan	Sekali, ketika dikeluarkan dari antrean	Tidak pernah — setiap taksiran dapat turun lagi
Struktur data yang dibutuhkan	Antrean prioritas yang mendukung `Extract-Min` dan `Decrease-Key`	Tidak ada; cukup daftar sisi
Waktu jalan	$\Theta(V^2)$ dengan larik; $O(E\lg V)$ dengan *binary heap*; $O(V\lg V + E)$ dengan *Fibonacci heap*	$\Theta(VE)$
Pada graf jarang, $E = \Theta(V)$	$O(V\lg V)$	$\Theta(V^2)$
Implementasi terdistribusi	Sulit — butuh minimum global	Mudah — tiap simpul hanya butuh tetangganya
Gagal ketika	Ada bobot sisi yang negatif	Sebuah siklus negatif terjangkau (dan ia melaporkannya)

Penerapan nyata, berikut alasannya. Penerapan tanpa alasannya hanyalah daftar nama; inilah alasan-alasannya.

- **Bellman–Ford: perutean vektor-jarak.** *Routing Information Protocol* (RIP), dan keluarga vektor-jarak pada umumnya, adalah Bellman–Ford yang dijalankan secara terdistribusi: setiap *router* menyimpan vektor jarak taksiran dan secara berkala mengirimkannya ke tetangga-tetangganya, yang lalu merelaksasi taksiran mereka sendiri terhadapnya. Bellman–Ford menjadi algoritma yang tepat di sini bukan karena biaya tautannya negatif — memang tidak — melainkan karena baris *terdistribusi* pada **Tabel 16.6**: merelaksasi sebuah sisi hanya membutuhkan dua *router* di ujung-ujungnya, sedangkan `Extract-Min` Dijkstra membutuhkan pandangan global yang tidak dimiliki satu *router* pun.

- **Bellman–Ford: arbitrase mata uang.** Ambillah kurs tukar r_{ij} lalu tetapkan $w(i,j) = -\ln r_{ij}$. Barisan transaksi yang melipatgandakan uang Anda lebih dari satu kali menjadi siklus berbobot total negatif, dan uji siklus negatif Bellman–Ford menemukannya. Inilah contoh baku bobot yang *sungguh-sungguh* negatif, dan ia menjawab keberatan yang wajar bahwa jarak tidak pernah negatif.
- **Dijkstra: aplikasi peta dan navigasi.** Jarak jalan dan waktu tempuh tak negatif, grafnya jarang, dan jawabannya dibutuhkan dalam hitungan milidetik. Dijkstra dengan *binary heap* — atau kerabatnya yang terarah-tujuan, A^* dan *contraction hierarchies* — adalah yang dijalankan setiap produk navigasi.
- **Dijkstra: perutean *link-state*.** OSPF menyebarkan seluruh topologi ke setiap *router*, lalu setiap *router* menjalankan Dijkstra secara lokal pada peta lengkapnya. Perhatikan pertukarannya: OSPF membayar biaya penyebaran topologi demi membeli kecepatan Dijkstra, keputusan yang tepat berlawanan dengan keputusan RIP.

16.7.4 Bagian 2 — kedua pseudokodenya

Gambar 16.3 dan **Gambar 16.4** adalah pseudokode kedua algoritma sebagaimana diberikan Bab 13. Mereproduksinya bukanlah inti bagian 2; *menandai perbedaannya*-lah intinya.

```
DIJKSTRA(G, w, s)
1  untuk tiap v ∈ V
2      lakukan d[v] ← ∞
3  d[s] ← 0
4  S ← ∅      ▷ himpunan simpul yang sudah tuntas
5  Q ← V
6  selama Q ≠ ∅
7      lakukan u ← EXTRACT-MIN(Q)
8          S ← S ∪ {u}
9          untuk tiap v ∈ Adj[u]
10             lakukan jika d[v] > d[u] + w(u, v)        relaksasi
11                 maka d[v] ← d[u] + w(u, v)             sisi
```

Gambar 16.3 Algoritma Dijkstra sebagaimana diberikan Bab 13. Baris 1–3 menginisialisasi, baris 7 mengeluarkan simpul belum-selesai terdekat, dan baris 9–11 yang disorot adalah relaksasinya.

```
BELLMAN-FORD(G, w, s)
01 untuk tiap v ∈ V[G]
02     d[v] ← ∞
03     π[v] ← NIL
04 d[s] ← 0
05 untuk i ← 1 sampai |V[G]| − 1 lakukan
06     untuk tiap sisi (u, v) ∈ E[G] lakukan
07         RELAX(u, v, w)
08 untuk tiap sisi (u, v) ∈ E[G] lakukan
09     jika d[v] > d[u] + w(u, v) maka kembalikan false
10 kembalikan true
```

Gambar 16.4 Algoritma Bellman–Ford sebagaimana diberikan Bab 13. Baris 1–4 menginisialisasi persis seperti Dijkstra; baris 5–7 adalah $|V| - 1$ putaran atas semua sisi; baris 8–10 adalah uji siklus negatif yang tidak ada padanannya pada Dijkstra.

Letakkan keduanya berdampingan dan empat perbedaan tampak dengan sendirinya. Sajikan sebagai daftar bernomor — itulah beda antara uraian dan analisis.

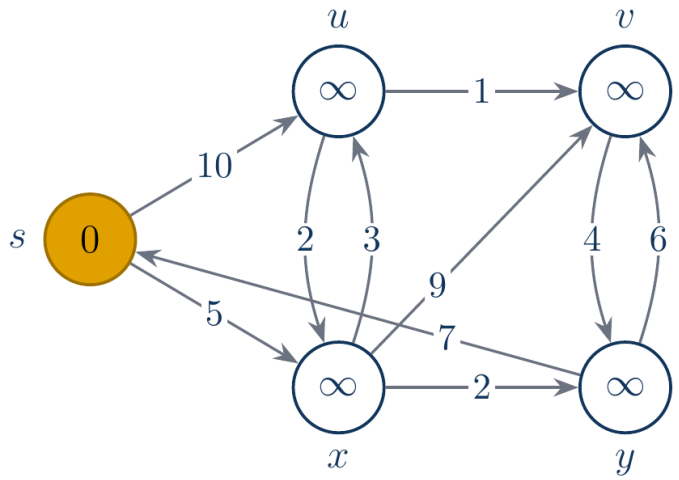
1. **Baris 1–3 identik.** Keduanya menetapkan $d[v] \leftarrow \infty$ untuk setiap v dan $d[s] \leftarrow 0$. Inisialisasi bukan tempat keduanya berbeda, dan menyatakannya lebih dahulu memperlihatkan bahwa Anda tahu apa yang dimiliki bersama.
2. **Baris 4–7 Dijkstra berhadapan dengan baris 5 Bellman–Ford.** Dijkstra memelihara himpunan S simpul yang sudah selesai dan antrean prioritas Q, lalu memilih simpul berikutnya dengan `Extract-Min`. Bellman–Ford tidak memilih apa-apa: ia berputar $|V| - 1$ kali atas seluruh daftar sisi. Pilihan Dijkstra adalah pilihan rakus; gelung Bellman–Ford adalah program dinamisnya.
3. **Dijkstra merelaksasi setiap sisi sekali; Bellman–Ford merelaksasi setiap sisi $|V| - 1$ kali.** Baris 9 Dijkstra beriterasi atas $Adj[u]$ untuk setiap u tepat sekali sepanjang seluruh jalannya, jadi setiap sisi direlaksasi tepat sekali. Baris 6 Bellman–Ford berada di dalam gelung baris 5, jadi setiap sisi direlaksasi $|V| - 1$ kali. Fakta struktural tunggal itu adalah seluruh perbedaan waktu jalannya: $O(E \lg V)$ berhadapan dengan $\Theta(VE)$.
4. **Bellman–Ford punya baris 8–10; Dijkstra tidak punya yang serupa.** Sapuan penutup menanyakan apakah masih ada sisi yang *dapat* direlaksasi sesudah $|V| - 1$ putaran. Jika ada, sebuah lintasan terpendek membutuhkan $\geq |V|$ sisi, artinya ia mengulang sebuah simpul, artinya ada siklus berbobot negatif; algoritmanya mengembalikan `false`. Dijkstra tidak dapat melakukan ini, sebab ia sudah membuang informasinya — simpul yang sudah dituntaskannya tidak pernah ditinjau lagi.

Mengapa $|V| - 1$ putaran sudah cukup. Inilah kalimat yang mengubah uraian menjadi bukti, dan ia murah. Lintasan terpendek tanpa siklus negatif tidak pernah mengulang simpul, jadi ia punya paling banyak $|V| - 1$ sisi. Sesudah putaran ke-i, $d[v]$ benar untuk setiap v yang lintasan terpendeknya memakai paling banyak i sisi — dengan induksi, sebab putaran ke-i merelaksasi setiap sisi, dan khususnya sisi terakhir lintasan itu. Karena itu sesudah $|V| - 1$ putaran setiap $d[v]$ benar.

16.7.5 Bagian 3 — ilustrasinya

Bagian 3 meminta ilustrasi yang *singkat*: dua gambar dan dua tabel bilangan sudah tepat, dan lebih dari itu hanyalah hiasan.

Dijkstra. **Gambar 16.5** adalah graf Bab 13 pada saat inisialisasi, dengan $d[s] = 0$ dan setiap taksiran lain ∞.

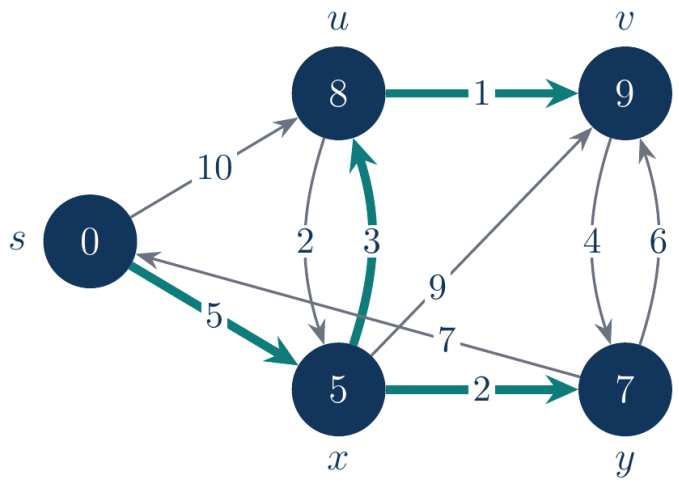


Gambar 16.5 Contoh Dijkstra pada saat inisialisasi: $d[s] = 0$, dan $d[u] = d[v] = d[x] = d[y] = \infty$. Bobot sisi $s \to u = 10$, $s \to x = 5$, $u \to v = 1$, $u \to x = 2$, $x \to u = 3$, $x \to v = 9$, $x \to y = 2$, $y \to v = 6$, $y \to s = 7$, $v \to y = 4$.

Kini penelusurannya, satu baris per `Extract-Min`, dengan hitungannya:

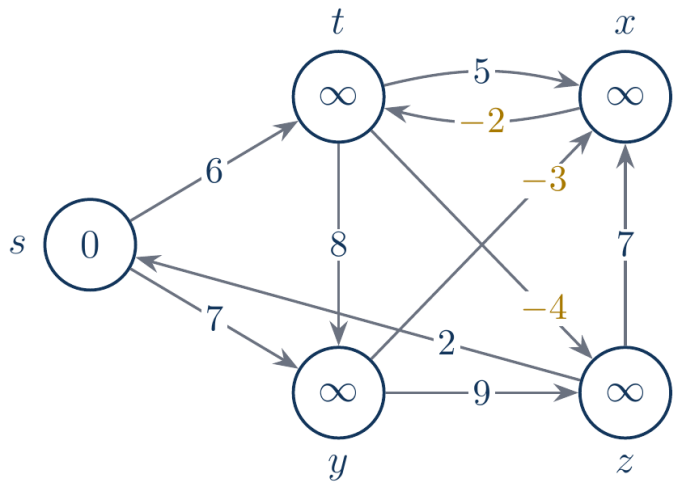
- **Keluarkan** s ($d = 0$). Relaksasi $s \to u$: $0 + 10 = 10 < \infty$, jadi $d[u] = 10$. Relaksasi $s \to x$: $0 + 5 = 5 < \infty$, jadi $d[x] = 5$.
- **Keluarkan** x ($d = 5$, yang terkecil). Relaksasi $x \to u$: $5 + 3 = 8 < 10$, jadi $d[u] = 8$. Relaksasi $x \to v$: $5 + 9 = 14 < \infty$, jadi $d[v] = 14$. Relaksasi $x \to y$: $5 + 2 = 7 < \infty$, jadi $d[y] = 7$.
- **Keluarkan** y ($d = 7$). Relaksasi $y \to v$: $7 + 6 = 13 < 14$, jadi $d[v] = 13$. Relaksasi $y \to s$: $7 + 7 = 14 \not< 0$, tidak berubah.
- **Keluarkan** u ($d = 8$). Relaksasi $u \to v$: $8 + 1 = 9 < 13$, jadi $d[v] = 9$. Relaksasi $u \to x$: $8 + 2 = 10 \not< 5$, tidak berubah.
- **Keluarkan** v ($d = 9$). Relaksasi $v \to y$: $9 + 4 = 13 \not< 7$, tidak berubah. Antrean kosong.

Gambar 16.6 adalah keadaan ketika algoritmanya berhenti, dengan setiap simpul biru tua — sudah dituntaskan — dan pohon lintasan terpendeknya ditebalkan.



Gambar 16.6 Contoh Dijkstra ketika berhenti: $d[s] = 0, d[x] = 5, d[y] = 7, d[u] = 8, d[v] = 9$. Sisi-sisi tebal adalah pohon lintasan terpendek. Bandingkan bilangannya dengan penelusuran tangan di Subbab 16.7.5.

Bellman–Ford. Gambar 16.7 adalah graf berbobot negatif Bab 13 pada saat inisialisasi.



Gambar 16.7 Contoh Bellman–Ford pada saat inisialisasi. Bobot: $s \to t = 6, s \to y = 7, t \to x = 5, t \to y = 8, t \to z = -4, x \to t = -2, y \to x = -3, y \to z = 9, z \to s = 2, z \to x = 7$. Tiga dari sepuluh bobotnya negatif, jadi Dijkstra tidak dapat dipakai.

Dengan sisi-sisi direlaksasi dalam urutan yang didaftar, putaran-putarannya berjalan seperti pada **Tabel 16.7**.

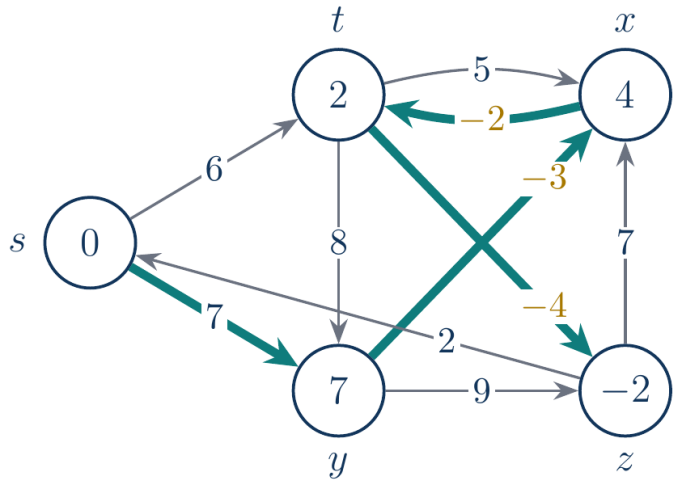
Tabel 16.7 Empat putaran Bellman–Ford pada graf **Gambar 16.7**. Perhatikan bahwa $d[t]$ *turun* dari 6 ke 2 pada putaran 2 — taksiran yang oleh Dijkstra sudah dituntaskan — dan bahwa putaran 4 tidak mengubah apa pun, yang lalu dikukuhkan oleh pemeriksaan penutupnya.

Sesudah	$d[s]$	$d[t]$	$d[x]$	$d[y]$	$d[z]$
inisialisasi	0	∞	∞	∞	∞
putaran 1	0	6	4	7	2
putaran 2	0	2	4	7	2
putaran 3	0	2	4	7	−2
putaran 4	0	2	4	7	−2

Periksalah putaran 1 dengan tangan. Dimulai dari $d[s] = 0$ dengan yang lain takhingga, relaksasi-relaksasinya secara berurutan adalah: $s \to t$ memberi $d[t] = 6$; $s \to y$ memberi $d[y] = 7$; $t \to x$

memberi $d[x] = 6 + 5 = 11$; $t \to y$ memberi $6 + 8 = 14 \not< 7$, tidak berubah; $t \to z$ memberi $d[z] = 6 - 4 = 2$; $x \to t$ memberi $11 - 2 = 9 \not< 6$; $y \to x$ memberi $7 - 3 = 4 < 11$, jadi $d[x] = 4$; $y \to z$ memberi $7 + 9 = 16 \not< 2$; $z \to s$ memberi $2 + 2 = 4 \not< 0$; $z \to x$ memberi $2 + 7 = 9 \not< 4$. Akhir putaran 1: (0,6,4,7,2) — sesuai dengan tabelnya.

Gambar 16.8 adalah keadaan ketika algoritmanya berhenti.



Gambar 16.8 Bellman–Ford ketika berhenti: $d[s] = 0$, $d[t] = 2$, $d[x] = 4$, $d[y] = 7$, $d[z] = -2$. Sisi-sisi tebal adalah pohon lintasan terpendek. Lintasan ke t adalah $s \to y \to x \to t$, berbiaya $7 - 3 - 2 = 2$ — tiga sisi, padahal sisi langsung $s \to t$ berbiaya 6.

16.7.6 Programnya

Kode 16.6 menjalankan kedua algoritma pada dua graf Bab 13 dan mencetak vektor taksiran sesudah setiap langkah, sehingga **Tabel 16.7** dan penelusuran Dijkstra dapat diperiksa secara mekanis.

Kode 16.6 Kedua algoritma Persoalan 16.5 pada dua graf Bab 13, mencetak vektor taksiran sesudah setiap pengeluaran dan sesudah setiap putaran.

```
// Persoalan 16.5: Dijkstra dan Bellman-Ford, dijalankan pada kedua graf contoh
// Bab 13, dengan taksiran d[v] dicetak sesudah tiap langkah.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

const int INF = INT_MAX / 4;
struct Edge { int u, v, w; };

void show(const string& tag, const vector<string>& nm, const vector<int>& d) {
    cout << setw(14) << tag << " :";
    for (size_t i = 0; i < d.size(); ++i) {
        cout << "  " << nm[i] << "=";
        if (d[i] >= INF) cout << setw(3) << "inf"; else cout << setw(3) << d[i];
    }
    cout << "\n";
}

void dijkstra(const vector<string>& nm, const vector<Edge>& E, int s) {
    int n = (int)nm.size();
    vector<vector<pair<int,int>>> adj(n);
    for (auto& e : E) adj[e.u].push_back({e.v, e.w});
    vector<int> d(n, INF); d[s] = 0;
    vector<bool> done(n, false);
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    Q.push({0, s});
    show("initialize", nm, d);
    while (!Q.empty()) {
        auto [du, u] = Q.top(); Q.pop();
        if (done[u]) continue;                 // entri basi: lewati saja
        done[u] = true;
        for (auto [v, w] : adj[u])
            if (d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
        show("extract " + nm[u], nm, d);
    }
}
```

Kode 16.6 (Lanjutan) *Kedua algoritma Persoalan 16.5 pada dua graf Bab 13, mencetak vektor taksiran sesudah setiap pengeluaran dan sesudah setiap putaran.*

```
bool bellmanFord(const vector<string>& nm, const vector<Edge>& E, int s) {
    int n = (int)nm.size();
    vector<int> d(n, INF); d[s] = 0;
    show("initialize", nm, d);
    for (int pass = 1; pass <= n - 1; ++pass) {
        for (auto& e : E)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) d[e.v] = d[e.u] + e.w;
        show("pass " + to_string(pass), nm, d);
    }
    for (auto& e : E)
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) return false;   // siklus negatif
    return true;
}

int main() {
    cout << "=== Dijkstra on the Chapter 13 graph (s,u,v,x,y) ===\n";
    vector<string> nmD = {"s","u","v","x","y"};
    vector<Edge> ED = {{0,1,10},{0,3,5},{1,2,1},{1,3,2},{3,1,3},
                       {3,2,9},{3,4,2},{4,2,6},{4,0,7},{2,4,4}};
    dijkstra(nmD, ED, 0);

    cout << "\n=== Bellman-Ford on the Chapter 13 graph (s,t,x,y,z) ===\n";
    vector<string> nmB = {"s","t","x","y","z"};
    vector<Edge> EB = {{0,1,6},{0,3,7},{1,2,5},{1,3,8},{1,4,-4},
                       {2,1,-2},{3,2,-3},{3,4,9},{4,0,2},{4,2,7}};
    bool ok = bellmanFord(nmB, EB, 0);
    cout << "returns " << (ok ? "true" : "false") << " (no negative cycle)\n";
    return 0;
}
```

Dikompilasi dengan `g++ -std=c++17 -O2 -Wall` lalu dijalankan:

```
=== Dijkstra on the Chapter 13 graph (s,u,v,x,y) ===
    initialize :  s=  0  u=inf  v=inf  x=inf  y=inf
     extract s :  s=  0  u= 10  v=inf  x=  5  y=inf
     extract x :  s=  0  u=  8  v= 14  x=  5  y=  7
     extract y :  s=  0  u=  8  v= 13  x=  5  y=  7
     extract u :  s=  0  u=  8  v=  9  x=  5  y=  7
     extract v :  s=  0  u=  8  v=  9  x=  5  y=  7

=== Bellman-Ford on the Chapter 13 graph (s,t,x,y,z) ===
    initialize :  s=  0  t=inf  x=inf  y=inf  z=inf
        pass 1 :  s=  0  t=  6  x=  4  y=  7  z=  2
        pass 2 :  s=  0  t=  2  x=  4  y=  7  z=  2
        pass 3 :  s=  0  t=  2  x=  4  y=  7  z= -2
        pass 4 :  s=  0  t=  2  x=  4  y=  7  z= -2
returns true (no negative cycle)
```

Setiap nilainya cocok dengan **Gambar 16.6**, **Gambar 16.8**, dan **Tabel 16.7**.

16.7.7 Memperlihatkan *mengapa* Dijkstra gagal, bukan sekadar *bahwa* ia gagal

Perbedaannya lebih meyakinkan jika Anda dapat memperagakannya. **Kode 16.7** mengerjakan dua hal: ia memperlihatkan graf empat simpul yang membuat Dijkstra mengembalikan jarak yang keliru, dan ia memperlihatkan Bellman–Ford menolak graf yang bersiklus negatif.

Kode 16.7 Kegagalan Dijkstra pada sisi negatif, dan deteksi siklus negatif oleh Bellman–Ford, pada dua graf terkecil yang memperlihatkan masing-masingnya.

```
// Mengapa Dijkstra butuh bobot taknegatif, dan apa gantinya kerja Bellman-Ford.
// Bagian 1: graf empat simpul dengan satu sisi negatif, tempat Dijkstra
//           melaporkan jarak yang sama sekali keliru.
// Bagian 2: graf dengan siklus berbobot negatif, yang dideteksi Bellman-Ford.
#include <iostream>
#include <string>
#include <vector>
#include <climits>
using namespace std;

const int INF = INT_MAX / 4;
struct Edge { int u, v, w; };

// Dijkstra, dengan nilainya DIBEKUKAN pada saat sebuah simpul meninggalkan
// antreannya -- persis yang dijanjikan algoritmanya dan yang dibaca pemanggil.
vector<int> dijkstra(int n, const vector<Edge>& E, int s) {
    vector<vector<pair<int,int>>> adj(n);
    for (auto& e : E) adj[e.u].push_back({e.v, e.w});
    vector<int> d(n, INF), frozen(n, INF);
    vector<bool> done(n, false);
    d[s] = 0;
    for (int it = 0; it < n; ++it) {
        int u = -1;
        for (int v = 0; v < n; ++v) if (!done[v] && (u == -1 || d[v] < d[u])) u = v;
        if (u == -1 || d[u] >= INF) break;
        done[u] = true;
        frozen[u] = d[u];                       // sudah terikat: tak pernah ditengok lagi
        for (auto [v, w] : adj[u])
            if (!done[v] && d[u] + w < d[v]) d[v] = d[u] + w;
    }
    return frozen;
}

bool bellmanFord(int n, const vector<Edge>& E, int s, vector<int>& d) {
    d.assign(n, INF); d[s] = 0;
    for (int i = 1; i <= n - 1; ++i)
        for (auto& e : E)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) d[e.v] = d[e.u] + e.w;
    for (auto& e : E)
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) return false;
    return true;
}

int main() {
    const string nm[4] = {"s", "a", "b", "c"};
```

Kode 16.7 (Lanjutan) *Kegagalan Dijkstra pada sisi negatif, dan deteksi siklus negatif oleh Bellman–Ford, pada dua graf terkecil yang memperlihatkan masing-masingnya.*

```
    // s->a 2, s->b 3, b->a -2, a->c 1.  Benarnya: s->b->a->c berbiaya 3-2+1 = 2.
    vector<Edge> G1 = {{0,1,2}, {0,2,3}, {2,1,-2}, {1,3,1}};
    vector<int> dj = dijkstra(4, G1, 0), bf;
    bellmanFord(4, G1, 0, bf);
    cout << "graph 1:  s->a = 2,  s->b = 3,  b->a = -2,  a->c = 1\n";
    cout << "  vertex   Dijkstra   Bellman-Ford\n";
    for (int i = 0; i < 4; ++i)
        cout << "     " << nm[i] << "          " << dj[i] << "             " << bf[i] <<
"\n";
    cout << "  the true shortest path to c is s->b->a->c = 3 + (-2) + 1 = 2;\n"
         << "  Dijkstra froze d[a] = 2 before the edge b->a could improve it.\n";

    // s->a 2, a->b 1, b->a -4: siklus a->b->a berbobot -3.
    vector<Edge> G2 = {{0,1,2}, {1,2,1}, {2,1,-4}};
    vector<int> d3;
    bool ok = bellmanFord(3, G2, 0, d3);
    cout << "\ngraph 2:  s->a = 2,  a->b = 1,  b->a = -4   (cycle weight -3)\n"
         << "  Bellman-Ford returns " << (ok ? "true" : "false")
         << " -- a negative-weight cycle is reachable from s\n";
    return 0;
}
```

Keluarannya:

```
graph 1:  s->a = 2,  s->b = 3,  b->a = -2,  a->c = 1
  vertex   Dijkstra   Bellman-Ford
     s          0             0
     a          2             1
     b          3             3
     c          3             2
  the true shortest path to c is s->b->a->c = 3 + (-2) + 1 = 2;
  Dijkstra froze d[a] = 2 before the edge b->a could improve it.

graph 2:  s->a = 2,  a->b = 1,  b->a = -4   (cycle weight -3)
  Bellman-Ford returns false -- a negative-weight cycle is reachable from s
```

Ikutilah graf 1 dengan tangan dan mekanismenya menjadi terang. Dijkstra mengeluarkan s, menetapkan $d[a] = 2$ dan $d[b] = 3$, lalu mengeluarkan a — karena $2 < 3$ — menyatakan $d[a] = 2$ tuntas, dan merelaksasi $a \to c$ sehingga $d[c] = 3$. Baru sesudahnya ia mengeluarkan b dan menemukan bahwa $s \to b \to a$ berbiaya $3 - 2 = 1$. Tetapi a sudah selesai; sisi keluarnya tidak pernah direlaksasi lagi, jadi $d[c]$ tetap 3 padahal jawaban yang benar 2. **Pilihan rakus Dijkstra hanya aman ketika memperpanjang sebuah lintasan tidak dapat membuatnya lebih pendek — yaitu ketika bobotnya tak negatif.**

16.7.8 Kesalahan yang paling sering terjadi pada persoalan ini

Yang pertama adalah menulis "Bellman–Ford lebih lambat, jadi pakailah Dijkstra kapan pun bisa" lalu berhenti. Kalimat itu benar, tetapi ia hanya menyebut satu baris **Tabel 16.6** dan tidak yang lain. Baris-baris yang membedakan jawaban yang baik dari jawaban yang sekadar cukup adalah *siklus negatif dideteksi* dan *tiap simpul hanya butuh tetangganya* — yang kedua adalah alasan *router* memakai algoritma yang secara asimtotik lebih buruk, yaitu pernyataan (12) Persoalan 16.1 lagi, dalam sistem yang nyata.

Yang kedua menyangkut bagian 3: menggambar grafnya tetapi tidak bilangannya. Ilustrasi algoritma lintasan terpendek yang tidak memperlihatkan $d[v]$ berubah adalah ilustrasi sebuah graf, bukan sebuah algoritma. Letakkan taksirannya di dalam simpul, seperti pada **Gambar 16.5** dan **Gambar 16.8**, dan perlihatkan paling sedikit dua keadaan antara.

Uji diri. Pada graf 1 **Kode 16.7**, ubahlah bobot $b \to a$ dari -2 menjadi $+2$. Apa yang kini dilaporkan Dijkstra dan Bellman–Ford, dan mengapa keduanya sepakat?

Jawaban. Keduanya melaporkan $d = (0,2,3,3)$ untuk (s, a, b, c). Dengan semua bobot tak negatif, pilihan rakusnya aman: ketika a dikeluarkan pada 2, tak ada lintasan lewat b yang dapat mengalahkannya, sebab lintasan seperti itu sudah berbiaya paling sedikit $d[b] = 3 > 2$ sebelum sisi terakhirnya — yang kini berbobot $+2$, bukan -2.

16.8 Persoalan Terkerjakan tentang Lintasan Terpendek Sumber Tunggal

Persoalan 16.5 membandingkan kedua algoritma besar sumber tunggal dengan kata-kata. Tiga persoalan dalam subbab ini membuat Anda *menjalankan* keduanya, dengan tangan, pada graf yang belum pernah Anda lihat, dengan setiap taksiran dan setiap pendahulu dituliskan sesudah setiap langkah. Ketiganya melatih Bab 13 dalam urutan pengajarannya: Dijkstra, lalu Bellman–Ford dengan uji siklus negatifnya, lalu algoritma DAG.

16.8.1 Persoalan 16.6 — Dijkstra, dengan seluruh tabel sesudah setiap iterasi

Persoalan 16.6. Jalankan algoritma Dijkstra dari s pada graf berarah dengan sepuluh sisi

$$s \to a\ (4),\quad s \to b\ (1),\quad b \to a\ (2),\quad a \to c\ (5),\quad b \to c\ (8),\quad b \to d\ (10),\quad c \to d\ (2),\quad c \to e\ (6),\quad d \to e\ (1),\quad a \to d\ (9).$$

Sesudah setiap `Extract-Min`, berikan taksiran $d[v]$ dan pendahulu $\pi[v]$ setiap simpul. Lalu berikan pohon lintasan terpendeknya, lintasan terpendek ke setiap simpul, dan banyaknya relaksasi yang berhasil.

Apa yang diuji. Mekanika algoritma Dijkstra (Subbab 13.7): pemilihan simpul bertaksiran terkecil di antara yang belum selesai, relaksasi sisi-sisi keluarnya dengan Persamaan (16.1), dan pembukuan larik pendahulu (Subbab 13.10) yang mengubah tabel bilangan menjadi rute yang sesungguhnya.

Penyelesaian. Semua bobot tak negatif, jadi Dijkstra dapat dipakai. Tuliskan S untuk himpunan simpul yang sudah selesai. Inisialisasi menetapkan $d[s] = 0$, untuk setiap simpul lain $d[v] = \infty$, setiap $\pi[v] = \text{NIL}$, dan $S = \emptyset$.

Iterasi 1. Taksiran yang belum selesai adalah s: 0, selebihnya ∞; keluarkan s. $S = \{s\}$. Relaksasi sisi-sisi keluarnya:

- $s \to a$: $d[s] + 4 = 0 + 4 = 4 < \infty$, jadi $d[a] \leftarrow 4, \pi[a] \leftarrow s$.
- $s \to b$: $0 + 1 = 1 < \infty$, jadi $d[b] \leftarrow 1, \pi[b] \leftarrow s$.

Iterasi 2. Belum selesai: a: 4, b: 1, c, d, e: ∞. Yang terkecil b pada 1; keluarkan. $S = \{s, b\}$.

- $b \to a$: $1 + 2 = 3 < 4$, jadi $d[a] \leftarrow 3$, $\pi[a] \leftarrow b$. Inilah perbaikan pertama atas taksiran yang sudah berhingga: sisi langsung $s \to a$ berbobot 4 kalah oleh rute dua-sisi $s \to b \to a$ berbobot 3.
- $b \to c$: $1 + 8 = 9 < \infty$, jadi $d[c] \leftarrow 9, \pi[c] \leftarrow b$.
- $b \to d$: $1 + 10 = 11 < \infty$, jadi $d[d] \leftarrow 11, \pi[d] \leftarrow b$.

Iterasi 3. Belum selesai: a: 3, c: 9, d: 11, e: ∞. Keluarkan a pada 3. $S = \{s, b, a\}$.

- $a \to c$: $3 + 5 = 8 < 9$, jadi $d[c] \leftarrow 8, \pi[c] \leftarrow a$.

- $a \to d$: $3 + 9 = 12 \not< 11$. Tidak berubah. Inilah satu-satunya relaksasi yang gagal sepanjang jalannya algoritma.

Iterasi 4. Belum selesai: c: 8, d: 11, e: ∞. Keluarkan c pada 8. $S = \{s, b, a, c\}$.

- $c \to d$: $8 + 2 = 10 < 11$, jadi $d[d] \leftarrow 10$, $\pi[d] \leftarrow c$.
- $c \to e$: $8 + 6 = 14 < \infty$, jadi $d[e] \leftarrow 14$, $\pi[e] \leftarrow c$.

