Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

DAFTAR ISI

Fondasi Perancangan dan Analisis Algoritma....8
1.1 Tentang Bab Ini....8
1.2 Algoritma dan Struktur Data — Dua Pilar Utama....9
1.3 Dua Tujuan Perancangan Algoritma....10
1.4 Paradigma Perancangan Algoritma....12
1.5 Kerangka Kerja Pemecahan Masalah Lima Fase....12
1.6 Kerangka Kerja dalam Aksi: Enam Soal Diselesaikan dari Awal....14
1.7 Pemetaan Setiap Soal ke Paradigma Perancangan....26
1.8 Profitina dalam Praktik: Di Mana Bab Ini Muncul pada Produk Nyata....26
1.9 Ringkasan Bab dan Poin-Poin Kunci....27
1.10 Pertanyaan Tinjauan....27
Lampiran 1.A — Catatan Verifikasi Kode Sumber Bab 1....28
Referensi....28

Menganalisis Algoritma: Model RAM, Insertion Sort, dan Pencarian....30
2.1 Rekap: Dari Kuliah 1 ke Kuliah 2....30
2.2 Sejarah Singkat Algoritma....30
2.3 Definisi Inti, Secara Presisi....31
2.4 Lima Tujuan Perancangan, Ditinjau Ulang....32
2.5 Model RAM untuk Komputasi....32
2.6 Cara Menganalisis Waktu Eksekusi....33
2.7 Studi Kasus: Insertion Sort — Analisis Lengkap....33
2.8 Kasus Terbaik, Terburuk, dan Rata-Rata....37
2.9 Pertumbuhan Fungsi: Mengapa Asimtotik Penting....38
2.10 Pencarian: Masalah Fundamental....39
2.11 Paradigma Perancangan, Ditinjau Ulang....42
2.12 Profitina dalam Praktik: Di Mana Bab Ini Muncul dalam Produk Nyata....44
2.13 Ringkasan Bab dan Poin-Poin Kunci....44
2.14 Pertanyaan Tinjauan....45
Lampiran 2.A — Log Verifikasi untuk Kode Sumber Bab 2....46
Referensi....46

Notasi Asimtotik, Rekurensi, dan Merge Sort....48
3.1 Rekap: Dari Kuliah 2 ke Kuliah 3....48
3.2 Memformalkan Notasi Asimtotik: O, Ω, dan Θ....49
3.3 Membuktikan Batas Asimtotik dari Definisi....50
3.4 Sifat-Sifat Notasi Asimtotik....50
3.5 Rekurensi: Dari Mana Asalnya....51
3.6 Menyelesaikan Rekurensi I: Metode Pohon Rekursi....51
3.7 Menyelesaikan Rekurensi II: Metode Substitusi....52
3.8 Menyelesaikan Rekurensi III: Master Theorem....53
3.9 Studi Kasus: Merge Sort — Analisis Lengkap....54
3.10 Mengapa Ini Penting: Merge Sort vs. Insertion Sort....58
3.11 Paradigma Perancangan Ditinjau Ulang: Divide and Conquer, Diformalkan....59
3.12 Profitina dalam Praktik: Di Mana Bab Ini Muncul dalam Produk Nyata....60
3.13 Ringkasan Bab dan Poin-Poin Kunci....60
3.14 Pertanyaan Tinjauan....61
Lampiran 3.A — Log Verifikasi untuk Kode Sumber Bab 3....62

Referensi 62

Soal Latihan: Fondasi Hingga Merge Sort 64

4.1 Rekap: Kuliah 1–3 Secara Singkat 64
4.2 Cara Menggunakan Bab Latihan Ini 65
4.3 Soal Latihan 65
4.4 Format Kuis 1: Esai Open-Book Take-Home 68
4.5 Profitina dalam Praktik: Self-Review Sebelum Submit 69
4.6 Ringkasan Bab 69
Lampiran 4.A — Daftar Periksa Mandiri (Tidak Dinilai) 71
Referensi 71

Heapsort, Antrian Prioritas, dan Quicksort 73

5.1 Rekap: Dari Kuliah 3 ke Kuliah 5 73
5.2 Wawasan Selection Sort: Percepat Titik Sumbatan, Bukan Seluruh Algoritma 74
5.3 Binary Heap: Pohon yang Tersembunyi di Dalam Larik 75
5.4 HEAPIFY: Mesin yang Menjaga Kejujuran Heap 77
5.5 BUILD-HEAP: Konstruksi O(n) yang Mengejutkan 78
5.6 HEAPSORT: O(n log n) dan In-Place 79
5.7 Antrian Prioritas: Aplikasi Heap Paling Terkenal 80
5.8 Quicksort: Pengurutan Tercepat dalam Praktik 82
5.9 Prosedur PARTITION 82
5.10 Algoritma QUICKSORT 83
5.11 Menganalisis Quicksort: Kasus Terbaik, Terburuk, dan Rata-Rata 84
5.12 Randomized Quicksort: Menjinakkan Kasus Terburuk 85
5.13 Lanskap Pengurutan Lengkap Sejauh Ini 86
5.14 Profitina dalam Praktik: Di Mana Bab Ini Muncul dalam Produk Nyata 87
5.15 Ringkasan Bab dan Poin-Poin Kunci 88
5.16 Pertanyaan Tinjauan 89
Lampiran 5.A — Log Verifikasi untuk Kode Sumber Bab 5 90
Referensi 90

Pohon Pencarian Biner dan Pohon AVL 92

6.1 Rekap: Dari Kuliah 5 ke Kuliah 6 92
6.2 Mengapa Larik Terurut Tidak Cukup: ADT Kamus 93
6.3 Binary Search Tree: Definisi dan Sifat BST 94
6.4 Kueri Statistik-Urutan: MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR 95
6.5 BST-SEARCH: Menyusuri Satu Jalur dari Akar 95
6.6 BST-INSERT: Jatuh ke Tempat Kosong 96
6.7 BST-DELETE: Tiga Kasus, Tiga Perbaikan 97
6.8 Menganalisis Operasi BST: Semuanya Bergantung pada Tinggi 98
6.9 Persoalan Keseimbangan: Bagaimana Urutan Penyisipan Menciptakan Pohon Degeneratif 98
6.10 Pohon AVL: Invarian Faktor-Keseimbangan 99
6.11 Memulihkan Keseimbangan: Empat Kasus Rotasi 100
6.12 AVL-INSERT: Satu Rotasi Membetulkan Seluruh Jalur 101
6.13 Mengapa AVL Menjamin O(log n): Batas Tinggi 103
6.14 Profitina dalam Praktik: Di Mana Bab Ini Muncul dalam Produk Nyata 103
6.15 Ringkasan Bab dan Poin-Poin Kunci 104
6.16 Pertanyaan Tinjauan 105
Lampiran 6.A — Log Verifikasi untuk Kode Sumber Bab 6 106
Referensi 106

Program Dinamis (Dynamic Programming)....108
7.1 Recap: Dari Kuliah 6 ke Kuliah 7....108
7.2 Apa Itu Program Dinamis? Substruktur Optimal dan Subpersoalan Tumpang-Tindih....109
7.3 Kisah Peringatan: Fibonacci Rekursif Naif....109
7.4 Memoization: Program Dinamis Top-Down....110
7.5 Tabulasi: Program Dinamis Bottom-Up....111
7.6 Soal Rod-Cutting dan Solusi Program Dinamisnya....112
7.7 Merekonstruksi Solusi Rod-Cutting Optimal....114
7.8 Perkalian Rantai Matriks: Mengapa Pengelompokan Penting....114
7.9 Solusi Program Dinamis untuk Perkalian Rantai Matriks....115
7.10 Longest Common Subsequence: Soal dan Rekurensi....116
7.11 Solusi Program Dinamis untuk LCS: Mengisi Tabel....116
7.12 Merekonstruksi LCS dengan Backtracking....117
7.13 Menganalisis Program Dinamis: Kapan Ia Menguntungkan?....118
7.14 Profitina dalam Praktik: Di Mana Bab Ini Muncul dalam Produk Nyata....119
7.15 Rangkuman Bab dan Poin-Poin Kunci....120
7.16 Soal Latihan....121
Lampiran 7.A — Log Verifikasi untuk Source Code Bab 7....122
Referensi....122

Soal Latihan: Heapsort Hingga Program Dinamis....124
8.1 Rekap: Kuliah 5–7 Secara Singkat....124
8.2 Cara Memakai Bab Latihan Ini....125
8.3 Soal Latihan....126
8.4 Format UTS: Open-Book, Take-Home Essay....128
8.5 Profitina dalam Praktik: Self-Review Sebelum Pengumpulan....129
8.6 Rangkuman Bab....129
Lampiran 8.A — Checklist Self-Check (Tidak Dinilai)....131
Referensi....131

Algoritma Greedy....133
9.1 Rekap: Dari Kuliah 8 ke Kuliah 9....133
9.2 Apa yang Membuat Algoritma Greedy Benar? Dua Syarat Struktural....134
9.3 Soal Activity Selection....135
9.4 Membuktikan Greedy Benar: Argumen Pertukaran....136
9.5 Huffman Coding: Kompresi Prefix-Free Optimal....137
9.6 Membaca Kode dan Menghitung Penghematannya....138
9.7 Fractional Knapsack: Keberhasilan Greedy Kedua....139
9.8 0/1 Knapsack: Di Mana Aturan Greedy yang Sama Gagal....140
9.9 Menganalisis Algoritma Greedy: Ke Mana Sebenarnya Waktu Dihabiskan....141
9.10 Algoritma Greedy dalam Praktik: Di Mana Bab Ini Muncul di Produk Nyata....141
9.11 Rangkuman Bab dan Poin Kunci....142
9.12 Pertanyaan Ulasan....143
Lampiran 9.A — Log Verifikasi untuk Source Code Bab 9....144
Referensi....144

Graf dan Minimum Spanning Tree: Algoritma Kruskal....146
10.1 Rekap: Dari Algoritma Greedy ke Graf....146
10.2 Dasar-Dasar Graf: Simpul, Sisi, dan Representasi....147
10.3 Masalah Minimum Spanning Tree dan Sifat Cut....147
10.4 Algoritma Kruskal....148