Iterasi 5. Belum selesai: d: 10, e: 14. Keluarkan d pada 10. $S = \{s, b, a, c, d\}$.

- $d \to e$: $10 + 1 = 11 < 14$, jadi $d[e] \leftarrow 11$, $\pi[e] \leftarrow d$.

Iterasi 6. Keluarkan e pada 11. Ia tidak mempunyai sisi keluar. Antreannya kosong dan algoritmanya berhenti.

Tabel 16.8 menghimpun keenam iterasi itu. Setiap sel ditulis $d[v]/\pi[v]$, dengan $-$ untuk NIL, dan sebuah sel ditebalkan ketika simpulnya dikeluarkan pada baris itu — sejak saat itu, menurut teorema kebenaran di Subbab 13.7.8, nilainya tidak pernah berubah.

Tabel 16.8 Algoritma Dijkstra pada graf Persoalan 16.6, satu baris per `Extract-Min`. Setiap sel adalah $d[v]/\pi[v]$; sel tebal adalah simpul yang dituntaskan pada baris itu.

Sesudah	Dikeluarkan	s	a	b	c	d	e
inisialisasi	—	$0/-$	$\infty/-$	$\infty/-$	$\infty/-$	$\infty/-$	$\infty/-$
1	s	**0/NIL**	$4/s$	$1/s$	$\infty/-$	$\infty/-$	$\infty/-$
2	b	$0/-$	$3/b$	**1/s**	$9/b$	$11/b$	$\infty/-$
3	a	$0/-$	**3/b**	$1/s$	$8/a$	$11/b$	$\infty/-$
4	c	$0/-$	$3/b$	$1/s$	**8/a**	$10/c$	$14/c$
5	d	$0/-$	$3/b$	$1/s$	$8/a$	**10/c**	$11/d$
6	e	$0/-$	$3/b$	$1/s$	$8/a$	$10/c$	**11/d**

Jawaban. $\delta(s,\cdot) = (s{:}\,0,\ a{:}\,3,\ b{:}\,1,\ c{:}\,8,\ d{:}\,10,\ e{:}\,11)$. Mengikuti π ke belakang dari setiap simpul memberikan rute-rutenya:

$$e \leftarrow d \leftarrow c \leftarrow a \leftarrow b \leftarrow s, \qquad \text{jadi lintasan ke } e \text{ adalah } s \to b \to a \to c \to d \to e, \quad 1 + 2 + 5 + 2 + 1 = 11.$$

Di sini pohon lintasan terpendeknya kebetulan berupa satu lintasan, $s \to b \to a \to c \to d \to e$; rute setiap simpul adalah awalan darinya. Dari sepuluh sisinya, setiap sisi direlaksasi tepat sekali, dan sembilan relaksasi berhasil; empat dari sembilan itu (di a, c, d, dan e) menurunkan taksiran yang sudah berhingga, yang dalam implementasi *heap* adalah sebuah `Decrease-Key`.

Memeriksa jawabannya secara mandiri. Ambillah d, simpul dengan rute bersaing terbanyak, lalu daftarkan semuanya: $s \to b \to d$ berbiaya $1 + 10 = 11$; $s \to a \to d$ berbiaya $4 + 9 = 13$; $s \to b \to a \to d$ berbiaya $1 + 2 + 9 = 12$; $s \to b \to c \to d$ berbiaya $1 + 8 + 2 = 11$; $s \to a \to c \to d$ berbiaya $4 + 5 + 2 = 11$; $s \to b \to a \to c \to d$ berbiaya $1 + 2 + 5 + 2 = 10$. Minimumnya 10, sesuai dengan tabelnya. Perhatikan bahwa tiga rute berbeda seri pada 11; Dijkstra memegang salah satunya, $s \to b \to d$, selama dua iterasi sebelum rute lewat a dan c menggantikannya.

Program. **Kode 16.8** mencetak tabel yang sama.

Kode 16.8 Algoritma Dijkstra pada graf Persoalan 16.6, mencetak pasangan $d[v]/\pi[v]$ untuk setiap simpul sesudah setiap pengeluaran, lalu setiap lintasan terpendek yang dibaca balik lewat π.

```
// Persoalan 16.6: algoritma Dijkstra dengan seluruh tabel (d, pi) dicetak sesudah
// tiap EXTRACT-MIN, sehingga setiap baris penelusuran tangan dapat diperiksa mesin.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

const int INF = INT_MAX / 4;
struct Edge { int u, v, w; };

int main() {
    vector<string> nm = {"s", "a", "b", "c", "d", "e"};
    vector<Edge> E = {{0,1,4}, {0,2,1}, {2,1,2}, {1,3,5}, {2,3,8},
                      {2,4,10}, {3,4,2}, {3,5,6}, {4,5,1}, {1,4,9}};
    int n = (int)nm.size();
    vector<vector<pair<int,int>>> adj(n);
    for (auto& e : E) adj[e.u].push_back({e.v, e.w});

    vector<int> d(n, INF), pi(n, -1);
    vector<bool> done(n, false);
    d[0] = 0;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    Q.push({0, 0});

    auto row = [&](const string& tag) {
        cout << setw(11) << left << tag << right;
        for (int v = 0; v < n; ++v) {
            string cell = (d[v] >= INF) ? "inf" : to_string(d[v]);
            cell += "/";
            cell += (pi[v] < 0) ? "-" : nm[pi[v]];
            cout << setw(8) << cell;
        }
        cout << "\n";
    };
    cout << setw(11) << left << "step" << right;
    for (int v = 0; v < n; ++v) cout << setw(8) << ("d/pi " + nm[v]);
    cout << "\n";
    row("init");
```

Kode 16.8 (Lanjutan) *Algoritma Dijkstra pada graf Persoalan 16.6, mencetak pasangan d[v]/\pi[v] untuk setiap simpul sesudah setiap pengeluaran, lalu setiap lintasan terpendek yang dibaca balik lewat \pi.*

```
    int step = 0;
    while (!Q.empty()) {
        auto [du, u] = Q.top(); Q.pop();
        if (done[u] || du > d[u]) continue;       // entri antrean yang basi
        done[u] = true;
        for (auto [v, w] : adj[u])
            if (d[u] + w < d[v]) { d[v] = d[u] + w; pi[v] = u; Q.push({d[v], v}); }
        row(to_string(++step) + ": " + nm[u]);
    }

    cout << "\nshortest paths from s:\n";
    for (int v = 0; v < n; ++v) {
        vector<int> path;
        for (int x = v; x != -1; x = pi[x]) path.push_back(x);
        cout << "  " << nm[v] << "  d = " << setw(2) << d[v] << "  path:";
        for (int i = (int)path.size() - 1; i >= 0; --i) cout << " " << nm[path[i]];
        cout << "\n";
    }
    return 0;
}
```

Ia mencetak:

```
step         d/pi s  d/pi a  d/pi b  d/pi c  d/pi d  d/pi e
init            0/-   inf/-   inf/-   inf/-   inf/-   inf/-
1: s            0/-     4/s     1/s   inf/-   inf/-   inf/-
2: b            0/-     3/b     1/s     9/b    11/b   inf/-
3: a            0/-     3/b     1/s     8/a    11/b   inf/-
4: c            0/-     3/b     1/s     8/a    10/c    14/c
5: d            0/-     3/b     1/s     8/a    10/c    11/d
6: e            0/-     3/b     1/s     8/a    10/c    11/d

shortest paths from s:
  s  d =  0  path: s
  a  d =  3  path: s b a
  b  d =  1  path: s b
  c  d =  8  path: s b a c
  d  d = 10  path: s b a c d
  e  d = 11  path: s b a c d e
```

Baris demi baris, keluarannya adalah **Tabel 16.8**.

Uji diri. Ubahlah bobot $s \to b$ dari 1 menjadi 5 lalu jalankan algoritmanya lagi. Berikan urutan pengeluarannya dan nilai d/π yang dihasilkan.

Jawaban. Urutan pengeluarannya menjadi s, a, b, c, d, e, dan nilai ketika berhenti adalah $a: 4/s$, $b: 5/s$, $c: 9/a$, $d: 11/c$, $e: 12/d$. Ketika b dikeluarkan pada 5, ketiga relaksasinya gagal — $5 + 2 = 7 \not< 4$, $5 + 8 = 13 \not< 9$, $5 + 10 = 15 \not< 13$ — jadi b berakhir sebagai daun pohon lintasan terpendek, yang kini berupa $s \to a \to c \to d \to e$ ditambah cabang tersendiri $s \to b$.

16.8.2 Persoalan 16.7 — Bellman–Ford, dan sebuah siklus negatif ditemukan lalu diperlihatkan

Persoalan 16.7. Sebuah graf berarah pada simpul-simpul s, a, b, c, d mempunyai tujuh sisi di bawah ini, dan Bellman–Ford merelaksasinya persis dalam urutan ini pada setiap putaran:

$$(b, c, -2), \quad (a, b, 4), \quad (s, a, 3), \quad (s, b, 6), \quad (c, d, 3), \quad (a, c, 5), \quad (d, b, 1).$$

(a) Jalankan Bellman–Ford dari s, dengan memberikan d/π sesudah masing-masing dari $|V| - 1 = 4$ putaran, dan hasil pemeriksaan penutupnya. (b) Kini ubahlah bobot $d \to b$ dari 1 menjadi -2 lalu jalankan lagi. Tunjukkan bahwa pemeriksaan penutupnya gagal, dan pakailah larik pendahulunya untuk memperlihatkan sebuah siklus berbobot negatif.

Apa yang diuji. Bellman–Ford persis sebagaimana diberikan Subbab 13.8, termasuk tiga hal yang mudah dinyatakan dan mudah dikeliru: bahwa *urutan* daftar sisi mengubah seberapa cepat taksiran-taksirannya mapan tetapi tidak mengubah di mana mereka mapan; bahwa pemeriksaan penutup adalah pendeteksi siklus negatifnya; dan bahwa larik pendahulu, yang biasanya menampung pohon lintasan terpendek, menampung sebuah siklus ketika ada siklus negatif (Subbab 13.8.8).

Penyelesaian (a). Inisialisasikan $d[s] = 0$, yang lain ∞, semua $\pi = \text{NIL}$. Perhatikan lebih dahulu siklus-siklus dalam grafnya: $b \to c \to d \to b$ berbobot $-2 + 3 + 1 = 2 > 0$, dan itulah satu-satunya siklus, jadi tidak ada siklus negatif dan algoritmanya seharusnya mengembalikan TRUE.

Putaran 1, merelaksasi dalam urutan yang didaftar:

1. (b, c): $d[b] = \infty$, tak ada yang dapat direlaksasi darinya.
2. (a, b): $d[a] = \infty$, tak ada.
3. (s, a): $0 + 3 = 3 < \infty$, jadi $d[a] = 3$, $\pi[a] = s$.

4. (s, b): $0 + 6 = 6 < \infty$, jadi $d[b] = 6, \pi[b] = s$.
5. (c, d): $d[c] = \infty$, tak ada.
6. (a, c): $3 + 5 = 8 < \infty$, jadi $d[c] = 8, \pi[c] = a$.
7. (d, b): $d[d] = \infty$, tak ada.

Tiga perubahan. Taksiran untuk d masih ∞ walaupun c sudah tercapai pada putaran ini — karena (c, d) diperiksa *sebelum* (a, c) memberi c taksiran yang berhingga. Urutan sisinya telah merugikan kita satu putaran.

Putaran 2:

1. (b, c): $6 + (-2) = 4 < 8$, jadi $d[c] = 4, \pi[c] = b$.
2. (a, b): $3 + 4 = 7 \not< 6$.
3. (s, a) dan 4. (s, b): tidak berubah.
4. (c, d): $4 + 3 = 7 < \infty$, jadi $d[d] = 7, \pi[d] = c$.
5. (a, c): $3 + 5 = 8 \not< 4$.
6. (d, b): $7 + 1 = 8 \not< 6$.

Dua perubahan. *Putaran 3 dan 4* tidak mengubah apa pun: setiap sisi diperiksa dan setiap ujinya gagal. *Pemeriksaan penutup* adalah sapuan kelima yang hanya menguji; karena putaran 4 tidak mengubah apa pun, sapuan itu pun tidak dapat menemukan apa-apa, dan algoritmanya mengembalikan TRUE.

Tabel 16.9 Bellman–Ford pada graf Persoalan 16.7(a), dengan $d[v]/\pi[v]$ sesudah setiap putaran. Taksiran-taksirannya mapan sesudah putaran 2; putaran 3 dan 4 tidak mengubah apa pun dan pemeriksaan penutupnya mengembalikan TRUE.

Sesudah	s	a	b	c	d	Perubahan
inisialisasi	$0/-$	$\infty/-$	$\infty/-$	$\infty/-$	$\infty/-$	—
putaran 1	$0/-$	$3/s$	$6/s$	$8/a$	$\infty/-$	3
putaran 2	$0/-$	$3/s$	$6/s$	$4/b$	$7/c$	2
putaran 3	$0/-$	$3/s$	$6/s$	$4/b$	$7/c$	0
putaran 4	$0/-$	$3/s$	$6/s$	$4/b$	$7/c$	0

Periksalah jawabannya terhadap sifat relaksasi lintasan. Lintasan terpendek ke d adalah $s \to b \to c \to d$, berbobot $6 - 2 + 3 = 7$, tiga sisi. Sisi pertamanya (s, b) direlaksasi pada putaran 1; sisi keduanya, (b, c), pada putaran 2; sisi ketiganya, (c, d), juga pada putaran 2, karena ia berada *sesudah* (b, c) dalam daftar. Jadi lintasan itu terelaksasi penuh, berurutan, pada akhir putaran 2 — tepat ketika $d[d]$ menjadi 7. Batas $|V| - 1 = 4$ putaran adalah kasus terburuk atas urutan sisi; urutan ini hanya membutuhkan dua.

Jawaban (a): $\delta(s,\cdot) = (s{:}\,0,\ a{:}\,3,\ b{:}\,6,\ c{:}\,4,\ d{:}\,7)$, pohon pendahulu $a \leftarrow s, b \leftarrow s, c \leftarrow b, d \leftarrow c$, dan algoritmanya mengembalikan TRUE.

Penyelesaian (b). Kini $b \to c \to d \to b$ berbobot $-2 + 3 + (-2) = -1 < 0$: sebuah siklus negatif, terjangkau dari s lewat $s \to b$. Karena itu bobot lintasan terpendek ke b, c, dan d adalah $-\infty$, dan Bellman–Ford harus menyatakannya dengan mengembalikan FALSE.

Putaran 1 identik dengan (a), sebab $d[d] = \infty$ sepanjang putaran itu: $(0,3,6,8,\infty)$.

Putaran 2: (b, c) memberi $d[c] = 6 - 2 = 4$ $(\pi[c] = b)$; (c, d) memberi $d[d] = 4 + 3 = 7$ $(\pi[d] = c)$; dan kini (d, b) memberi $7 + (-2) = 5 < 6$, jadi $d[b] = 5, \pi[b] = d$. Tiga perubahan.

Putaran 3: (b, c): $5 - 2 = 3 < 4$, jadi $d[c] = 3$; (c, d): $3 + 3 = 6 < 7$, jadi $d[d] = 6$; (d, b): $6 - 2 = 4 < 5$, jadi $d[b] = 4$. Tiga perubahan — setiap taksiran pada siklus itu turun tepat 1, bobot satu kali mengelilinginya.

Putaran 4: ketiga sisi yang sama menyala lagi: $d[c] = 2$, $d[d] = 5$, $d[b] = 3$.

Pemeriksaan penutup: (b, c): $d[b] + w(b, c) = 3 + (-2) = 1 < 2 = d[c]$. Sebuah sisi masih dapat direlaksasi sesudah $|V| - 1$ putaran, jadi algoritmanya mengembalikan FALSE.

Tabel 16.10 Bellman–Ford pada graf Persoalan 16.7(b). Ketiga taksiran pada siklus itu turun 1 pada setiap putaran, dan pemeriksaan penutupnya menemukan bahwa (b, c) masih dapat direlaksasi, sebab $3 + (-2) = 1 < 2$.

Sesudah	s	a	b	c	d	Perubahan
inisialisasi	$0/-$	$\infty/-$	$\infty/-$	$\infty/-$	$\infty/-$	—
putaran 1	$0/-$	$3/s$	$6/s$	$8/a$	$\infty/-$	3
putaran 2	$0/-$	$3/s$	$5/d$	$4/b$	$7/c$	3
putaran 3	$0/-$	$3/s$	$4/d$	$3/b$	$6/c$	3
putaran 4	$0/-$	$3/s$	$3/d$	$2/b$	$5/c$	3

Memperlihatkan siklusnya. Deteksi saja hanya mengatakan *bahwa* ada siklus negatif; larik pendahulunya mengatakan *di mana*. Mulailah di c, kepala sisi yang gagal dalam pemeriksaan, lalu ikutilah π ke belakang $|V| = 5$ kali: $c \to b \to d \to c \to b \to d$. Kita berdiri di d. Mengapa lima langkah cukup untuk berada *pada* siklusnya? Rantai enam simpul yang diambil dari hanya lima simpul pasti mengulang salah satunya, jadi sesudah $|V|$ langkah penelusurannya sudah memasuki putaran yang tak pernah dapat ditinggalkannya; simpul yang dicapai berada di atasnya. (Bahwa rantai dari simpul yang masih dapat diperbaiki menuju sebuah siklus, dan bukan kembali ke s, adalah argumen subgraf pendahulu dalam Cormen dkk. [1, ch. 22].) Kini berjalanlah dari d sampai kita kembali ke d: $d \to \pi[d] = c \to \pi[c] = b \to \pi[b] = d$. Membalik urutan perjumpaan kita memberi siklusnya searah sisi-sisinya:

$$b \to c \to d \to b, \qquad w = -2 + 3 + (-2) = -1.$$

Program. **Kode 16.9** menjalankan kedua bagian.

Kode 16.9 Bellman–Ford putaran demi putaran pada graf-graf Persoalan 16.7, dengan banyaknya perubahan pada setiap putaran, pemeriksaan penutup, dan — ketika pemeriksaan itu gagal — siklus negatif yang dipulihkan dari larik pendahulu.

```
// Persoalan 16.7: Bellman-Ford putaran demi putaran, mula-mula pada graf bersisi
// negatif tanpa siklus negatif, lalu pada graf yang sama dengan satu bobot diubah
// sehingga muncul siklus negatif -- yang dideteksi putaran tambahan dan kemudian
// diperlihatkan oleh larik pendahulu.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <climits>
using namespace std;

const int INF = INT_MAX / 4;
struct Edge { int u, v, w; };
const vector<string> nm = {"s", "a", "b", "c", "d"};

void show(const string& tag, const vector<int>& d, const vector<int>& pi) {
    cout << setw(8) << left << tag << right;
    for (size_t v = 0; v < d.size(); ++v) {
        string cell = (d[v] >= INF) ? "inf" : to_string(d[v]);
        cell += "/";
        cell += (pi[v] < 0) ? "-" : nm[pi[v]];
        cout << setw(8) << cell;
    }
    cout << "\n";
}

void run(const vector<Edge>& E) {
    int n = (int)nm.size();
    vector<int> d(n, INF), pi(n, -1);
    d[0] = 0;
    cout << setw(8) << left << "pass" << right;
    for (int v = 0; v < n; ++v) cout << setw(8) << ("d/pi " + nm[v]);
    cout << "\n";
    show("init", d, pi);
    for (int pass = 1; pass <= n - 1; ++pass) {
        int changes = 0;
        for (auto& e : E)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) {
                d[e.v] = d[e.u] + e.w; pi[e.v] = e.u; ++changes;
            }
        show(to_string(pass) + " (" + to_string(changes) + ")", d, pi);
    }
    // putaran tambahan: masih adakah yang dapat direlaksasi?
    for (auto& e : E)
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) {
            cout << "check: edge " << nm[e.u] << "->" << nm[e.v] << " still relaxes ("
```

Kode 16.9 (Lanjutan) *Bellman–Ford putaran demi putaran pada graf-graf Persoalan 16.7, dengan banyaknya perubahan pada setiap putaran, pemeriksaan penutup, dan — ketika pemeriksaan itu gagal — siklus negatif yang dipulihkan dari larik pendahulu.*

```
                << d[e.u] << " + (" << e.w << ") = " << d[e.u] + e.w << " < " << d[e.v]
                << ")  =>  return FALSE\n";
            int x = e.v;
            pi[x] = e.u;                          // catat sisi yang memperbaiki
            for (int i = 0; i < n; ++i) x = pi[x]; // mundur n langkah: kini di siklus
            vector<int> cyc = {x};
            for (int y = pi[x]; y != x; y = pi[y]) cyc.push_back(y);
            cout << "negative cycle:";
            int total = 0;
            for (int i = (int)cyc.size() - 1; i >= 0; --i) cout << " " << nm[cyc[i]];
            cout << " " << nm[cyc.back()] << "\n";
            for (size_t i = 0; i < cyc.size(); ++i) {
                int to = cyc[i], from = cyc[(i + 1) % cyc.size()];
                for (auto& f : E) if (f.u == from && f.v == to) total += f.w;
            }
            cout << "its weight: " << total << "\n";
            return;
        }
    cout << "check: no edge relaxes  =>  return TRUE (no negative cycle reachable)\n";
}

int main() {
    // Daftar sisi, dalam urutan yang direlaksasi setiap putaran.
    vector<Edge> G = {{2,3,-2}, {1,2,4}, {0,1,3}, {0,2,6},
                      {3,4,3}, {1,3,5}, {4,2,1}};
    cout << "Graph 1: d->b has weight 1 (cycle b->c->d->b weighs -2+3+1 = 2)\n";
    run(G);
    G[6].w = -2;
    cout << "\nGraph 2: d->b has weight -2 (cycle b->c->d->b weighs -2+3-2 = -1)\n";
    run(G);
    return 0;
}
```

Ia mencetak:

```
Graph 1: d->b has weight 1 (cycle b->c->d->b weighs -2+3+1 = 2)
pass      d/pi s  d/pi a  d/pi b  d/pi c  d/pi d
init        0/-   inf/-   inf/-   inf/-   inf/-
1 (3)       0/-     3/s     6/s     8/a   inf/-
2 (2)       0/-     3/s     6/s     4/b     7/c
3 (0)       0/-     3/s     6/s     4/b     7/c
4 (0)       0/-     3/s     6/s     4/b     7/c
check: no edge relaxes  =>  return TRUE (no negative cycle reachable)

Graph 2: d->b has weight -2 (cycle b->c->d->b weighs -2+3-2 = -1)
pass      d/pi s  d/pi a  d/pi b  d/pi c  d/pi d
init        0/-   inf/-   inf/-   inf/-   inf/-
1 (3)       0/-     3/s     6/s     8/a   inf/-
2 (3)       0/-     3/s     5/d     4/b     7/c
3 (3)       0/-     3/s     4/d     3/b     6/c
4 (3)       0/-     3/s     3/d     2/b     5/c
check: edge b->c still relaxes (3 + (-2) = 1 < 2)  =>  return FALSE
negative cycle: b c d b
its weight: -1
```

Bilangan di dalam kurung sesudah setiap nomor putaran adalah banyaknya relaksasi yang berhasil, dan ia cocok dengan kolom terakhir **Tabel 16.9** dan **Tabel 16.10**. Cacahan nol pada bagian (a) juga mengisyaratkan pengoptimalan murah yang disebut Bab 13: **putaran tanpa perubahan berarti tidak akan ada lagi yang berubah**, sebab putaran berikutnya akan memeriksa bilangan-bilangan yang persis sama, jadi algoritmanya boleh berhenti lebih awal. Pada bagian (b) tak ada putaran yang pernah tenang — siklus negatif membuat setiap taksiran di

atasnya terus turun selamanya — dan itu cara lain untuk melihat mengapa pemeriksaannya harus gagal.

Uji diri. Pada bagian (a), baliklah urutan daftar sisinya lalu jalankan Bellman–Ford lagi. Berapa putaran yang mengubah sesuatu, dan apakah nilai-nilai akhirnya sama?

Jawaban. Dalam urutan $(d, b, 1), (a, c, 5), (c, d, 3), (s, b, 6), (s, a, 3), (a, b, 4), (b, c, -2)$, putaran 1 hanya menetapkan $d[b] = 6$, $d[a] = 3$, lalu $d[c] = 6 - 2 = 4$ (sisi-sisi keluar dari a sudah diperiksa sebelum $d[a]$ berhingga). Putaran 2 menetapkan $d[d] = 4 + 3 = 7$ lewat (c, d). Putaran 3 tidak mengubah apa pun. Jadi dua putaran mengubah sesuatu, dan nilai-nilai akhirnya sama, (0,3,6,4,7) — sebagaimana mestinya, sebab jawabannya tidak bergantung pada urutan; hanya kecepatan mencapainya yang bergantung.

16.8.3 Persoalan 16.8 — lintasan terpendek pada graf berarah taksiklik

Persoalan 16.8. Sebuah graf berarah taksiklik pada simpul-simpul s, a, b, c, d, e mempunyai sisi-sisi

$$s \to a\ (6),\ \ s \to b\ (3),\ \ b \to a\ (-2),\ \ b \to c\ (4),\ \ a \to c\ (1),\ \ a \to d\ (5),\ \ c \to d\ (-3),\ \ c \to e\ (2),\ \ d \to e\ (1),\ \ b \to d\ (7),$$

dengan senarai ketetanggaan yang disimpan persis dalam urutan ini. (a) Temukan urutan topologis dengan pencarian mendalam dari s, dengan memberikan waktu penemuan dan waktu selesainya. (b) Hitunglah bobot lintasan terpendek dan pendahulunya dari s dengan merelaksasi dalam urutan itu, memperlihatkan d/π sesudah setiap simpul diproses. (c) Jelaskan mengapa antrean prioritas tidak dibutuhkan, dan mengapa algoritma Dijkstra bukan alat yang tepat walaupun ia kebetulan berhasil di sini.

Apa yang diuji. Pengurutan topologis dengan pencarian mendalam (Bab 10) digabungkan dengan relaksasi (Bab 13): algoritma Subbab 13.9, yang kebenarannya bertumpu pada fakta bahwa setiap sisi mengarah *maju* dalam urutan topologis. **Pseudokode 16.2** adalah prosedurnya.

Pseudokode 16.2 Lintasan terpendek sumber tunggal pada graf berarah taksiklik: satu pengurutan topologis, lalu satu relaksasi per sisi.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  for each vertex v in G.V
3      v.d = infinity;  v.pi = NIL
4  s.d = 0
5  for each vertex u, taken in topologically sorted order
6      for each vertex v in G.Adj[u]
7          RELAX(u, v, w)
```

Penyelesaian (a). Jalankan DFS dari s, dengan tetangga diambil menurut urutan senarai, dan sebuah jam yang berdetak pada setiap penemuan dan setiap penyelesaian.

- s ditemukan pada waktu 1. Tetangga pertamanya a.
- a ditemukan pada 2. Tetangga pertamanya c.
- c ditemukan pada 3. Tetangga pertamanya d.
- d ditemukan pada 4. Satu-satunya tetangganya e.
- e ditemukan pada 5; tanpa tetangga; e selesai pada 6.
- Kembali di d: tak ada tetangga lagi; d selesai pada 7.

- Kembali di c: tetangga berikutnya e sudah selesai; c selesai pada 8.
- Kembali di a: tetangga berikutnya d sudah selesai; a selesai pada 9.
- Kembali di s: tetangga berikutnya b belum ditemukan; b ditemukan pada 10. Tetangga-tetangganya a, c, d semuanya sudah selesai; b selesai pada 11.
- s selesai pada 12.

Mendaftar simpul-simpulnya menurut waktu selesai yang *menurun* memberi urutan topologis

$$s\ (12),\quad b\ (11),\quad a\ (9),\quad c\ (8),\quad d\ (7),\quad e\ (6).$$

Periksalah bahwa setiap sisi mengarah maju: $s \to a$, $s \to b$ (posisi 1 ke 3, 1 ke 2); $b \to a$, $b \to c$, $b \to d$ (2 ke 3, 4, 5); $a \to c$, $a \to d$ (3 ke 4, 5); $c \to d$, $c \to e$ (4 ke 5, 6); $d \to e$ (5 ke 6). Kesepuluhnya maju. Perhatikan bahwa urutan abjad $s, a, b, \ldots$ *tidak* dapat dipakai: sisi $b \to a$ akan mengarah mundur.

Penyelesaian (b). Inisialisasikan $d[s] = 0$, yang lain ∞. Proseslah simpul-simpulnya dalam urutan topologis dan relaksasikan setiap sisi keluar sekali.

- s $(d = 0)$: $s \to a$: $0 + 6 = 6$, jadi $d[a] = 6, \pi[a] = s$. $s \to b$: $0 + 3 = 3$, jadi $d[b] = 3, \pi[b] = s$.
- b $(d = 3)$: $b \to a$: $3 + (-2) = 1 < 6$, jadi $d[a] = 1$, $\pi[a] = b$. $b \to c$: $3 + 4 = 7$, jadi $d[c] = 7$, $\pi[c] = b$. $b \to d$: $3 + 7 = 10$, jadi $d[d] = 10, \pi[d] = b$.
- a $(d = 1)$: $a \to c$: $1 + 1 = 2 < 7$, jadi $d[c] = 2$, $\pi[c] = a$. $a \to d$: $1 + 5 = 6 < 10$, jadi $d[d] = 6$, $\pi[d] = a$.
- c $(d = 2)$: $c \to d$: $2 + (-3) = -1 < 6$, jadi $d[d] = -1$, $\pi[d] = c$. $c \to e$: $2 + 2 = 4$, jadi $d[e] = 4$, $\pi[e] = c$.
- d $(d = -1)$: $d \to e$: $-1 + 1 = 0 < 4$, jadi $d[e] = 0, \pi[e] = d$.
- e $(d = 0)$: tanpa sisi keluar.

Tabel 16.11 Algoritma DAG pada Persoalan 16.8, satu baris per simpul yang diproses dalam urutan topologis. Nilai setiap simpul sudah tuntas pada saat ia diproses, sebab semua sisi masuknya berasal dari simpul yang diproses lebih awal.

Diproses	s	a	b	c	d	e
s	$0/-$	$6/s$	$3/s$	$\infty/-$	$\infty/-$	$\infty/-$
b	$0/-$	$1/b$	$3/s$	$7/b$	$10/b$	$\infty/-$
a	$0/-$	$1/b$	$3/s$	$2/a$	$6/a$	$\infty/-$
c	$0/-$	$1/b$	$3/s$	$2/a$	$-1/c$	$4/c$
d	$0/-$	$1/b$	$3/s$	$2/a$	$-1/c$	$0/d$
e	$0/-$	$1/b$	$3/s$	$2/a$	$-1/c$	$0/d$

Jawaban (b): $\delta(s,\cdot) = (s{:}\,0,\ a{:}\,1,\ b{:}\,3,\ c{:}\,2,\ d{:}\,-1,\ e{:}\,0)$, dan lintasan terpendek ke e adalah $s \to b \to a \to c \to d \to e$ berbobot $3 - 2 + 1 - 3 + 1 = 0$.

Penyelesaian (c). Ketika u diproses, setiap sisi yang *masuk* ke u berasal dari simpul yang lebih awal dalam urutan topologis, yang sudah diproses dan sisi-sisi keluarnya — termasuk yang menuju u — sudah direlaksasi. Jadi menurut sifat relaksasi lintasan, $d[u]$ sudah tuntas pada saat itu, apa pun tanda bobot-bobotnya. Urutan simpul-simpul menjadi tuntas sudah ditetapkan di muka oleh pengurutan topologisnya, jadi tidak ada yang perlu *dipilih*, dan antrean prioritas, yang satu-satunya tugasnya memilih, tidak dibutuhkan. Biaya total: $\Theta(V + E)$ untuk pengurutan ditambah $\Theta(V + E)$ untuk relaksasi, di sini $6 + 10$ langkah masing-masing.

Dijkstra, pada graf ini, kebetulan memberi jawaban yang benar: ia mengeluarkan s (0), b (3), a (1), c (2), d (−1), e (0) dan tak satu pun simpul yang sudah tuntas diperbaiki sesudahnya. Tetapi itu keberuntungan, bukan teorema: bukti kebenarannya (Subbab 13.7.8) membutuhkan bobot tak negatif, dan nilai-nilai pengeluaran 0,3,1,2, −1,0 bahkan tidak menaik, padahal justru invarian itulah yang diandalkan bukti tersebut. Program yang benar pada satu masukan tanpa bukti di belakangnya adalah tebakan yang diperingatkan Bab 8.

Program. **Kode 16.10** mengerjakan kedua langkah.

Kode 16.10 Lintasan terpendek pada DAG Persoalan 16.8: pengurutan topologis secara mendalam, lalu satu relaksasi per sisi dalam urutan topologis, dengan d/π dicetak sesudah setiap simpul.