10.5 Membuktikan Kruskal Benar melalui Sifat Cut....150
10.6 Struktur Data Union-Find (Disjoint-Set)....151
10.7 Menganalisis Waktu Jalan Kruskal....151
10.8 Algoritma Graf dalam Praktik: Di Mana Bab Ini Muncul dalam Produk Nyata....152
10.9 Ringkasan Bab dan Poin-Poin Kunci....152
10.10 Pertanyaan Tinjauan....153
Lampiran 10.A — Log Verifikasi untuk Kode Sumber Bab 10....155
Referensi....155

Algoritma Prim untuk Minimum Spanning Tree....158
11.1 Rekap: Dari Kruskal ke Prim....158
11.2 Algoritma Prim: Menumbuhkan Satu Pohon....159
11.3 Jejak Prim pada Jaringan Kandidat....159
11.4 Mengapa Prim Mencapai MST yang Sama dengan Kruskal....161
11.5 Membuktikan Prim Benar melalui Sifat Cut....162
11.6 Mengimplementasikan Priority Queue....162
11.7 Menganalisis Waktu Jalan Prim....163
11.8 Profitina dalam Praktik: Memilih Algoritma MST....163
11.9 Ringkasan Bab dan Poin-Poin Kunci....164
11.10 Pertanyaan Tinjauan....165
Lampiran 11.A — Log Verifikasi untuk Kode Sumber Bab 11....166
Referensi....166

Soal Latihan: Algoritma Greedy Hingga Minimum Spanning Tree....168
12.1 Rekap: Kuliah 9–11 secara Ringkas....168
12.2 Cara Memakai Bab Latihan Ini....169
12.3 Soal Latihan....170
12.4 Format Kuis 2: Open-Book, Take-Home Essay....173
12.5 Profitina in Practice: Tinjau-Ulang Sendiri Sebelum Pengumpulan....174
12.6 Ringkasan Bab....174
Lampiran 12.A — Checklist Tinjau-Diri (Non-Dinilai)....176
Referensi....176

Lintasan Terpendek Sumber-Tunggal: Algoritma Dijkstra dan Bellman-Ford....178
13.1 Rekap: Dari Satu Pohon ke Banyak Lintasan....178
13.2 Masalah Lintasan Terpendek dan Optimal Substructure....179
13.3 Algoritma Dijkstra: Finalisasi Greedy....179
13.4 Membuktikan Kebenaran Dijkstra: Mengapa Bobot Non-Negatif Penting....181
13.5 Mengapa Dijkstra Gagal pada Bobot Negatif....182
13.6 Algoritma Bellman-Ford: Relaksasi Semuanya, Berulang Kali....183
13.7 Mendeteksi Siklus Berbobot Negatif....184
13.8 Menganalisis dan Membandingkan Waktu Jalan....185
13.9 Profitina dalam Praktik: Latensi, Kredit, dan Keamanan Siklus....186
13.10 Ringkasan Bab dan Poin-Poin Kunci....186
13.11 Soal Latihan....187
Lampiran 13.A — Log Verifikasi untuk Kode Sumber Bab 13....189
Referensi....189

Analisis Amortisasi....192
14.1 Rekap: Dari Biaya Satu Operasi ke Total Suatu Urutan....192
14.2 Metode Agregat: Tabel Dinamis Berlipat Ganda....193
14.3 Metode Akuntansi: PUSH dan MULTIPOP pada Stack....194

14.4 Metode Potensial: Increment Binary Counter 196
14.5 Membandingkan Ketiga Metode 197
14.6 Profitina dalam Praktik: Cache, Batch, dan Counter Versi 198
14.7 Ringkasan Bab dan Poin-Poin Kunci 198
14.8 Soal Latihan 199
Lampiran 14.A — Log Verifikasi untuk Kode Sumber Bab 14 201
Referensi 201

Lintasan Terpendek Semua-Pasangan 204
15.1 Rekap: Satu Sumber adalah Bab 13. Setiap Pasangan adalah Bab Ini 204
15.2 Rekurensi Floyd-Warshall 205
15.3 Melacak Floyd-Warshall Secara Manual, Pass demi Pass 206
15.4 Merekonstruksi Lintasan Sebenarnya, Bukan Cuma Jaraknya 208
15.5 Algoritma Warshall: Transitive Closure sebagai Kasus Khusus 208
15.6 Floyd-Warshall vs. Menjalankan Dijkstra atau Bellman-Ford V Kali 209
15.7 Profitina dalam Praktik: Matriks Latensi Antar-Hub 210
15.8 Ringkasan Bab dan Poin-Poin Kunci 210
15.9 Soal Tinjauan 211
Lampiran 15.A — Log Verifikasi untuk Kode Sumber Bab 15 213
Referensi 214

Soal Latihan: Lintasan Terpendek Sampai Lintasan Terpendek Semua-Pasangan 216
16.1 Rekap: Kuliah 13–15 Secara Singkat 216
16.2 Cara Memakai Bab Latihan Ini 217
16.3 Soal Latihan 218
16.4 Format Ujian Akhir Semester: Esai Open-Book, Take-Home 222
16.5 Profitina dalam Praktik: Tinjau Ulang Diri Sebelum Menyerahkan 222
16.6 Ringkasan Bab 223
Lampiran 16.A — Checklist Cek-Diri (Tidak Dinilai) 224
Referensi 224

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 1

Fondasi Perancangan dan Analisis Algoritma

Dari masalah dunia nyata menjadi program yang teruji dan berjalan benar: pola pikir yang wajib dimiliki setiap perancang algoritma sebelum menulis satu baris kode pun.

Tujuan Pembelajaran — setelah mempelajari bab ini, mahasiswa mampu:
(1) Menjelaskan perbedaan antara masalah, algoritma, program, dan struktur data. (2) Menyatakan dan menerapkan dua tujuan universal perancangan algoritma — kebenaran (correctness) dan efisiensi (efficiency). (3) Menyebutkan lima paradigma utama perancangan algoritma dan mengenali paradigma mana yang cocok untuk suatu masalah baru. (4) Menerapkan Kerangka Kerja Lima Fase (Pahami, Abstraksi, Rancang, Analisis, Implementasi) untuk menyelesaikan masalah baru dari awal. (5) Membaca dan menulis argumen kebenaran sederhana serta estimasi kompleksitas Big-O untuk algoritma dasar.

Satu Toolchain, Tiga Sistem Operasi — Windows 11, macOS, Ubuntu

OS	Instalasi sekali saja	Kompilasi & jalankan (identik di mana pun)
Windows 11	MinGW-w64: winget install BrechtSanders.WinLibs (atau WSL2: wsl --install)	g++ -Wall -Wextra -std=c++17 -O2 -o prog prog.cpp lalu prog.exe / ./prog
macOS	xcode-select --install (clang Apple menyahut sebagai g++)	g++ -Wall -Wextra -std=c++17 -O2 -o prog prog.cpp lalu ./prog
Ubuntu/Linux	sudo apt update && sudo apt install build-essential	g++ -Wall -Wextra -std=c++17 -O2 -o prog prog.cpp lalu ./prog

Semua contoh di buku ini berjalan identik di Windows 11, macOS, dan Ubuntu/Linux; perintah hanya berbeda saat instalasi. Pilih baris Anda sekali dan ikuti terus di laptop Anda sendiri.

1.1 Tentang Bab Ini

Selamat datang di mata kuliah Perancangan dan Analisis Algoritma (Design and Analysis of Algorithms, DAA). Mata kuliah ini bukan tentang satu bahasa pemrograman tertentu, bukan pula tentang pustaka (library) atau kerangka kerja (framework) tertentu. Mata kuliah ini tentang cara BERPIKIR — mengembangkan cara yang disiplin dan dapat diulang untuk mengubah masalah dunia nyata yang berantakan menjadi solusi yang terbukti benar sekaligus terbukti cepat.

Setelah menyelesaikan mata kuliah ini, mahasiswa mampu: memodelkan masalah komputasional secara presisi, dari suatu skenario dunia nyata dengan masukan dan keluaran yang jelas; merancang algoritma menggunakan strategi yang sesuai — greedy, divide-and-conquer, pemrograman dinamis, atau paradigma lainnya; menganalisis kebenaran dan kompleksitas, yaitu membuktikan bahwa algoritma selalu memberikan jawaban yang benar serta menghitung dengan tepat seberapa cepat algoritma berjalan dan seberapa besar memori yang dibutuhkan; dan mengimplementasikan rancangan tersebut secara bersih, menerjemahkannya menjadi kode yang berjalan dengan struktur data yang sesuai.

Perhatikan urutannya.
Pahami dulu, baru rancang, baru analisis, baru menulis kode. Kode adalah langkah TERAKHIR, bukan yang pertama. Kesalahan paling umum yang dilakukan mahasiswa baru adalah langsung menuju keyboard. Buku ini akan melatih kebiasaan tersebut, karena proses berpikir sebelum menulis kode itulah yang membedakan seorang insinyur yang kebetulan beruntung dari seorang insinyur yang secara konsisten menghasilkan perangkat lunak yang benar dan efisien.

1.2 Algoritma dan Struktur Data — Dua Pilar Utama

1.2.1 Apa Itu Algoritma?

Algoritma adalah prosedur komputasional langkah demi langkah yang mengubah masukan (input) menjadi keluaran (output) — sebuah resep yang presisi: diberikan bahan tertentu (masukan), ikuti langkah tertentu, dan hasilkan hidangan (keluaran). Secara formal, setiap algoritma harus memenuhi lima sifat berikut.

- Masukan (Input) — nol atau lebih nilai yang diberikan dari luar.
- Keluaran (Output) — setidaknya satu nilai dihasilkan sebagai hasil.
- Kepastian (Definiteness) — setiap langkah dirumuskan secara presisi, tanpa ada ruang ambiguitas.
- Keterhinggaan (Finiteness) — prosedur harus berhenti pada akhirnya, untuk setiap masukan yang valid.
- Keefektifan (Effectiveness) — setiap langkah cukup mendasar sehingga, pada prinsipnya, dapat dikerjakan oleh manusia dengan pensil dan kertas.

Program adalah implementasi dari suatu algoritma dalam bahasa pemrograman tertentu — dalam buku ini, C++. Algoritma adalah gagasan; program adalah wujud nyatanya. Banyak program yang berbeda, dalam bahasa yang berbeda-beda, dapat mengimplementasikan algoritma yang persis sama.

1.2.2 Apa Itu Struktur Data?

Struktur data adalah cara mengatur, menyimpan, dan mengelola data agar dapat diakses dan diubah secara efisien. Pemilihan struktur data dapat menjadi pembeda antara algoritma yang selesai dalam hitungan milidetik dan algoritma yang membutuhkan waktu berjam-jam untuk masukan yang sama persis.

Analogi dapur
Algoritma ibarat resep masakan — urutan langkahnya. Struktur data ibarat peralatan dapur — panci, wajan, dan oven. Resep yang persis sama, dikerjakan dengan peralatan yang lebih baik, menghasilkan hidangan yang sama namun lebih cepat. Sepanjang buku ini, gagasan algoritmik yang sama akan berjalan jauh lebih cepat begitu dipasangkan dengan struktur data yang tepat.

Struktur data	Keunggulan utama	Contoh penggunaan di buku ini

Array / vector	Akses terindeks O(1), memori berurutan	Insertion Sort, Binary Search (Bab 1–2)
Linked list	Penyisipan/penghapusan O(1) pada posisi tertentu	Sekuens dinamis, adjacency list
Stack / queue	Akses terbatas LIFO / FIFO	Simulasi rekursi, penelusuran BFS
Hash map (unordered_map)	Pencarian rata-rata O(1) berdasarkan kunci	Memoisasi untuk pemrograman dinamis
BST seimbang (map, set)	Operasi terurut O(log n)	Himpunan dinamis terurut, statistik urutan
Heap / priority queue	Ekstraksi min/max O(log n)	Algoritma Dijkstra, pengodean Huffman
Graf (adjacency list/matrix)	Memodelkan relasi/jaringan	Lintasan terpendek, pohon rentang minimum

1.3 Dua Tujuan Perancangan Algoritma

Setiap kali kita merancang sebuah algoritma, ada tepat dua sifat yang harus dijamin. Setiap teknik dalam buku ini pada akhirnya melayani salah satu dari dua tujuan berikut.

1.3.1 Tujuan 1 — Kebenaran (Correctness)

Sebuah algoritma dikatakan benar jika ia menghasilkan keluaran yang tepat untuk SETIAP kemungkinan masukan yang valid — bukan hanya untuk contoh yang kebetulan diuji, dan bukan hanya untuk masukan berukuran kecil. Terdapat beberapa teknik standar untuk membuktikan kebenaran secara rigor:

- Bukti langsung — menunjukkan bahwa logika algoritma selalu mengarah ke jawaban yang benar.
- Bukti dengan kontradiksi — mengasumsikan algoritma memberi jawaban salah, lalu menurunkan kontradiksi logis.
- Invarian loop (loop invariant) — mengidentifikasi suatu sifat yang berlaku sebelum dan sesudah setiap iterasi loop, lalu menunjukkan bahwa sifat tersebut mengimplikasikan hasil yang benar ketika loop berakhir.
- Induksi — membuktikan kasus dasar, lalu menunjukkan bahwa jika algoritma benar untuk masukan berukuran k, algoritma juga benar untuk ukuran k+1.

Contoh terapan: apa arti "benar" untuk masalah Pengurutan?

Perhatikan masalah pengurutan (sorting): diberikan sekuens $a_1, a_2, \ldots, a_n$, hasilkan permutasi $b_1, b_2, \ldots, b_n$ sedemikian sehingga $b_1 \le b_2 \le \ldots \le b_n$. Algoritma pengurutan yang benar harus memenuhi DUA syarat sekaligus, untuk setiap masukan: keluaran harus terurut, dan keluaran harus merupakan permutasi dari masukan — setiap elemen pada masukan muncul tepat satu kali pada keluaran, tidak ada yang diciptakan maupun dihilangkan.

Jika algoritma Anda mengubah [2, 5, 4, 10, 7] menjadi [2, 4, 5, 7, 10], maka algoritma tersebut benar. Namun jika ia justru menghasilkan [1, 2, 3, 4, 5], keluaran tersebut memang terurut — tetapi bukan permutasi dari masukan, sehingga algoritma tersebut salah, betapapun "rapi" keluarannya terlihat.

1.3.2 Tujuan 2 — Efisiensi (Efficiency)

Algoritma harus menggunakan waktu dan memori seminimal mungkin. Dua algoritma bisa sama-sama benar namun berperilaku sangat berbeda seiring bertambahnya ukuran masukan.

> **Satu juta angka, dua algoritma**
>
> Mengurutkan satu juta angka: Bubble Sort, dengan kompleksitas $O(n^2)$, membutuhkan sekitar 10^{12} operasi — berjam-jam komputasi. Merge Sort, dengan kompleksitas $O(n \log n)$, membutuhkan sekitar 2×10^7 operasi — hanya milidetik. Keduanya benar. Satu di antaranya kurang lebih sejuta kali lebih cepat.

Gambar 1.1 — Bagaimana jumlah operasi bertumbuh terhadap ukuran masukan n untuk enam kelas kompleksitas umum. Perhatikan bagaimana $O(n^2)$ dan $O(2^n)$ meledak, sementara $O(\log n)$ dan $O(n)$ tetap nyaris datar.

> **Aturan praktis**
>
> Komputer modern mampu menjalankan sekitar 10^8 hingga 10^9 operasi sederhana per detik. Jika algoritma Anda melakukan lebih dari sekitar 10^8 operasi untuk ukuran masukan sesuai batasan (constraints) suatu soal, algoritma tersebut sangat mungkin melampaui batas waktu yang wajar — baik dalam tugas kuliah, online judge lomba pemrograman, maupun kode produksi.

1.3.3 Kedua Tujuan Bekerja Bersama

Kebenaran tanpa efisiensi tidak berguna — program pada akhirnya melampaui batas waktu atau kehabisan memori. Efisiensi tanpa kebenaran lebih buruk lagi — program yang cepat namun memberi

jawaban salah dapat menimbulkan kerugian nyata. Anda membutuhkan keduanya secara bersamaan, dan seluruh isi buku ini membahas cara mencapai keduanya sekaligus.

1.4 Paradigma Perancangan Algoritma

Tidak ada satu resep tunggal untuk merancang algoritma. Sebaliknya, ilmu komputer telah mengumpulkan beberapa strategi umum yang ampuh — paradigma — yang masing-masing cocok untuk kategori masalah yang luas. Mengenali paradigma mana yang dibutuhkan suatu masalah baru adalah salah satu keterampilan paling berharga yang akan diberikan mata kuliah ini.

Paradigma	Gagasan inti	Contoh baku
Incremental (brute force)	Memproses elemen satu per satu sambil menjaga status berjalan.	Menelusuri array untuk mencari nilai maksimum.
Greedy	Pada setiap langkah, buat pilihan optimal lokal dan berharap mengarah ke optimum global.	Algoritma Kruskal untuk pohon rentang minimum.
Divide and conquer	Pecah masalah menjadi sub-masalah lebih kecil dengan tipe sama, selesaikan secara rekursif, gabungkan.	Merge Sort.
Pemrograman dinamis	Seperti divide-and-conquer, tetapi sub-masalah SALING TUMPANG TINDIH; simpan hasil untuk hindari komputasi ulang.	Menghitung bilangan Fibonacci, atau Soal 6 di bawah.
Matematis / teori bilangan	Memanfaatkan bentuk tertutup atau identitas aljabar yang melewati brute force sepenuhnya.	Algoritma Euclid untuk GCD.

1.5 Kerangka Kerja Pemecahan Masalah Lima Fase

Setiap masalah dalam buku ini — bahkan setiap masalah yang akan Anda hadapi sebagai perancang algoritma — diselesaikan menggunakan kerangka kerja lima fase yang sama. Menginternalisasi kerangka kerja ini adalah hasil paling berharga dari bab ini.

Gambar 1.2 — Kerangka Kerja Lima Fase yang digunakan di seluruh buku ini. Fase Analisis dapat mengembalikan Anda ke fase Rancang; alur putar-balik ini wajar dan memang diharapkan, bukan sebuah kegagalan.

- Fase 1 — PAHAMI: baca soal, identifikasi masukan/keluaran/batasan, tuliskan ulang dengan kata-kata sendiri, dan buat contoh kecil secara manual.
- Fase 2 — ABSTRAKSI: lepaskan cerita/bungkus soal. Apa inti matematisnya? Sifat apa yang menentukan jawabannya? Di sinilah proses berpikir yang sesungguhnya terjadi.
- Fase 3 — RANCANG: pilih paradigma dari Bagian 1.4, tuliskan algoritma dalam pseudocode atau bahasa biasa, dan tentukan struktur data yang dibutuhkan.
- Fase 4 — ANALISIS: buktikan kebenaran (apakah algoritma bekerja untuk SEMUA masukan, bukan hanya yang telah dicoba?), hitung kompleksitas waktu dan ruang, dan periksa apakah sesuai dengan batasan soal.
- Fase 5 — IMPLEMENTASI: tulis kode sebagai terjemahan langsung dan bersih dari rancangan Fase 3. Jika kode terlihat rumit, kemungkinan rancangannya perlu disederhanakan terlebih dahulu.

> **Kesalahan paling umum mahasiswa**
> Sebagian besar mahasiswa langsung meloncat ke Fase 5. Tahan keinginan tersebut. Empat fase pertama itulah yang membuat Fase 5 menjadi nyaris trivial — melewatkannya justru membuat Fase 5 terasa sulit.