```
// Persoalan 16.8: lintasan terpendek sumber tunggal pada DAG bersisi negatif.
// Langkah 1 adalah pengurutan topologis dengan pencarian mendalam (Bab 10);
// langkah 2 merelaksasi sisi keluar tiap simpul sekali, dalam urutan itu (Bab 13).
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

const int INF = INT_MAX / 4;
const vector<string> nm = {"s", "a", "b", "c", "d", "e"};
vector<vector<pair<int,int>>> adj(6);
vector<int> color(6, 0), order;                  // 0 putih, 1 abu-abu, 2 hitam

void dfs(int u) {
    color[u] = 1;
    for (auto [v, w] : adj[u]) if (color[v] == 0) dfs(v);
    color[u] = 2;
    order.push_back(u);                          // ditambahkan ketika u selesai
}

int main() {
    auto add = [](int u, int v, int w) { adj[u].push_back({v, w}); };
    add(0,1,6); add(0,2,3); add(2,1,-2); add(2,3,4); add(1,3,1);
    add(1,4,5); add(3,4,-3); add(3,5,2); add(4,5,1); add(2,4,7);
    int n = (int)nm.size();

    for (int u = 0; u < n; ++u) if (color[u] == 0) dfs(u);
    reverse(order.begin(), order.end());         // waktu selesai menurun
    cout << "topological order:";
    for (int v : order) cout << " " << nm[v];
    cout << "\n\n";

    vector<int> d(n, INF), pi(n, -1);
    d[0] = 0;
    cout << "process";
    for (int v = 0; v < n; ++v) cout << setw(8) << ("d/pi " + nm[v]);
    cout << "\n";
    for (int u : order) {
        if (d[u] < INF)
            for (auto [v, w] : adj[u])
                if (d[u] + w < d[v]) { d[v] = d[u] + w; pi[v] = u; }
        cout << setw(7) << nm[u];
        for (int v = 0; v < n; ++v) {
```

Kode 16.10 (Lanjutan) *Lintasan terpendek pada DAG Persoalan 16.8: pengurutan topologis secara mendalam, lalu satu relaksasi per sisi dalam urutan topologis, dengan d/\pi dicetak sesudah setiap simpul.*

```
            string cell = (d[v] >= INF) ? "inf" : to_string(d[v]);
            cell += "/";
            cell += (pi[v] < 0) ? "-" : nm[pi[v]];
            cout << setw(8) << cell;
        }
        cout << "\n";
    }
    cout << "\nshortest path to e:";
    vector<int> path;
    for (int x = 5; x != -1; x = pi[x]) path.push_back(x);
    for (int i = (int)path.size() - 1; i >= 0; --i) cout << " " << nm[path[i]];
    cout << "   weight " << d[5] << "\n";
    return 0;
}
```

Ia mencetak:

```
topological order: s b a c d e

process  d/pi s  d/pi a  d/pi b  d/pi c  d/pi d  d/pi e
      s     0/-     6/s     3/s   inf/-   inf/-   inf/-
      b     0/-     1/b     3/s     7/b    10/b   inf/-
      a     0/-     1/b     3/s     2/a     6/a   inf/-
      c     0/-     1/b     3/s     2/a    -1/c     4/c
      d     0/-     1/b     3/s     2/a    -1/c     0/d
      e     0/-     1/b     3/s     2/a    -1/c     0/d

shortest path to e: s b a c d e   weight 0
```

Urutan topologis dan setiap barisnya sepakat dengan perhitungan tangan pada **Tabel 16.11**.

Uji diri. Ubahlah bobot $b \to a$ dari -2 menjadi $+2$. Hitunglah ulang d/π untuk setiap simpul.

Jawaban. a: $5/b$, b: $3/s$, c: $6/a$, d: $3/c$, e: $4/d$. (Di a: $\min(6,3+2) = 5$. Di c: $\min(3+4,5+1) = 6$. Di d: $\min(3+7,5+5,6-3) = 3$ — seri antara 10 dan 10 diputuskan uji tegasnya untuk b, lalu c mengalahkan keduanya. Di e: $\min(6+2,3+1) = 4$.)

16.9 Persoalan Terkerjakan tentang Analisis Amortisasi

Bab 14 memperkenalkan tiga metode untuk satu pertanyaan: *berapa biaya total sebuah barisan n operasi, ketika beberapa di antaranya mahal?* Dua persoalan dalam subbab ini menerapkan ketiga metode itu pada struktur data yang sama, berdampingan, agar Anda melihat bahwa ketiganya adalah tiga cara membukukan satu fakta, bukan tiga teori yang berbeda. Kedua persoalan memakai identitas metode potensial dari Subbab 14.6, yang layak dituliskan sekali lagi sebelum kita mulai. Jika operasi ke-i berbiaya sebenarnya c_i dan membawa strukturnya dari keadaan D_{i-1} ke keadaan D_i, **biaya amortisasi**-nya terhadap fungsi potensial Φ adalah

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}), \tag{16.6}$$

dan dengan menjumlahkan atas $i = 1, \ldots, n$, potensial-potensialnya berteleskop:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0), \qquad \text{jadi} \qquad \sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i \ \text{ bilamana } \Phi(D_n) \ge \Phi(D_0). \tag{16.7}$$

Syarat $\Phi(D_n) \ge \Phi(D_0)$ harus berlaku untuk *setiap* n, sebab kita menginginkan batasnya untuk setiap awalan barisan; itulah sebabnya orang biasanya mengambil $\Phi(D_0) = 0$ dan membuktikan $\Phi \ge 0$ sepanjang waktu (Subbab 14.6.3).

16.9.1 Persoalan 16.9 — pencacah biner, dengan tiga cara dan dengan induksi

Persoalan 16.9. Sebuah pencacah biner k-bit $A[0..k-1]$ mulai dari nol, dan prosedur `INCREMENT` pada **Pseudokode 16.3** dipanggil n kali. Membalik satu bit berbiaya 1. Tunjukkan bahwa biaya total n panggilan itu kurang dari $2n$, sehingga biaya amortisasi per panggilan $O(1)$, dengan empat cara: (a) dengan metode agregat, (b) dengan metode akuntansi, (c) dengan metode potensial, dan (d) dengan induksi atas n. Tabulasikan delapan penambahan pertama untuk semuanya.

Pseudokode 16.3 `INCREMENT` pada pencacah biner k-bit: nolkan bit-bit 1 di ujung, lalu jadikan bit 0 yang pertama bernilai 1.

```
INCREMENT(A, k)
i = 0
while i < k and A[i] == 1
    A[i] = 0
    i = i + 1
if i < k
    A[i] = 1
```

Apa yang diuji. Ketiga metode Bab 14 (Subbab 14.3.3, 14.5.4, dan 14.6.5) pada struktur tempat ketiganya pertama kali diajarkan, ditambah pengamatan bahwa metode potensial pada dasarnya adalah hipotesis induksi yang diperkuat — langkah yang sama dengan bukti substitusi di Bab 4.

Mengapa batas naifnya tidak berguna. Satu `INCREMENT` dapat membalik semua k bit (dari $0111\ldots1$ menjadi $1000\ldots0$), jadi n panggilan berbiaya paling banyak nk. Benar, dan tidak berguna: ia membebani setiap panggilan seolah-olah panggilan terburuk. **Gambar 16.9** memperlihatkan enam belas penambahan pertama sebuah pencacah 8-bit dengan bit-bit yang terbalik diarsir; hampir setiap panggilan hanya membalik satu atau dua bit.

Nilai pencacah	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Biaya total
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Gambar 16.9 Pencacah biner 8-bit dari 0 sampai 16. Bit yang diarsir adalah yang terbalik dalam perjalanan ke nilai berikutnya, dan kolom terakhir adalah jumlah berjalan pembalikan — 31 sesudah enam belas penambahan, tepat di bawah $2 \times 16 = 32$.

Penyelesaian (a): metode agregat. Hitunglah per *bit*, bukan per *panggilan*. Bit $A[0]$ terbalik pada setiap panggilan: n kali. Bit $A[1]$ terbalik pada setiap panggilan kedua: $\lfloor n/2 \rfloor$ kali. Secara umum bit $A[i]$ terbalik tepat ketika i bit terendahnya semuanya 1 sebelum panggilan, yang terjadi sekali dalam setiap 2^i panggilan, jadi ia terbalik $\lfloor n/2^i \rfloor$ kali. Karena itu biaya total $C(n)$ adalah

$$C(n) = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = n \cdot \frac{1}{1 - 1/2} = 2n. \tag{16.8}$$

Biaya amortisasi per panggilan adalah $C(n)/n < 2$. Periksalah Persamaan (16.8) pada $n = 16$: $16 + 8 + 4 + 2 + 1 = 31 < 32$, tepat entri terakhir **Gambar 16.9**.

Penyelesaian (b): metode akuntansi. Tarifkan biaya **2** pada setiap `INCREMENT`. Ketika sebuah bit ditetapkan dari 0 menjadi 1 (baris 6, yang terjadi paling banyak sekali per panggilan), belanjakan 1 dari tarif itu untuk pembalikan tersebut dan tinggalkan 1 lainnya *di atas bit itu* sebagai kredit. Ketika sebuah bit dikembalikan dari 1 menjadi 0 (baris 3), bayarlah dengan kredit yang duduk di atasnya. Invarian yang harus dibuktikan adalah bahwa tabungannya tidak pernah negatif, dan ia berlaku karena alasan sederhana: **setiap bit 1 membawa tepat satu satuan kredit**, jadi tabungannya sama dengan banyaknya bit 1, yang tidak pernah negatif. Pengembalian di baris 2–4 karena itu selalu sudah dibayar di muka, setiap panggilan membayar paling banyak 2 dari sakunya sendiri, dan n panggilan berbiaya paling banyak $2n$.

Penyelesaian (c): metode potensial. Misalkan $\Phi(D_i) = b_i$, banyaknya bit 1 sesudah panggilan ke-i. Maka $\Phi(D_0) = 0$ dan $\Phi \geq 0$ selalu. Andaikan panggilan ke-i mengembalikan t_i bit di baris 2–4. Maka biaya sebenarnya paling banyak $t_i + 1$ (t_i pengembalian ditambah paling banyak satu penetapan), dan banyaknya bit 1 berubah paling banyak sebesar $-t_i + 1$:

$$b_i \leq b_{i-1} - t_i + 1.$$

(Kesamaan berlaku kecuali pencacahnya meluap, ketika semua k bit dikembalikan dan tak satu pun ditetapkan.) Menurut Persamaan (16.6),

$$\hat{c}_i \;=\; c_i + b_i - b_{i-1} \;\le\; (t_i + 1) + (1 - t_i) \;=\; 2.$$

Setiap biaya amortisasi paling banyak 2, potensialnya mulai dari 0 dan tidak pernah negatif, jadi menurut Persamaan (16.7) biaya sebenarnya total paling banyak $2n$.

Ambillah satu panggilan mahal dan periksalah hitungannya. Panggilan 8 membawa 0111 ke 1000: $t_8 = 3$ pengembalian dan satu penetapan, jadi $c_8 = 4$; potensialnya turun dari $b_7 = 3$ ke $b_8 = 1$; biaya amortisasinya $\hat{c}_8 = 4 + 1 - 3 = 2$. Panggilan 16 membawa 01111 ke 10000: $c_{16} = 5$, Φ turun dari 4 ke 1, dan $\hat{c}_{16} = 5 + 1 - 4 = 2$. Panggilan yang mahal dibayar oleh potensial yang dibangun panggilan-panggilan murah.

Penyelesaian (d): dengan induksi. Klaim $C(n) < 2n$ tidak dapat membawa dirinya sendiri melalui induksi: mengetahui $C(n-1) < 2(n-1)$ tidak mengatakan apa-apa yang berguna tentang panggilan ke-n, yang dapat berbiaya sebesar k. Perbaikannya — siasat penguatan yang sama dengan bukti substitusi di Bab 4 — adalah membawa serta besaran kedua. Misalkan $\nu(n)$ adalah banyaknya bit 1 sesudah n panggilan, lalu buktikan

$$P(n): \qquad C(n) + \nu(n) \le 2n.$$

Dasar, $n = 0$. $C(0) = 0$, $\nu(0) = 0$, dan $0 \le 0$.

Langkah. Anggaplah $C(n-1) + \nu(n-1) \le 2(n-1)$. Panggilan ke-$n$ mengubah t bit 1 di ujung menjadi 0 dan satu bit 0 menjadi 1 (abaikan luapan, yang hanya membantu), jadi ia berbiaya $t + 1$ dan mengubah banyaknya bit 1 sebesar $-t + 1$. Karena itu

$$\begin{aligned} C(n) + \nu(n) \;&=\; (C(n-1) + t + 1) + (\nu(n-1) - t + 1) \\ &=\; C(n-1) + \nu(n-1) + 2 \;\le\; 2(n-1) + 2 \;=\; 2n. \end{aligned}$$

Karena $\nu(n) \ge 1$ untuk $n \ge 1$ (pencacahnya tidak nol sesudah penambahan yang tidak meluap), $C(n) \le 2n - 1 < 2n$. ■

Perhatikan apa besaran yang diperkuat itu: $C(n) + \nu(n)$ adalah total sebenarnya ditambah potensial dari (c). **Metode potensial adalah induksi dengan hipotesis yang diperkuat, ditulis sebagai pembukuan.**

Delapan panggilan pertama, semua metode sekaligus. **Tabel 16.12** menghitung setiap besaran dengan tangan. Kolom *tabungan* adalah saldo akuntansi sesudah panggilan (tarif yang dibayar dikurangi biaya sebenarnya), dan kolom $\hat{c}_i$ adalah Persamaan (16.6) dengan Φ = banyaknya bit 1.

Tabel 16.12 Pencacah biner dengan ketiga metode untuk delapan panggilan pertama. Jumlah berjalan $C(i)$ tetap di bawah $2i$ (agregat), tabungannya tidak pernah negatif dan selalu sama dengan banyaknya bit 1 (akuntansi), dan setiap biaya amortisasi tepat 2 (potensial).

i	Pencacah	t_i	c_i	$C(i)$	$2i$	Tabungan	Φ_i	$c_i=c_i+\Phi_i-\Phi_{i-1}$
1	0001	0	1	1	2	1	1	$1+1-0=2$
2	0010	1	2	3	4	1	1	$2+1-1=2$
3	0011	0	1	4	6	2	2	$1+2-1=2$
4	0100	2	3	7	8	1	1	$3+1-2=2$
5	0101	0	1	8	10	2	2	$1+2-1=2$
6	0110	1	2	10	12	2	2	$2+2-2=2$
7	0111	0	1	11	14	3	3	$1+3-2=2$
8	1000	3	4	15	16	1	1	$4+1-3=2$

Kesamaan kolom *tabungan* dan Φ_i bukanlah kebetulan: kredit metode akuntansi dan potensial metode potensial adalah bilangan yang sama, yang satu disimpan di atas bit-bitnya dan yang lain dihitung dari bit-bit itu.

Program. **Kode 16.11** menjalankan enam belas panggilan dengan ketiga buku besarnya, lalu memeriksa Persamaan (16.8) untuk n yang lebih besar.

Kode 16.11 Pencacah biner Persoalan 16.9 dengan total agregat, tabungan akuntansi, dan biaya amortisasi metode potensial dicetak untuk setiap panggilan, disusul jumlah agregat untuk n yang lebih besar.

```
// Persoalan 16.9: pencacah biner k-bit dianalisis dengan tiga cara sekaligus.
//   agregat   : jumlah berjalan biaya sebenarnya, dibandingkan dengan 2n
//   akuntansi : tarif 2 per INCREMENT; tabungan tidak boleh negatif
//   potensial : Phi = banyaknya bit 1; biaya amortisasi = c + Phi_i - Phi_{i-1}
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

const int K = 8;

int main() {
    int A[K] = {0};
    long long total = 0, bank = 0;
    int phiPrev = 0;
    cout << "  i  counter   t  cost  total   2i  bank  Phi  amortized\n";
    for (int i = 1; i <= 16; ++i) {
        int j = 0, t = 0;
        while (j < K && A[j] == 1) { A[j] = 0; ++j; ++t; }   // nolkan bit 1 di ujung
        if (j < K) A[j] = 1;                                 // jadikan satu bit 0 bernilai
1
        int cost = t + (j < K ? 1 : 0);
        total += cost;
        bank += 2 - cost;                  // ditarif 2, dibelanjakan biaya sebenarnya
        int phi = 0;
        for (int b = 0; b < K; ++b) phi += A[b];
        int amort = cost + phi - phiPrev;
        phiPrev = phi;
        string bits;
        for (int b = K - 1; b >= 0; --b) bits += char('0' + A[b]);
        cout << setw(3) << i << "  " << bits << setw(3) << t << setw(6) << cost
             << setw(7) << total << setw(5) << 2 * i << setw(6) << bank
             << setw(5) << phi << setw(8) << amort << "\n";
    }
    // rumus agregat, diperiksa untuk n yang lebih besar
    for (long long n : {16LL, 100LL, 1000LL, 1000000LL}) {
        long long c = 0;
        for (long long p = 1; p <= n; p *= 2) c += n / p;   // jumlah floor(n / 2^i)
        cout << "n = " << setw(7) << n << ":  total flips = " << setw(7) << c
             << "  <  2n = " << 2 * n << "\n";
    }
    return 0;
}
```

Ia mencetak:

```
  i  counter   t  cost  total   2i  bank  Phi  amortized
  1  00000001  0     1      1    2     1    1        2
  2  00000010  1     2      3    4     1    1        2
  3  00000011  0     1      4    6     2    2        2
  4  00000100  2     3      7    8     1    1        2
  5  00000101  0     1      8   10     2    2        2
  6  00000110  1     2     10   12     2    2        2
  7  00000111  0     1     11   14     3    3        2
  8  00001000  3     4     15   16     1    1        2
  9  00001001  0     1     16   18     2    2        2
 10  00001010  1     2     18   20     2    2        2
 11  00001011  0     1     19   22     3    3        2
 12  00001100  2     3     22   24     2    2        2
 13  00001101  0     1     23   26     3    3        2
 14  00001110  1     2     25   28     3    3        2
 15  00001111  0     1     26   30     4    4        2
 16  00010000  4     5     31   32     1    1        2
n =      16:  total flips =      31  <  2n = 32
n =     100:  total flips =     197  <  2n = 200
n =    1000:  total flips =    1994  <  2n = 2000
n = 1000000:  total flips = 1999993  <  2n = 2000000
```

Delapan baris pertamanya adalah **Tabel 16.12**; total 31 pada $i = 16$ adalah entri terakhir **Gambar 16.9**; dan pada $n = 1000$ totalnya $1994 = \sum_i \lfloor 1000/2^i \rfloor$, amortisasi 1.994 per panggilan.

Uji diri. Andaikan membalik bit i berbiaya 2^i alih-alih 1 (bit tinggi mahal untuk ditulis). Dengan metode agregat, temukan biaya total $n = 2^m$ penambahan dari nol, dan biaya amortisasi per panggilan.

Jawaban. Bit i terbalik $\lfloor n/2^i \rfloor = 2^{m-i}$ kali dengan biaya 2^i masing-masing, menyumbang tepat $2^m = n$ untuk setiap $i = 0,1,\ldots,m$. Totalnya $(m+1)\,n = n(\lg n + 1)$, jadi biaya amortisasinya $\Theta(\lg n)$ per panggilan — tidak lagi konstan. Amortisasi menyebarkan biaya; ia tidak melenyapkannya.

16.9.2 Persoalan 16.10 — tabel dinamis, dengan tiga cara, dan sebuah potensial keliru yang menggoda

Persoalan 16.10. Sebuah tabel dinamis mulai kosong dengan $size = 0$. `TABLE-INSERT` (**Pseudokode 16.4**) menulis satu butir; jika tabelnya penuh, ia lebih dahulu mengalokasikan tabel baru berukuran dua kali lipat (ukuran 1 ketika ukuran lamanya 0), menyalin setiap butir ke sana, dan membebaskan tabel lama. Menulis atau menyalin satu butir berbiaya 1. (a) Berikan biaya sebenarnya c_i penyisipan ke-i. (b) Tunjukkan bahwa n penyisipan berbiaya kurang dari $3n$ dengan metode agregat, (c) dengan metode akuntansi bertarif 3, dan (d) dengan metode potensial dengan

$$\Phi(T) \;=\; 2 \cdot num[T] - size[T]. \tag{16.9}$$

(e) Tabulasikan sembilan penyisipan pertama. (f) Jelaskan apa yang salah dengan potensial $\Phi(T) = num[T] - size[T]$.

Pseudokode 16.4 `TABLE-INSERT` untuk tabel yang berlipat dua ketika penuh. Baris 3–6 adalah perluasan yang mahal; baris 7 adalah penyisipan dasarnya.

```
TABLE-INSERT(T, x)
if T.size == 0
    allocate T.table with 1 slot;  T.size = 1
if T.num == T.size
    allocate new-table with 2 * T.size slots
    insert all items in T.table into new-table
    free T.table;  T.table = new-table;  T.size = 2 * T.size
insert x into T.table
T.num = T.num + 1
```

Apa yang diuji. Tabel dinamis Subbab 14.7, dianalisis dengan ketiga metode dari Subbab 14.7.3 sampai 14.7.5, dan dua syarat yang harus dipenuhi sebuah fungsi potensial.

Penyelesaian (a). Penyisipan ke-i mendapati $num = i - 1$ butir di dalam tabel. Perluasan terjadi tepat ketika tabelnya penuh, yaitu ketika $i - 1$ sama dengan ukuran saat itu, dan ukurannya berjalan 1,2,4,8, ... Jadi

$$c_i = \begin{cases} i & \text{jika } i-1 \text{ tepat berupa pangkat 2} \quad (i-1 \text{ salinan, ditambah 1} \\ 1 & \text{selainnya.} \end{cases}$$

(Penyisipan yang paling pertama mengalokasikan tabel berukuran 1 dan menulis satu butir: $c_1 = 1$.) Penyisipan yang mahal adalah $i = 2,3,5,9,17, \ldots$, berbiaya $2,3,5,9,17, \ldots$.

Penyelesaian (b): agregat. Setiap penyisipan membayar 1 untuk operasi tulisnya sendiri, yang berjumlah n. Di atas itu, salinan-salinan pada perluasan berjumlah $1 + 2 + 4 + \cdots + 2^j$ dengan $2^j \leq n - 1$, dan jumlah geometri kurang dari dua kali suku terbesarnya:

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j < n + 2n = 3n.$$

Untuk $n = 17$: operasi tulisnya berbiaya 17, salinan pada $i = 2,3,5,9,17$ berbiaya $1 + 2 + 4 + 8 + 16 = 31$, dan totalnya 48 memang di bawah $3 \times 17 = 51$.

Penyelesaian (c): akuntansi. Tarifkan **3** pada setiap penyisipan. Belanjakan 1 untuk menulis butirnya. Simpanlah 2 sebagai kredit, dan bayangkan sebagai berikut: 1 satuan membayar pemindahan *butir ini* pada perluasan berikutnya, dan 1 satuan membayar pemindahan *satu butir lama* yang sudah dipindahkan pada perluasan sebelumnya dan sudah membelanjakan kreditnya sendiri. Mengapa ini seimbang? Tepat sesudah perluasan ke ukuran $2m$, tabelnya berisi $m + 1$ butir, dengan m butir lama yang kreditnya sudah habis. Sebelum perluasan berikutnya, $m - 1$ butir lagi datang, masing-masing meninggalkan 2 satuan, ditambah butir yang datang bersama perluasan itu: tabungannya berisi $2m$ satuan ketika tabel berisi $2m$ butir itu penuh — tepat cukup untuk memindahkan kesemua $2m$ butir. Jadi tabungannya tidak pernah negatif, dan n penyisipan berbiaya paling banyak $3n$.

Penyelesaian (d): potensial. Periksalah lebih dahulu kedua syarat untuk Persamaan (16.9). Tabel kosong mempunyai $num = size = 0$, jadi $\Phi(T_0) = 0$. Sesudah penyisipan pertama tabelnya selalu paling sedikit setengah penuh — tepat setengah penuh ditambah satu segera sesudah perluasan, sebab tabelnya berlipat dua dari keadaan penuh dan satu butir ditambahkan, dan penyisipan hanya menaikkan num — jadi $2\,num \geq size$ dan $\Phi \geq 0$. Kini hitunglah $\hat{c}_i$ menurut Persamaan (16.6), dengan menuliskan num_i, $size_i$ untuk nilai-nilai sesudah penyisipan i.

Kasus 1: tanpa perluasan. Maka $size_i = size_{i-1}$, $num_i = num_{i-1} + 1$, dan $c_i = 1$:

$$\hat{c}_i \;=\; 1 + (2\,num_i - size_i) - (2(num_i - 1) - size_i) \;=\; 1 + 2 \;=\; 3.$$

Kasus 2: perluasan. Maka $size_{i-1} = num_{i-1} = num_i - 1$, $size_i = 2\,size_{i-1} = 2(num_i - 1)$, dan $c_i = num_i$ ($num_i - 1$ salinan ditambah 1 operasi tulis):

$$\begin{aligned}\hat{c}_i \;&=\; num_i + (2\,num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\ &=\; num_i + 2 - (num_i - 1) \;=\; 3.\end{aligned}$$

(Untuk $i = 1$, ketika ukurannya berubah dari 0 menjadi 1, $\hat{c}_1 = 1 + (2 - 1) - 0 = 2$.) Setiap biaya amortisasi paling banyak 3, Φ tidak pernah turun di bawah $\Phi(T_0) = 0$, jadi menurut Persamaan (16.7) biaya sebenarnya total n penyisipan paling banyak $3n$. Potensial adalah rekening tabungannya: ia naik 2 pada setiap penyisipan murah dan dibelanjakan sekaligus untuk penyalinan.

Penyelesaian (e). **Tabel 16.13** mengerjakan setiap perhitungan itu untuk sembilan penyisipan pertama.

Tabel 16.13 Tabel dinamis Persoalan 16.10 dengan ketiga metode untuk sembilan penyisipan pertama. Totalnya tetap di bawah $3i$, tabungannya tetap positif, dan biaya amortisasinya 3 sejak penyisipan kedua — termasuk pada penyisipan mahal $i = 2,3,5,9$.

i	$size_i$	num_i	c_i	$\sum c$	$3i$	Tabungan	$\Phi_i = 2num_i - size_i$	c_i
1	1	1	1	1	3	2	$2 - 1 = 1$	$1 + 1 - 0 = 2$
2	2	2	2	3	6	3	$4 - 2 = 2$	$2 + 2 - 1 = 3$
3	4	3	3	6	9	3	$6 - 4 = 2$	$3 + 2 - 2 = 3$
4	4	4	1	7	12	5	$8 - 4 = 4$	$1 + 4 - 2 = 3$
5	8	5	5	12	15	3	$10 - 8 = 2$	$5 + 2 - 4 = 3$
6	8	6	1	13	18	5	$12 - 8 = 4$	$1 + 4 - 2 = 3$
7	8	7	1	14	21	7	$14 - 8 = 6$	$1 + 6 - 4 = 3$
8	8	8	1	15	24	9	$16 - 8 = 8$	$1 + 8 - 6 = 3$
9	16	9	9	24	27	3	$18 - 16 = 2$	$9 + 2 - 8 = 3$

Tabungannya selalu $\Phi_i + 1$: satu satuan lebih itu adalah kembalian dari penyisipan pertama, yang biaya amortisasinya 2 tetapi ditarif 3.

Penyelesaian (f): potensial keliru yang menggoda. Mengapa $2\,num - size$ dan bukan sesuatu yang lebih sederhana? Cobalah $\Phi(T) = num - size$, yang juga 0 pada tabel kosong dan tampak sama wajarnya. Hitunglah: pada Kasus 1, $\Delta\Phi = +1$ dan $\hat{c}_i = 1 + 1 = 2$; pada Kasus 2, $\Delta\Phi = (num_i - 2(num_i - 1)) - 0 = 2 - num_i$ dan $\hat{c}_i = num_i + 2 - num_i = 2$. Setiap biaya amortisasi paling banyak 2! Seandainya ini analisis yang sah, n penyisipan akan berbiaya paling banyak $2n$ — padahal pada $n = 17$ kita menghitung total sebenarnya 48, jauh di atas $2 \times 17 = 34$. "Bukti" itu gagal karena Φ ini **negatif**: tabel yang belum penuh mempunyai $num < size$, dan pada $n = 17$, $\Phi = 17 - 32 = -15$. Menurut Persamaan (16.7), $\sum c_i = \sum \hat{c}_i - \Phi(D_n) + \Phi(D_0) = 33 + 15 = 48$ —

jumlah biaya amortisasinya, 33, mengecilkan kenyataan tepat sebesar utang yang ditumpuk potensialnya. Potensial yang boleh turun di bawah nilai awalnya meminjam dari masa depan, dan batas yang diberikannya tidak bernilai. **Selalu periksalah $\Phi(D_i) \geq \Phi(D_0)$ sebelum memercayai sebuah batas amortisasi.**

Pilihan menggoda kedua, $\Phi(T) = num$, memang memenuhi $\Phi \geq 0$ tetapi gagal dengan cara lain: pada Kasus 2, $\hat{c}_i = num_i + 1$, yang bukan konstanta, jadi ia tidak membuktikan apa pun yang lebih baik daripada batas naifnya. Potensial yang baik harus besar *tepat ketika operasi mahal sudah dekat* — tabel penuh, Φ besar — dan kecil sesaat sesudahnya. $2\,num - size$ adalah fungsi paling sederhana yang berbentuk seperti itu: ia naik dari 2 sampai *size* selama tabelnya terisi.

Program. Kode 16.12 mencetak ketiga buku besar untuk tujuh belas penyisipan dan total agregat untuk n yang lebih besar.

Kode 16.12 Tabel dinamis berlipat dua Persoalan 16.10, dengan biaya sebenarnya, jumlah berjalan terhadap $3i$, tabungan akuntansi, potensial $2\,num - size$, dan biaya amortisasi setiap penyisipan.

```
// Persoalan 16.10: TABLE-INSERT ke tabel yang berlipat dua ketika penuh, dianalisis
// dengan metode agregat, akuntansi, dan potensial berdampingan.
//   biaya sebenarnya c_i = 1, ditambah (i-1) salinan bila tabel sedang penuh
//   akuntansi      : tarif 3 per penyisipan; tabungan = tarif - biaya sebenarnya
//   potensial      : Phi = 2*num - size (dan Phi = 0 untuk tabel kosong)
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    long long size = 0, num = 0, total = 0, bank = 0, phiPrev = 0;
    cout << "  i  size  num  c_i  total   3i  bank  Phi  amortized\n";
    for (int i = 1; i <= 17; ++i) {
        long long cost = 1;                        // menulis butir baru
        if (num == size) {                         // penuh: alokasikan dan salin
            cost += num;                           // num butir dipindahkan
            size = (size == 0) ? 1 : 2 * size;
        }
        ++num;
        total += cost;
        bank += 3 - cost;
        long long phi = 2 * num - size;
        long long amort = cost + phi - phiPrev;
        phiPrev = phi;
        cout << setw(3) << i << setw(6) << size << setw(5) << num << setw(5) << cost
             << setw(7) << total << setw(5) << 3 * i << setw(6) << bank
             << setw(5) << phi << setw(8) << amort << "\n";
    }
    for (long long n : {17LL, 1000LL, 1000000LL}) {
        long long s = 0, sz = 0, c = 0;
        for (long long k = 1; k <= n; ++k) {
            c += 1;
            if (s == sz) { c += s; sz = sz ? 2 * sz : 1; }
            ++s;
        }
        cout << "n = " << setw(7) << n << ":  total cost = " << setw(7) << c
             << "  <  3n = " << 3 * n << "\n";
    }
    return 0;
}
```

Ia mencetak:

```
  i  size  num  c_i  total   3i  bank  Phi  amortized
  1     1    1    1      1    3     2    1          2
  2     2    2    2      3    6     3    2          3
  3     4    3    3      6    9     3    2          3
  4     4    4    1      7   12     5    4          3
  5     8    5    5     12   15     3    2          3
  6     8    6    1     13   18     5    4          3
  7     8    7    1     14   21     7    6          3
  8     8    8    1     15   24     9    8          3
  9    16    9    9     24   27     3    2          3
 10    16   10    1     25   30     5    4          3
 11    16   11    1     26   33     7    6          3
 12    16   12    1     27   36     9    8          3
 13    16   13    1     28   39    11   10          3
 14    16   14    1     29   42    13   12          3
 15    16   15    1     30   45    15   14          3
 16    16   16    1     31   48    17   16          3
 17    32   17   17     48   51     3    2          3
n =      17:  total cost =      48  <  3n = 51
n =    1000:  total cost =    2023  <  3n = 3000
n = 1000000:  total cost = 2048575  <  3n = 3000000
```

Sembilan baris pertamanya adalah **Tabel 16.13**, dan total agregatnya — 48 untuk $n = 17$, 2023 untuk $n = 1000$, 2048575 untuk $n = 10^6$ — semuanya berada di bawah $3n$.

Uji diri. Andaikan tabel yang penuh bertambah dengan **4** slot tetap (ukuran 4,8,12, …), mulai dari ukuran 0. Temukan biaya total 17 penyisipan, dan tunjukkan dengan metode agregat bahwa n penyisipan berbiaya $\Theta(n^2)$.

Jawaban. Perluasan terjadi ketika $num = 0,4,8,12,16$ dan menyalin 0,4,8,12,16 butir: 40 salinan ditambah 17 operasi tulis, total 57. Secara umum, dengan $m = \lfloor (n-1)/4 \rfloor$ perluasan sesudah yang pertama, salinannya berjumlah $4(1 + 2 + \cdots + m) = 2m(m+1) \approx n^2/8$, jadi totalnya $\Theta(n^2)$ dan biaya amortisasi per penyisipan $\Theta(n)$. (Untuk $n = 1000$: $m = 249$, total $1000 + 2 \cdot 249 \cdot 250 = 125{,}500$.) Bertumbuh dengan *faktor*-lah yang membuat tabel murah; bertumbuh dengan *konstanta* tidak.

16.10 Persoalan Terkerjakan tentang Lintasan Terpendek Semua Pasangan

Bab 15 meminta bobot lintasan terpendek antara *setiap* pasangan simpul terurut, dan menjawabnya dengan dua algoritma yang semangatnya hampir tidak mungkin lebih berbeda: Floyd–Warshall, program dinamis tiga baris atas matriks, dan algoritma Johnson, sebuah reduksi yang mengubah bobot-bobotnya sehingga Dijkstra boleh dipakai. Dua persoalan dalam subbab ini menjalankan keduanya, dengan tangan, pada graf empat simpul yang sama dengan satu sisi negatif, lalu memeriksa bahwa keduanya sepakat.

Graf yang dipakai pada kedua persoalan mempunyai simpul 1,2,3,4 dan tujuh sisi

$$1 \to 2\ (4), \quad 1 \to 4\ (5), \quad 2 \to 3\ (-2), \quad 3 \to 1\ (3), \quad 3 \to 4\ (2), \quad 4 \to 2\ (1), \quad 4 \to 3\ (6).$$

Ia mempunyai lima siklus: $1 \to 2 \to 3 \to 1$ berbobot $4 - 2 + 3 = 5$, $2 \to 3 \to 4 \to 2$ berbobot $-2 + 2 + 1 = 1$, $3 \to 4 \to 3$ berbobot $2 + 6 = 8$, $1 \to 4 \to 2 \to 3 \to 1$ berbobot $5 + 1 - 2 + 3 = 7$, dan $1 \to 4 \to 3 \to 1$ berbobot $5 + 6 + 3 = 14$. Semuanya positif, jadi lintasan terpendek terdefinisi dengan baik walaupun satu sisinya negatif.

16.10.1 Persoalan 16.11 — Floyd–Warshall, setiap matriks dihitung sel demi sel

Persoalan 16.11. Jalankan algoritma Floyd–Warshall pada graf di atas. Berikan matriks $D^{(0)}, D^{(1)}, D^{(2)}, D^{(3)}, D^{(4)}$ dan matriks pendahulu $\Pi^{(0)}, \dots, \Pi^{(4)}$, dengan memperlihatkan untuk setiap ronde k setiap perbandingan yang dapat mengubah sebuah entri. Lalu bacalah lintasan terpendek $1 \rightsquigarrow 4$, $4 \rightsquigarrow 1$, dan $3 \rightsquigarrow 2$ dari $\Pi^{(4)}$, dan jelaskan bagaimana algoritmanya akan menyingkapkan siklus negatif.

Apa yang diuji. Rekurensi Floyd–Warshall dari Subbab 15.6. Misalkan $d_{ij}^{(k)}$ adalah bobot lintasan terpendek dari i ke j yang semua simpul *antara*-nya terletak di $\{1, \dots, k\}$. Lintasan seperti itu entah sama sekali tidak memakai k, entah melewati k tepat sekali dan terbelah di sana menjadi dua potong yang simpul antaranya terletak di $\{1, \dots, k-1\}$. Karena itu

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{jika } k = 0, \\ \min(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{jika } k \geq 1, \end{cases} \tag{16.10}$$

dengan w_{ij} adalah bobot sisinya, 0 pada diagonal dan ∞ jika tidak ada sisi. Aturan pendahulunya menyertainya: ketika argumen kedua yang menang, $\pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$ — langkah terakhir menuju j adalah langkah terakhir rute dari k ke j.

Penyelesaian. *Ronde 0.* $D^{(0)}$ adalah matriks bobot, dan $\Pi^{(0)}$ mempunyai $\pi_{ij} = i$ di mana pun ada sisi $i \to j$, dan NIL di tempat lain. **Tabel 16.14** memperlihatkan keduanya.

Tabel 16.14 Matriks-matriks awal Persoalan 16.11: $D^{(0)}$ adalah matriks bobot, dan $\Pi^{(0)}$ mencatat ekor setiap sisi.

$D^{(0)}$	1	2	3	4		$\Pi^{(0)}$	1	2	3	4
1	0	4	∞	5		**1**	NIL	1	NIL	1
2	∞	0	-2	∞		**2**	NIL	NIL	2	NIL
3	3	∞	0	2		**3**	3	NIL	NIL	3
4	∞	1	6	0		**4**	NIL	4	4	NIL

Satu fakta menghemat sebagian besar pekerjaannya: pada ronde k, sebuah entri d_{ij} hanya dapat berubah jika **kedua** d_{ik} (kolom k) dan d_{kj} (baris k) berhingga, dan tidak pernah ketika $i = k$ atau $j = k$ (melewati k untuk mencapai k tidak menambahkan apa-apa). Jadi pada setiap ronde kita hanya perlu melihat entri-entri berhingga di kolom k dan baris k.

Ronde 1 (bolehkah lintasan melewati simpul 1?). Kolom 1, di luar diagonal: $d_{21} = \infty$, $d_{31} = 3$, $d_{41} = \infty$. Baris 1: $d_{12} = 4$, $d_{13} = \infty$, $d_{14} = 5$. Hanya $i = 3$ yang entrinya berhingga, dan hanya $j = 2,4$:

- d_{32}: $\min(\infty,\ 3 + 4) = 7$. **Berubah**; $\pi_{32} = \pi_{12} = 1$. (Rute $3 \to 1 \to 2$.)
- d_{34}: $\min(2,\ 3 + 5 = 8) = 2$. Tidak berubah.

Ronde 2 (lewat simpul 1 dan 2). Kolom 2: $d_{12} = 4$, $d_{32} = 7$, $d_{42} = 1$. Baris 2: $d_{21} = \infty$, $d_{23} = -2$, $d_{24} = \infty$. Hanya $j = 3$:

- d_{13}: $\min(\infty,\ 4 + (-2)) = 2$. **Berubah**; $\pi_{13} = \pi_{23} = 2$.
- d_{33}: $\min(0,\ 7 + (-2) = 5) = 0$. Tidak berubah — diagonalnya tetap 0, sebagaimana mestinya tanpa siklus negatif.
- d_{43}: $\min(6,\ 1 + (-2)) = -1$. **Berubah**; $\pi_{43} = \pi_{23} = 2$. Sisi negatif itu telah membuat rute $4 \to 2 \to 3$ lebih murah daripada sisi langsung berbobot 6.

Ronde 3 (lewat 1, 2, 3). Kolom 3: $d_{13} = 2$, $d_{23} = -2$, $d_{43} = -1$. Baris 3: $d_{31} = 3$, $d_{32} = 7$, $d_{34} = 2$. Kini kesembilan kombinasinya berhingga:

- $i = 1$: $d_{11} = \min(0,\ 2+3) = 0$; $d_{12} = \min(4,\ 2+7) = 4$; $d_{14} = \min(5,\ 2+2) = 4$, **berubah**, $\pi_{14} = \pi_{34} = 3$.
- $i = 2$: $d_{21} = \min(\infty,\ -2+3) = 1$, **berubah**, $\pi_{21} = \pi_{31} = 3$; $d_{22} = \min(0,\ -2+7) = 0$; $d_{24} = \min(\infty,\ -2+2) = 0$, **berubah**, $\pi_{24} = \pi_{34} = 3$.
- $i = 4$: $d_{41} = \min(\infty,\ -1+3) = 2$, **berubah**, $\pi_{41} = \pi_{31} = 3$; $d_{42} = \min(1,\ -1+7) = 1$; $d_{44} = \min(0,\ -1+2) = 0$.

Ronde 4 (semua simpul diizinkan). Kolom 4: $d_{14} = 4$, $d_{24} = 0$, $d_{34} = 2$. Baris 4: $d_{41} = 2$, $d_{42} = 1$, $d_{43} = -1$:

- $i = 1$: $d_{11} = \min(0,6) = 0$; $d_{12} = \min(4,\ 4+1=5) = 4$; $d_{13} = \min(2,\ 4-1=3) = 2$.
- $i = 2$: $d_{21} = \min(1,\ 0+2) = 1$; $d_{22} = \min(0,\ 1) = 0$; $d_{23} = \min(-2,\ 0-1=-1) = -2$.
- $i = 3$: $d_{31} = \min(3,\ 2+2=4) = 3$; $d_{32} = \min(7,\ 2+1) = 3$, **berubah**, $\pi_{32} = \pi_{42} = 4$; $d_{33} = \min(0,\ 2-1=1) = 0$.

Tabel 16.15 menghimpun kedelapan perubahan sepanjang jalannya algoritma.

Tabel 16.15 Setiap entri yang berubah selama Floyd–Warshall pada Persoalan 16.11, berikut perbandingan yang mengubahnya. Perhatikan bahwa d_{32} berubah dua kali: rute yang ditemukan pada ronde 1 dikalahkan oleh rute yang ditemukan pada ronde 4.

Ronde k	Entri	Lama	Calon $d_{ik}+d_{kj}$	Baru	π_{ij} Baru
1	d_{32}	∞	$3+4=7$	7	1
2	d_{13}	∞	$4+(-2)=2$	2	2
2	d_{43}	6	$1+(-2)=-1$	-1	2
3	d_{14}	5	$2+2=4$	4	3
3	d_{21}	∞	$-2+3=1$	1	3
3	d_{24}	∞	$-2+2=0$	0	3
3	d_{41}	∞	$-1+3=2$	2	3
4	d_{32}	7	$2+1=3$	3	4

Jawaban. Matriks-matriks akhirnya adalah

$$D^{(4)} = \begin{pmatrix} 0 & 4 & 2 & 4 \\ 1 & 0 & -2 & 0 \\ 3 & 3 & 0 & 2 \\ 2 & 1 & -1 & 0 \end{pmatrix}, \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 2 & 3 \\ 3 & \text{NIL} & 2 & 3 \\ 3 & 4 & \text{NIL} & 3 \\ 3 & 4 & 2 & \text{NIL} \end{pmatrix}.$$

Membaca lintasan dari $\Pi^{(4)}$ berjalan mundur dari tujuannya, dengan memakai baris i sepanjang waktu:

- $1 \rightsquigarrow 4$: $\pi_{14} = 3, \pi_{13} = 2, \pi_{12} = 1$. Lintasannya $1 \to 2 \to 3 \to 4$, berbobot $4 - 2 + 2 = 4 = d_{14}$.
- $4 \rightsquigarrow 1$: $\pi_{41} = 3, \pi_{43} = 2, \pi_{42} = 4$. Lintasannya $4 \to 2 \to 3 \to 1$, berbobot $1 - 2 + 3 = 2 = d_{41}$.
- $3 \rightsquigarrow 2$: $\pi_{32} = 4, \pi_{34} = 3$. Lintasannya $3 \to 4 \to 2$, berbobot $2 + 1 = 3 = d_{32}$. (Rute ronde 1, $3 \to 1 \to 2$ berbobot 7, dengan benar ditinggalkan.)

Siklus negatif. Siklus negatif yang melewati simpul i, paling lambat pada ronde ketika semua simpulnya diizinkan, akan memberi rute dari i kembali ke i yang berbobot negatif, jadi ia akan membuat $d_{ii} < 0$. Karena itu Floyd–Warshall mendeteksi siklus negatif tanpa biaya tambahan:

sesudah ronde terakhir, lihatlah diagonalnya (Subbab 15.6.14). Di sini setiap entri diagonalnya 0, jadi tidak ada siklus negatif.

Program. **Kode 16.13** mencetak setiap matriks, dengan tanda bintang pada setiap entri yang berubah pada ronde itu.

Kode 16.13 Floyd–Warshall pada graf Persoalan 16.11, mencetak $D^{(k)}$ dan $\Pi^{(k)}$ untuk $k = 0, \ldots, 4$ dan menandai setiap entri yang berubah, disusul uji siklus negatif pada diagonalnya.

```
// Persoalan 16.11: Floyd-Warshall pada graf empat simpul dengan satu sisi negatif,
// mencetak setiap matriks jarak D(0)..D(4) dan setiap matriks pendahulu
// Pi(0)..Pi(4), serta menandai dengan * setiap entri yang berubah pada ronde itu.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;

const int INF = 1000000;
const int n = 4;
struct Edge { int u, v, w; };

int D[n + 1][n + 1], P[n + 1][n + 1];          // berbasis 1; P = 0 berarti NIL
bool changed[n + 1][n + 1];

void print(int k) {
    cout << "D(" << k << ")                          Pi(" << k << ")\n";
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= n; ++j) {
            string c = (D[i][j] >= INF) ? "inf" : to_string(D[i][j]);
            if (changed[i][j]) c += "*";
            cout << setw(6) << c;
        }
        cout << "        ";
        for (int j = 1; j <= n; ++j) {
            if (P[i][j] == 0) cout << setw(4) << "NIL"; else cout << setw(4) << P[i][j];
        }
        cout << "\n";
    }
    cout << "\n";
}

int main() {
    vector<Edge> E = {{1,2,4}, {1,4,5}, {2,3,-2}, {3,1,3},
                      {3,4,2}, {4,2,1}, {4,3,6}};
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j) { D[i][j] = (i == j) ? 0 : INF; P[i][j] = 0; }
    for (auto& e : E) { D[e.u][e.v] = e.w; P[e.u][e.v] = e.u; }
    print(0);
```

Kode 16.13 (Lanjutan) *Floyd–Warshall pada graf Persoalan 16.11, mencetak D^{(k)} dan \Pi^{(k)} untuk k = 0, \ldots, 4 dan menandai setiap entri yang berubah, disusul uji siklus negatif pada diagonalnya.*

```
    for (int k = 1; k <= n; ++k) {
        for (int i = 1; i <= n; ++i)
            for (int j = 1; j <= n; ++j) {
                changed[i][j] = false;
                if (D[i][k] < INF && D[k][j] < INF && D[i][k] + D[k][j] < D[i][j]) {
                    D[i][j] = D[i][k] + D[k][j];   // lewat k saja
                    P[i][j] = P[k][j];             // langkah terakhir disalin dari rute k
                    changed[i][j] = true;
                }
            }
        print(k);
    }

    bool negCycle = false;
    for (int i = 1; i <= n; ++i) if (D[i][i] < 0) negCycle = true;
    cout << "negative diagonal entry: " << (negCycle ? "yes" : "no") << "\n";
    return 0;
}
```

Ia mencetak:

```
D(0)                          Pi(0)
     0     4   inf     5          NIL   1 NIL   1
   inf     0    -2   inf          NIL NIL   2 NIL
     3   inf     0     2            3 NIL NIL   3
   inf     1     6     0          NIL   4   4 NIL

D(1)                          Pi(1)
     0     4   inf     5          NIL   1 NIL   1
   inf     0    -2   inf          NIL NIL   2 NIL
     3    7*     0     2            3   1 NIL   3
   inf     1     6     0          NIL   4   4 NIL

D(2)                          Pi(2)
     0     4    2*     5          NIL   1   2   1
   inf     0    -2   inf          NIL NIL   2 NIL
     3     7     0     2            3   1 NIL   3
   inf     1   -1*     0          NIL   4   2 NIL

D(3)                          Pi(3)
     0     4     2    4*          NIL   1   2   3
    1*     0    -2    0*            3 NIL   2   3
     3     7     0     2            3   1 NIL   3
    2*     1    -1     0            3   4   2 NIL

D(4)                          Pi(4)
     0     4     2     4          NIL   1   2   3
     1     0    -2     0            3 NIL   2   3
     3    3*     0     2            3   4 NIL   3
     2     1    -1     0            3   4   2 NIL

negative diagonal entry: no
```

Entri-entri berbintang tepat merupakan kedelapan baris **Tabel 16.15**. Waktu jalannya $\Theta(n^3)$: di sini $4^3 = 64$ perbandingan, yang sebagian besarnya dilewati penyelesaian tangan berkat pengamatan "baris k dan kolom k harus berhingga".

Uji diri. Ubahlah bobot $4 \to 2$ dari 1 menjadi -1. Siklus mana yang menjadi negatif, dan apa yang diperlihatkan diagonal $D^{(4)}$?

Jawaban. Siklus $2 \to 3 \to 4 \to 2$ kini berbobot $-2 + 2 - 1 = -1$. Menjalankan algoritmanya memberi $D^{(4)}$ dengan diagonal $(0, -1, -1, -2)$: entri 2, 3, dan 4 negatif, yang menandai siklus itu,

sedangkan $d_{11} = 0$ karena simpul 1 tidak berada di atasnya. Entri $d_{44} = -2$ adalah bobot mengelilingi siklus itu *dua kali*, yang merupakan peringatan tersendiri: begitu ada siklus negatif, bilangan-bilangan dalam D bukan lagi bobot lintasan terpendek apa pun, dan hanya tanda-tandanya pada diagonal yang boleh dipercaya.

16.10.2 Persoalan 16.12 — pembobotan ulang Johnson, dikerjakan selengkapnya

Persoalan 16.12. Selesaikan persoalan semua pasangan untuk graf yang sama dengan algoritma Johnson. (a) Tambahkan simpul baru q dengan sisi berbobot 0 ke setiap simpul, jalankan Bellman–Ford dari q, dan berikan potensial $h(v) = \delta(q, v)$. (b) Hitunglah setiap sisi yang dibobot ulang $w'(u, v)$ dan tunjukkan bahwa tak satu pun negatif. (c) Jalankan Dijkstra dari setiap simpul pada graf yang dibobot ulang, lalu ubahlah hasilnya kembali menjadi jarak yang sebenarnya. (d) Tunjukkan bahwa pembobotan ulang tidak dapat mengubah lintasan mana yang terpendek, dan bandingkan dengan Persoalan 16.11.

Apa yang diuji. Subbab 15.8: pembobotan ulang (*reweighting*)

$$w'(u, v) \;=\; w(u, v) + h(u) - h(v) \tag{16.11}$$

dengan h dipilih sebagai bobot lintasan terpendek dari sebuah simpul bantu, dua sifatnya — ia mempertahankan lintasan terpendek, dan ia membuat setiap bobot tak negatif — dan pembukuan yang mengubah jawaban Dijkstra atas w' kembali menjadi jawaban atas w.

Penyelesaian (a). Simpul baru q mempunyai sisi $q \to 1, q \to 2, q \to 3, q \to 4$, semuanya berbobot 0; tidak ada yang masuk ke q, jadi tidak ada siklus baru yang tercipta dan graf barunya bersiklus negatif hanya jika graf lamanya demikian. Jalankan Bellman–Ford dari q pada kelima simpul, dengan merelaksasi keempat sisi-q lebih dahulu, lalu ketujuh sisi asli dalam urutan yang didaftar.

Putaran 1. Sisi-sisi q menetapkan $h(1) = h(2) = h(3) = h(4) = 0$. Lalu: $1 \to 2$: $0 + 4 \not< 0$; $1 \to 4$: $0 + 5 \not< 0$; $2 \to 3$: $0 + (-2) = -2 < 0$, jadi $h(3) = -2$; $3 \to 1$: $-2 + 3 = 1 \not< 0$; $3 \to 4$: $-2 + 2 = 0 \not< 0$; $4 \to 2$: $0 + 1 \not< 0$; $4 \to 3$: $0 + 6 \not< -2$.

Putaran 2 sampai 4 tidak mengubah apa pun, dan pemeriksaan penutupnya tidak menemukan sisi yang dapat diperbaiki, jadi tidak ada siklus negatif. Potensial-potensialnya adalah

$$h(1) = 0, \qquad h(2) = 0, \qquad h(3) = -2, \qquad h(4) = 0.$$

Setiap $h(v)$ adalah bobot cara termurah mencapai v dengan mulai dari mana saja (sebab q dapat melangkah ke simpul mana pun tanpa biaya). Hanya simpul 3 yang dapat dicapai lebih murah daripada gratis, lewat sisi negatif $2 \to 3$.

Penyelesaian (b). Terapkan Persamaan (16.11) pada setiap sisi. **Tabel 16.16** memperlihatkan hitungannya.

Tabel 16.16 Pembobotan ulang Johnson pada graf Persoalan 16.12. Setiap sisi yang dibobot ulang tak negatif, dan sisi negatif $2 \to 3$ telah menjadi sisi berbobot 0.

Sisi (u,v)	$w(u,v)$	$h(u)$	$h(v)$	$w'(u,v)=w+h(u)-h(v)$
(1,2)	4	0	0	$4+0-0=4$
(1,4)	5	0	0	$5+0-0=5$
(2,3)	-2	0	-2	$-2+0+2=0$
(3,1)	3	-2	0	$3-2-0=1$
(3,4)	2	-2	0	$2-2-0=0$
(4,2)	1	0	0	$1+0-0=1$
(4,3)	6	0	-2	$6+0+2=8$

Setiap w' tak negatif, dan bukan karena kebetulan. Karena $h(v) = \delta(q, v)$, ketaksamaan segitiga Bab 13 memberi $h(v) \le h(u) + w(u, v)$ untuk setiap sisi, yang dapat disusun ulang menjadi $w(u, v) + h(u) - h(v) \ge 0$. Perhatikan juga sisi *mana* yang keluar tepat 0: (2,3) dan (3,4) — sisi-sisi pada lintasan terpendek dari q, tempat ketaksamaan segitiga berlaku dengan kesamaan.

Penyelesaian (c). Dengan bobot tak negatif, Dijkstra sah dipakai. Jalankan dari setiap sumber u untuk memperoleh $\hat{d}(u, v)$, jaraknya di bawah w'. Inilah jalannya dari sumber 4, selengkapnya:

- Keluarkan 4 ($\hat{d} = 0$): relaksasi $4 \to 2$ menjadi $0 + 1 = 1$, dan $4 \to 3$ menjadi $0 + 8 = 8$.
- Keluarkan 2 ($\hat{d} = 1$): relaksasi $2 \to 3$: $1 + 0 = 1 < 8$, jadi $\hat{d}(4,3) = 1$.
- Keluarkan 3 ($\hat{d} = 1$): relaksasi $3 \to 1$ menjadi $1 + 1 = 2$; $3 \to 4$: $1 + 0 \not< 0$.
- Keluarkan 1 ($\hat{d} = 2$): $1 \to 2$: $2 + 4 \not< 1$; $1 \to 4$: $2 + 5 \not< 0$.

Jadi $\hat{d}(4,\cdot) = (2,1,1,0)$. Untuk mengubahnya kembali, batalkan pergeserannya. Sebuah lintasan dari u ke v, menurut identitas teleskopik Persamaan (16.12) di bawah, mempunyai bobot-w' yang sama dengan bobot-w-nya ditambah $h(u) - h(v)$, jadi

$$\delta(u, v) \;=\; \hat{d}(u, v) - h(u) + h(v).$$

Untuk $u = 4$ (dengan $h(4) = 0$): $\delta(4,1) = 2 - 0 + 0 = 2$, $\delta(4,2) = 1 - 0 + 0 = 1$, $\delta(4,3) = 1 - 0 + (-2) = -1$, $\delta(4,4) = 0$. Ketiga sumber lainnya berjalan dengan cara yang sama; program di bawah mencetak keempat barisnya.

Penyelesaian (d). Misalkan $p = v_0 \to v_1 \to \cdots \to v_k$ adalah lintasan mana pun. Bobot ulangnya adalah

$$w'(p) \;=\; \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \;=\; w(p) + h(v_0) - h(v_k), \qquad (16.12)$$

sebab setiap $h(v_i)$ di bagian dalam ditambahkan sekali dan dikurangkan sekali. Koreksi $h(v_0) - h(v_k)$ hanya bergantung pada kedua ujungnya, jadi **setiap** lintasan dari v_0 ke v_k tergeser dengan besar yang *sama*, dan urutan lintasan-lintasan itu menurut bobot — khususnya, yang mana yang terpendek — tidak berubah. Periksalah pada lintasan $1 \to 2 \to 3$: $w = 4 - 2 = 2$, $w' = 4 + 0 = 4$, dan memang $2 + h(1) - h(3) = 2 + 0 + 2 = 4$.

Program. Kode 16.14 mengerjakan keempat langkahnya.

Kode 16.14 Algoritma Johnson pada graf Persoalan 16.12: Bellman–Ford dari simpul bantu, pembobotan ulang berikut hitungannya, dan Dijkstra dari setiap sumber, dengan jarak $\hat{d}$ di bawah w' dan jarak D yang dipulihkan berdampingan.

```
// Persoalan 16.12: algoritma Johnson pada graf empat simpul Persoalan 16.11.
//   1. tambahkan simpul q = 0 dengan sisi berbobot 0 ke setiap simpul
//   2. Bellman-Ford dari q memberikan potensial h(v) = delta(q, v)
//   3. bobot ulang: w'(u,v) = w(u,v) + h(u) - h(v), yang tak pernah negatif
//   4. Dijkstra dari setiap simpul pada w', lalu batalkan pergeserannya
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
using namespace std;

const int INF = 1000000;
const int n = 4;
struct Edge { int u, v, w; };

int main() {
    vector<Edge> E = {{1,2,4}, {1,4,5}, {2,3,-2}, {3,1,3},
                      {3,4,2}, {4,2,1}, {4,3,6}};

    // Langkah 1 dan 2: Bellman-Ford dari q atas E ditambah sisi q->v berbobot 0.
    vector<Edge> Eq;
    for (int v = 1; v <= n; ++v) Eq.push_back({0, v, 0});
    for (auto& e : E) Eq.push_back(e);
    vector<int> h(n + 1, INF);
    h[0] = 0;
    for (int pass = 1; pass <= n; ++pass) {          // |V'| - 1 = n putaran
        for (auto& e : Eq)
            if (h[e.u] < INF && h[e.u] + e.w < h[e.v]) h[e.v] = h[e.u] + e.w;
        cout << "Bellman-Ford pass " << pass << ":";
        for (int v = 1; v <= n; ++v) cout << "  h(" << v << ")=" << setw(2) << h[v];
        cout << "\n";
    }
    for (auto& e : Eq)
        if (h[e.u] + e.w < h[e.v]) { cout << "negative cycle\n"; return 0; }

    // Langkah 3: bobot ulang.
    cout << "\nreweighted edges:\n";
    vector<vector<pair<int,int>>> adj(n + 1);
    for (auto& e : E) {
        int w2 = e.w + h[e.u] - h[e.v];
        cout << "  w'(" << e.u << "," << e.v << ") = " << setw(2) << e.w << " + (" <<
setw(2)
             << h[e.u] << ") - (" << setw(2) << h[e.v] << ") = " << w2 << "\n";
        adj[e.u].push_back({e.v, w2});
    }
```

Kode 16.14 (Lanjutan) *Algoritma Johnson pada graf Persoalan 16.12: Bellman–Ford dari simpul bantu, pembobotan ulang berikut hitungannya, dan Dijkstra dari setiap sumber, dengan jarak \hat{d} di bawah w' dan jarak D yang dipulihkan berdampingan.*

```
    // Langkah 4: Dijkstra dari setiap sumber pada w', lalu D[u][v] = dhat + h(v) - h(u).
    cout << "\n  source   dhat(u,1..4)          D(u,1..4)\n";
    for (int s = 1; s <= n; ++s) {
        vector<int> d(n + 1, INF);
        d[s] = 0;
        priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
        Q.push({0, s});
        while (!Q.empty()) {
            auto [du, u] = Q.top(); Q.pop();
            if (du > d[u]) continue;
            for (auto [v, w] : adj[u])
                if (d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
        }
        cout << setw(8) << s << "  ";
        for (int v = 1; v <= n; ++v) cout << setw(5) << d[v];
        cout << "     ";
        for (int v = 1; v <= n; ++v) cout << setw(5) << d[v] + h[v] - h[s];
        cout << "\n";
    }
    return 0;
}
```

Ia mencetak:

```
Bellman-Ford pass 1:  h(1)= 0  h(2)= 0  h(3)=-2  h(4)= 0
Bellman-Ford pass 2:  h(1)= 0  h(2)= 0  h(3)=-2  h(4)= 0
Bellman-Ford pass 3:  h(1)= 0  h(2)= 0  h(3)=-2  h(4)= 0
Bellman-Ford pass 4:  h(1)= 0  h(2)= 0  h(3)=-2  h(4)= 0

reweighted edges:
  w'(1,2) =  4 + ( 0) - ( 0) = 4
  w'(1,4) =  5 + ( 0) - ( 0) = 5
  w'(2,3) = -2 + ( 0) - (-2) = 0
  w'(3,1) =  3 + (-2) - ( 0) = 1
  w'(3,4) =  2 + (-2) - ( 0) = 0
  w'(4,2) =  1 + ( 0) - ( 0) = 1
  w'(4,3) =  6 + ( 0) - (-2) = 8

  source   dhat(u,1..4)          D(u,1..4)
       1      0    4    4    4         0    4    2    4
       2      1    0    0    0         1    0   -2    0
       3      1    1    0    0         3    3    0    2
       4      2    1    1    0         2    1   -1    0
```

Kolom-kolom D tepat merupakan baris-baris $D^{(4)}$ dari Persoalan 16.11. Dua algoritma yang dibangun di atas gagasan yang sama sekali berbeda — program dinamis atas simpul antara yang diizinkan, dan reduksi ke Dijkstra lewat potensial — sepakat pada keenam belas entrinya.

Mana yang akan Anda pakai? Pada empat simpul, tidak ada bedanya. Secara umum, Floyd–Warshall berbiaya $\Theta(V^3)$ dan Johnson berbiaya satu Bellman–Ford, $O(VE)$, ditambah V kali Dijkstra, $O(VE\lg V)$ dengan *binary heap*. Untuk graf padat, $E = \Theta(V^2)$, $O(V^3\lg V)$ milik Johnson kalah oleh gelung rangkap tiga Floyd–Warshall yang sederhana; untuk graf jarang, $E = \Theta(V)$, $O(V^2\lg V)$ milik Johnson menang dengan faktor hampir V. Persoalan 16.17 di subbab berikutnya membandingkan keduanya dengan kata-kata dan pada graf yang lebih besar.

Uji diri. Ubahlah bobot $1 \to 4$ dari 5 menjadi -1 (ini tidak menciptakan siklus negatif). Temukan potensial-potensial yang baru, sisi-sisi yang dibobot ulang, dan baris pertama yang baru dari matriks jaraknya.

Jawaban. Bellman–Ford dari q memberi $h = (0,0,-2,-1)$: simpul 4 kini tercapai seharga -1 lewat $1 \to 4$. Sisi-sisi yang dibobot ulang adalah $w'(1,2) = 4$, $w'(1,4) = -1 + 0 + 1 = 0$, $w'(2,3) = 0$, $w'(3,1) = 1$, $w'(3,4) = 2 - 2 + 1 = 1$, $w'(4,2) = 1 - 1 - 0 = 0$, $w'(4,3) = 6 - 1 + 2 = 7$ — semuanya tak negatif. Dijkstra dari 1 memberi $\hat{d}(1,\cdot) = (0,0,0,0)$, dan dengan mengubahnya kembali, $\delta(1,\cdot) = (0,\ 0-0+0,\ 0-0-2,\ 0-0-1) = (0,0,-2,-1)$. Rute ke 2 kini $1 \to 4 \to 2$ berbobot $-1 + 1 = 0$.

16.11 Persoalan Terkerjakan Lanjutan

Sembilan persoalan dalam subbab ini meninjau kembali bagian buku selebihnya dengan bentuk yang sama seperti Persoalan 16.1 sampai 16.5: lima belas pernyataan singkat, sebuah algoritma untuk dirancang dan dianalisis, sebuah induksi, sebuah bukti tiga bagian tentang jaminan sebuah algoritma, dan perbandingan dua algoritma untuk tugas yang sama. Persoalan-persoalan itu datang dalam dua kelompok, Persoalan 16.13 sampai 16.17 dan Persoalan 16.18 sampai 16.21, dan setiap kelompok layak dicoba dalam satu kali duduk, dengan penyelesaiannya ditutup, sebelum membaca lebih lanjut.

16.11.1 Persoalan 16.13 — lima belas pernyataan lagi

Persoalan 16.13. Tandailah setiap pernyataan **B** atau **S**, dengan pembenaran satu baris.

1. Algoritma Kruskal dapat diimplementasikan agar berjalan dalam waktu $O(E\lg V)$.
2. Pada graf berarah yang bersiklus, pengurutan topologis tetap ada jika siklusnya berpanjang 2.
3. Algoritma rakus selalu menghasilkan penyelesaian optimal untuk persoalan *knapsack* pecahan.
4. Algoritma rakus selalu menghasilkan penyelesaian optimal untuk persoalan *knapsack* 0/1.
5. Pengodean Huffman memberikan kata kode terpendek kepada simbol yang paling sering muncul.
6. Pencarian melebar menghitung lintasan terpendek pada graf tak berbobot.
7. Pencarian mendalam pada graf tak berarah dapat menghasilkan sisi silang (*cross edge*).
8. Floyd–Warshall berjalan dalam waktu $\Theta(V^3)$ dan memakai ruang $\Theta(V^2)$.
9. Algoritma Johnson secara asimtotik lebih baik daripada Floyd–Warshall pada graf jarang.
10. Biaya amortisasi sebuah operasi dapat lebih besar daripada biaya kasus terburuknya.
11. Dalam metode potensial, fungsi potensial boleh menjadi negatif asalkan kembali ke nol pada akhirnya.
12. Tabel dinamis yang berlipat dua ketika meluap mempunyai biaya penyisipan amortisasi $O(1)$.
13. Merelaksasi sebuah sisi dapat menaikkan $d[v]$.
14. Setiap sisi pohon merentang minimum adalah sisi teringan yang menyeberangi suatu potongan.
15. Lintasan terpendek pada graf tanpa siklus negatif tidak pernah mengulang simpul.

Penyelesaian. (1) **B** — mengurutkan E sisi mendominasi, $O(E\lg E) = O(E\lg V)$ karena $E < V^2$ sehingga $\lg E < 2\lg V$. (2) **S** — siklus apa pun, sepanjang apa pun, membuat pengurutan topologis mustahil; siklus-2 $u \to v \to u$ tetaplah siklus. (3) **B** — ambillah butir menurut nilai-per-satuan-berat yang menurun dan belah butir terakhirnya; argumen pertukaran membuktikannya optimal. (4) **S** — dengan butir $(w, v) = (10,60), (20,100), (30,120)$ dan kapasitas 50, rakus menurut rasio mengambil dua yang pertama seharga 160, padahal $\{2,3\}$ memberi 220. (5) **B** — simbol berfrekuensi lebih tinggi digabungkan lebih belakangan, jadi ia duduk lebih dekat ke akar; secara

formal, jika $f(a) > f(b)$ maka $\text{depth}(a) \leq \text{depth}(b)$ dalam pohon Huffman. (6) **B** — BFS menghitung lintasan bersisi paling sedikit, yang terpendek ketika setiap bobot 1. (7) **S** — DFS tak berarah hanya menghasilkan sisi pohon dan sisi balik; calon sisi silang pasti sudah dijelajahi dari ujung yang lain lebih dahulu. (8) **B** — tiga gelung bersarang, satu matriks $V \times V$ jika diperbarui di tempat. (9) **B** — Johnson $O(V^2 \lg V + VE)$ dengan *Fibonacci heap*; pada graf jarang dengan $E = \Theta(V)$ itu $O(V^2 \lg V)$, berhadapan dengan $\Theta(V^3)$. (10) **B** — metode akuntansi dengan sengaja menarif lebih pada operasi murah; PUSH pada tumpukan berbiaya kasus terburuk 1 dan berbiaya amortisasi 2. (11) **S** — metode potensial menuntut $\Phi(D_i) \geq \Phi(D_0)$ untuk *setiap* i, bukan hanya pada akhirnya, agar Persamaan (16.7) membatasi total sebenarnya untuk setiap awalan barisan; Persoalan 16.10(f) memperlihatkan apa yang salah jika tidak demikian. (12) **B** — n penyisipan berbiaya kurang dari $3n$ (Persoalan 16.10), jadi $O(1)$ masing-masing. (13) **S** — relaksasi hanya pernah menurunkan $d[v]$; ia dijaga oleh $d[v] > d[u] + w(u, v)$. (14) **B** — itulah sifat potongan dibaca terbalik: untuk $e \in T$, menghapus e membelah T menjadi dua komponen, dan e adalah sisi teringan yang menyeberangi potongan itu. (15) **B** — mengulang simpul berarti memuat siklus, yang bobotnya ≥ 0, jadi menghapusnya tidak memperpanjang lintasan.

16.11.2 Persoalan 16.14 — lintasan terpanjang pada DAG

Persoalan 16.14. Rancanglah algoritma berwaktu polinomial yang menemukan lintasan terpanjang dari sebuah simpul sumber pada **graf berarah taksiklik** berbobot. Berikan asumsi, pseudokode, dan analisis kompleksitas waktunya, lalu jelaskan mengapa pendekatan yang sama gagal pada graf berarah umum.

Apa yang diuji. Persoalan 16.8 yang dibalik: algoritma DAG bekerja dengan bobot *apa pun*, jadi ia bekerja juga dengan bobot yang dinegasikan, dan "terpendek dengan $-w$" adalah "terpanjang dengan w".

Penyelesaian. *Asumsi.* Masukannya adalah graf berarah taksiklik $G = (V, E)$ dengan bobot sisi riil w dan sumber s; "lintasan terpanjang" berarti bobot total maksimum; sebuah lintasan tidak mengunjungi simpul dua kali; simpul yang tidak terjangkau dari s berpanjang $-\infty$.

Rancangan. DAG mempunyai urutan topologis — urutan linear tempat setiap sisi mengarah maju (Bab 10). Proseslah simpul-simpulnya dalam urutan itu dan relaksasikan setiap sisi keluar, dengan mengambil **maksimum** alih-alih minimum, seperti pada **Pseudokode 16.5**.

Pseudokode 16.5 Lintasan terpanjang dari s pada DAG berbobot: algoritma lintasan terpendek DAG dengan perbandingan yang dibalik.