1.6 Kerangka Kerja dalam Aksi: Enam Soal Diselesaikan dari Awal

Bagian sisa bab ini menerapkan Kerangka Kerja Lima Fase pada enam soal nyata dari online judge E-Olymp dan SPOJ — online judge yang sama yang akan digunakan untuk soal-soal latihan mata kuliah ini (lihat latihan Bab 4 dan panduan kumpulan soal SPOJ pendamping). Setiap soal mengilustrasikan paradigma yang berbeda dari Bagian 1.4. Kode sumber C++ yang lengkap, telah dikompilasi, dan telah diuji untuk setiap soal tersedia pada paket kode pendamping (`code/chapter01/`).

1.6.1 Soal 1: Columns (E-Olymp 10281) — Greedy / Komputasi Langsung

Fase 1 — Pahami

Masukan: n kolom kubus satuan, kolom ke-i memiliki tinggi a_i ($1 \le a_i \le 1000$, $1 \le n \le 1000$). Keluaran: jumlah minimum warna yang dibutuhkan untuk mewarnai semua kubus sedemikian sehingga setiap deret horizontal kubus yang bersebelahan (tanpa celah) dan setiap kolom mendapat warna yang semuanya berbeda.

```
Baris 6: [X]
Baris 5: [X][X]
Baris 4: [X][X]      [X]      <- celah di kolom 3 (tinggi 2 < 4)
Baris 3: [X][X]      [X]
Baris 2: [X][X][X][X]         <- keempat kolom hadir
Baris 1: [X][X][X][X]
         kol1 kol2 kol3 kol4   tinggi = [6, 5, 2, 4]
```

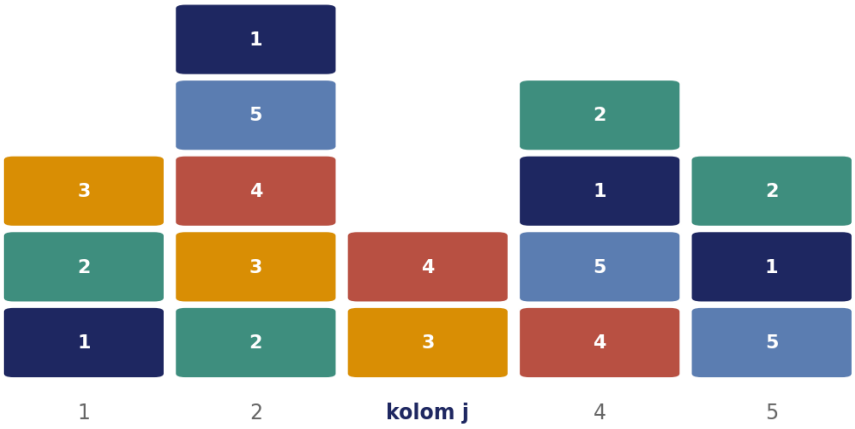
Fase 2 — Abstraksi

Dua batasan harus dipenuhi sekaligus. Secara vertikal, kolom setinggi h membutuhkan h warna berbeda, sehingga dibutuhkan minimal $\max(a_i)$ warna. Secara horizontal, baris 1 selalu memuat seluruh n kolom, membentuk satu deret sepanjang n, sehingga dibutuhkan minimal n warna. Misalkan $C = \max(n, \max(a_i))$. Mewarnai kubus pada (kolom j, baris r) dengan warna $((j + r) \bmod C) + 1$ menjamin warna berbeda di dalam kolom mana pun (paling banyak setinggi C kubus) maupun di dalam deret horizontal mana pun (paling banyak selebar C kubus, karena tidak mungkin melebihi n).

> **Wawasan kunci**
> Jawabannya persis $\max(n, \max(a_1, \ldots, a_n))$ — C warna terbukti perlu (dari dua batasan di atas) sekaligus cukup (dari konstruksi eksplisit di atas).

Soal 1 · Columns — argumen pewarnaan

Warnai kubus di (kolom j, baris r) dengan ((j + r) mod C) + 1. C = maks(n, tinggi maksimum).



Gambar 1.3 — Konstruksi pewarnaan: kubus (kolom j, baris r) diwarnai ((j + r) mod C) + 1. Tidak ada kolom maupun deret horizontal yang mengulang warna.

Fase 3 — Rancang

Baca n. Inisialisasi jawaban = n. Untuk setiap tinggi a_i, tetapkan jawaban = max(jawaban, a_i). Cetak jawaban.

Fase 4 — Analisis

Waktu: O(n) — satu kali pemindaian dari kiri ke kanan. Ruang: O(1). Kebenaran: terbukti di atas melalui konstruksi eksplisit.

Fase 5 — Implementasi

```cpp
/*
 * Chapter 1 - Problem 1: Columns  (E-Olymp 10281)
 * Paradigm : Greedy / Direct Computation
 *
 * Pseudocode:
 *   read n
 *   answer <- n
 *   for each height a_i:
 *       answer <- max(answer, a_i)
 *   print answer
 *
 * Proof of correctness (see textbook Section 1.6.1):
 *   Let C = max(n, max(a_i)). Coloring cube at (column j, row r) with
 *   color ((j + r) mod C) + 1 guarantees every column (<= C cubes) and
 *   every horizontal run (<= C cubes, bounded by n) gets distinct colors.
 *   Hence C colors are sufficient, and C is clearly necessary, so the
 *   answer is exactly C.
 *
 * Complexity: Time O(n), Space O(1).
 */
#include <cstdio>
#include <algorithm>
using namespace std;

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    int answer = n;
    for (int i = 0; i < n; i++) {
        int height;
        scanf("%d", &height);
        answer = max(answer, height);
    }
    printf("%d\n", answer);
    return 0;
}
```

1.6.2 Soal 2: Journey (E-Olymp 10280) — Greedy / Observasi Matematis

Fase 1 — Pahami

Masukan: n kota terletak pada satu garis lurus, kota ke-i pada koordinat x_i ($1 \le x_i \le 1000$, $1 \le n \le 100$). Tugas: mulai dari kota pilihan mana pun, kunjungi setiap kota minimal satu kali dan kembali ke titik awal, dengan total jarak tempuh seminimal mungkin.

Fase 2 — Abstraksi

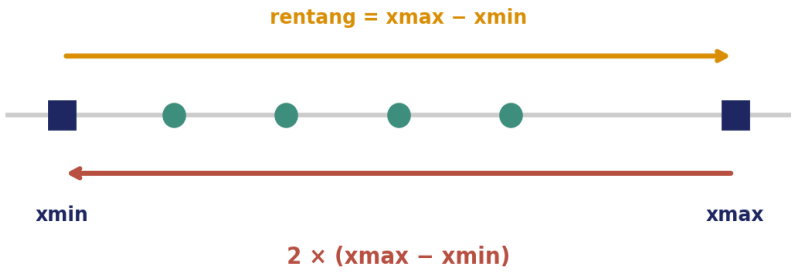
Lepaskan bungkus ceritanya: kita memiliki n titik pada garis bilangan dan membutuhkan tur tertutup terpendek yang mengunjungi semuanya. Pada garis, pergerakan hanya bisa ke kiri atau ke kanan, dan setiap pembalikan arah membuang jarak; rencana optimalnya adalah bergerak dari satu titik ekstrem ke titik ekstrem lainnya lalu kembali. Misalkan x_{min} dan x_{maks} adalah koordinat terkecil dan terbesar. Menelusuri $[x_{min}, x_{maks}]$ pada kedua arah mengunjungi semua titik, dan secara aljabar posisi awal saling meniadakan, sehingga totalnya selalu $2 \times (x_{maks} - x_{min})$.

Wawasan kunci

Jawaban = 2 × (max(x_i) − min(x_i)). Tidak dibutuhkan pencarian atas kota awal maupun urutan kunjungan — geometri garis membuat solusi optimal menjadi ekspresi bentuk tertutup.

Soal 2 · Journey — menyusuri rentang dua kali

Setiap perjalanan tertutup yang mengunjungi seluruh titik di [xmin, xmax] melintasi rentang itu minimal dua kali.



Gambar 1.4 — Setiap perjalanan tertutup pada garis harus melintasi rentang [xmin, xmax] minimal dua kali — sekali pergi, sekali kembali.

Fase 3 & 4 — Rancang dan Analisis

Lacak dmin dan dmax berjalan sambil membaca koordinat dalam satu pemindaian; cetak 2 × (dmax − dmin). Waktu O(n), ruang O(1). Kebenaran: tur tertutup mana pun pada garis harus menelusuri seluruh rentang [dmin, dmax] minimal dua kali — sekali untuk masing-masing arah — dan rencana ini persis mencapai batas bawah tersebut.

Fase 5 — Implementasi

```cpp
/*
 * Chapter 1 - Problem 2: Journey  (E-Olymp 10280)
 * Paradigm : Greedy / Mathematical Observation
 *
 * Pseudocode:
 *   read n
 *   track dmin, dmax over all coordinates
 *   print 2 * (dmax - dmin)
 *
 * Proof of correctness: On a 1-D line, any closed tour visiting every
 * point in [xmin, xmax] must traverse that interval at least twice
 * (once in each direction), and traversing it exactly twice suffices.
 * The starting point cancels out algebraically.
 *
 * Complexity: Time O(n), Space O(1).
 */
#include <cstdio>
#include <algorithm>
using namespace std;

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    int dmin = 1000000, dmax = -1000000;
    for (int i = 0; i < n; i++) {
        int x;
        scanf("%d", &x);
        dmin = min(dmin, x);
        dmax = max(dmax, x);
    }
    printf("%d\n", 2 * (dmax - dmin));
    return 0;
}
```

1.6.3 Soal 3: Unique Numbers (SPOJ DCEPC901 / 12073) — Teori Bilangan (GCD)

Fase 1 — Pahami

Mulai dengan himpunan {0, 1, …, n−1}. Lakukan k operasi secara berurutan, masing-masing bekerja atas hasil operasi sebelumnya: tipe 0 dengan nilai x mengganti setiap angka i menjadi (i + x) mod n; tipe 1 dengan nilai x mengganti setiap angka i menjadi (i × x) mod n. Setelah setiap operasi, laporkan berapa banyak angka berbeda (unik) yang tersisa.

Fase 2 — Abstraksi

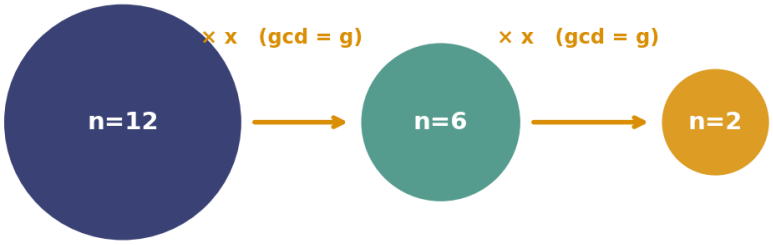
Penjumlahan dengan konstanta modulo n adalah suatu bijeksi — jika a ≠ b maka a+x ≠ b+x (mod n) — sehingga penjumlahan tidak pernah mengubah jumlah nilai berbeda. Perkalian berbeda: bayangan (image) dari f(i) = i×x mod n, untuk i = 0..n−1, persis merupakan himpunan kelipatan dari g = gcd(n, x), yaitu {0, g, 2g, …, n−g}, yang beranggotakan n/g elemen. Dengan merangkai operasi-operasi tersebut, kita cukup melacak satu variabel “ukuran efektif berjalan” n′: setiap operasi kali dengan x menetapkan n′ ← n′/gcd(n′, x), dan setiap operasi tambah membiarkan n′ tidak berubah.

Wawasan kunci

Lacak n sebagai satu variabel berjalan. Pada operasi kali, n ← n / gcd(n, x); pada operasi tambah, n tidak berubah. Cetak n setelah setiap operasi.

Soal 3 · Unique Numbers — perkalian mempersempit semesta

Mengalikan dengan x memetakan {0..n−1} ke kelipatan gcd(n, x): semesta menyusut menjadi n / gcd(n, x).



Gambar 1.5 — Setiap operasi kali-dengan-x menyusutkan semesta nilai berbeda dari n menjadi n / gcd(n, x); operasi tambah tidak mengubahnya.

Fase 3 & 4 — Rancang dan Analisis

Baca n dan k. Untuk setiap operasi (p, x): jika p = 1 (kali), perbarui n ← n / gcd(n, x); cetak n pada kedua kasus. Waktu: O(k log n) — satu komputasi GCD per operasi. Ruang: O(1).

Fase 5 — Implementasi

```cpp
/*
 * Chapter 1 - Problem 3: Unique Numbers  (SPOJ DCEPC901 / 12073)
 * Paradigm : Number Theory (GCD)
 *
 * Key insight: addition mod n is a bijection (never changes the count
 * of distinct values). Multiplication by x maps {0..n-1} onto the
 * multiples of gcd(n, x), shrinking the universe to n / gcd(n, x).
 *
 * Pseudocode:
 *   read n, k
 *   for each operation (p, x):
 *       if p == 1: n <- n / gcd(n, x)
 *       print n
 *
 * Complexity: Time O(k log n), Space O(1).
 */
#include <cstdio>
#include <algorithm>
using namespace std;

int main() {
    long long n, k, p, x;
    if (scanf("%lld %lld", &n, &k) != 2) return 0;
    while (k--) {
        scanf("%lld %lld", &p, &x);
        if (p & 1) n /= __gcd(n, x);
        printf("%lld\n", n);
    }
    return 0;
}
```

1.6.4 Soal 4: Minimum & Maximum Numbers (SPOJ 7667) — Konstruksi Digit Greedy

Fase 1 — Pahami

Untuk setiap T kasus uji, diberikan N (banyak digit, $1 \le N \le 18$) dan K (banyak digit berbeda yang dibutuhkan, $1 \le K \le 10$), cetak bilangan bulat positif N-digit terkecil dan terbesar (tanpa nol di depan) yang menggunakan tepat K digit berbeda.

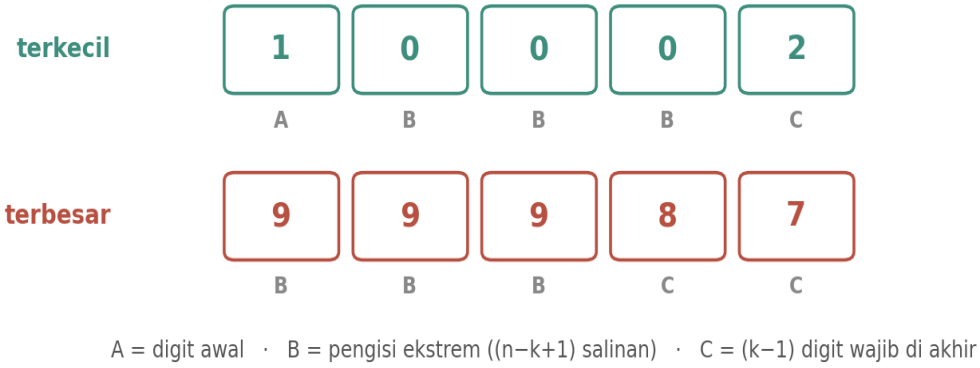
Fase 2 — Abstraksi

Jika K = 1, jawabannya adalah satu digit yang diulang N kali — digit tersebut tidak boleh 0 karena aturan larangan nol di depan, sehingga nilai minimum adalah 111…1 dan nilai maksimum adalah 999…9. Untuk K ≥ 2, besaran (magnitude) suatu bilangan hampir seluruhnya ditentukan oleh digit-digit paling signifikan, sehingga strategi greedy dapat diterapkan: untuk nilai minimum, tempatkan digit awal terkecil yang dapat dipakai (1), isi (N−K+1) posisi berikutnya dengan 0, lalu tempatkan sisa K−2 digit berbeda yang dibutuhkan (2, 3, …, K−1) di posisi paling akhir, tempat pengaruhnya terhadap besaran paling kecil. Untuk nilai maksimum, cerminkan strategi tersebut dengan digit 9.

> **Verifikasi (N=5, K=3)**
> Minimum: 1 + 000 + 2 = 10002 (digit yang dipakai: {0,1,2}, tepat 3 digit berbeda). Maksimum: 999 + 87 = 99987 (digit yang dipakai: {7,8,9}, tepat 3 digit berbeda). Keduanya sesuai.

Soal 4 · Min / Max Numbers — nilai ekstrem di ujung

N = 5 digit, K = 3 digit berbeda: digit awal + pengisi ekstrem, lalu digit wajib ditambahkan di akhir.



Gambar 1.6 — Urutan penempatan digit untuk N=5, K=3: digit awal dan pengisi ekstrem ditempatkan lebih dulu (paling menentukan nilai), digit wajib yang berbeda ditambahkan di akhir.

Fase 3 & 4 — Rancang dan Analisis

Waktu: O(N) per kasus uji, dibangun melalui penempatan digit langsung, bukan pencarian. Ruang: O(1). Kebenaran mengikuti prinsip umum bahwa pilihan greedy pada digit paling signifikan mendominasi besaran bilangan, sehingga menetapkannya lebih dulu dan hanya menyentuh digit paling tidak signifikan untuk memenuhi syarat banyaknya digit berbeda tidak dapat dikalahkan oleh susunan lain mana pun.

Fase 5 — Implementasi

```cpp
/*
 * Chapter 1 - Problem 4: Minimum & Maximum Numbers  (SPOJ 7667)
 * Paradigm : Greedy Digit Construction
 *
 * Given N (digit count) and K (distinct digit count), construct the
 * smallest and largest N-digit numbers (no leading zero) using exactly
 * K distinct digits, by placing extreme digits at the most significant
 * positions and introducing the remaining required digits last.
 *
 * Complexity: Time O(N) per test case, Space O(1).
 */
#include <cstdio>
using namespace std;

int main() {
    int t;
    if (scanf("%d", &t) != 1) return 0;
    while (t--) {
        int n, k;
        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;
}
```

1.6.5 Soal 5: Interesting Sequence (E-Olymp 4894) — Manipulasi Bit / Pola

Fase 1 — Pahami

Sebuah sekuens dibangun mulai dari [0]: secara berulang, salin sekuens saat ini, ganti setiap 0→1, 1→2, 2→0 pada salinan tersebut, lalu gabungkan (append) ke sekuens asli — menghasilkan [0] → [0,1] → [0,1,1,2] → [0,1,1,2,1,2,2,0] → …. Diberikan m (berindeks mulai 1, hingga $2^{63}-1$), tentukan elemen ke-m.

Fase 2 — Abstraksi

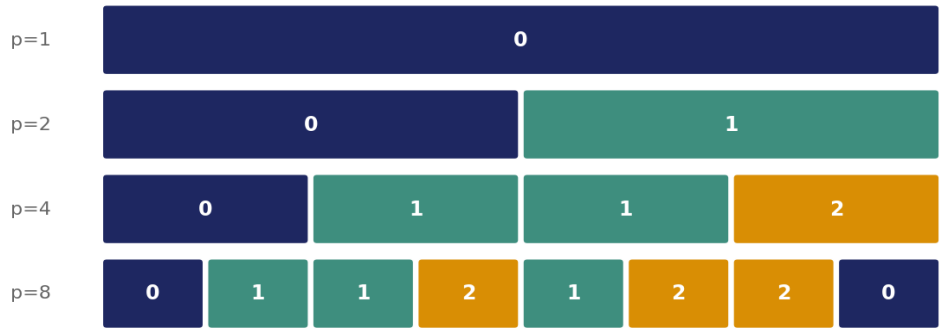
Aturan penggantian 0→1→2→0 sederhananya adalah “tambah 1 mod 3.” Karena sekuens berlipat ganda pada setiap langkah pembangunan, posisi berindeks-0 p dapat ditelaah bit demi bit: setiap kali bit paling signifikan p saat ini bernilai 1, kita berada pada “separuh kedua” yang dihasilkan langkah salin-dan-tambah, menyumbang satu +1 (mod 3) lagi. Menjumlahkan kontribusi ini atas setiap bit-1 dari p menghasilkan value(p) = popcount(p) mod 3.