```
LONGEST-PATH-DAG(G, w, s)
1  topologically sort V(G)
2  for each v in V:  dist[v] = -infinity;  par[v] = NIL
3  dist[s] = 0
4  for each u in topological order
5      if dist[u] > -infinity
6          for each edge (u,v) in Adj[u]
7              if dist[u] + w(u,v) > dist[v]
8                  dist[v] = dist[u] + w(u,v);  par[v] = u
9  return dist, par
```

Kebenaran. Ketika u diproses, setiap pendahulu u sudah diproses, sebab urutan topologis meletakkan ekor setiap sisi sebelum kepalanya. Jadi $dist[u]$ sudah tuntas ketika dipakai. Secara formal, dengan induksi atas urutan topologisnya: klaim *"$dist[u]$ sama dengan bobot lintasan terpanjang dari s ke u ketika u diproses"* berlaku untuk simpul pertama dan terpelihara karena setiap sisi masuk u sudah direlaksasi tepat sekali sebelum u dicapai.

Kompleksitas. Pengurutan topologis $\Theta(V + E)$; gelung relaksasinya menyentuh setiap sisi tepat sekali, jadi totalnya $\Theta(V + E)$. Ruang $\Theta(V)$.

Mengapa ia gagal pada graf berarah umum. Dua alasan, dan keduanya sebaiknya diberikan. Pertama, tidak ada urutan topologis, jadi argumen "proses pendahulu lebih dahulu" tidak punya pijakan. Kedua dan lebih mendasar, **lintasan sederhana terpanjang pada graf berarah umum adalah NP-sulit**: lintasan Hamilton adalah lintasan sederhana terpanjang ketika semua bobotnya 1. Ketaksiklikan bukan sekadar kemudahan; ia adalah seluruh alasan persoalan ini mudah.

Sebuah contoh terkerjakan. Ambillah $V = \{1, \dots, 7\}$ dan sisi-sisi $1 \to 2$ (6), $1 \to 3$ (4), $2 \to 4$ (5), $2 \to 5$ (9), $3 \to 4$ (3), $3 \to 5$ (7), $4 \to 6$ (2), $4 \to 7$ (8), $5 \to 6$ (4), $6 \to 7$ (3). Dalam urutan topologis 1,3,2,5,4,6,7:

$$dist[3] = 4, \quad dist[2] = 6, \quad dist[5] = \max(6 + 9,\ 4 + 7) = 15, \quad dist[4] = \max(6 + 5,\ 4 + 3) = 11,$$

$$dist[6] = \max(11 + 2,\ 15 + 4) = 19, \quad dist[7] = \max(11 + 8,\ 19 + 3) = 22.$$

Lintasan terpanjangnya $1 \to 2 \to 5 \to 6 \to 7$ berbobot $6 + 9 + 4 + 3 = 22$. **Kode 16.15** mengukuhkannya, dengan memakai pengurutan topologis Kahn yang berbasis antrean alih-alih DFS Persoalan 16.8, sehingga kedua metode Bab 10 muncul dalam bab ini.

Kode 16.15 Lintasan terpanjang pada DAG berbobot untuk Persoalan 16.14: pengurutan topologis Kahn disusul satu relaksasi pemaksimum per sisi.

```
// Persoalan 16.14: lintasan terpanjang pada DAG berbobot, menurut urutan topologis.
// Pengurutan topologis (Bab 11) + satu pengenduran per sisi (Bab 13) = Theta(V+E).
#include <iostream>
#include <iomanip>
#include <vector>
#include <stack>
#include <climits>
using namespace std;

const int NEG = INT_MIN / 4;

int main() {
    const int n = 7;                              // simpul 1..7
    vector<vector<pair<int,int>>> adj(n + 1);
    auto add = [&](int u, int v, int w) { adj[u].push_back({v, w}); };
    add(1,2,6); add(1,3,4); add(2,4,5); add(2,5,9); add(3,4,3);
    add(3,5,7); add(4,6,2); add(4,7,8); add(5,6,4); add(6,7,3);

    vector<int> indeg(n + 1, 0);
    for (int u = 1; u <= n; ++u) for (auto [v, w] : adj[u]) ++indeg[v];

    vector<int> order;                            // pengurutan topologis Kahn
    vector<int> ind = indeg;
    stack<int> S;
    for (int v = 1; v <= n; ++v) if (ind[v] == 0) S.push(v);
    while (!S.empty()) {
        int u = S.top(); S.pop();
        order.push_back(u);
        for (auto [v, w] : adj[u]) if (--ind[v] == 0) S.push(v);
    }
    cout << "topological order:";
    for (int v : order) cout << " " << v;
    cout << "\n\n";

    vector<int> dist(n + 1, NEG), par(n + 1, 0);
    dist[1] = 0;
    cout << "relaxing in topological order\n  v  dist[v]  via\n";
    for (int u : order) {
        if (dist[u] == NEG) continue;             // tak terjangkau dari sumbernya
        for (auto [v, w] : adj[u])
            if (dist[u] + w > dist[v]) { dist[v] = dist[u] + w; par[v] = u; }
    }
    for (int v : order)
        cout << setw(3) << v << setw(9) << dist[v] << setw(6) << par[v] << "\n";
```

Kode 16.15 (Lanjutan) *Lintasan terpanjang pada DAG berbobot untuk Persoalan 16.14: pengurutan topologis Kahn disusul satu relaksasi pemaksimum per sisi.*

```
    int best = 1;
    for (int v = 1; v <= n; ++v) if (dist[v] > dist[best]) best = v;
    vector<int> path;
    for (int v = best; v != 0; v = par[v]) path.push_back(v);
    cout << "\nlongest path has weight " << dist[best] << ":";
    for (int i = (int)path.size() - 1; i >= 0; --i) cout << " " << path[i];
    cout << "\n";
    return 0;
}
```

```
topological order: 1 3 2 5 4 6 7

relaxing in topological order
  v  dist[v]  via
  1        0    0
  3        4    1
  2        6    1
  5       15    2
  4       11    2
  6       19    5
  7       22    6

longest path has weight 22: 1 2 5 6 7
```

Inilah metode lintasan kritis (*critical path method*) dalam penjadwalan proyek (Subbab 13.9.7): jika simpul-simpulnya adalah tonggak dan bobotnya durasi, lintasan terpanjang adalah panjang terpendek yang mungkin dari seluruh proyek.

Uji diri. Tambahkan sisi $3 \to 6$ berbobot 20. Berapa lintasan terpanjangnya sekarang?

Jawaban. $dist[6] = \max(11+2,\ 15+4,\ 4+20) = 24$ dan $dist[7] = \max(11+8,\ 24+3) = 27$: lintasan $1 \to 3 \to 6 \to 7$ berbobot $4+20+3 = 27$. Perhatikan bahwa urutan topologis 1,3,2,5,4,6,7 tetap sah, sebab sisi barunya mengarah maju.

16.11.3 Persoalan 16.15 — pohon dengan n simpul mempunyai $n-1$ sisi

Persoalan 16.15. Buktikan dengan induksi bahwa sebuah pohon — graf tak berarah yang terhubung dan taksiklik — dengan n simpul mempunyai tepat $n-1$ sisi.

Penyelesaian. *Klaim.* Setiap graf tak berarah yang terhubung dan taksiklik pada $n \geq 1$ simpul mempunyai tepat $n-1$ sisi. *Bukti, dengan induksi kuat atas n.*

Dasar, $n = 1$**.** Satu simpul, terhubung dan taksiklik, tidak mempunyai sisi, dan $1-1=0$.

Hipotesis. Anggaplah setiap pohon pada n' simpul mempunyai $n'-1$ sisi, untuk semua $1 \leq n' < n$.

Langkah. Misalkan T adalah pohon pada $n \geq 2$ simpul. Pertama, T mempunyai simpul berderajat 1. (Berjalanlah dari simpul mana pun tanpa langsung berbalik; karena T berhingga dan taksiklik, perjalanan itu tidak dapat kembali ke simpul yang sudah dikunjungi, jadi ia harus berhenti, dan ia hanya dapat berhenti di simpul tanpa sisi yang belum terpakai — sebuah daun.) Misalkan ℓ adalah simpul seperti itu dan e satu-satunya sisinya. Buanglah ℓ dan e untuk memperoleh T'.

T' taksiklik, sebab membuang sesuatu tidak dapat menciptakan siklus. T' terhubung: lintasan mana pun di T antara dua simpul T' tidak dapat *melewati* ℓ, sebab memasuki ℓ memakai satu-satunya sisinya dan keluar darinya harus memakai sisi itu lagi. Dan T' mempunyai $n-1$ simpul, dengan $1 \leq n-1 < n$, jadi hipotesisnya memberi $|E(T')| = (n-1)-1 = n-2$. Dengan menambahkan kembali ℓ dan e,

$$|E(T)| = |E(T')| + 1 = n-2+1 = n-1. \qquad \blacksquare$$

Alternatifnya, dalam dua baris. Bangunlah T dengan menambahkan simpul satu demi satu, masing-masing disambungkan ke bagian yang sudah ada dengan tepat satu sisi — satu sisi per simpul sesudah yang pertama. Ini induksi yang sama dinyatakan ke depan. Dan inilah fakta yang dipakai diam-diam di Persoalan 16.4: menghapus satu sisi dari tur n simpul menyisakan $n-1$ sisi pada n simpul, terhubung — sebuah pohon merentang.

16.11.4 Persoalan 16.16 — pengodean Huffman, dalam tiga bagian

Persoalan 16.16. Algoritma Huffman berulang kali menggabungkan dua simbol yang paling jarang muncul. Buktikan (1) bahwa kode yang dihasilkannya bebas-awalan (*prefix-free*), sehingga dapat diurai tanpa ambigu; (2) bahwa waktu jalannya $O(n\lg n)$ untuk n simbol; (3) bahwa kode yang dihasilkannya mempunyai panjang harapan minimum di antara semua kode bebas-awalan.

Penyelesaian (1): bebas-awalan. Dalam pohon yang dibangun Huffman, sebuah simbol hanya diletakkan pada **daun**: setiap penggabungan menciptakan simpul *dalam* baru yang anak-anaknya adalah dua pohon yang digabungkan, dan simbol tidak pernah dipindahkan dari daun. Sebuah kata kode adalah barisan pilihan kiri/kanan pada lintasan dari akar ke daun. Jika kata kode x merupakan awalan kata kode y, maka daun untuk x terletak pada lintasan menuju daun untuk y — mustahil, sebab daun tidak mempunyai keturunan. Karena itu tak satu pun kata kode menjadi awalan kata kode lain, dan penguraiannya tidak ambigu: bacalah bit sampai sebuah daun tercapai, keluarkan simbolnya, mulai lagi dari akar. ■

Penyelesaian (2): $O(n\lg n)$. Bangunlah antrean prioritas-min dari n simbol: $O(n)$ dengan `Build-Min-Heap`. Gelungnya berjalan $n-1$ kali; setiap iterasi mengerjakan dua `Extract-Min` dan satu `Insert`, masing-masing $O(\lg n)$. Totalnya $O(n)+O(n\lg n)=O(n\lg n)$. Jika frekuensinya datang sudah terurut, dua antrean biasa menggantikan *heap*-nya dan seluruh algoritmanya menjadi $\Theta(n)$.

Penyelesaian (3): keoptimalan. Dua lema, lalu sebuah induksi — bentuk baku setiap bukti rakus (Bab 9). Tuliskan $f(c)$ untuk frekuensi simbol c dan $d(c)$ untuk kedalamannya dalam pohon; biaya sebuah pohon adalah $\sum_c f(c)\,d(c)$.

Lema A (pilihan rakus). Misalkan x dan y adalah dua simbol berfrekuensi terendah. Ada kode awalan optimal yang di dalamnya x dan y bersaudara pada kedalaman maksimum. *Bukti.* Ambillah pohon optimal T mana pun dan misalkan a, b adalah dua saudara pada kedalaman maksimum (simpul dalam terdalam mempunyai dua anak daun, sebab pohon optimal itu penuh). Tukarkan x dengan a. Biayanya berubah sebesar

$$\begin{aligned} f(x)\,d(a)+f(a)\,d(x)-f(x)\,d(x)-f(a)\,d(a) \\ = \; -(f(a)-f(x))(d(a)-d(x)) \;\le\; 0, \end{aligned}$$

sebab $f(x)\le f(a)$ dan $d(x)\le d(a)$. Jadi pertukaran itu tidak menaikkan biaya. Kerjakan hal yang sama untuk y dan b. Hasilnya tetap optimal dan mempunyai x, y sebagai saudara terdalam. ▫

Lema B (substruktur optimal). Misalkan T adalah pohon optimal untuk abjad C, dan misalkan x, y bersaudara di dalamnya dengan induk z. Misalkan $C' = C \setminus \{x,y\} \cup \{z\}$ dengan $f(z)=f(x)+f(y)$, dan misalkan T' adalah T dengan daun x, y dihapus. Maka $\text{cost}(T)=\text{cost}(T')+f(x)+f(y)$, dan T' optimal untuk C'. *Bukti.* Identitas biayanya adalah hitungan: x dan y duduk satu aras di bawah z, jadi $f(x)d(x)+f(y)d(y)=(f(x)+f(y))(d(z)+1)=f(z)d(z)+f(x)+f(y)$. Untuk keoptimalannya, jika suatu T'' mengalahkan T' pada C', memasang kembali x dan y di bawah z dalam T'' akan mengalahkan T pada C. ▫

Induksi. Atas $n=|C|$. Untuk $n=2$ pohon mana pun memberikan satu bit untuk tiap simbol dan semuanya optimal. Untuk $n>2$, Huffman menggabungkan x, y menjadi z dan berekursi pada C', yang mempunyai $n-1$ simbol; menurut induksi ia mengembalikan pohon optimal T' untuk C'; menurut Lema B pohon yang diperoleh dengan memasang kembali x dan y berbiaya $\text{cost}(T')+f(x)+f(y)$, dan menurut Lema A suatu pohon optimal untuk C mempunyai x, y sebagai saudara

terdalam sehingga, lagi-lagi menurut Lema B, berbiaya paling sedikit sebesar itu. Jadi pohon Huffman optimal. ■

Soal Latihan 8 di Subbab 16.19 membangun kode Huffman untuk enam tanda layanan Profitina Laundry dan memeriksa hitungannya dengan sebuah program.

16.11.5 Persoalan 16.17 — Floyd–Warshall berhadapan dengan Johnson

Persoalan 16.17. Floyd–Warshall dan Johnson sama-sama menyelesaikan persoalan lintasan terpendek semua pasangan. (1) Uraikan perbedaan keduanya dan kapan masing-masing sebaiknya dipilih; (2) tuliskan pseudokode masing-masing dan tunjukkan di mana keduanya berbeda; (3) ilustrasikan masing-masing pada graf kecil.

Penyelesaian (1): perbedaannya. Floyd–Warshall adalah program dinamis atas *simpul mana yang boleh menjadi simpul antara sebuah lintasan*, Persamaan (16.10). Ia tiga gelung bersarang, berwaktu $\Theta(V^3)$ dan berruang $\Theta(V^2)$, ia menerima sisi negatif, dan ia mendeteksi siklus negatif lewat entri diagonal yang negatif. Johnson adalah sebuah *reduksi*: jalankan Bellman–Ford sekali dari simpul tambahan untuk memperoleh potensial h, bobot ulang setiap sisi dengan Persamaan (16.11) — yang membuat setiap bobot tak negatif dan mempertahankan lintasan terpendek — lalu jalankan Dijkstra dari setiap simpul. Itu $O(VE)$ untuk Bellman–Ford ditambah V kali Dijkstra, memberi $O(V^2 \lg V + VE)$ dengan *Fibonacci heap*. **Pilih Floyd–Warshall ketika grafnya padat atau V kecil; pilih Johnson ketika grafnya jarang.** Pada $V = 1000$, $E = 3000$: Floyd–Warshall mengerjakan 10^9 langkah dasar, Johnson sekitar 10^7.

Penyelesaian (2): pseudokode. **Pseudokode 16.6** meletakkan keduanya berdampingan.

Pseudokode 16.6 Floyd–Warshall (kiri) dan Johnson (kanan). Floyd–Warshall tidak pernah mengubah bobotnya; Johnson mengubahnya sekali, di baris 4, sehingga Dijkstra menjadi sah.

```
FLOYD-WARSHALL(W)
1 D = W
2 for k = 1 to n
3   for i = 1 to n
4     for j = 1 to n
5       D[i][j] = min(D[i][j],
6                     D[i][k]+D[k][j])
7 return D
```

```
JOHNSON(G, w)
1 add vertex q joined to all v with w(q,v)=0
2 if BELLMAN-FORD(G', w, q) = false: report neg. cycle
3 h(v) = delta(q,v) for all v
4 w'(u,v) = w(u,v) + h(u) - h(v)  for all (u,v)
5 for each u in V
6   run DIJKSTRA(G, w', u) giving dhat(u,v)
7   D[u][v] = dhat(u,v) - h(u) + h(v)
8 return D
```

Perbedaan yang perlu ditunjuk: Floyd–Warshall tidak pernah mengubah bobotnya dan membayar $\Theta(V^3)$; Johnson mengubah bobotnya sekali sehingga Dijkstra menjadi sah, dan membayar V kali Dijkstra. Pembobotan ulang itu sahih karena potensial-potensialnya berteleskop sepanjang lintasan mana pun, Persamaan (16.12), sehingga setiap lintasan di antara dua ujung yang sama tergeser dengan konstanta yang sama dan peminimumnya tidak berubah. Ketaknegatifan w' menyusul dari ketaksamaan segitiga pada h: $h(v) \leq h(u) + w(u, v)$.

Penyelesaian (3): ilustrasi. Persoalan 16.11 dan 16.12 sudah menjalankan kedua algoritma dengan tangan pada graf empat simpul. **Kode 16.16** menjalankan keduanya pada graf lima simpul yang lebih besar dengan dua sisi negatif, mencetak $D^{(0)}$ sampai $D^{(5)}$, potensial-potensialnya, sisi-sisi yang dibobot ulang, dan jawaban Johnson.

Kode 16.16 Floyd–Warshall dan Johnson pada satu graf lima simpul, dengan keenam matriks antara, potensial-potensial, dan sisi-sisi yang dibobot ulang dicetak.

```
// Persoalan 16.17: Floyd-Warshall berhadapan dengan Johnson pada graf yang sama.
// Lima simpul, satu sisi negatif, tanpa siklus negatif.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

const int INF = 1000000;
const int n = 5;
struct Edge { int u, v, w; };
vector<Edge> E = {{1,2,3},{1,3,8},{1,5,-4},{2,4,1},{2,5,7},
                  {3,2,4},{4,1,2},{4,3,-5},{5,4,6}};

void printMat(const string& tag, int D[6][6]) {
    cout << tag << "\n";
    for (int i = 1; i <= n; ++i) {
        cout << "   ";
        for (int j = 1; j <= n; ++j) {
            if (D[i][j] >= INF) cout << setw(5) << "inf";
            else                cout << setw(5) << D[i][j];
        }
        cout << "\n";
    }
    cout << "\n";
}

int main() {
    int D[6][6];
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j) D[i][j] = (i == j) ? 0 : INF;
    for (auto& e : E) D[e.u][e.v] = min(D[e.u][e.v], e.w);
    printMat("D(0)  -- the weight matrix W", D);

    for (int k = 1; k <= n; ++k) {
        for (int i = 1; i <= n; ++i)
            for (int j = 1; j <= n; ++j)
                if (D[i][k] < INF && D[k][j] < INF && D[i][k] + D[k][j] < D[i][j])
                    D[i][j] = D[i][k] + D[k][j];   // penjaga: inf + w tetap inf
        printMat("D(" + to_string(k) + ")", D);
    }
```

Kode 16.16 (Lanjutan) *Floyd–Warshall dan Johnson pada satu graf lima simpul, dengan keenam matriks antara, potensial-potensial, dan sisi-sisi yang dibobot ulang dicetak.*

```
    // ---- Johnson: bobot baru dari h hasil Bellman-Ford, lalu Dijkstra n kali.
    vector<int> h(n + 1, 0);                    // h = jarak dari simpul baru 0
    for (int pass = 1; pass <= n; ++pass)       // sisi 0->v semuanya berbobot 0
        for (auto& e : E)
            if (h[e.u] + e.w < h[e.v]) h[e.v] = h[e.u] + e.w;
    cout << "Johnson's potentials h(v):";
    for (int v = 1; v <= n; ++v) cout << "  h(" << v << ")=" << h[v];
    cout << "\n\nreweighted edges  w'(u,v) = w(u,v) + h(u) - h(v):\n   ";
    for (auto& e : E)
        cout << e.u << "->" << e.v << "=" << e.w + h[e.u] - h[e.v] << "  ";
    cout << "\n\n";

    vector<vector<pair<int,int>>> adj(n + 1);
    for (auto& e : E) adj[e.u].push_back({e.v, e.w + h[e.u] - h[e.v]});

    int J[6][6];
    for (int s = 1; s <= n; ++s) {
        vector<int> d(n + 1, INF); d[s] = 0;
        priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
        Q.push({0, s});
        while (!Q.empty()) {
            auto [du, u] = Q.top(); Q.pop();
            if (du > d[u]) continue;
            for (auto [v, w] : adj[u])
                if (d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
        }
        for (int v = 1; v <= n; ++v)
            J[s][v] = (d[v] >= INF) ? INF : d[v] - h[s] + h[v];   // batalkan pergeserannya
    }
    printMat("Johnson's answer", J);

    bool same = true;
    for (int i = 1; i <= n; ++i) for (int j = 1; j <= n; ++j) if (D[i][j] != J[i][j]) same
= false;
    cout << "Floyd-Warshall and Johnson agree on every entry: "
         << (same ? "yes" : "no") << "\n";
    return 0;
}
```

Keluarannya:

```
D(0)  -- the weight matrix W
       0    3    8  inf   -4
     inf    0  inf    1    7
     inf    4    0  inf  inf
       2  inf   -5    0  inf
     inf  inf  inf    6    0

D(1)
       0    3    8  inf   -4
     inf    0  inf    1    7
     inf    4    0  inf  inf
       2    5   -5    0   -2
     inf  inf  inf    6    0

D(2)
       0    3    8    4   -4
     inf    0  inf    1    7
     inf    4    0    5   11
       2    5   -5    0   -2
     inf  inf  inf    6    0

D(3)
       0    3    8    4   -4
     inf    0  inf    1    7
     inf    4    0    5   11
       2   -1   -5    0   -2
     inf  inf  inf    6    0

D(4)
       0    3   -1    4   -4
       3    0   -4    1   -1
       7    4    0    5    3
       2   -1   -5    0   -2
       8    5    1    6    0

D(5)
       0    1   -3    2   -4
       3    0   -4    1   -1
       7    4    0    5    3
       2   -1   -5    0   -2
       8    5    1    6    0

Johnson's potentials h(v):  h(1)=0  h(2)=-1  h(3)=-5  h(4)=0  h(5)=-4

reweighted edges  w'(u,v) = w(u,v) + h(u) - h(v):
   1->2=4  1->3=13  1->5=0  2->4=0  2->5=10  3->2=0  4->1=2  4->3=0  5->4=2

Johnson's answer
       0    1   -3    2   -4
       3    0   -4    1   -1
       7    4    0    5    3
       2   -1   -5    0   -2
       8    5    1    6    0

Floyd-Warshall and Johnson agree on every entry: yes
```

Perhatikan bahwa setiap sisi yang dibobot ulang tak negatif — 4,13,0,0,10,0,2,0,2 — tepat syarat yang dibutuhkan Dijkstra, dibuat dengan sengaja.

16.11.6 Persoalan 16.18 — lima belas pernyataan sekali lagi

Persoalan 16.18. Tandailah setiap pernyataan **B** atau **S**, dengan pembenaran satu baris.

1. Heapsort mengurutkan di tempat dan berjalan dalam $\Theta(n \lg n)$ pada setiap kasus.
2. `BUILD-MAX-HEAP` berjalan dalam $\Theta(n \lg n)$.

3. Pohon AVL bertinggi h memuat paling sedikit $F(h+3)-1$ simpul, dengan F adalah barisan Fibonacci.
4. Penelusuran *inorder* pohon pencarian biner mencetak kunci-kuncinya dalam urutan terurut.
5. Setiap pengurutan berbasis perbandingan mengerjakan $\Omega(n\lg n)$ perbandingan pada kasus terburuk.
6. Memoisasi dan tabulasi dari bawah ke atas selalu mempunyai waktu jalan asimtotik yang sama.
7. Persoalan dengan substruktur optimal selalu dapat diselesaikan secara rakus.
8. Hutan himpunan-lepas (*disjoint-set forest*) dengan gabungan menurut peringkat dan pemampatan lintasan mendukung m operasi pada n unsur dalam $O(m\,\alpha(n))$.
9. Dalam algoritma Prim, himpunan sisi yang terpilih selalu membentuk satu pohon.
10. Dalam algoritma Kruskal, himpunan sisi yang terpilih selalu membentuk satu pohon.
11. Algoritma Dijkstra mengunjungi setiap sisi tepat sekali.
12. Metode agregat, akuntansi, dan potensial dapat memberikan biaya amortisasi yang berbeda untuk operasi yang sama.
13. Program dinamis *knapsack* 0/1 berjalan dalam waktu polinomial terhadap ukuran masukannya.
14. Graf dengan V simpul dan lebih dari $V-1$ sisi pasti memuat siklus.
15. Membuang sisi terberat sebuah siklus tidak pernah memutus graf yang terhubung.

Penyelesaian. (1) **B**. (2) **S** — ia $\Theta(n)$; sebagian besar simpul berada dekat dasar, dan $\sum_h \lceil n/2^{h+1}\rceil O(h) = O(n)$. (3) **B** — dari $N(h) = 1 + N(h-1) + N(h-2)$. (4) **B**. (5) **B** — pohon keputusannya mempunyai $n!$ daun, jadi tingginya $\geq \lg(n!) = \Theta(n\lg n)$. (6) **S** — keduanya sepakat ketika setiap subpersoalan dibutuhkan, tetapi memoisasi dapat melewati subpersoalan yang akan diisi tabelnya, dan karena itu dapat lebih cepat secara asimtotik pada sebagian masukan; tabulasi menghindari ongkos rekursi dan dapat lebih cepat dalam praktik pada masukan lain. (7) **S** — substruktur optimal perlu tetapi tidak cukup; sifat pilihan rakus adalah hipotesis tambahannya, dan *knapsack* 0/1 mempunyai yang pertama tanpa yang kedua. (8) **B**. (9) **B** — Prim menumbuhkan satu pohon dari akarnya; sisi-sisi yang terpilih terhubung pada setiap langkah. (10) **S** — Kruskal memelihara *hutan* yang baru menjadi satu pohon pada sisi yang paling akhir. (11) **S** — ia merelaksasi setiap sisi paling banyak sekali, tetapi sisi dari simpul yang tak terjangkau tidak pernah direlaksasi sama sekali. (12) **B** — ketiga metode boleh membebani operasi yang *sama* secara berbeda: metode agregat memberi setiap operasi biaya rata-rata yang sama, sedangkan metode akuntansi dan potensial dapat menarif `PUSH` 2 dan `POP` 0. Yang harus mereka sepakati adalah bahwa total amortisasinya membatasi total sebenarnya, bukan bagaimana total itu dibagi; pada Persoalan 16.9 ketiganya kebetulan sepakat panggilan demi panggilan. (13) **S** — $\Theta(nW)$ adalah *pseudo*-polinomial: W eksponensial terhadap banyaknya bit yang dipakai untuk menuliskannya, jadi waktu jalannya tidak polinomial terhadap ukuran masukan. Persoalan 16.19 membahas tepat hal ini. (14) **B** — hutan pada V simpul mempunyai paling banyak $V-1$ sisi (Persoalan 16.15, diterapkan pada setiap komponen), jadi lebih dari $V-1$ sisi memaksa adanya siklus. (15) **B** — sifat siklus: kedua ujungnya tetap terhubung lewat sisa siklusnya.

16.11.7 Persoalan 16.19 — *knapsack* 0/1, dan arti "polinomial"

Persoalan 16.19. Rancanglah algoritma untuk persoalan *knapsack* 0/1: n butir dengan berat bulat w_i dan nilai v_i, serta kapasitas bulat W; pilihlah himpunan bagian bernilai total maksimum

dengan berat total paling banyak W. Berikan asumsi, pseudokode, analisis kompleksitas waktu, dan pernyataan yang dibenarkan dengan cermat tentang apakah algoritma Anda berjalan dalam waktu polinomial.

Penyelesaian. *Asumsi.* w_i, v_i, W adalah bilangan bulat tak negatif; setiap butir boleh diambil paling banyak sekali (itulah arti 0/1); nilai himpunan kosong adalah 0.

Rancangan. Misalkan $c[i][j]$ adalah nilai terbaik yang dapat diperoleh dari i butir pertama dengan kapasitas j yang tersedia. Butir i entah ditinggalkan — nilai $c[i-1][j]$ — entah diambil, yang membutuhkan $w_i \leq j$ dan menghasilkan $v_i + c[i-1][j-w_i]$:

$$c[i][j] = \begin{cases} 0 & i = 0 \text{ atau } j = 0, \\ c[i-1][j] & w_i > j, \\ \max(c[i-1][j],\ v_i + c[i-1][j-w_i]) & \text{selainnya.} \end{cases} \tag{16.13}$$

Pseudokode. **Pseudokode 16.7** mengisi tabel Persamaan (16.13) baris demi baris.

Pseudokode 16.7 Program dinamis *knapsack* 0/1: n baris berisi $W + 1$ entri, masing-masing dihitung dalam waktu konstan.

```
KNAPSACK(w, v, n, W)
for j = 0 to W:  c[0][j] = 0
for i = 1 to n
  c[i][0] = 0
  for j = 1 to W
    if w[i] > j: c[i][j] = c[i-1][j]
    else:        c[i][j] = max(c[i-1][j], v[i] + c[i-1][j-w[i]])
return c[n][W]
```

Kompleksitas. Dua gelung bersarang berisi n dan W iterasi dengan badan $\Theta(1)$: waktu $\Theta(nW)$ dan ruang $\Theta(nW)$, yang dapat dikurangi menjadi ruang $\Theta(W)$ dengan hanya menyimpan satu baris.

Apakah ia polinomial? **Tidak, dan menjawab ya adalah jebakannya.** Masukannya terdiri atas $2n + 1$ bilangan. Menuliskan W membutuhkan $\lceil \lg(W + 1) \rceil$ bit, jadi jika masukannya berukuran b bit maka W dapat sebesar 2^b. Waktu jalan $\Theta(nW)$ karena itu eksponensial terhadap *ukuran* masukan, walaupun polinomial terhadap *nilai* masukan. Algoritma seperti itu disebut **pseudo-polinomial**. Ia cepat ketika W kecil — jadwal mesin 60 menit, sistem koin 100 rupiah — dan tidak berguna ketika W bilangan 64-bit. *Knapsack* 0/1 NP-sulit, dan algoritma yang sungguh polinomial akan membuktikan P = NP.

Contoh terkerjakan. Lima layanan tambahan Profitina Laundry, masing-masing memakai menit mesin dan menghasilkan nilai dalam ribuan rupiah, dengan anggaran sepuluh menit: kanji (3,30), kilat (4,50), pewangi (5,60), lipat (2,20), setrika (6,70). **Kode 16.17** mengisi tabelnya (nama-nama layanan dalam program tetap dalam bahasa Inggris: *starch*, *express*, *perfume*, *fold*, *iron*).

Kode 16.17 Program dinamis *knapsack* 0/1 Persoalan 16.19, mencetak seluruh tabel c dan menyusun kembali himpunan yang terpilih dengan menelusurinya mundur.