Verifikasi

p=0 (000_2): popcount=0 → 0. p=3 (011_2): popcount=2 → 2. p=7 (111_2): popcount=3 → 0. Ketiganya sesuai dengan nilai sekuens hasil konstruksi langsung.

Soal 5 · Interesting Sequence — popcount mod 3

Konstruksi penggandaan: salin urutan, tambah 1 (mod 3) pada salinan → value(p) = popcount(p) mod 3.



Gambar 1.7 — Setiap langkah penggandaan menyalin baris dan menambah 1 (mod 3) pada salinannya, sehingga value(p) selalu sama dengan popcount(p) mod 3.

Fase 3 & 4 — Rancang dan Analisis

Ubah kueri m yang berindeks-1 menjadi posisi berindeks-0 p = m−1, hitung popcount(p) menggunakan instruksi popcount perangkat keras, lalu cetak hasilnya mod 3. Waktu: O(1) per kueri. Ruang: O(1).

Fase 5 — Implementasi

```cpp
/*
 * Chapter 1 - Problem 5: Interesting Sequence  (E-Olymp 4894)
 * Paradigm : Bit Manipulation / Mathematical Pattern
 *
 * The sequence doubles each step by copying itself with every value
 * incremented mod 3. This is equivalent to: value(p) = popcount(p) mod 3
 * for 0-indexed position p. (See textbook Section 1.6.5 for the full
 * derivation from the recursive construction.)
 *
 * Complexity: Time O(1) per query (popcount is a single CPU instruction).
 */
#include <cstdio>
using namespace std;
typedef unsigned long long ull;

int main() {
    long long q;
    if (scanf("%lld", &q) != 1) return 0;
    while (q--) {
        unsigned long long x;
        scanf("%llu", &x);
        int pc = __builtin_popcountll(x - 1);
        printf("%d\n", pc % 3);
    }
    return 0;
}
```

1.6.6 Soal 6: Recursive Function 1 (E-Olymp 10296) — Pemrograman Dinamis / Memoisasi

Fase 1 — Pahami

Sebuah fungsi didefinisikan dengan f(0) = 1 dan, untuk n > 0, f(n) = f(⌊n/2⌋) + f(⌊n/3⌋). Diberikan satu bilangan bulat n (hingga 10^{18}), hitung f(n) dalam batas waktu satu detik.

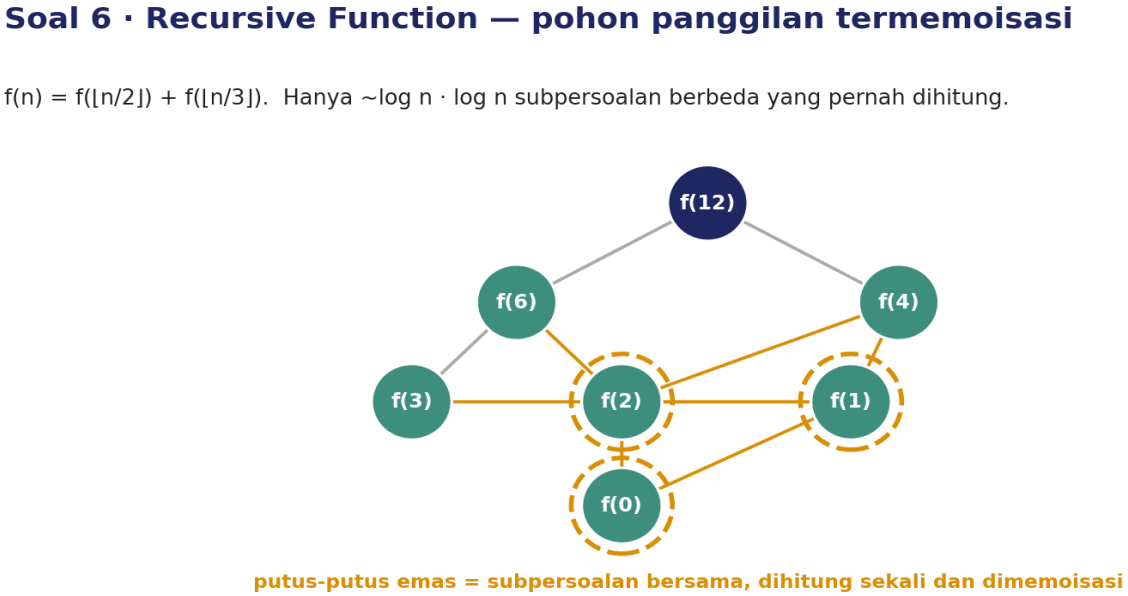
Fase 2 — Abstraksi

Rekursi naif bercabang menjadi dua pemanggilan pada setiap tingkat, yang bersifat eksponensial tanpa caching. Pengamatan kuncinya adalah setiap nilai yang pernah dikueri berbentuk $\lfloor n / (2^a \times 3^b) \rfloor$ untuk suatu bilangan bulat non-negatif a dan b, dengan a dibatasi sekitar $\log_2(10^{18}) \approx 60$ dan b sekitar $\log_3(10^{18}) \approx 38$. Ini memberikan paling banyak sekitar $60 \times 38 \approx 2.280$ sub-masalah berbeda — sangat kecil, walaupun n sendiri bisa sangat besar. Array yang diindeks oleh n mustahil digunakan (membutuhkan 10^{18} entri), tetapi hash map yang hanya menyimpan sub-masalah yang benar-benar dikunjungi membuat memoisasi menjadi mudah.

```
f(12) = f(6) + f(4)
  f(6) = f(3) + f(2)
    f(3) = f(1) + f(1)
      f(1) = f(0) + f(0) = 1 + 1 = 2      [simpan f(1)=2]
    f(3) = 2 + 2 = 4                      [simpan f(3)=4]
    f(2) = f(1) + f(0) = 2 + 1 = 3        [simpan f(2)=3]
  f(6) = 4 + 3 = 7                        [simpan f(6)=7]
  f(4) = f(2) + f(1) = 3 + 2 = 5          [keduanya sudah tersimpan]
f(12) = 7 + 5 = 12
```

Wawasan kunci

Walaupun n mencapai 10^{18}, hanya ada sekitar 2.000 sub-masalah berbeda yang pernah muncul. Tabel memo berbasis map mengubah rekursi eksponensial menjadi komputasi yang cepat dan kecil.



Gambar 1.8 — Pohon panggilan untuk f(12). Simpul putus-putus emas dicapai lewat lebih dari satu jalur — persis subpersoalan yang diselamatkan memoisasi dari penghitungan ulang.

Fase 3 & 4 — Rancang dan Analisis

Waktu: O(log²n · log log n) — sekitar 2.000 sub-masalah, masing-masing dengan satu pencarian map O(log). Ruang: O(log²n) untuk tabel memo.

Fase 5 — Implementasi

```cpp
/*
 * Chapter 1 - Problem 6: Recursive Function 1  (E-Olymp 10296)
 * Paradigm : Dynamic Programming / Memoization
 *
 * f(0) = 1, f(n) = f(floor(n/2)) + f(floor(n/3)) for n > 0.
 * Although n can be as large as 10^18, the number of DISTINCT
 * sub-problems reachable from n is only ~ log2(n) * log3(n) ~ 2,280,
 * so memoizing with a map (not an array) makes this trivially fast.
 *
 * Complexity: Time O(log^2 n * log log n), Space O(log^2 n).
 */
#include <cstdio>
#include <map>
using namespace std;
typedef long long ll;

map<ll, ll> memo;

ll f(ll u) {
    if (memo.count(u)) return memo[u];
    ll result = f(u / 2) + f(u / 3);
    memo[u] = result;
    return result;
}

int main() {
    memo[0] = 1;
    ll n;
    if (scanf("%lld", &n) != 1) return 0;
    printf("%lld\n", f(n));
    return 0;
}
```

1.7 Pemetaan Setiap Soal ke Paradigma Perancangan

Salah satu tujuan utama mata kuliah ini adalah melatih kemampuan mengenali, dengan cepat, paradigma mana yang dibutuhkan suatu soal baru. Tabel berikut merangkum keenam soal yang baru saja diselesaikan.

Soal	Paradigma	Sinyal pengenal
Columns	Greedy / komputasi langsung	Jawaban tereduksi menjadi satu fungsi max() atas dua batasan sederhana.
Journey	Greedy / observasi matematis	Geometri 1 dimensi; hanya dua arah pergerakan yang mungkin.
Unique Numbers	Teori bilangan (GCD)	Operasi modular berulang; lacak satu invarian, bukan seluruh status.
Min & Max Numbers	Konstruksi digit greedy	Besaran bilangan didominasi oleh digit paling signifikan.
Interesting Sequence	Manipulasi bit / pola	Konstruksi pelipatgandaan rekursif; cari rumus bit tertutup.
Recursive Function 1	Pemrograman dinamis / memoisasi	Pemanggilan rekursif dengan sedikit argumen BERBEDA yang benar-benar tercapai.

1.8 Profitina dalam Praktik: Di Mana Bab Ini Muncul pada Produk Nyata

> **Tentang kotak ini**
> Sepanjang buku ini, kotak seperti ini menghubungkan materi kuliah dengan Profitina, sebuah produk AI Visibility (Generative Engine Optimization) nyata buatan Indonesia yang dikembangkan penulis. Kotak ini menjelaskan DI MANA dan MENGAPA suatu konsep dipakai dalam sistem yang benar-benar berjalan — bukan detail implementasi, logika bisnis, atau kode sumber proprietary Profitina, yang tetap bersifat confidential. Lihat dokumen pendamping 'Profitina: Tinjauan Teknis Non-Confidential' untuk gambaran lengkapnya.