```
// Persoalan 16.19: knapsack 0/1 dengan pemrograman dinamis, seluruh tabelnya.
// Lima layanan tambahan, masing-masing memakan menit-mesin dan menghasilkan rupiah.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

int main() {
    const int n = 5, W = 10;
    string name[n + 1] = {"", "starch", "express", "perfume", "fold", "iron"};
    int w[n + 1] = {0, 3, 4, 5, 2, 6};           // menit-mesin
    int v[n + 1] = {0, 30, 50, 60, 20, 70};      // ribu rupiah

    vector<vector<int>> c(n + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= n; ++i)
        for (int j = 0; j <= W; ++j)
            c[i][j] = (w[i] > j) ? c[i-1][j]
                                 : max(c[i-1][j], c[i-1][j - w[i]] + v[i]);

    cout << "      j =";
    for (int j = 0; j <= W; ++j) cout << setw(4) << j;
    cout << "\n";
    for (int i = 0; i <= n; ++i) {
        cout << setw(8) << (i == 0 ? string("-") : name[i]) << " ";
        for (int j = 0; j <= W; ++j) cout << setw(4) << c[i][j];
        cout << "\n";
    }

    cout << "\nbest value = " << c[n][W] << "\nchosen:";
    int j = W, used = 0;
    for (int i = n; i >= 1; --i)
        if (c[i][j] != c[i-1][j]) { cout << " " << name[i]; used += w[i]; j -= w[i]; }
    cout << "\nminutes used = " << used << " of " << W << "\n";
    return 0;
}
```

```
      j =   0   1   2   3   4   5   6   7   8   9  10
       -    0   0   0   0   0   0   0   0   0   0   0
  starch    0   0   0  30  30  30  30  30  30  30  30
 express    0   0   0  30  50  50  50  80  80  80  80
 perfume    0   0   0  30  50  60  60  80  90 110 110
    fold    0   0  20  30  50  60  70  80  90 110 110
    iron    0   0  20  30  50  60  70  80  90 110 120

best value = 120
chosen: iron express
minutes used = 10 of 10
```

Periksalah dengan tangan: kilat dan setrika memberi $50 + 70 = 120$ dalam $4 + 6 = 10$ menit, dan tak ada himpunan bagian lain yang muat dalam sepuluh menit yang mengalahkannya — $\{$ pewangi, kilat $\}$ memberi 110 dalam 9 menit, $\{$ pewangi, kanji, lipat $\}$ memberi 110 dalam 10. Perhatikan juga aturan penyusunan kembalinya: $c[i][j] \neq c[i-1][j]$ berarti butir i diambil.

Uji diri. Naikkan anggarannya menjadi 11 menit. Berapa nilai terbaiknya, dan layanan apa saja?

Jawaban. 130, dicapai oleh dua himpunan bagian yang berbeda: pewangi dan setrika ($60 + 70$ dalam $5 + 6 = 11$ menit), dan kilat, pewangi, dan lipat ($50 + 60 + 20$ dalam $4 + 5 + 2 = 11$ menit). Tabelnya hanya menghitung $c[5][11] = 130$; himpunan optimal mana yang dilaporkan penelusuran mundurnya bergantung pada urutan butir-butirnya, sama seperti subbarisan menaik terpanjang pada Persoalan 16.2 yang tidak tunggal.

16.11.8 Persoalan 16.20 — hampiran-2 untuk penutup simpul

Persoalan 16.20. `APPROX-VERTEX-COVER` berulang kali memilih sisi yang belum tertutup (u, v) secara sembarang, menambahkan **kedua** u dan v ke penutupnya, dan menghapus setiap sisi yang bersisian dengan u atau v. Buktikan (1) bahwa himpunan yang dikembalikannya adalah penutup simpul (*vertex cover*); (2) bahwa waktu jalannya polinomial; (3) bahwa algoritma itu mempunyai batas rasio $\rho(n) = 2$.

Penyelesaian (1): ia mengembalikan penutup simpul. Gelungnya baru berakhir ketika tidak ada lagi sisi yang belum tertutup — algoritmanya menyingkirkan sebuah sisi dari pertimbangan tepat ketika salah satu ujungnya sudah masuk ke C. Jadi pada akhirnya setiap sisi graf asli mempunyai paling sedikit satu ujung di C, dan itulah definisi penutup simpul. ■

Penyelesaian (2): polinomial. Dengan senarai ketetanggaan, setiap sisi diperiksa sejumlah konstan kali: sekali ketika dipilih, dan sekali ketika dihapus karena salah satu ujungnya terpilih. Jadi waktu jalannya $O(V + E)$. ■

Penyelesaian (3): batas rasio 2. Misalkan A adalah himpunan sisi yang *dipilih* algoritmanya (bukan yang dihapusnya). Dua pengamatan.

A adalah pemasangan (*matching*). Ketika sisi (u, v) dipilih, setiap sisi yang menyentuh u atau v dihapus, jadi tak ada sisi terpilih sesudahnya yang berbagi ujung dengannya. Karena itu sisi-sisi A saling lepas berpasangan.

Setiap penutup simpul memuat paling sedikit satu ujung setiap sisi A, dan ujung-ujung itu berbeda. Misalkan C^* adalah penutup optimal. Setiap sisi A harus tertutup, jadi C^* memuat paling sedikit satu dari kedua ujungnya; dan karena sisi-sisi A tidak berbagi ujung, itu adalah $|A|$ simpul yang berbeda. Karena itu $|C^*| \geq |A|$.

Algoritmanya mengembalikan $2|A|$ simpul, dua per sisi yang dipilih, semuanya berbeda menurut sifat pemasangan. Jadi

$$|C| = 2|A| \leq 2|C^*|, \qquad \frac{|C|}{|C^*|} \leq 2. \qquad \blacksquare$$

Buktinya berkerangka sama dengan Persoalan 16.4: temukan besaran yang menjadi batas bawah optimumnya ($|A|$ di sini, $d(T)$ di sana), lalu tunjukkan bahwa jawaban algoritmanya paling banyak dua kali besaran itu. **Kode 16.18** menjalankannya berhadapan dengan gaya kasar.

Kode 16.18 Hampiran-2 untuk penutup simpul pada Persoalan 16.20, dengan pemasangan yang dibangunnya secara tersirat dan optimum sejati yang ditemukan dengan pencarian menyeluruh.

```
// Persoalan 16.20: APPROX-VERTEX-COVER. Ambil sisi tak tertutup mana pun, masukkan
// KEDUA ujungnya ke penutup, buang tiap sisi yang disentuhnya, ulangi. Batas nisbah 2.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <set>
using namespace std;

struct Edge { int u, v; };

int main() {
    const int n = 7;
    string nm[n] = {"a","b","c","d","e","f","g"};
    vector<Edge> E = {{0,1},{1,2},{2,3},{3,4},{3,5},{4,5},{4,6},{2,6}};

    // ---- hampirannya
    set<int> C;
    vector<bool> covered(E.size(), false);
    vector<Edge> matching;
    for (size_t i = 0; i < E.size(); ++i) {
        if (covered[i]) continue;
        int u = E[i].u, v = E[i].v;
        C.insert(u); C.insert(v);
        matching.push_back(E[i]);
        cout << "pick edge (" << nm[u] << "," << nm[v] << ")  ->  cover gains "
             << nm[u] << " and " << nm[v] << "\n";
        for (size_t k = 0; k < E.size(); ++k)
            if (E[k].u == u || E[k].v == u || E[k].u == v || E[k].v == v) covered[k] =
true;
    }
    cout << "\napproximate cover C =";
    for (int v : C) cout << " " << nm[v];
    cout << "   |C| = " << C.size()
         << "   (matching size " << matching.size() << ")\n";

    // ---- optimumnya, gaya kasar atas semua 2^n upahimpunan
    int best = n + 1, bestMask = 0;
    for (int mask = 0; mask < (1 << n); ++mask) {
        bool ok = true;
        for (auto& e : E)
            if (!((mask >> e.u) & 1) && !((mask >> e.v) & 1)) { ok = false; break; }
        if (!ok) continue;
        int sz = __builtin_popcount(mask);
        if (sz < best) { best = sz; bestMask = mask; }
    }
    cout << "optimal cover C* =";
```

Kode 16.18 (Lanjutan) *Hampiran-2 untuk penutup simpul pada Persoalan 16.20, dengan pemasangan yang dibangunnya secara tersirat dan optimum sejati yang ditemukan dengan pencarian menyeluruh.*

```
    for (int v = 0; v < n; ++v) if ((bestMask >> v) & 1) cout << " " << nm[v];
    cout << "   |C*| = " << best << "\n";

    cout << fixed << setprecision(4)
         << "\nratio |C|/|C*| = " << (double)C.size() / best
         << "   (the theorem guarantees at most 2)\n"
         << "check |C*| >= matching size:  " << best << " >= " << matching.size() << "\n"
         << "check |C| = 2 * matching:     " << C.size() << " = " << 2 * matching.size() <<
"\n";
    return 0;
}
```

```
pick edge (a,b)  ->  cover gains a and b
pick edge (c,d)  ->  cover gains c and d
pick edge (e,f)  ->  cover gains e and f

approximate cover C = a b c d e f   |C| = 6   (matching size 3)
optimal cover C* = a c d e   |C*| = 4

ratio |C|/|C*| = 1.5000   (the theorem guarantees at most 2)
check |C*| >= matching size:  4 >= 3
check |C| = 2 * matching:     6 = 6
```

16.11.9 Persoalan 16.21 — Prim berhadapan dengan Kruskal

Persoalan 16.21. Algoritma Prim dan Kruskal sama-sama membangun pohon merentang minimum. (1) Uraikan perbedaan keduanya dan kapan masing-masing sebaiknya dipilih; (2) tuliskan pseudokode masing-masing dan tunjukkan di mana keduanya berbeda; (3) ilustrasikan masing-masing pada graf kecil.

Penyelesaian (1): perbedaannya. Keduanya adalah wujud algoritma MST generik Bab 11: berulang kali tambahkan sisi yang **aman**, dibenarkan oleh sifat potongan. Prim menumbuhkan *satu pohon* dari akarnya, selalu mengambil sisi teringan yang meninggalkannya; ia membutuhkan antrean prioritas dan berjalan dalam $\Theta(V^2)$ dengan larik atau $O(E\lg V)$ dengan *binary heap*. Kruskal menumbuhkan *hutan*, mengurutkan semua sisi dan mengambil masing-masing jika ia menyambungkan dua komponen yang berbeda; ia membutuhkan hutan himpunan-lepas dan berjalan dalam $O(E\lg E) = O(E\lg V)$, didominasi pengurutannya. **Pilih Prim pada graf padat**, tempat $\Theta(V^2)$ mengalahkan $O(V^2\lg V)$ dan tidak dibutuhkan pengurutan; **pilih Kruskal pada graf jarang**, atau ketika sisi-sisinya datang sudah terurut, dan dalam hal itu Kruskal $O(E\,\alpha(V))$ — praktis linear, menurut analisis amortisasi *union–find* di Subbab 14.8.

Penyelesaian (2): pseudokode. **Pseudokode 16.8** meletakkan keduanya berdampingan.

Pseudokode 16.8 Prim (kiri) dan Kruskal (kanan). Baris 5 masing-masing adalah tempat keduanya berbeda: simpul terdekat ke pohon, berhadapan dengan sisi berikutnya yang menyambungkan dua komponen.

```
PRIM(G, w, r)                          KRUSKAL(G, w)
1 for each v: key[v]=inf, pi[v]=NIL    1 A = empty
2 key[r] = 0                           2 for each v: MAKE-SET(v)
3 Q = V                                3 sort E by weight, ascending
4 while Q not empty                    4 for each (u,v) in E, in order
5   u = EXTRACT-MIN(Q)                 5   if FIND-SET(u) != FIND-SET(v)
6   for each v in Adj[u]               6     A = A + {(u,v)}
7     if v in Q and w(u,v) < key[v]    7     UNION(u,v)
8       pi[v] = u; key[v] = w(u,v)     8 return A
```

Perbedaan strukturalnya ada di baris 5 masing-masing. `EXTRACT-MIN` milik Prim bertanya *"simpul mana yang paling dekat ke pohon yang sudah saya miliki?"*; `FIND-SET` milik Kruskal bertanya *"apakah sisi ini menyambungkan dua hal yang belum tersambung?"* Prim lokal dan bertahap; Kruskal global dan terurut. Baris 8 Prim juga kerabat relaksasi: Dijkstra akan menetapkan `key[v] = key[u] + w(u,v)`, Prim menetapkan `key[v] = w(u,v)` — jarak ke pohon alih-alih jarak ke sumber.

Penyelesaian (3): ilustrasi. Pakailah keenam titik laundry pada **Tabel 16.5**. Kedua algoritma menghasilkan pohon yang sama berbobot 22 — dan karena pohon itu tunggal di sini, memang harus demikian. **Tabel 16.17** meletakkan penelusuran keduanya berdampingan.

Tabel 16.17 Prim dan Kruskal pada graf enam simpul **Tabel 16.5**. Prim menambahkan $F - E$ lalu $E - D$; Kruskal menambahkan $D - E$ lalu $E - F$. *Urutannya* berbeda; *himpunan* sisinya $\{AF, AB, BC, DE, EF\}$ dan bobotnya 22 tidak.

Prim dari A	Sisi	w	Kum.		Kruskal	Sisi	w	Putusan	Kum.
1	(akar A)	0	0		1	$A - F$	3	terima	3
2	$A - F$	3	3		2	$A - B$	4	terima	7
3	$A - B$	4	7		3	$B - C$	5	terima	12
4	$B - C$	5	12		4	$D - E$	5	terima	17
5	$F - E$	5	17		5	$E - F$	5	terima	22
6	$E - D$	5	22		—	berhenti: 5 sisi			22

Rincian yang mengajar: tiga sisi berbobot 5, jadi teorema ketunggalan di Subbab 16.3.9 tidak berlaku — namun pencarian menyeluruh atas seluruh $\binom{15}{5} = 3003$ himpunan bagian berlima sisi menemukan tepat satu pohon merentang berbobot 22. Bobot kembar mengizinkan pohon merentang minimum kedua; ia tidak menghasilkannya.

16.12 Persoalan-Persoalan Bab Ini Dibandingkan

Sesudah mengerjakan dua puluh satu persoalan, mundurlah selangkah dan tanyakan apa yang dipakai masing-masing. **Tabel 16.18** meletakkan semuanya berdampingan.

Tabel 16.18 Kedua puluh satu persoalan bab ini sekilas: jenis masing-masing, alat yang menyelesaikannya dan di mana buku ini mengajarkannya, dan biaya penyelesaiannya.

Persoalan	Subbab	Jenisnya	Alat utama, dan di mana ia diajarkan	Biaya penyelesaiannya
16.1 lima belas pernyataan	16.3	pengenalan di seluruh buku	O berhadapan dengan Ω, kasus terburuk berhadapan dengan rata-rata (Bab 3, 5, 6, 11)	—
16.2 subbarisan menaik	16.4	rancangan dan analisis	pemrograman dinamis, subpersoalan yang diperkuat (Bab 7)	$\Theta(n^2)$; $O(n\lg n)$
16.3 pohon biner diperluas	16.5	bukti induksi	induksi kuat atas simpul dalam (Bab 3, 6)	—
16.4 TSP metrik	16.6	jaminan tiga bagian	MST, penelusuran *preorder*, ketaksamaan segitiga (Bab 10, 11, 13)	$\Theta(n^2)$, rasio 2
16.5 Bellman–Ford vs Dijkstra	16.7	perbandingan	relaksasi dan urutannya (Bab 13)	$\Theta(VE)$; $O(E\lg V)$
16.6 tabel Dijkstra	16.8.1	penelusuran	`Extract-Min` ditambah relaksasi (§13.7)	$O(E\lg V)$
16.7 siklus negatif	16.8.2	penelusuran dan deteksi	$\lvert V\rvert - 1$ putaran, pemeriksaan penutup, penelusuran π (§13.8)	$\Theta(VE)$
16.8 lintasan terpendek DAG	16.8.3	penelusuran	pengurutan topologis DFS ditambah relaksasi (§10, §13.9)	$\Theta(V + E)$
16.9 pencacah biner	16.9.1	amortisasi, empat cara	agregat, akuntansi, potensial, induksi (§14.3–14.6)	total $< 2n$
16.10 tabel dinamis	16.9.2	amortisasi, tiga cara	potensial $2\,num - size$ (§14.7)	total $< 3n$
16.11 Floyd–Warshall	16.10.1	penelusuran	rekurensi atas simpul antara (§15.6)	$\Theta(V^3)$
16.12 Johnson	16.10.2	penelusuran	pembobotan ulang dengan potensial (§15.8)	$O(V^2\lg V + VE)$
16.13 lima belas pernyataan	16.11.1	pengenalan di seluruh buku	Bab 9–15	—
16.14 lintasan terpanjang DAG	16.11.2	rancangan dan analisis	urutan topologis, relaksasi pemaksimum (§13.9)	$\Theta(V + E)$
16.15 sisi pohon	16.11.3	bukti induksi	membuang sebuah daun (Bab 10, 11)	—
16.16 Huffman	16.11.4	jaminan tiga bagian	argumen pertukaran, substruktur optimal (Bab 9)	$O(n\lg n)$
16.17 Floyd–Warshall vs Johnson	16.11.5	perbandingan	padat berhadapan dengan jarang (Bab 15)	$\Theta(V^3)$; $O(V^2\lg V + VE)$
16.18 lima belas pernyataan	16.11.6	pengenalan di seluruh buku	Bab 5–14	—

Tabel 16.18 (Lanjutan) *Kedua puluh satu persoalan bab ini sekilas: jenis masing-masing, alat yang menyelesaikannya dan di mana buku ini mengajarkannya, dan biaya penyelesaiannya.*

Persoalan	Subbab	Jenisnya	Alat utama, dan di mana ia diajarkan	Biaya penyelesaiannya
16.19 *knapsack* 0/1	16.11.7	rancangan dan analisis	pemrograman dinamis; waktu pseudo-polinomial (Bab 7)	$\Theta(nW)$
16.20 penutup simpul	16.11.8	jaminan tiga bagian	pemasangan menjadi batas bawah optimum	$O(V+E)$, rasio 2
16.21 Prim vs Kruskal	16.11.9	perbandingan	sifat potongan; *heap* berhadapan dengan *union–find* (Bab 11, 14)	$\Theta(V^2)$; $O(E\lg V)$

Apa yang sama pada semuanya.

1. **Setiap persoalan mempunyai pendekatan yang tampak jelas tetapi keliru atau terlalu lemah.** Menjawab pernyataan (11) menurut nama notasinya, menyelesaikan persoalan subbarisan yang bersebelahan, berinduksi atas tinggi, menyatakan bahwa pemintasan memendekkan, memercayai Dijkstra dengan sisi negatif, membatasi pencacah dengan nk, memilih potensial $num - size$, menambahkan konstanta pada setiap sisi alih-alih potensial. Sebagian besar nilai setiap penyelesaian terletak pada *menjelaskan mengapa* pendekatan yang tampak jelas itu gagal.
2. **Setiap jawaban diperiksa dua kali.** Sebuah bukti dan sebuah program; penelusuran tangan dan tabel yang dicetak; Floyd–Warshall berhadapan dengan Johnson; hampirannya berhadapan dengan gaya kasar; tiga metode amortisasi berhadapan satu sama lain.
3. **Beberapa gagasan yang sama berulang.** Relaksasi muncul pada tujuh persoalan (16.5 sampai 16.8, 16.12, 16.14, 16.17); hipotesis yang diperkuat (subpersoalan yang *berakhir di* $S[i]$, total pencacah *ditambah* bit-bit 1-nya) pada dua; batas bawah optimum pada dua; fungsi potensial — h milik Johnson dan Φ milik Bab 14 adalah gagasan yang sama dalam dua baju — pada tiga.

> **WAWASAN KUNCI: SATU GAGASAN, DUA BAB.** Potensial h milik Johnson dan potensial amortisasi Φ mengerjakan tugas yang sama. Masing-masing menambahkan pada setiap langkah sebuah koreksi yang *berteleskop* sepanjang barisan langkah mana pun — $h(u) - h(v)$ sepanjang sebuah lintasan, $\Phi(D_i) - \Phi(D_{i-1})$ sepanjang barisan operasi — sehingga totalnya hanya berubah sebesar suku batasnya. Bandingkan Persamaan (16.12) dengan Persamaan (16.7): pada keduanya, suku-suku di bagian dalam saling meniadakan dan hanya kedua ujungnya yang tersisa. Johnson memakai koreksi itu untuk membuat setiap sisi tak negatif; metode potensial memakainya untuk membuat setiap operasi murah. Begitu Anda melihat bahwa keduanya siasat yang sama, Anda akan menemukannya lagi dalam dual sebuah program linear (Subbab 16.15).

16.13 Latihan Uji Diri Singkat

Empat puluh pertanyaan dengan jawaban satu baris, merentang keenam belas bab. Tutuplah jawabannya, kerjakan daftarnya dari atas ke bawah, lalu periksalah diri Anda. Setiap yang meleset menunjuk bab yang perlu dibaca ulang. Dua atau tiga latihan per bab, berurutan menurut bab.

1. **Lima fase kerangka penyelesaian persoalan?** Memahami, mengabstraksi, merancang, menganalisis, mengimplementasikan. (Bab 1)
2. **Mengapa membaca kendala sebelum merancang?** Kendala adalah anggaran waktu: $n \leq 10^6$ menuntut $O(n)$ atau $O(n\lg n)$; $n \leq 500$ mengizinkan $O(n^3)$. (Bab 1, 4)
3. **Berapa biaya indeks larik dalam model RAM?** Satu satuan, seperti operasi aritmetika atau perbandingan mana pun. (Bab 2)
4. **Tiga kewajiban sebuah invarian gelung?** Inisialisasi, pemeliharaan, terminasi. (Bab 2)

5. **Kasus terbaik dan terburuk pengurutan sisip, dan masukan penyebab masing-masing?** $\Theta(n)$ pada larik terurut, $\Theta(n^2)$ pada larik terurut terbalik. (Bab 2)
6. **Definisikan** $f(n) = O(g(n))$**.** Terdapat $c > 0, n_0$ dengan $0 \leq f(n) \leq c\, g(n)$ untuk semua $n \geq n_0$. (Bab 3)
7. O **berhadapan dengan** Ω **dalam satu kalimat?** O membatasi dari atas, Ω dari bawah, Θ dari keduanya. (Bab 3, Subbab 16.3.10)
8. **Tiga kasus Teorema Master?** $f = O(n^{\log_b a - \varepsilon}) \Rightarrow \Theta(n^{\log_b a})$; $f = \Theta(n^{\log_b a}) \Rightarrow \Theta(n^{\log_b a} \lg n)$; $f = \Omega(n^{\log_b a + \varepsilon})$ dengan syarat keteraturan $\Rightarrow \Theta(f)$. (Bab 3)
9. **Selesaikan** $T(n) = 2T(n/2) + \Theta(n)$ **dan** $T(n) = T(n-1) + \Theta(n)$**.** $\Theta(n \lg n)$ dan $\Theta(n^2)$ — merge sort dan quicksort kasus terburuk. (Bab 3, 5)
10. **Bagaimana sebuah program sebaiknya diperiksa sebelum dipercaya?** Terhadap contohnya, lalu terhadap masukan sah terkecil dan terbesar, lalu terhadap gaya kasar pada kasus-kasus kecil acak. (Bab 4)
11. **Mengapa** `BUILD-MAX-HEAP` $\Theta(n)$**?** Sebagian besar simpul berada dekat dasar; $\sum_h \lceil n/2^{h+1} \rceil O(h) = O(n)$. (Bab 5)
12. **Waktu dan ruang tambahan heapsort?** Selalu $\Theta(n \lg n)$, tambahan $O(1)$ — di tempat. (Bab 5)
13. **Apa yang dibeli dengan mengacak poros quicksort?** Harapan $\Theta(n \lg n)$ pada setiap masukan; kasus terburuk $\Theta(n^2)$ tetap ada, tetapi tak ada lawan yang dapat memaksakannya. (Bab 5)
14. **Batas bawah pengurutan berbasis perbandingan, dan mengapa?** $\Omega(n \lg n)$: pohon keputusan dengan $n!$ daun bertinggi $\geq \lg(n!) = \Theta(n \lg n)$. (Bab 5)
15. **Mengapa penelusuran *inorder* BST terurut?** Subpohon kiri $<$ akar $<$ subpohon kanan, jadi kiri–akar–kanan mengeluarkan kunci yang menaik; induksi. (Bab 6)
16. **Batas tinggi AVL, dan dari mana asalnya?** $h \leq 1.4405 \lg(n+2) - 0.3277$, dari $N(h) = 1 + N(h-1) + N(h-2)$. (Bab 6)
17. **Berapa rotasi yang dapat ditimbulkan satu penyisipan AVL? Satu penghapusan?** Paling banyak dua; sampai $\Theta(\lg n)$. (Bab 6)
18. **Apa yang membuat sebuah persoalan cocok untuk pemrograman dinamis alih-alih bagi-dan-kuasai?** Subpersoalan yang bertumpang tindih ditambah substruktur optimal. (Bab 7)
19. **Memoisasi berhadapan dengan tabulasi, dalam satu baris?** Keduanya menyimpan setiap subpersoalan sekali; memoisasi mempertahankan rekursinya dan mengisi sesuai kebutuhan, tabulasi menghapusnya dan mengisi menurut urutan ketergantungan. (Bab 7)
20. **Empat langkah merancang program dinamis?** Cirikan struktur optimalnya, tuliskan rekurensinya, hitung dari bawah ke atas, susun kembali dari pilihan-pilihan yang dicatat. (Bab 7)
21. **Apa yang membuat bukti yang belum selesai tetap berguna?** Strukturnya: klaimnya, tekniknya, kasus dasarnya, hipotesis yang tepat, langkah-langkah yang sudah dibenarkan, dan label yang jujur pada celahnya. (Bab 8, Subbab 16.2.2)
22. **Dua hipotesis yang dibutuhkan bukti rakus?** Sifat pilihan rakus dan substruktur optimal. (Bab 9)
23. **Mengapa rakus menyelesaikan *knapsack* pecahan tetapi tidak 0/1?** Pecahan memungkinkan Anda mengisi kapasitas tepat dengan rasio terbaik; 0/1 tidak dapat membelah butir, jadi pilihan terbaik secara lokal dapat memboroskan kapasitas. (Bab 9)

24. **Dalam satu baris, mengapa Huffman optimal?** Dua simbol berfrekuensi terendah dapat dijadikan saudara terdalam tanpa kerugian (pilihan rakus), dan menggabungkan keduanya menyisakan contoh yang lebih kecil dari persoalan yang sama (substruktur optimal). (Bab 9, Persoalan 16.16)
25. **BFS dan DFS berbeda dalam apa, persisnya?** Wadahnya: antrean FIFO memberi urutan bersisi paling sedikit; tumpukan, tersurat maupun rekursif, memberi urutan mendalam. (Bab 10)
26. **Apa yang diberikan DFS dan tidak diberikan BFS?** Waktu penemuan dan waktu selesai, dan karena itu sisi balik, deteksi siklus, pengurutan topologis, dan komponen terhubung kuat. (Bab 10, Persoalan 16.8)
27. **Senarai ketetanggaan atau matriks ketetanggaan?** Senarai untuk graf jarang (ruang $\Theta(V + E)$, $\Theta(\deg u)$ untuk memindai u); matriks untuk graf padat atau ketika "apakah (u, v) sebuah sisi?" harus $\Theta(1)$. (Bab 10)
28. **Nyatakan sifat potongan.** Untuk setiap potongan yang dihormati hutan saat ini, sisi teringan yang menyeberanginya aman untuk ditambahkan. (Bab 11)
29. **Prim atau Kruskal?** Prim pada graf padat ($\Theta(V^2)$ dengan larik); Kruskal pada graf jarang, atau ketika sisi-sisinya sudah terurut. (Bab 11)
30. **Apakah pengurutan topologis ada untuk setiap graf berarah?** Tidak — tepat untuk yang taksiklik; siklus sepanjang apa pun melarangnya. (Bab 10, 11)
31. **Apa yang dibutuhkan evaluasi rekayasa atas algoritma yang diimplementasikan?** Sebaran masukan yang dinyatakan, waktu yang diukur, perbandingan terhadap garis dasar, dan kesimpulan yang menyebut pertukarannya. (Bab 12)
32. **Apa itu relaksasi?** `if d[v] > d[u] + w(u,v) then d[v] = d[u] + w(u,v)` — ia hanya dapat menurunkan taksiran. (Bab 13, Persamaan (16.1))
33. **Mengapa Dijkstra gagal pada sisi negatif?** Ia menuntaskan simpul ketika mengeluarkannya, dan sisi negatif yang datang belakangan dapat memperbaikinya. (Bab 13, Subbab 16.7.7)
34. **Mengapa $|V| - 1$ putaran pada Bellman–Ford, dan apa gunanya satu putaran lagi?** Lintasan terpendek tanpa siklus negatif mempunyai paling banyak $|V| - 1$ sisi, dan putaran ke-i membuat setiap lintasan i-sisi benar; putaran tambahan yang masih merelaksasi sebuah sisi membuktikan adanya siklus negatif. (Bab 13, Persoalan 16.7)
35. **Tiga teknik amortisasi?** Agregat (total atas n operasi, dibagi n), akuntansi (tarifkan biaya dan jaga kredit tetap tak negatif), potensial (definisikan Φ dan pakailah Persamaan (16.6)). (Bab 14)
36. **Mengapa tabel dinamis yang berlipat dua $O(1)$ amortisasi per penyisipan?** n penyisipan berbiaya $n + \sum_j 2^j < 3n$. (Bab 14, Persoalan 16.10)
37. **Syarat apa yang harus dipenuhi fungsi potensial?** $\Phi(D_i) \geq \Phi(D_0)$ untuk semua i; biasanya diambil $\Phi(D_0) = 0$ dan $\Phi \geq 0$. (Bab 14, Persoalan 16.10(f))
38. **Rekurensi Floyd–Warshall, dengan kata-kata?** Lintasan $i \to j$ terpendek yang hanya memakai $\{1..k\}$ sebagai simpul antara entah menghindari k, entah berjalan $i \rightsquigarrow k \rightsquigarrow j$. (Bab 15, Persamaan (16.10))
39. **Mengapa pembobotan ulang Johnson mempertahankan lintasan terpendek?** Potensialnya berteleskop: setiap lintasan $u \rightsquigarrow v$ tergeser dengan $h(u) - h(v)$ yang sama. (Bab 15, Persamaan (16.12))
40. **Satu pertanyaan yang harus diajukan sebelum menulis algoritma apa pun?** Apa bentuk masukannya, dan apa bentuk jawabannya — sebab keduanya bersama-sama menamai algoritmanya. (Bab 16, Subbab 16.14)

16.14 Tabel Pemilihan Algoritma

Inilah halaman yang layak disimpan. Setiap barisnya adalah sebuah *bentuk* — cara sebuah persoalan tampak ketika Anda pertama kali membacanya — dan algoritma yang dituntut bentuk itu, berikut alasan dan biayanya. Tabel ini sengaja disusun menurut bentuk alih-alih menurut algoritma, sebab pada saat Anda membutuhkannya, Anda mengenal persoalannya dan belum mengenal jawabannya.

Tabel 16.19 Tabel pemilihan algoritma. Bacalah kolom kiri sampai sebuah baris cocok dengan persoalan di depan Anda; tiga kolom lainnya adalah jawabannya, pembenaran yang akan diminta dari Anda, dan anggaran yang harus Anda penuhi.

Persoalannya tampak seperti ini	Pakailah	Mengapa	Biaya
Mengurutkan n butir, memori sempit	Heapsort	Di tempat, dan $\Theta(n\lg n)$ tanpa kasus buruk	$\Theta(n\lg n)$, ruang $O(1)$
Mengurutkan n butir, kecepatan rata-rata paling penting	Quicksort teracak	Konstanta kecil, lokalitas baik	Harapan $\Theta(n\lg n)$
Mengurutkan n butir, kunci yang sama harus mempertahankan urutannya	Merge sort	Satu-satunya yang stabil di antara ketiganya	$\Theta(n\lg n)$, ruang $\Theta(n)$
Mengurutkan n bilangan bulat kecil dalam rentang yang diketahui	*Counting sort* atau *radix sort*	Bukan pengurutan berbasis perbandingan, jadi lantai $\Omega(n\lg n)$ tidak berlaku	$\Theta(n+k)$
Mencari dalam larik terurut yang statis	Pencarian biner	Pembagian dua hanya sahih karena lariknya terurut	$\Theta(\lg n)$
Mencari dengan penyisipan dan penghapusan, penelusuran terurut dibutuhkan	Pohon AVL atau merah-hitam	Mempertahankan urutan *dan* pembaruan $O(\lg n)$	$O(\lg n)$
Mencari dengan penyisipan dan penghapusan, urutan tidak pernah dibutuhkan	Tabel *hash*	Harapan $\Theta(1)$; tanpa urutan	Harapan $\Theta(1)$
Berulang kali mengambil yang terkecil atau terbesar	*Binary heap*	`Extract-Min` dalam $\Theta(\lg n)$, `Build` dalam $\Theta(n)$	$\Theta(\lg n)$ per operasi
Memilih sebanyak mungkin butir yang tidak bertumpang tindih	Rakus menurut waktu selesai paling awal	Sifat pilihan rakus berlaku; dibuktikan di Bab 9	$\Theta(n\lg n)$
Mengisi kapasitas dengan barang yang dapat dibagi	Rakus menurut kerapatan nilai	*Knapsack* pecahan; argumen pertukaran	$\Theta(n\lg n)$
Mengisi kapasitas dengan barang yang tidak dapat dibagi	Pemrograman dinamis atas kapasitas	Rakus gagal; rekurensi $c[i][j]$	$\Theta(nW)$, pseudo-polinomial
Memampatkan simbol dengan frekuensi yang diketahui	Huffman	Kode awalan optimal	$O(n\lg n)$
Subpersoalan bertumpang tindih, jawaban dibangun dari potongan optimal	Pemrograman dinamis	Ingat, jangan hitung ulang	bergantung persoalan
Subpersoalan saling bebas	Bagi dan kuasai	Tak ada yang perlu diingat	Teorema Master

Tabel 16.19 (Lanjutan) *Tabel pemilihan algoritma. Bacalah kolom kiri sampai sebuah baris cocok dengan persoalan di depan Anda; tiga kolom lainnya adalah jawabannya, pembenaran yang akan diminta dari Anda, dan anggaran yang harus Anda penuhi.*

Persoalannya tampak seperti ini	Pakailah	Mengapa	Biaya
Menjangkau setiap simpul, sisi paling sedikit dari satu sumber	BFS	Antreannya memberi jarak yang tak menurun	$\Theta(V+E)$
Mendeteksi siklus, mengurutkan tugas, menemukan komponen	DFS	Waktu penemuan/selesai menyingkap struktur	$\Theta(V+E)$
Mengurutkan tugas yang berprasyarat	Pengurutan topologis	Ada tepat untuk DAG	$\Theta(V+E)$
Menyambungkan segalanya dengan biaya terkecil	Prim (padat) atau Kruskal (jarang)	Sifat potongan membuat sisi rakusnya aman	$\Theta(V^2)$ / $O(E\lg V)$
Lintasan terpendek, satu sumber, bobot ≥ 0	Dijkstra	Rakus aman ketika memperpanjang tidak dapat memendekkan	$O(E\lg V)$
Lintasan terpendek, satu sumber, sebagian bobot negatif	Bellman–Ford	Merelaksasi semuanya $\lvert V\rvert - 1$ kali; mendeteksi siklus negatif	$\Theta(VE)$
Lintasan terpendek, satu sumber, grafnya DAG	Relaksasi dalam urutan topologis	Satu putaran cukup; juga menyelesaikan lintasan *terpanjang*	$\Theta(V+E)$
Lintasan terpendek antara setiap pasangan, graf padat	Floyd–Warshall	Tiga gelung, ruang $\Theta(V^2)$	$\Theta(V^3)$
Lintasan terpendek antara setiap pasangan, graf jarang	Johnson	Bobot ulang sekali, lalu V kali Dijkstra	$O(V^2\lg V + VE)$
Barisan operasi, salah satunya sesekali mahal	Analisis amortisasi	Langkah terburuk bukan langkah rata-rata	—
Jawaban eksaknya NP-sulit dan n besar	Hampiran dengan rasio yang dibuktikan	Jaminan mengalahkan heuristik	bergantung persoalan
Sesuatu harus dipelihara selagi butir-butir berdatangan	Hutan himpunan-lepas, *heap*, atau pohon seimbang	Pilih menurut operasi yang paling sering Anda kerjakan	$O(\alpha(n))$ / $O(\lg n)$

Dua aturan untuk memakai **Tabel 16.19** dengan jujur.

Pertama, bentuknya berasal dari masukan dan keluaran yang diminta, bukan dari kata-kata persoalannya. "Temukan cara termurah menyambungkan semua desa" dan "rangkailah sebuah cip dengan tembaga sesedikit mungkin" adalah baris yang sama. "Temukan rute terpendek yang mengunjungi setiap desa sekali" tampak mirip dan merupakan baris yang sama sekali lain — baris kedua dari bawah, yang NP-sulit.

Kedua, sebuah baris adalah titik berangkat, bukan vonis. Kolom biayanya asimtotik, dan pernyataan (12) Persoalan 16.1 adalah keberatan abadi terhadap membacanya sebagai kebenaran mutlak. Ukurlah sebelum Anda berkomitmen.

16.15 Ke Mana Melangkah Selanjutnya

Buku ini mencakup bahan setara satu semester, dan pokok bahasannya jauh lebih luas. Enam wilayah besar ditinggalkan, dan maksud subbab ini adalah bahwa Anda kini cukup berbekal untuk memasuki setiap satunya. Setiap paragraf menyebutkan apa wilayah itu, mengapa ia

berada di luar cakupan, apa yang sudah Anda miliki yang akan membawa Anda masuk, dan satu buku untuk memulai.

Aliran jaringan (*network flow*). Graf berarah yang sisi-sisinya mempunyai *kapasitas*, sebuah sumber dan sebuah muara; pertanyaannya adalah berapa banyak yang dapat didorong dari yang satu ke yang lain. Metode Ford–Fulkerson menemukan aliran maksimum dengan berulang kali mendorong sepanjang lintasan penambah (*augmenting path*) dalam graf residu; teorema aliran-maks potongan-min mengatakan jawabannya sama dengan kapasitas potongan teringan yang memisahkan sumber dari muara, salah satu dualitas terindah dalam pokok bahasan ini. Ia ditinggalkan karena membutuhkan satu bab penuh tentang graf residu sebelum apa pun dapat dibuktikan. Semua hal lainnya sudah Anda miliki: BFS menemukan lintasan penambahnya (itulah Edmonds–Karp, dan BFS-lah alasan ia berhenti dalam $O(VE^2)$), dan kosakata potongannya milik Bab 11. Pemasangan bipartit, pemilihan proyek, segmentasi citra, dan batas penutup simpul Persoalan 16.20 semuanya persoalan aliran yang menyamar. Mulailah dengan [5, ch. 7], yang mengembangkan aliran dari nol lalu menghabiskan seratus halaman untuk persoalan-persoalan yang tereduksi kepadanya; lalu [1, ch. 24].

NP-kelengkapan dan hampiran. Persoalan 16.4 membutuhkan hampiran-2 karena persoalan eksaknya tak tertangani (*intractable*), tetapi ia tidak pernah mengatakan apa arti *tak tertangani*. Teorinya: sebuah persoalan berada di NP jika penyelesaian yang diusulkan dapat diperiksa dalam waktu polinomial; ia NP-lengkap jika setiap persoalan di NP tereduksi kepadanya dalam waktu polinomial; dan tak seorang pun tahu apakah P = NP, walaupun hampir semua orang percaya tidak. Pedagang keliling, *knapsack* 0/1, penutup simpul, siklus Hamilton, pewarnaan graf, dan keterpenuhan boolean semuanya NP-lengkap, dan Anda telah berjumpa empat dari keenamnya dalam buku ini. Keterampilan yang harus diperoleh adalah **reduksi**: menunjukkan bahwa persoalan baru paling sedikit sesulit persoalan yang sudah dikenal — Subbab 16.6.6 mengerjakan satu, dari siklus Hamilton ke TSP umum. Lalu datang separuh yang konstruktif — algoritma hampiran, dengan batas rasio yang dibuktikan seperti yang Anda tulis di Persoalan 16.4 dan 16.20. Katalog klasiknya adalah [6]; perlakuan modern atas sisi hampirannya adalah [7]; [1, ch. 34–35] adalah versi ringkasnya.

Pencocokan string (*string matching*). Diberikan teks berpanjang n dan pola berpanjang m, temukan setiap kemunculannya. Metode naifnya $\Theta(nm)$; Knuth–Morris–Pratt [8] mencapai $\Theta(n + m)$ dengan menghitung di muka, untuk setiap awalan pola, panjang awalan sejati terpanjangnya yang juga akhiran — sehingga ketidakcocokan tidak pernah mengirim penunjuk teks mundur. Rabin–Karp mengerjakannya dengan *hash* bergulir; Boyer–Moore biasanya tercepat dalam praktik dengan memindai polanya dari belakang. Di balik pencocokan terbentang pohon akhiran dan larik akhiran, yang menjawab "di mana substring ini muncul?" untuk substring kueri *apa pun* sesudah prapemrosesan linear. Kedua prasyaratnya sudah Anda miliki: gagasan otomatonnya adalah mesin keadaan atas pola, dan analisisnya amortisasi — bukti KMP adalah argumen agregat yang persis berbentuk pencacah biner Persoalan 16.9. Rujukan bakunya, yang kedalamannya masih belum tertandingi, adalah [9]; [1, ch. 32] adalah pengantarnya.

Geometri komputasional. Titik, ruas garis, poligon: apakah ruas ini memotong ruas itu, apa selubung cembung titik-titik ini, pasangan mana yang terdekat, titik mana yang paling dekat ke sebuah kueri. Teknik-tekniknya adalah yang sudah Anda miliki — bagi dan kuasai memberi pasangan terdekat dalam $\Theta(n\lg n)$ dan merupakan penerapan buku teks Teorema Master; *Graham's scan* mengurutkan menurut sudut lalu membuat satu lintasan dengan tumpukan, yaitu algoritma rakus dengan analisis amortisasi berbasis tumpukan; algoritma garis sapu adalah antrean prioritas ditambah pohon pencarian seimbang, yaitu Bab 5 dan 6 digabungkan. Isi

barunya adalah *predikat*-nya: uji orientasi, dan kecermatan titik-mengambang yang dituntutnya. Buku bakunya adalah [10].

Pemrograman linear. Maksimumkan fungsi linear peubah-peubah riil dengan kendala ketaksamaan linear. Ia mencakup banyak sekali hal: lintasan terpendek, aliran maksimum, dan persoalan penugasan semuanya program linear, dan dual-dualnya adalah argumen potongan-min dan potensial yang sudah Anda pakai — potensial h Johnson pada Persoalan 16.12 adalah penyelesaian layak bagi dual program linear lintasan terpendek, dan itulah sebabnya kotak wawasan di Subbab 16.12 menemukan siasat teleskopik yang sama di dua bab. Metode simpleks eksponensial pada kasus terburuk dan sangat baik dalam praktik (pernyataan (12) Persoalan 16.1, dalam bentuknya yang paling murni); metode titik-interior polinomial. Ketika peubah-peubahnya diharuskan bulat, persoalannya menjadi NP-sulit, dan *relaksasi LP disusul pembulatan* adalah salah satu sumber utama algoritma hampiran dengan rasio yang dibuktikan. Pintu masuk yang pendek dan mudah dibaca adalah [11]; [1, ch. 29] memberikan inti algoritmiknya.

Algoritma teracak dan daring. Dua gagasan terpisah yang berdampingan karena keduanya mengganti "kasus terburuk atas masukan" dengan sesuatu yang lebih berguna. Algoritma **teracak** melempar koin: quicksort teracak di Bab 5 adalah yang pertama Anda jumpai, dan pelajarannya berlaku umum — pengacakan mengubah "masukan buruk" menjadi "nasib buruk", yang tidak dapat diatur oleh lawan. Di baliknya ada seleksi teracak, *treap* dan *skip list*, analisis *hashing*, potongan-min Karger, dan metode Monte Carlo dengan galat satu sisi. Algoritma **daring** harus menjawab sebelum seluruh masukannya tiba — *cache* yang memutuskan apa yang dikeluarkan, server yang memutuskan lift mana yang dikirim — dan dinilai menurut *rasio kompetitif*-nya terhadap optimum luring yang mengetahui masa depan. Rasio itu adalah objek sejenis dengan $\rho(n) = 2$ pada Persoalan 16.4, dan itulah sebabnya kedua pokok bahasan itu berdampingan. Mulailah dengan [12] untuk pengacakan dan [13] untuk teori daring.

Catatan ketujuh, bukan sebuah wilayah: **kebiasaan mengukur.** Setiap pernyataan asimtotik dalam buku ini adalah klaim tentang n yang besar, dan setiap satunya dapat diperiksa di mesin Anda sendiri dalam sepuluh menit. [14] adalah buku tunggal terbaik tentang praktik memilih dan menyetel algoritma, dan katalog persoalannya — "saya punya ini, saya ingin itu, inilah yang diketahui" — adalah versi profesional **Tabel 16.19**.

16.16 Profitina dalam Praktik

Pokok bahasan bab ini adalah *memilih*, dan memilih adalah tempat sebuah tim rekayasa menghabiskan waktu algoritmanya. Profitina — platform visibilitas AI (GEO, *Generative Engine Optimization*) yang beroperasi di `app.profitina.com` — adalah tempat yang berguna untuk menyaksikannya. Ia memindai situs web sebuah merek dan menilai seberapa siap situs itu untuk *dikutip oleh mesin jawab AI*, dari sinyal-sinyal nyata seperti keterambilan konten, data terstruktur, dan kredibilitas teknis, lalu membantu memperbaiki apa yang ditemukannya. Hampir setiap baris **Tabel 16.19** muncul di suatu tempat dalam produk ini, dan dalam setiap kasus alasan pilihannya adalah alasan yang diberikan buku ini.

Antrean perayapan adalah antrean prioritas. Memindai sebuah situs berarti mengambil halaman-halamannya, dan halaman-halaman itu tidak setara: beranda dan halaman yang paling banyak ditaut layak diambil lebih dahulu, sebab skor yang berguna dapat dilaporkan sebelum perayapan selesai. Itulah `Extract-Min` pada *heap* yang dikunci dengan taksiran prioritas — baris Bab 5 dalam tabel itu, dipilih karena operasi yang paling sering dikerjakan adalah "berikan saya halaman belum-diambil yang paling bernilai".

Situs adalah graf, dan kedalaman adalah jarak lintasan terpendek. Perayapan dimulai dari satu URL lalu mengikuti tautan. Berapa klik sebuah halaman terletak dari beranda adalah jarak lintasan terpendeknya dalam graf tak berbobot, dan tepat itulah yang dihitung pencarian melebar — kasus merosot Dijkstra ketika setiap sisi berbobot 1 dan antrean prioritasnya menjadi antrean biasa. Urutan BFS sengaja dipilih alih-alih mendalam: perayapan yang terpotong oleh anggaran lalu sudah mencakup halaman-halaman yang *dangkal*, yang paling mungkin dijangkau mesin jawab. Itulah latihan 25 di Subbab 16.13, dipakai sebagai keputusan produk.

Irama pemindaian ulang adalah persoalan penjadwalan. Setiap merek mempunyai tingkat langganan yang menentukan seberapa sering ia dipindai ulang secara otomatis, dan penjadwalnya harus memuatkan pemindaian hari itu ke dalam kapasitas yang tersedia, dengan peringatan ketika sebuah skor bergerak. Menjaga pemindaian yang jatuh tempo berikutnya tetap di depan adalah antrean prioritas lagi, dikunci menurut waktu; memilih pemindaian mana yang muat dalam satu jam yang sibuk adalah baris pemilihan-kegiatan dalam tabel itu.

Menghapus duplikasi halaman yang nyaris sama adalah *hashing* ditambah pencocokan string. Banyak situs menyajikan konten yang sama di beberapa URL. Mengkanonikkan URL adalah persoalan tabel *hash*; mendeteksi bahwa dua URL yang *berbeda* mengembalikan teks yang pada dasarnya sama adalah persoalan string — salah satu tempat wilayah pencocokan string di Subbab 16.15 membuktikan gunanya.

Tren skor mingguan adalah struktur amortisasi. Dasbor analitik historis menyimpan tren skor mingguan per merek, yang dapat diekspor sebagai CSV, dan riwayat itu bertambah satu baris per merek per pemindaian, selamanya. Tabel yang menampungnya dan berlipat dua ketika penuh membayar tepat harga yang dihitung pada Persoalan 16.10: $O(1)$ amortisasi per penyisipan, dengan sesekali penyalinan mahal yang tidak boleh jatuh di tengah permintaan seorang pengguna. Persoalan 16.10(f) adalah peringatan yang menyertainya — analisis yang "membuktikan" batas yang lebih baik dengan membiarkan potensialnya menjadi negatif sedang meminjam dari masa depan, dan masa depan selalu mengirim tagihannya.

Memeriksa dua kali adalah gerbang rilis. Setiap jawaban dalam bab ini dikukuhkan dengan cara kedua: Floyd–Warshall berhadapan dengan Johnson, hampiran berhadapan dengan gaya kasar, tiga buku besar amortisasi berhadapan satu sama lain. Profitina menerapkan disiplin yang sama dalam skala besar: seluruh rangkaian uji otomatisnya, jauh lebih dari seribu uji, harus lulus sebelum rilis mana pun, dan sebuah rilis baru melaporkan keberhasilan sesudah versi yang dipasang menjawab pemeriksaan kesehatannya.

Satu aturan mengatur cara semua ini dipakai: platform itu *menyarankan*; pelanggannya yang *memutuskan* — skor adalah taksiran dari sinyal-sinyal yang diukur, tidak pernah jaminan untuk dikutip.

16.16.1 Studi kasus terkerjakan: Profitina Laundry

Profitina Laundry adalah usaha pengajaran buku ini — sebuah binatu lingkungan dengan satu mesin setrika, satu mobil boks, dan enam titik jemput, bukan produk yang sesungguhnya. Pada malam terakhir buku ini ia mempunyai tiga pertanyaan, dan seluruh intinya adalah bahwa ketiganya pertanyaan yang *berbeda* sehingga menuntut tiga algoritma yang berbeda.

- *Pekerjaan setrika mana yang dapat diselesaikan satu mesin hari ini?* — sekumpulan selang waktu, pilih sebanyak mungkin yang tidak bertumpang tindih: **pemilihan kegiatan secara rakus** (Bab 9).

- *Dalam urutan apa mobil boksnya menjemput?* — sebuah metrik pada enam titik, temukan tur tertutup yang pendek: **pohon merentang minimum ditambah penelusuran *preorder***, dengan jaminan Persoalan 16.4 (Bab 11).
- *Seberapa jauh setiap lingkungan lewat jalan?* — graf berbobot dengan bobot tak negatif dan satu sumber: **Dijkstra** (Bab 13).

Kode 16.19 menjawab ketiganya dari satu himpunan data.

Kode 16.19 Perencana malam Profitina Laundry: tiga pertanyaan dari satu himpunan data, dijawab oleh algoritma rakus, pohon merentang minimum dengan penelusuran *preorder*, dan Dijkstra — satu baris **Tabel 16.19** masing-masing.

```
// Profitina Laundry: perencana malam hari. Satu himpunan data, tiga pertanyaan,
// tiga algoritma berbeda -- dipilih menurut bentuk tiap pertanyaan, bukan kebiasaan.
//   Q1  pekerjaan setrika mana yang muat di satu mesin?  pemilihan aktivitas rakus (Bab 9)
//   Q2  dengan urutan apa vannya menjemput?              MST + penelusuran preorder (Bab
11, 16.6)
//   Q3  berapa jauh tiap titik antar lewat jalan?        Dijkstra (Bab 13)
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>
using namespace std;

struct Job { string id; int s, f; };

const int N = 6;
const string pt[N] = {"Depot","Keputih","Gebang","Semolowaru","Manyar","Klampis"};
const int  PX[N] = {0, 4, 8, 6, 2, 0};
const int  PY[N] = {0, 0, 1, 5, 6, 3};

int dist2[N][N];                                    // metrik garis lurus (Manhattan)
vector<int> kid[N];
vector<int> tourOrder;

void preorder(int u) { tourOrder.push_back(u); for (int v : kid[u]) preorder(v); }

string hhmm(int m) {                                // menit sesudah pukul 08:00
    char b[16]; snprintf(b, sizeof b, "%02d:%02d", 8 + m / 60, m % 60); return b;
}

int main() {
    // ---------- Q1: pemilihan aktivitas secara rakus ----------
    vector<Job> job = {{"J1",0,60},{"J2",30,90},{"J3",60,110},{"J4",45,150},
                       {"J5",115,165},{"J6",150,210},{"J7",100,140},{"J8",170,240}};
    sort(job.begin(), job.end(), [](const Job& a, const Job& b) { return a.f < b.f; });
```

Kode 16.19 (Lanjutan) *Perencana malam Profitina Laundry: tiga pertanyaan dari satu himpunan data, dijawab oleh algoritma rakus, pohon merentang minimum dengan penelusuran preorder, dan Dijkstra — satu baris Tabel 16.19 masing-masing.*

```
    cout << "Q1  one press machine, jobs sorted by finishing time\n";
    int last = -1, chosen = 0;
    for (auto& j : job) {
        bool take = j.s >= last;
        cout << "    " << j.id << "  " << hhmm(j.s) << "-" << hhmm(j.f)
             << (take ? "   TAKE" : "   skip (overlaps)") << "\n";
        if (take) { last = j.f; ++chosen; }
    }
    cout << "    jobs scheduled = " << chosen << "\n\n";

    // ---------- Q2: pohon merentang minimum, lalu tur hampiran 2 ----------
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            dist2[i][j] = abs(PX[i] - PX[j]) + abs(PY[i] - PY[j]);

    vector<int> key(N, INT_MAX), par(N, -1);
    vector<bool> in(N, false);
    key[0] = 0;
    int mst = 0;
    for (int it = 0; it < N; ++it) {
        int u = -1;
        for (int v = 0; v < N; ++v) if (!in[v] && (u == -1 || key[v] < key[u])) u = v;
        in[u] = true; mst += key[u];
        for (int v = 0; v < N; ++v)
            if (!in[v] && dist2[u][v] < key[v]) { key[v] = dist2[u][v]; par[v] = u; }
    }
    for (int v = 0; v < N; ++v) if (par[v] != -1) kid[par[v]].push_back(v);
    for (int v = 0; v < N; ++v) sort(kid[v].begin(), kid[v].end());
    preorder(0);

    int tour = 0;
    for (int i = 0; i < N; ++i) tour += dist2[tourOrder[i]][tourOrder[(i + 1) % N]];
    cout << "Q2  collection route (MST weight " << mst
         << ", so no tour is shorter than " << mst << ")\n    ";
    for (int i = 0; i < N; ++i) cout << pt[tourOrder[i]] << (i + 1 < N ? " -> " : " -> ");
    cout << pt[0] << "\n    full walk costs 2 x MST = " << 2 * mst
         << "; dropping repeats gives " << tour
         << " blocks, and the theorem of 16.6 guarantees\n"
         << "    this is at most twice the unknown optimum\n\n";

    // ---------- Q3: jarak jalan terpendek dari depot ----------
    vector<vector<pair<int,int>>> road(N);
    auto link = [&](int a, int b, int w) { road[a].push_back({b, w}); road[b].push_back({a,
w}); };
    link(0,1,4); link(0,5,3); link(1,2,5); link(2,3,6); link(3,4,5);
    link(4,5,5); link(1,3,7); link(1,5,7); link(0,4,8);
```

Kode 16.19 (Lanjutan) *Perencana malam Profitina Laundry: tiga pertanyaan dari satu himpunan data, dijawab oleh algoritma rakus, pohon merentang minimum dengan penelusuran preorder, dan Dijkstra — satu baris Tabel 16.19 masing-masing.*

```
    vector<int> d(N, INT_MAX);
    d[0] = 0;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    Q.push({0, 0});
    while (!Q.empty()) {
        auto [du, u] = Q.top(); Q.pop();
        if (du > d[u]) continue;                    // entri antrean yang basi
        for (auto [v, w] : road[u])
            if (d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
    }
    cout << "Q3  shortest road distance from the depot\n";
    vector<int> ord(N); for (int i = 0; i < N; ++i) ord[i] = i;
    sort(ord.begin(), ord.end(), [&](int a, int b) { return d[a] < d[b]; });
    for (int v : ord)
        cout << "    " << setw(12) << left << pt[v] << right << setw(4) << d[v] << "
blocks\n";
    return 0;
}
```

Dikompilasi dengan `g++ -std=c++17 -O2 -Wall` lalu dijalankan:

```
Q1  one press machine, jobs sorted by finishing time
    J1  08:00-09:00   TAKE
    J2  08:30-09:30   skip (overlaps)
    J3  09:00-09:50   TAKE
    J7  09:40-10:20   skip (overlaps)
    J4  08:45-10:30   skip (overlaps)
    J5  09:55-10:45   TAKE
    J6  10:30-11:30   skip (overlaps)
    J8  10:50-12:00   TAKE
    jobs scheduled = 4

Q2  collection route (MST weight 22, so no tour is shorter than 22)
    Depot -> Keputih -> Gebang -> Klampis -> Manyar -> Semolowaru -> Depot
    full walk costs 2 x MST = 44; dropping repeats gives 40 blocks, and the theorem of 16.6
guarantees
    this is at most twice the unknown optimum