Mesin pemindaian (scanning engine) Profitina harus berulang kali memutuskan, untuk suatu nama bisnis dan domain, strategi ekstraksi sinyal (signal-extraction) mana yang dijalankan lebih dulu — sebuah contoh sehari-hari dari pemilihan paradigma. Pemeriksaan yang murah namun bernilai tinggi diperlakukan secara greedy (Bagian 1.4): jalankan dulu pemeriksaan yang paling mungkin mengonfirmasi atau menyingkirkan suatu hipotesis, persis penalaran yang sama yang membenarkan langkah greedy pada soal Columns dan Journey di atas. Ketika Profitina harus menghindari pengulangan komputasi mahal pada banyak kueri bisnis yang serupa (misalnya, mengestimasi ulang sinyal kategori atau lokal untuk bisnis-bisnis di kota dan sektor yang sama), sistem menggunakan cache bergaya memoisasi yang diberi kunci berdasarkan sidik jari (fingerprint) masukan yang telah dinormalisasi — secara konsep sama dengan tabel memo berbasis map pada Soal 6, hanya diterapkan pada beban kerja produksi, bukan pada online judge lomba pemrograman. Kebenaran dan efisiensi

(Bagian 1.3) diperlakukan sebagai persyaratan yang sama-sama tidak dapat ditawar dalam praktik rekayasa Profitina sendiri: pemindaian yang berjalan cepat namun mengembalikan sinyal yang salah sama sekali tidak dapat diterima seperti halnya pemindaian yang benar namun terlalu lambat untuk melayani pengguna secara interaktif.

1.9 Ringkasan Bab dan Poin-Poin Kunci

- Sebuah algoritma harus BENAR sekaligus EFISIEN — salah satu saja tidak cukup.
- Selalu ikuti Lima Fase: Pahami → Abstraksi → Rancang → Analisis → Implementasi. Jangan pernah langsung meloncat ke kode.
- Fase ABSTRAKSI adalah tempat proses berpikir yang sesungguhnya terjadi: reduksi masalah ke inti matematisnya.
- Kenali paradigma Anda — greedy, divide-and-conquer, pemrograman dinamis, teori bilangan, manipulasi bit — masing-masing memiliki sinyal pengenal.
- Struktur data yang tepat dapat mengubah masalah yang mustahil menjadi trivial.

"Esensi perancangan algoritma bukanlah menghafal solusi. Esensinya adalah belajar melihat struktur matematis tersembunyi dalam suatu masalah — abstraksi yang mereduksi sesuatu yang kompleks menjadi sesuatu yang sederhana."

1.10 Pertanyaan Tinjauan

Pertanyaan berikut untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Kuliah 2.

1. Berikan contoh (selain pengurutan) suatu masalah di mana algoritma yang salah-tetapi-cepat dan algoritma yang benar-tetapi-lambat sama-sama tampak menggoda untuk dikumpulkan, dan jelaskan mana yang akan Anda pilih beserta alasannya.
2. Pada soal Columns, mengapa tidak cukup hanya menjamin batasan vertikal (warna ≥ tinggi kolom maksimum)? Buatlah contoh kecil yang menunjukkan bahwa mengabaikan batasan horizontal memberikan warna yang terlalu sedikit.
3. Pada Soal 3 (Unique Numbers), mengapa operasi TAMBAH tidak pernah mengubah jumlah nilai berbeda, sedangkan operasi KALI dapat mengubahnya? Jawab dalam istilah bijeksi.
4. Untuk Soal 6 (Recursive Function 1), estimasikan berapa banyak sub-masalah berbeda yang akan ada jika relasi rekurensinya justru $f(n) = f(\lfloor n/2 \rfloor) + f(\lfloor n/2 \rfloor)$ (kedua suku dibagi 2). Apakah memoisasi akan tetap membantu sebesar sebelumnya? Mengapa atau mengapa tidak.
5. Identifikasi satu bagian dari sistem yang Anda gunakan sehari-hari (aplikasi peta, aplikasi pesan, mesin pencari) yang Anda duga menggunakan strategi greedy, dan jelaskan apa yang mungkin menjadi 'pilihan optimal lokal'-nya.

Lampiran 1.A — Catatan Verifikasi Kode Sumber Bab 1

Setiap program yang disertakan dalam buku ini dikompilasi dan dijalankan terhadap contoh yang telah diverifikasi secara manual sebelum disertakan, sebagai kebijakan tetap. Keenam program di bawah ini dikompilasi dengan `g++ -O2 -std=c++17 -Wall` (hanya peringatan scanf yang tidak berbahaya, nol error) dan menghasilkan keluaran yang persis sesuai prediksi pada Bagian 1.6.

```
$ printf "4\n6 5 2 4\n" | ./01_columns
6                                   # ekspektasi 6 = max(n=4, max(a_i)=6)

$ printf "3\n5 1 3\n" | ./02_journey
8                                   # ekspektasi 2*(5-1) = 8

$ printf "6 2\n1 2\n0 1\n" | ./03_unique_numbers
3
3                                   # ekspektasi 6/gcd(6,2)=3, tidak berubah
oleh tambah

$ printf "1\n5 3\n" | ./04_min_max_numbers
10002 99987                         # sesuai verifikasi terapan Bagian 1.6.4

$ printf "3\n1 4 8\n" | ./05_interesting_sequence
0
2
0                                   # ekspektasi popcount(m-1) mod 3 untuk m =
1, 4, 8

$ printf "12\n" | ./06_recursive_function
12                                  # sesuai penelusuran f(12) pada Bagian
1.6.6
```

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Bab 1–4).

[2] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, edisi ke-4. Lulu Press, 2020–2023.

[3] R. Soelaiman, S. Candra, M. M. 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 (JATIT), Vol. 97, No. 8, 2019.

[4] R. Soelaiman, Y. H. Pratama, M. M. I. Subakti, Y. Purwananto. “Genetic Algorithm Approach for Automated Generation of Neural Networks Architecture for Robust Digit Recognition.” JATIT, Vol. 98, No. 17, 2020.

[5] E-Olymp Online Judge, https://www.e-olymp.com (soal yang diacu: 10281, 10280, 4894, 10296).

[6] SPOJ (Sphere Online Judge), https://www.spoj.com (soal yang diacu: DCEPC901/12073, 7667).

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 2

Menganalisis Algoritma: Model RAM, Insertion Sort, dan Pencarian

Cara mengukur "cepat" tanpa stopwatch, mengapa Insertion Sort adalah studi kasus pertama yang sempurna, dan bagaimana larik terurut mengubah pencarian O(n) menjadi O(log n).

Tujuan Pembelajaran — setelah bab ini Anda mampu:
(1) Menjelaskan model RAM dan menggunakannya untuk menurunkan fungsi waktu-eksekusi T(n) suatu algoritma sederhana. (2) Menyatakan dan membuktikan invarian loop untuk algoritma iteratif, dengan Insertion Sort sebagai contoh utama. (3) Membedakan waktu eksekusi kasus terbaik, terburuk, dan rata-rata, serta menjelaskan mengapa mata kuliah ini berfokus pada kasus terburuk. (4) Membandingkan laju pertumbuhan (O(1), O(log n), O(n), O(n log n), $O(n^2)$, $O(2^n)$) dan memprediksi algoritma mana yang selesai dalam batas waktu. (5) Mengimplementasikan dan membandingkan Linear Search dan Binary Search, serta menjelaskan ide divide-and-conquer di balik yang terakhir.

2.1 Rekap: Dari Kuliah 1 ke Kuliah 2

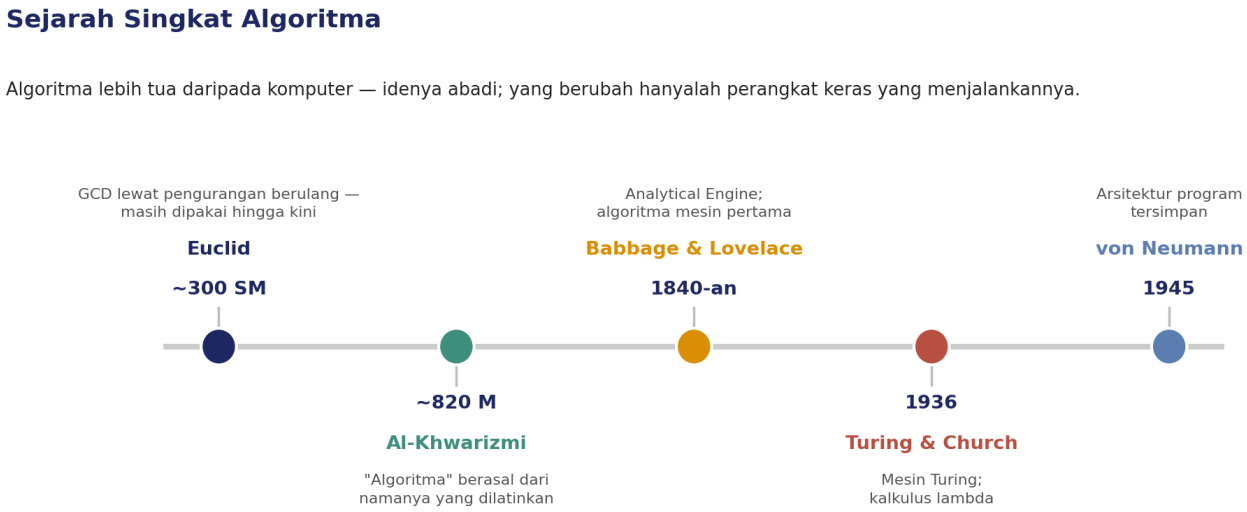
Kuliah 1 memberi Anda Kerangka Kerja Lima Fase dan menggunakannya untuk menyelesaikan enam soal konkret. Bab ini mundur selangkah dan membangun perangkat teoretis yang menjelaskan MENGAPA solusi-solusi itu berhasil: model presisi tentang apa arti "satu langkah", teknik standar (invarian loop) untuk membuktikan kebenaran algoritma iteratif, dan kosakata laju pertumbuhan yang mengubah "cepat" dan "lambat" menjadi pernyataan presisi yang dapat dibandingkan.

Gambar 2.1 — Posisi bab ini dalam peta jalan 16 kuliah DAA. Kuliah 3 akan memformalkan notasi asimtotik yang dipakai secara informal di bab ini.