Q3  shortest road distance from the depot
    Depot          0 blocks
    Klampis        3 blocks
    Keputih        4 blocks
    Manyar         8 blocks
    Gebang         9 blocks
    Semolowaru    11 blocks
```

Perhatikan apa yang dilakukan jadwal rakus itu terhadap J4: ia berjalan dari 08:45 sampai 10:30, yang panjang, dan mengambilnya akan menghalangi J3 maupun J5. Mengurutkan menurut waktu *selesai* alih-alih waktu mulai atau durasi itulah yang membuat algoritmanya melewatinya, dan bukti pilihan rakus di Bab 9-lah yang membuat pelewatan itu terbukti benar, bukan sekadar masuk akal. Empat pekerjaan itu optimal, bukan sekadar baik. Dan lihatlah jarak-jarak dari depo pada Q3: 0, 3, 4, 8, 9, 11 blok — persis jarak Manhattan kisi Q2 untuk keenam titik yang sama. Itu kebetulan jaringan jalan ini, bukan hukum: Q2 dihitung dari koordinat dan Q3 dari sembilan ruas jalan, yang merupakan dua graf yang berbeda. Tutuplah jalan dari depo ke Klampis, dan Q3 berubah sedangkan Q2 tidak. Dua pertanyaan tentang "jarak" adalah pertanyaan tentang dua graf yang berbeda, dan masing-masing mendapat algoritmanya sendiri.

16.17 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

1. **Menjawab pernyataan yang diduga alih-alih pernyataan yang tertulis.** Garis bawahilah kata kerja dan kata pembatasnya — *pencarian linear*, *paling sedikit*, *selalu*, *tidak mungkin* — sebelum memutuskan (Subbab 16.3.12).
2. **Memberikan batas O tanpa menyebut kasusnya.** "Quicksort adalah $O(n^2)$" benar untuk kasus terburuk dan menyesatkan di tempat lain. Sebutkan *terburuk*, *rata-rata*, atau *harapan*, setiap kali.
3. **Mencampuradukkan subbarisan dengan substring.** Subbarisan mempertahankan urutan tetapi tidak kebersebelahan. Nyatakan ini dalam asumsi Anda sebelum merancang apa pun (Subbab 16.4.10).
4. **Menghitung nilai program dinamis tetapi tidak penyelesaiannya.** Simpanlah larik pendahulunya dan telusuri balik. "Temukan subbarisan menaik terpanjang" berarti hasilkanlah subbarisan itu.
5. **Memakai induksi lemah di tempat yang membutuhkan induksi kuat.** Jika langkah induksinya membelah objek berukuran n menjadi potongan berukuran k dan $n-1-k$, hipotesisnya harus mencakup *setiap* ukuran yang lebih kecil (Subbab 16.5.8).
6. **Menyatakan sebuah langkah yang membutuhkan hipotesis.** "Pemintasan membuat penelusurannya lebih pendek" hanya benar di bawah ketaksamaan segitiga. Sebutkan hipotesisnya di tempat Anda memakainya (Subbab 16.6.10).
7. **Membuktikan batas rasio tanpa batas bawah untuk optimumnya.** Setiap bukti semacam itu berkerangka sama: temukan sesuatu yang paling banyak sebesar optimumnya ($d(T)$, $|A|$), lalu tunjukkan bahwa jawabannya paling banyak dua kali besaran itu.
8. **Memperlakukan Dijkstra sebagai universal.** Ia rakus, dan rakus aman di sini hanya karena memperpanjang lintasan tidak dapat memendekkannya. Satu sisi negatif dan penuntasannya keliru (Subbab 16.7.7). Dan jawaban yang benar pada satu graf bersisi negatif, seperti pada Persoalan 16.8, tidak membuktikan apa-apa.
9. **Mengeluarkan simpul yang keliru dalam penelusuran Dijkstra.** Simpul berikutnya adalah simpul belum-selesai dengan taksiran *saat ini* yang terkecil, termasuk taksiran yang diturunkan pada iterasi sebelumnya — bukan yang terkecil ketika pertama kali dicapai. Tuliskan seluruh baris **Tabel 16.8** sebelum memilih.
10. **Melupakan pemeriksaan penutup Bellman–Ford, atau menghentikannya terlalu cepat.** Tanpa sapuan tambahan tidak ada deteksi siklus negatif; dan jalan keluar awal hanya sah sesudah putaran *tanpa* perubahan, bukan sesudah putaran dengan sedikit perubahan (Persoalan 16.7).
11. **Mengambil urutan topologis dari waktu selesai yang menaik.** Urutannya adalah waktu selesai yang *menurun*; waktu selesai yang menaik mendaftar setiap sisi mundur (Persoalan 16.8).
12. **Membiarkan $\infty + w$ meluap.** Dalam kode, ∞ adalah bilangan besar, dan menambahkan bobot negatif padanya menghasilkan bilangan besar yang bukan lagi "takhingga". Jagalah setiap relaksasi dengan `d[u] < INF`, seperti yang dilakukan setiap daftar kode dalam bab ini.
13. **Memilih fungsi potensial tanpa memeriksa $\Phi(D_i) \geq \Phi(D_0)$.** Potensial yang menjadi negatif "membuktikan" batas yang keliru — Persoalan 16.10(f) "membuktikan" $2n$ untuk tabel yang biaya sebenarnya pada $n = 17$ adalah 48, bukan 34.
14. **Mencampuradukkan biaya amortisasi dengan biaya kasus rata-rata.** Batas amortisasi berlaku untuk *setiap* barisan, pada kasus terburuk, tanpa peluang di mana pun. Batas kasus rata-rata bergantung pada sebaran masukan.

15. **Meletakkan k di gelung yang keliru pada Floyd–Warshall.** Simpul antara k harus menjadi gelung *terluar*; dengan k di gelung terdalam, d_{ij} dituntaskan sebelum rute-rute yang menjadi tumpuannya diketahui (Subbab 15.6.6).
16. **Lupa membatalkan pembobotan ulang Johnson.** Dijkstra atas w' mengembalikan $\hat{d}(u,v)$, bukan $\delta(u,v)$; jawabannya adalah $\hat{d}(u,v) - h(u) + h(v)$ (Persoalan 16.12).
17. **Menambahkan konstanta pada setiap sisi untuk menghilangkan bobot negatif.** Lintasan dengan lebih banyak sisi mendapat tambahan lebih besar, jadi lintasan terpendeknya dapat berubah (Soal Latihan 1(d)). Potensial Johnson berhasil justru karena koreksinya berteleskop menjadi besaran yang hanya bergantung pada kedua ujungnya.
18. **Menyebut algoritma pseudo-polinomial sebagai polinomial.** *Knapsack* $\Theta(nW)$ eksponensial terhadap *ukuran* masukan (Persoalan 16.19).
19. **Mengilustrasikan graf alih-alih algoritma.** Ilustrasi harus memperlihatkan taksiran-taksiran berubah. Paling sedikit dua keadaan antara.
20. **Membaca Tabel 16.19 sebagai vonis alih-alih titik berangkat.** Kolom biayanya asimtotik; konstanta, memori, dan ukuran masukan tetap menentukan (pernyataan (12) Persoalan 16.1).

16.18 Ringkasan Bab, dan Sepatah Kata Penutup

Bab ini mengonsolidasikan Bab 13 sampai 15 dan memadukan seluruh buku dengan mengerjakan dua puluh satu persoalan, masing-masing diturunkan dari asas paling dasar, masing-masing diperiksa oleh program yang sudah dikompilasi dan dijalankan.

Hasil-hasil pokoknya, dihimpun:

- **Persoalan 16.1.** Sembilan benar, enam salah. Yang salah adalah (3) — pencarian linear $\Theta(n)$ entah lariknya terurut entah tidak; (8) — penyisipan ke senarai terurut $\Theta(n)$ karena separuh yang murah tidak menyelamatkan separuh yang mahal; (11) — O adalah batas atas, bukan batas bawah; (12) — konstanta, memori, ukuran masukan, dan kestabilan semuanya kadang-kadang mengungguli asimtotik; (13) — urutan mendalam dan urutan melebar sepakat pada 26 dari 38 graf bersimpul empat, dan pada lintasan bersimpul tiga.
- **Persoalan 16.2.** $L[i]$, panjang subbarisan menaik terpanjang yang *berakhir di* $S[i]$, memenuhi Persamaan (16.2). Dua gelung bersarang memberi $\Theta(n^2)$; larik pendahulu memberi subbarisannya sendiri; larik `tail` yang terurut dengan pencarian biner memberi $O(n\lg n)$.
- **Persoalan 16.3.** $X(n) = n + 1$ dengan induksi kuat atas banyaknya simpul dalam, membelah di akar, Persamaan (16.3); k saling meniadakan, dan itulah sebabnya hasilnya tidak bergantung pada bentuk pohon.
- **Persoalan 16.4.** $d(T) \leq d(L^*)$ karena menghapus satu sisi dari tur optimal menyisakan pohon merentang; $d(W) = 2d(T)$ karena penelusuran penuh memakai setiap sisi pohon dua kali; $d(L) \leq d(W)$ menurut ketaksamaan segitiga; bersama-sama, Persamaan (16.5). Pada enam titik nyata rasio yang teramati adalah $40/28 = 1.4286$.
- **Persoalan 16.5.** Kedua algoritma hanya berbeda dalam urutan dan kelipatan relaksasinya; perbedaan itulah alasan *router* memakai yang satu dan peta memakai yang lain.
- **Persoalan 16.6 sampai 16.8.** Dijkstra menuntaskan simpul dalam urutan jarak yang menaik dan merelaksasi setiap sisi sekali; Bellman–Ford membutuhkan sampai $|V| - 1$ putaran, lebih sedikit dengan urutan sisi yang beruntung, dan satu putaran lagi yang masih merelaksasi sesuatu membuktikan adanya siklus negatif, yang lalu diperlihatkan oleh larik pendahulunya; pada DAG, satu putaran dalam urutan topologis cukup untuk bobot apa pun.

- **Persoalan 16.9 dan 16.10.** Pencacah biner berbiaya kurang dari $2n$ dan tabel berlipat dua kurang dari $3n$ untuk n operasi, menurut metode agregat, akuntansi, maupun potensial; kredit akuntansi dan potensialnya adalah bilangan yang sama, dan potensial yang boleh menjadi negatif membuktikan batas yang keliru.
- **Persoalan 16.11 dan 16.12.** Floyd–Warshall menghitung $D^{(0)}, \ldots, D^{(n)}$ menurut Persamaan (16.10) dalam $\Theta(V^3)$; Johnson membobot ulang menurut Persamaan (16.11), yang sifat teleskopiknya mempertahankan setiap lintasan terpendek, dan mencapai matriks yang sama dengan V kali Dijkstra.
- **Persoalan 16.13 sampai 16.21** meninjau kembali bagian buku selebihnya: lintasan terpanjang pada DAG, banyaknya sisi sebuah pohon, keoptimalan Huffman, kedua algoritma semua pasangan dibandingkan, *knapsack* pseudo-polinomial, hampiran-2 penutup simpul, dan Prim berhadapan dengan Kruskal.

Di sekeliling persoalan-persoalan itu, bab ini meninggalkan untuk Anda sebuah metode untuk persoalan berbagian dan bukti yang macet, empat puluh latihan uji diri di seluruh enam belas bab, tabel pemilihan algoritma yang disusun menurut bentuk persoalan alih-alih nama algoritma, dan peta enam wilayah — aliran jaringan, NP-kelengkapan dan hampiran, pencocokan string, geometri komputasional, pemrograman linear, serta algoritma teracak dan daring — yang kini siap Anda masuki.

Sepatah kata penutup

Enam belas bab yang lalu buku ini dimulai dengan seorang pengayuh becak dan seorang penjual siomay, dan janji bahwa seseorang yang tidak tahu apa-apa dapat memulai di halaman pertama dan menyelesaikannya dengan kemampuan mengimplementasikan bahannya. Jika Anda sudah tiba di sini, janji itu telah ditepati di sisi Anda, dan itulah sisi yang lebih berat.

Ada baiknya kita cermat tentang apa yang kini Anda miliki, sebab itu bukan apa yang dikira kebanyakan orang diberikan buku seperti ini.

Anda tidak memiliki katalog algoritma. Katalog dapat Anda cari. Yang Anda miliki adalah kemampuan untuk memandang persoalan yang belum pernah Anda lihat — sederet selang waktu, jaringan jalan dengan tol negatif, tabel yang harus tumbuh tanpa batas, bukti yang tidak kunjung selesai pada percobaan pertama — dan *mengenali bentuknya*. Persoalan 16.4 adalah buktinya. Buku ini tidak pernah mengajarkan algoritma hampiran. Anda diberi persoalan dari pokok bahasan yang tidak dicakupnya, dan semua yang Anda butuhkan sudah ada di tangan Anda: sebuah pohon merentang, sebuah penelusuran, dan satu ketaksamaan. Itu bukan keberuntungan. Itulah gunanya pendidikan algoritma yang sungguh-sungguh.

Anda juga memiliki sesuatu yang lebih langka, yaitu kebiasaan untuk tidak memercayai diri sendiri. Setiap jarak, setiap bobot pohon, setiap matriks, dan setiap batas amortisasi dalam bab ini dihitung dua kali — sekali dengan tangan dan sekali dengan program — dan keduanya sepakat. Ketika suatu hari keduanya tidak sepakat, Anda akan tahu bahwa salah satunya keliru dan bahwa menemukan yang mana itulah pekerjaannya. Kebiasaan itu akan bertahan lebih lama daripada setiap batas asimtotik dalam buku ini.

Dan Anda memiliki hitungan tentang akibat. Pohon merentang minimum tidak berbiaya lebih daripada tur terbaik. Tabel berlipat dua berbiaya kurang dari tiga per penyisipan, dan tabel yang tumbuh dengan konstanta berbiaya kuadratik. Satu sisi negatif cukup untuk membuat pilihan rakus tidak aman, dan satu potensial yang turun di bawah nol cukup untuk membuat sebuah bukti keliru. Algoritma adalah disiplin tempat perasaan samar tentang biaya digantikan oleh

bilangan yang dapat Anda pertahankan, dan penggantian itu, lebih daripada teknik tunggal mana pun, adalah hal yang layak Anda bawa keluar dari buku ini.

Semoga berhasil — dan tetaplah luar biasa. Selebihnya kini menjadi tugas Anda.

16.19 Soal Latihan

1. Tandailah setiap pernyataan Benar atau Salah dan berikan pembenaran satu kalimat. (a) Penyisipan ke *binary heap* berisi n unsur berbiaya $O(\lg n)$. (b) Setiap graf berarah tanpa siklus mempunyai paling sedikit satu simpul berderajat-masuk nol. (c) Jika setiap bobot sisi graf terhubung dinaikkan dengan konstanta yang sama $c > 0$, pohon merentang minimumnya tidak berubah. (d) Jika setiap bobot sisi dinaikkan dengan konstanta yang sama $c > 0$, setiap lintasan terpendek tidak berubah. (e) Algoritma rakus yang optimal untuk semua masukan berukuran ≤ 10 optimal untuk semua masukan.
2. Jalankan **Pseudokode 16.1** dengan tangan pada $S = \langle 4,12,1,9,5,15,7,11 \rangle$. Berikan larik L dan P serta subbarisan yang dihasilkan penelusuran baliknya.
3. Buktikan bahwa pohon biner diperluas dengan n simpul dalam mempunyai seluruhnya $2n + 1$ simpul, lalu simpulkan berapa penunjuk kosong yang dimuat pohon biner *aslinya*.
4. Ambillah keenam titik $A = (3,0)$, $B = (3,2)$, $C = (0,3)$, $D = (0,6)$, $E = (0,0)$, $F = (3,5)$ dengan metrik Manhattan. Bangunlah pohon merentang minimum berakar di A, berikan tur *preorder* dan panjangnya, temukan tur optimalnya, dan laporkan rasionya. Bandingkan dengan 1.4286 di Subbab 16.6.7.
5. Telusurilah Dijkstra dari A pada graf berarah dengan sisi $A \to B$ (7), $A \to C$ (9), $A \to F$ (14), $B \to C$ (10), $B \to D$ (15), $C \to D$ (11), $C \to F$ (2), $F \to E$ (9), $D \to E$ (6). Berikan urutan pengeluarannya, setiap perubahan taksiran, dan pohon lintasan terpendeknya.
6. Tinjaulah lintasan $s \to a \to b \to c$ dengan setiap sisi berbobot 1. Berapa putaran yang dibutuhkan Bellman–Ford jika sisi-sisinya direlaksasi dalam urutan $(s, a), (a, b), (b, c)$? Dalam urutan $(b, c), (a, b), (s, a)$? Apa artinya ini bagi batas $|V| - 1$?
7. Susunlah contoh empat simpul yang **tidak** memenuhi ketaksamaan segitiga dan yang membuat algoritma Persoalan 16.4 mengembalikan tur lebih dari dua kali optimumnya. Sebutkan langkah bukti mana yang gagal.
8. Enam tanda layanan Profitina Laundry muncul dengan frekuensi W: 40, D: 25, P: 15, F: 10, S: 6, I: 4 dalam sampel seratus tanda. Bangunlah kode Huffmannya, berikan panjang kata kodenya, hitunglah jumlah bitnya, dan bandingkan dengan kode berpanjang tetap.
9. Sebuah tumpukan mendukung `PUSH`, `POP`, dan `MULTIPOP(k)`, yang mengeluarkan $\min(k, |S|)$ butir. Dengan fungsi potensial $\Phi(S) = |S|$, tunjukkan bahwa biaya amortisasi setiap operasi itu $O(1)$, dan karena itu setiap barisan n operasi berbiaya $O(n)$.
10. Untuk setiap bentuk persoalan, sebutkan algoritmanya dari **Tabel 16.19**, berikan biayanya, dan katakan alasannya dalam satu anak kalimat. (a) Seorang kurir harus mengunjungi 12 alamat lalu kembali, meminimumkan jarak, dan jaraknya garis lurus. (b) Sebuah sistem *build* harus mengurutkan 400 unit kompilasi yang saling bergantung. (c) Pemeriksa ejaan harus memutuskan apakah masing-masing dari 10^6 kata ada dalam kamus tetap berisi 10^5 kata. (d) Sebuah gim harus menemukan jalur termurah menyeberangi kisi yang sebagian petaknya memulihkan nyawa (biaya negatif). (e) Pemantau jaringan harus melaporkan jarak antara setiap pasangan dari 200 *router*.
11. Hitunglah lintasan terpendek dari s pada DAG dengan sisi $s \to a$ (2), $s \to b$ (6), $a \to b$ (3), $a \to c$ (7), $b \to c$ (−4), $b \to d$ (2), $c \to d$ (1), dengan memberikan urutan topologisnya dan d/π sesudah setiap simpul diproses.

12. Sebuah pencacah biner mendukung `INCREMENT` dan `RESET`, dengan `RESET` menjadikan setiap bit 0 dan berbiaya sebanyak bit yang tadinya 1. Dengan fungsi potensial, tunjukkan bahwa setiap barisan n operasi yang dimulai dari nol berbiaya paling banyak $2n$.
13. Jalankan Floyd–Warshall pada graf tiga simpul dengan sisi $1 \to 2\,(5)$, $1 \to 3\,(6)$, $2 \to 3\,(-1)$, $3 \to 1\,(2)$. Berikan $D^{(0)}$ sampai $D^{(3)}$.
14. Untuk graf Soal Latihan 13, hitunglah potensial h Johnson dan sisi-sisi yang dibobot ulang, lalu pakailah keduanya untuk memulihkan $\delta(1,3)$ dari jalannya Dijkstra dari simpul 1. Kemudian jelaskan, dengan sebuah bilangan dari graf ini, mengapa menambahkan konstanta 3 pada setiap sisi bukan alternatif yang aman.

16.20 Penyelesaian Soal Latihan

1. (a) **Benar** — letakkan unsur barunya di ujung lalu naikkan sepanjang lintasan sedalam $\Theta(\lg n)$. (b) **Benar** — DAG mempunyai urutan topologis, dan simpul pertamanya berderajat-masuk nol; setara dengan itu, ikutilah sisi masuk ke belakang dan ketaksiklikan memaksa perjalanan itu berhenti. (c) **Benar** — setiap pohon merentang mempunyai tepat $V - 1$ sisi, jadi bobot setiap pohon merentang naik tepat $(V - 1)c$ dan urutan pohon-pohon merentang menurut bobot tidak berubah. (d) **Salah** — lintasan yang berbeda mempunyai banyak sisi yang berbeda, jadi lintasan k-sisi bertambah kc sedangkan lintasan dengan lebih sedikit sisi bertambah lebih kecil; menambahkan konstanta karena itu dapat mengubah lintasan mana yang terpendek. Inilah sebabnya Johnson membobot ulang dengan potensial yang berteleskop dan bukan dengan konstanta (Soal Latihan 14). (e) **Salah** — keoptimalan pada masukan kecil tidak membuktikan apa pun tentang masukan besar; rakus untuk *knapsack* 0/1 optimal pada banyak contoh kecil dan keliru secara umum.

2. Mengerjakan gelung luarnya: $L[1] = 1, P[1] = 0$; $L[2] = 2, P[2] = 1$ (karena $4 < 12$); $L[3] = 1, P[3] = 0$; $L[4] = 2, P[4] = 1$ $(4 < 9)$; $L[5] = 2, P[5] = 1$ $(4 < 5)$; $L[6] = 3, P[6] = 2$ $(12 < 15$ dengan $L[2] = 2)$; $L[7] = 3, P[7] = 5$ $(5 < 7$ dengan $L[5] = 2)$; $L[8] = 4, P[8] = 7$ $(7 < 11$ dengan $L[7] = 3)$.

Tabel 16.20 Tabel-tabel untuk Soal Latihan 2. Maksimumnya $L[8] = 4$, jadi penelusuran baliknya dimulai pada $k = 8$.

i	1	2	3	4	5	6	7	8
$S[i]$	4	12	1	9	5	15	7	11
$L[i]$	1	2	1	2	2	3	3	4
$P[i]$	0	1	0	1	1	2	5	7

Tabel 16.20 menghimpunnya. Telusuri balik dari $k = 8$: $11 \to P[8] = 7 \to 7 \to P[7] = 5 \to 5 \to P[5] = 1 \to 4 \to P[1] = 0$. Dibalik: $\langle 4,5,7,11 \rangle$, berpanjang 4. (Perhatikan bahwa $\langle 4,9,15 \rangle$ berpanjang 3 dan $\langle 1,5,7,11 \rangle$ juga berpanjang 4 — jawabannya tidak tunggal.)

3. Setiap simpul dalam pohon biner diperluas mempunyai tepat dua anak dan setiap simpul luar tidak beranak, dan menurut Subbab 16.5.3 terdapat n simpul dalam dan $n + 1$ simpul luar, jadi totalnya $n + (n + 1) = 2n + 1$ — selalu ganjil. Simpul-simpul luar tepat merupakan penunjuk kosong pohon aslinya, jadi pohon biner asli dengan n simpul memuat $n + 1$ penunjuk kosong. Periksalah pada satu simpul: $n = 1$, dua anak kosong, $1 + 1 = 2$. Periksalah pada **Gambar 16.1**: $n = 4$, lima penunjuk kosong, total $2(4) + 1 = 9$ simpul dalam pohon diperluasnya — empat lingkaran dan lima persegi.

4. Jarak Manhattan: $AB = 2$, $AC = 6$, $AD = 9$, $AE = 3$, $AF = 5$, $BC = 4$, $BD = 7$, $BE = 5$, $BF = 3$, $CD = 3$, $CE = 3$, $CF = 5$, $DE = 6$, $DF = 4$, $EF = 8$. Prim dari A mengambil $A - B$ (2), lalu $A - E$ (3), lalu $E - C$ (3), lalu $C - D$ (3), lalu $B - F$ (3): $d(T) = 2 + 3 + 3 + 3 + 3 = 14$. Berakar di A anak-anaknya adalah $A \to \{B, E\}$, $B \to \{F\}$, $E \to \{C\}$, $C \to \{D\}$, jadi daftar *preorder*-nya A, B, F, E, C, D dan turnya berbiaya $2 + 3 + 8 + 3 + 3 + 9 = 28$. Pencarian menyeluruh atas 60 tur memberi optimum $A \to B \to F \to D \to C \to E \to A$ seharga $2 + 3 + 4 + 3 + 3 + 3 = 18$. Rasionya $28/18 = 1.5556$ — lebih buruk daripada 1.4286 di Subbab 16.6.7, dan tetap jauh di bawah jaminan 2. Langkah yang mahal adalah $F \to E$ seharga 8: penelusuran *preorder* terpaksa melompat dari dasar satu cabang ke puncak cabang lain, dan tepat di situlah algoritma ini merugi.

5. Urutan pengeluarannya A, B, C, F, D, E.

- Keluarkan A ($d = 0$): $d[B] = 7, d[C] = 9, d[F] = 14$.
- Keluarkan B ($d = 7$): $7 + 10 = 17 \not< 9$, C tidak berubah; $d[D] = 7 + 15 = 22$.
- Keluarkan C ($d = 9$): $d[D] = 9 + 11 = 20 < 22$; $d[F] = 9 + 2 = 11 < 14$.
- Keluarkan F ($d = 11$): $d[E] = 11 + 9 = 20$.
- Keluarkan D ($d = 20$): $20 + 6 = 26 \not< 20$, tidak berubah.
- Keluarkan E ($d = 20$): tanpa sisi keluar.

Jarak yang dihasilkan $A = 0$, $B = 7$, $C = 9$, $F = 11$, $D = 20$, $E = 20$; pohon lintasan terpendek $B \leftarrow A$, $C \leftarrow A$, $F \leftarrow C$, $D \leftarrow C$, $E \leftarrow F$. Perhatikan bahwa $d[D]$ maupun $d[F]$ diperbaiki sesudah penetapan pertamanya — sebuah taksiran bersifat sementara sampai simpulnya dikeluarkan. (D dan E seri pada 20; yang mana pun boleh dikeluarkan lebih dahulu.)

6. Dalam urutan $(s, a), (a, b), (b, c)$, **satu** putaran cukup: putaran tunggal itu menetapkan $d[a] = 1$, lalu $d[b] = 2$, lalu $d[c] = 3$. Dalam urutan terbalik $(b, c), (a, b), (s, a)$, setiap putaran memajukan garis depannya tepat satu sisi: putaran 1 hanya menetapkan $d[a] = 1$; putaran 2 menetapkan $d[b] = 2$; putaran 3 menetapkan $d[c] = 3$. Dengan $|V| = 4$ itu $|V| - 1 = 3$ putaran, jadi batasnya tercapai. Pelajarannya, yang juga diperlihatkan Persoalan 16.7: $|V| - 1$ adalah kasus terburuk atas urutan sisi, bukan cacahan putaran yang sungguh dibutuhkan. Urutan yang baik — urutan topologis, bila ada — menguranginya menjadi satu putaran, yang tepat merupakan algoritma DAG **Pseudokode 16.2**.

7. Ambillah A, B, C, D dengan $d(A, B) = d(A, C) = d(A, D) = d(B, C) = d(B, D) = 1$ dan $d(C, D) = 100$. Ketaksamaan segitiga gagal pada $d(C, D) = 100 > d(C, A) + d(A, D) = 2$. Prim dari A mengambil $A - B$, $A - C$, $A - D$, semuanya berbobot 1, jadi $d(T) = 3$; daftar *preorder*-nya A, B, C, D dan turnya berbiaya $1 + 1 + 100 + 1 = 103$. Optimumnya $A \to C \to B \to D \to A = 1 + 1 + 1 + 1 = 4$, jadi rasionya $103/4 = 25.75$, jauh di atas 2. Langkah yang gagal adalah Langkah 3 di Subbab 16.6.5: pemintasan dari C ke D mengganti potongan dua-sisi $C \to A \to D$ berbiaya 2 dengan satu sisi berbiaya 100, dan hanya ketaksamaan segitiga yang melarang hal itu.

8. Menggabungkan dua yang terkecil berulang kali: $4 + 6 = 10$; $10 + 10 = 20$; $15 + 20 = 35$; $25 + 35 = 60$; $40 + 60 = 100$. Panjang kata kodenya mengikuti kedalamannya: W di kedalaman 1, D di 2, P di 3, F di 4, dan S, I di 5. Jumlah bitnya

$$\begin{aligned} &40(1) + 25(2) + 15(3) + 10(4) + 6(5) + 4(5) \\ &\qquad = 40 + 50 + 45 + 40 + 30 + 20 = 225. \end{aligned}$$

Jumlah nilai simpul-simpul dalamnya $10 + 20 + 35 + 60 + 100 = 225$ — bilangan yang sama, yang merupakan cara alternatif baku untuk menghitung biaya Huffman. Kode berpanjang tetap

untuk enam simbol membutuhkan $\lceil \lg 6 \rceil = 3$ bit masing-masing, jadi 300 bit; penghematannya $(300 - 225)/300 = 25\%$. **Kode 16.20** mengukuhkannya.

Kode 16.20 Kode Huffman Soal Latihan 8, dengan barisan penggabungannya, kata-kata kodenya, dan perbandingannya terhadap kode berpanjang tetap tiga bit.

```
// Soal latihan 8: bangunlah kode Huffman untuk tanda layanan Profitina Laundry
// lalu bandingkan dengan kode berpanjang tetap.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <map>
using namespace std;

struct Node { int f; char c; Node *l, *r; };
struct Cmp { bool operator()(Node* a, Node* b) const { return a->f > b->f; } };

void codes(Node* t, string s, map<char,string>& out) {
    if (!t->l && !t->r) { out[t->c] = s.empty() ? "0" : s; return; }
    codes(t->l, s + "0", out);
    codes(t->r, s + "1", out);
}

int main() {
    vector<pair<char,int>> freq = {{'W',40},{'D',25},{'P',15},{'F',10},{'S',6},{'I',4}};
    priority_queue<Node*, vector<Node*>, Cmp> Q;
    for (auto [c, f] : freq) Q.push(new Node{f, c, nullptr, nullptr});

    int internalSum = 0;
    while (Q.size() > 1) {
        Node* a = Q.top(); Q.pop();
        Node* b = Q.top(); Q.pop();
        Node* z = new Node{a->f + b->f, '*', a, b};
        internalSum += z->f;
        cout << "merge " << setw(3) << a->f << " + " << setw(3) << b->f
             << " = " << setw(3) << z->f << "\n";
        Q.push(z);
    }
    map<char,string> code;
    codes(Q.top(), "", code);
```

Kode 16.20 (Lanjutan) *Kode Huffman Soal Latihan 8, dengan barisan penggabungannya, kata-kata kodenya, dan perbandingannya terhadap kode berpanjang tetap tiga bit.*

```
    int total = 0, n = 0;
    cout << "\n sym  freq  code      bits\n";
    for (auto [c, f] : freq) {
        total += f * (int)code[c].size();
        n += f;
        cout << setw(4) << c << setw(6) << f << setw(7) << code[c]
             << setw(10) << f * (int)code[c].size() << "\n";
    }
    cout << "\nHuffman total          = " << total << " bits\n"
         << "sum of internal nodes  = " << internalSum << " bits (same number)\n"
         << "fixed-length 3 bits    = " << 3 * n << " bits\n"
         << "saving                 = " << 100.0 * (3 * n - total) / (3 * n) << " %\n";
    return 0;
}
```

```
merge   4 +   6 =  10
merge  10 +  10 =  20
merge  15 +  20 =  35
merge  25 +  35 =  60
merge  40 +  60 = 100

 sym  freq  code      bits
   W    40      0        40
   D    25     10        50
   P    15    110        45
   F    10   1110        40
   S     6  11111        30
   I     4  11110        20

Huffman total          = 225 bits
sum of internal nodes  = 225 bits (same number)
fixed-length 3 bits    = 300 bits
saving                 = 25 %
```

9. Ambillah $\Phi(S) = |S|$, banyaknya butir di dalam tumpukan. Ia memenuhi $\Phi(S) \geq 0 = \Phi(S_0)$, jadi ia potensial yang sah. Biaya amortisasinya diberikan oleh Persamaan (16.6).

- `PUSH`: biaya sebenarnya 1; tumpukannya bertambah satu, jadi $\Delta\Phi = +1$; $\hat{c} = 1 + 1 = 2$.
- `POP`: biaya sebenarnya 1; $\Delta\Phi = -1$; $\hat{c} = 1 - 1 = 0$.
- `MULTIPOP(k)`: misalkan $m = \min(k, |S|)$ adalah banyaknya yang sungguh dikeluarkan. Biaya sebenarnya m; $\Delta\Phi = -m$; $\hat{c} = m - m = 0$.

Setiap biaya amortisasi paling banyak 2, sebuah konstanta. Karena $\Phi(S_n) \geq \Phi(S_0)$, Persamaan (16.7) membatasi total sebenarnya dengan total amortisasinya:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(S_n) + \Phi(S_0) \leq \sum_{i=1}^{n} \hat{c}_i \leq 2n.$$

Jadi setiap barisan n operasi berbiaya $O(n)$, walaupun satu `MULTIPOP` dapat berbiaya $\Theta(n)$. Intuisi yang disandikan potensialnya: setiap `PUSH` membayar satu satuan di muka untuk pengeluaran kelak butir yang didorongnya, jadi mengeluarkan butir itu gratis ketika terjadi.

10. (a) Dua belas alamat dengan jarak garis lurus adalah **TSP metrik**; penyelesaian eksak masih terjangkau pada $n = 12$ dengan pemrograman dinamis atas himpunan bagian dalam $\Theta(2^n n^2)$, dan hampiran-2 MST-ditambah-*preorder* Persoalan 16.4 berjalan dalam $\Theta(n^2)$ dengan jaminan yang dibuktikan — pakailah hampiran itu jika jawabannya dibutuhkan berulang kali. (b) Empat ratus unit yang saling bergantung adalah **pengurutan topologis**, $\Theta(V + E)$, sebab tuntutannya tepat "setiap unit sesudah prasyaratnya", yang dapat dipenuhi tepat ketika graf ketergantungannya taksiklik. (c) Sejuta kueri keanggotaan terhadap kamus tetap adalah **tabel *hash***, harapan $\Theta(1)$ per kueri, sebab urutan tidak pernah dibutuhkan; larik terurut dengan pencarian biner $\Theta(\lg n)$ adalah alternatifnya jika memori sangat sempit. (d) Kisi dengan sebagian petak berbiaya negatif menyingkirkan Dijkstra, jadi pakailah **Bellman–Ford**, $\Theta(VE)$, yang juga melaporkan siklus negatif jika petak-petak pemulih itu membentuk putaran yang dapat dieksploitasi selamanya. (e) Semua pasangan di antara 200 *router* cukup padat sehingga **Floyd–Warshall** dengan $\Theta(V^3) = 8 \times 10^6$ langkah dasar paling sederhana dan paling cepat ditulis; Johnson baru lebih disukai jika grafnya jauh lebih besar dan jarang.

11. Setiap sisi mengarah maju dalam s, a, b, c, d, jadi itulah urutan topologisnya. Pemrosesan: s — $d[a] = 2$ $(\pi = s)$, $d[b] = 6$ $(\pi = s)$. a — $a \to b$: $2 + 3 = 5 < 6$, jadi $d[b] = 5$, $\pi[b] = a$; $a \to c$: $d[c] = 9, \pi[c] = a$. b — $b \to c$: $5 + (-4) = 1 < 9$, jadi $d[c] = 1, \pi[c] = b$; $b \to d$: $d[d] = 7, \pi[d] =$

b. c — $c \to d$: $1 + 1 = 2 < 7$, jadi $d[d] = 2$, $\pi[d] = c$. d — tanpa sisi keluar. Hasilnya $\delta = (s{:}\,0,\ a{:}\,2,\ b{:}\,5,\ c{:}\,1,\ d{:}\,2)$, dan lintasan ke d adalah $s \to a \to b \to c \to d$ berbobot $2 + 3 - 4 + 1 = 2$.

12. Misalkan Φ adalah banyaknya bit 1, jadi $\Phi(D_0) = 0$ dan $\Phi \geq 0$. `INCREMENT` dianalisis persis seperti pada Persoalan 16.9(c): $\hat{c} \leq 2$. `RESET` pada pencacah dengan b bit 1 berbiaya b dan menurunkan potensialnya dari b menjadi 0, jadi $\hat{c} = b + (0 - b) = 0$. Setiap biaya amortisasi paling banyak 2, jadi menurut Persamaan (16.7) setiap n operasi berbiaya paling banyak $2n$. Dalam istilah akuntansi: kredit yang ditinggalkan pada setiap bit 1 ketika bit itu ditetapkan membayar penolannya, entah penolan itu dikerjakan oleh `INCREMENT` berikutnya entah oleh `RESET`. (Agar biaya `RESET` sebesar banyaknya bit 1 dan bukan k, pencacahnya harus tahu di mana bit 1 tertingginya — satu indeks tambahan, dipelihara dalam $O(1)$ per operasi.)

13. $D^{(0)} = \begin{pmatrix} 0 & 5 & 6 \\ \infty & 0 & -1 \\ 2 & \infty & 0 \end{pmatrix}$. Ronde 1: hanya $d_{31} = 2$ yang berhingga di kolom 1, jadi $d_{32} = \min(\infty, 2 + 5) = 7$ dan $d_{33} = \min(0{,}2 + 6) = 0$: $D^{(1)} = \begin{pmatrix} 0 & 5 & 6 \\ \infty & 0 & -1 \\ 2 & 7 & 0 \end{pmatrix}$. Ronde 2: hanya $d_{23} = -1$ yang berhingga di baris 2, jadi $d_{13} = \min(6{,}5 - 1) = 4$ dan $d_{33} = \min(0{,}7 - 1) = 0$: $D^{(2)} = \begin{pmatrix} 0 & 5 & 4 \\ \infty & 0 & -1 \\ 2 & 7 & 0 \end{pmatrix}$. Ronde 3: baris 3 adalah $(2{,}7{,}\cdot)$, jadi $d_{21} = \min(\infty, -1 + 2) = 1$, $d_{12} = \min(5{,}4 + 7) = 5$, $d_{11} = \min(0{,}4 + 2) = 0$, $d_{22} = \min(0, -1 + 7) = 0$: $D^{(3)} = \begin{pmatrix} 0 & 5 & 4 \\ 1 & 0 & -1 \\ 2 & 7 & 0 \end{pmatrix}$. Diagonalnya nol, jadi tidak ada siklus negatif; satu-satunya siklus, $1 \to 2 \to 3 \to 1$ dan $1 \to 3 \to 1$, berbobot 6 dan 8.

14. Dari simpul bantu q, setiap h mulai dari 0; sisi $2 \to 3$ menurunkan $h(3)$ menjadi -1, dan tidak ada lagi yang membaik, jadi $h = (0{,}0, -1)$. Menurut Persamaan (16.11): $w'(1{,}2) = 5 + 0 - 0 = 5$, $w'(1{,}3) = 6 + 0 + 1 = 7$, $w'(2{,}3) = -1 + 0 + 1 = 0$, $w'(3{,}1) = 2 - 1 - 0 = 1$ — semuanya tak negatif. Dijkstra dari 1 atas w' mencapai 2 pada 5 lalu 3 pada $\min(7{,}5 + 0) = 5$, jadi $\hat{d}(1{,}3) = 5$ dan $\delta(1{,}3) = 5 - h(1) + h(3) = 5 - 0 - 1 = 4$, sepakat dengan $D^{(3)}$ Soal Latihan 13. Kini tambahkan 3 pada setiap sisi saja: sisi langsung $1 \to 3$ menjadi $6 + 3 = 9$ sedangkan lintasan dua-sisi $1 \to 2 \to 3$ menjadi $(5 + 3) + (-1 + 3) = 10$, jadi lintasan "terpendek" akan beralih ke sisi langsung, yang bobot sebenarnya 6 lebih buruk daripada optimum sebenarnya 4. Konstanta itu menghukum lintasan menurut banyaknya sisi; potensial hanya menghukumnya menurut kedua ujungnya.

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," *Operations Research Forum*, vol. 3, no. 20, 2022. (Originally Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.)

[3] A. R. Karlin, N. Klein, and S. Oveis Gharan, "A (slightly) improved approximation algorithm for metric TSP," in *Proc. 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, 2021, pp. 32–45. doi: 10.1145/3406325.3451009. Journal version: *Operations Research*, 2023, doi: 10.1287/opre.2022.2338.

[4] L. Gurvits, N. Klein, and J. Leake, "From trees to polynomials and back again: New capacity bounds with applications to TSP," in *Proc. 51st International Colloquium on Automata, Languages, and Programming (ICALP)*, LIPIcs vol. 297, 2024, paper 79. doi: 10.4230/LIPIcs.ICALP.2024.79.

[5] J. Kleinberg and É. Tardos, *Algorithm Design*. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman, 1979.

[7] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2011.

[8] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[9] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge, U.K.: Cambridge Univ. Press, 1997.

[10] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Berlin, Germany: Springer, 2008.

[11] J. Matoušek and B. Gärtner, *Understanding and Using Linear Programming*. Berlin, Germany: Springer, 2007.

[12] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 1995.

[13] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[14] S. S. Skiena, *The Algorithm Design Manual*, 3rd ed. Cham, Switzerland: Springer, 2020.

[15] *Programming Languages — C++*, ISO/IEC Standard 14882:2017, International Organization for Standardization, Geneva, Switzerland, 2017.

Materi lintasan terpendek Persoalan 16.5 sampai 16.8 mengikuti Cormen dkk. [1, ch. 22], analisis amortisasi Persoalan 16.9 dan 16.10 mengikuti [1, ch. 16], dan algoritma semua pasangan Persoalan 16.11, 16.12, dan 16.17 mengikuti [1, ch. 23]; gambar **Gambar 16.2**, **Gambar 16.3**, **Gambar 16.4**, **Gambar 16.5**, **Gambar 16.6**, **Gambar 16.7**, dan **Gambar 16.8** adalah gambar Bab 13, **Gambar 16.9** adalah gambar Bab 14, dan **Gambar 16.1** mengilustrasikan Persoalan 16.3. Program dinamis Persoalan 16.2 dan 16.19, bukti rakus Persoalan 16.16, serta argumen graf Persoalan 16.3, 16.15, dan 16.21 mengikuti [1, ch. 14, 15, 20, 21]; hampiran-2 TSP metrik Persoalan 16.4 dan batas penutup simpul Persoalan 16.20 adalah [1, §35.1–35.2], dan perbaikan 3/2 yang disebut di Subbab 16.6.9 adalah [2]. Wilayah-wilayah Subbab 16.15 diperkenalkan dalam [5] untuk aliran jaringan, [6] dan [7] untuk NP-kelengkapan dan hampiran, [9] dan [8] untuk pencocokan string, [10] untuk geometri komputasional, [11] untuk pemrograman linear, serta [12] dan [13] untuk algoritma teracak dan daring; [14] adalah katalog praktisi yang versi satu halamannya adalah **Tabel 16.19**. Setiap daftar kode dalam bab ini sesuai dengan [15] dan telah dikompilasi dengan `g++ -std=c++17 -O2 -Wall` lalu dijalankan; keluaran yang dikutip adalah keluaran yang dihasilkannya.