2.2 Sejarah Singkat Algoritma

Kata "algoritma" berasal dari nama matematikawan Persia Muhammad al-Khwarizmi (sekitar 780–850 M), yang namanya dilatinkan menjadi Algorismus; risalahnya tentang aljabar dan aritmetika membentuk cara dunia berhitung hingga kini. Algoritma pertama yang ditemui hampir setiap mahasiswa jauh lebih tua lagi: metode Euclid untuk pembagi persekutuan terbesar, berasal dari sekitar

300 SM — berusia lebih dari 2.300 tahun, dan masih menjadi metode yang Anda pakai pada soal Unique Numbers di Kuliah 1. Abad ke-19 memberi kita Analytical Engine karya Charles Babbage dan algoritma karya Ada Lovelace yang ditujukan untuk eksekusi mesin — program pertama yang pernah ditulis untuk mesin yang bahkan belum ada. Abad ke-20 kemudian memformalkan komputasi itu sendiri: Mesin Turing karya Alan Turing (1936), kalkulus lambda karya Alonzo Church, dan arsitektur program-tersimpan karya John von Neumann, yang dipakai hampir semua komputer sejak saat itu.



Gambar 2.2 — Algoritma lebih tua daripada komputer. Idenya abadi; yang berubah hanyalah perangkat keras yang menjalankannya.

> **Wawasan kunci**
> Algoritma lebih tua daripada komputer, lebih dari dua ribu tahun lebih tua. IDE-nya abadi — yang berubah sepanjang abad hanyalah perangkat keras yang mengeksekusinya.

2.3 Definisi Inti, Secara Presisi

2.3.1 Apa Itu Algoritma, Secara Formal?

Algoritma adalah prosedur komputasi yang terdefinisi dengan baik, yang menerima suatu nilai (atau himpunan nilai) sebagai masukan dan menghasilkan suatu nilai (atau himpunan nilai) sebagai keluaran — "esensi" dari sebuah komputasi, terlepas dari bahasa pemrograman apa pun. Setiap algoritma harus memenuhi lima properti, pertama kali diberikan di Kuliah 1 dan diulang di sini secara lengkap:

- Masukan — nol atau lebih nilai dipasok dari luar.
- Keluaran — setidaknya satu nilai dihasilkan.
- Ketegasan (definiteness) — setiap langkah dispesifikasikan secara presisi dan tidak ambigu.
- Keberhinggaan (finiteness) — algoritma berhenti setelah sejumlah langkah yang berhingga, untuk setiap masukan yang valid.
- Keefektifan (effectiveness) — setiap langkah cukup dasar sehingga secara prinsip dapat dilakukan seseorang dengan pensil dan kertas.

2.3.2 Masalah Algoritmik vs. Solusi Algoritmik

Sebuah masalah algoritmik menspesifikasikan hubungan masukan-keluaran yang diinginkan — APA yang perlu dihitung, bukan BAGAIMANA — dan mengizinkan tak hingga banyak masukan valid. Masalah SORTING, misalnya: masukan sebuah barisan n bilangan, keluaran sebuah permutasi dari bilangan-bilangan tersebut dalam urutan tidak menurun. Sebuah solusi algoritmik (sebuah algoritma) menjelaskan BAGAIMANA menghitung keluaran dari masukan; untuk masalah apa pun ada tak hingga banyak algoritma yang benar, ada yang cepat, ada yang lambat, ada yang elegan, ada yang jelek.

Tantangan sentral mata kuliah ini

Di antara tak hingga banyak algoritma yang benar untuk suatu masalah, temukan yang EFISIEN. Kebenaran saja adalah separuh pekerjaan yang mudah.

2.4 Lima Tujuan Perancangan, Ditinjau Ulang

Kuliah 1 memperkenalkan dua tujuan utama — kebenaran dan efisiensi — ditambah tiga tujuan sekunder yang lebih relevan dalam konteks rekayasa perangkat lunak: ketangguhan (menangani masukan buruk secara baik), adaptabilitas (mudah dimodifikasi), dan reusabilitas (dapat dipakai ulang di program lain).

Lima Tujuan Perancangan

Mata kuliah ini fokus hampir sepenuhnya pada dua tujuan berbintang — tiga lainnya lebih relevan di rekayasa perangkat lunak.

Gambar 2.3 — Mata kuliah ini fokus hampir sepenuhnya pada dua tujuan berbintang. Tiga lainnya adalah subjek yang tepat untuk mata kuliah rekayasa perangkat lunak.

Wawasan kunci

Di mata kuliah ini kita fokus hampir sepenuhnya pada Kebenaran dan Efisiensi. Tiga tujuan lainnya penting dalam praktik, tetapi itu adalah subjek mata kuliah yang berbeda.

2.5 Model RAM untuk Komputasi

Sebelum kita bisa mengukur kecepatan...

...kita perlu menyepakati apa arti “satu langkah”. Bagaimana cara menghitungnya?

Kita menggunakan model Random Access Machine (RAM) — komputer ideal dengan aturan sederhana: setiap instruksi "sederhana" (aritmetika, perpindahan data, alur kendali) memakan tepat satu langkah waktu; setiap akses memori — membaca atau menulis sel mana pun — memakan satu langkah waktu, terlepas dari lokasinya; dan tipe data adalah bilangan bulat dan bilangan titik-mengambang berukuran wajar. Model RAM adalah penyederhanaan — komputer nyata memiliki cache, pipeline, dan banyak inti — tetapi cukup akurat untuk memprediksi kinerja dan, yang terpenting, untuk membandingkan algoritma secara adil.

> **Mengapa model RAM, bukan stopwatch?**
> Jika kita mengukur waktu nyata (wall-clock), algoritma yang sama akan tampak "lebih cepat" di laptop yang lebih baru. Model RAM tidak bergantung pada mesin: ia menghitung OPERASI, bukan detik, sehingga memungkinkan kita membandingkan algoritma secara adil, selamanya, terlepas dari komputer siapa pun yang menjalankannya.

2.6 Cara Menganalisis Waktu Eksekusi

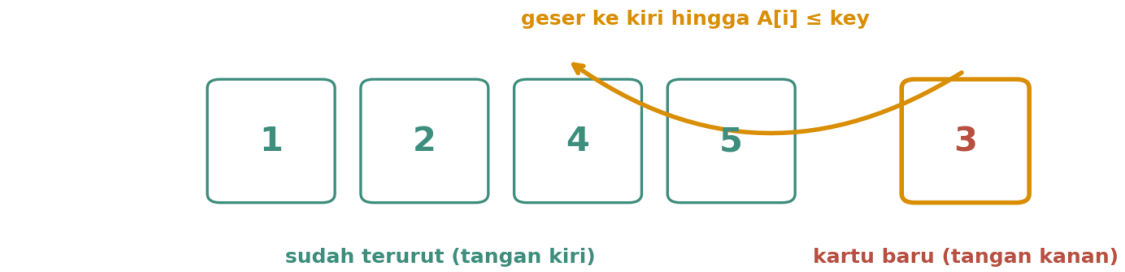
Kita menyatakan waktu eksekusi sebagai fungsi T(n) dari ukuran masukan n. Untuk menemukan T(n): hitung berapa kali setiap baris algoritma dieksekusi; berikan biaya untuk setiap baris (sebuah konstanta, di bawah model RAM); dan jumlahkan semua biaya. Ukuran masukan n sendiri bergantung pada masalahnya — jumlah elemen untuk sorting, dimensi untuk perkalian matriks, jumlah simpul (dan sering juga sisi, m) untuk algoritma graf, atau jumlah bit untuk masalah teori bilangan. Waktu eksekusi adalah total banyaknya operasi primitif yang dieksekusi — BUKAN waktu nyata, melainkan hitungan yang proporsional terhadap waktu aktual pada mesin yang wajar.

2.7 Studi Kasus: Insertion Sort — Analisis Lengkap

Insertion Sort adalah algoritma "sungguhan" pertama yang kita analisis secara lengkap. Ia mendemonstrasikan setiap konsep yang dibutuhkan bab ini: bukti kebenaran, pembedaan kasus terbaik/terburuk/rata-rata, dan analisis asimtotik — semuanya dalam satu algoritma kecil yang jujur.

2.7.1 Intuisi: Menyusun Kartu di Tangan

Bayangkan memegang kartu remi di tangan kiri, sudah terurut. Dengan tangan kanan Anda mengambil kartu baru dari meja, lalu menggesernya ke posisi yang benar di antara kartu-kartu terurut dengan membandingkan dari kanan ke kiri. Itulah persis cara kerja Insertion Sort pada sebuah larik — Gambar 2.4 menunjukkan analogi ini secara langsung, dan Gambar 2.5 menunjukkan ide yang sama dalam kode, satu posisi larik setiap kali.



Gambar 2.4 — Intuisi fisik di balik Insertion Sort: kartu baru digeser ke kiri hingga menemukan tempatnya yang benar di antara kartu-kartu yang sudah terurut.

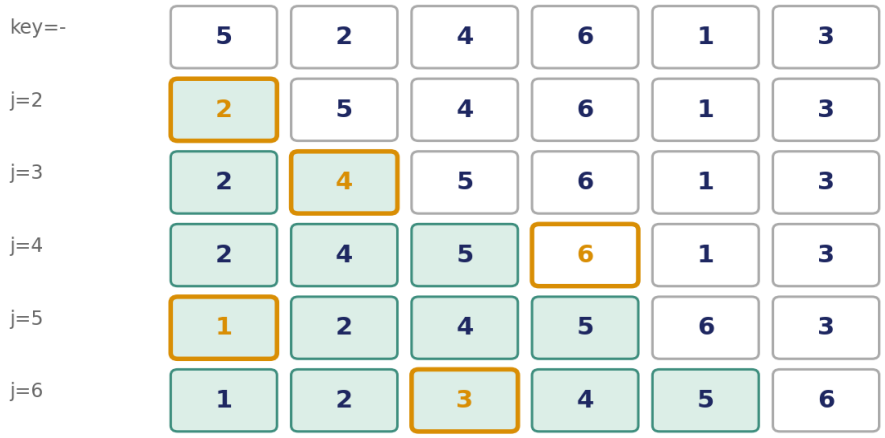
2.7.2 Algoritmanya

Strategi: proses elemen dari kiri ke kanan. Untuk setiap elemen baru (“key”), geser elemen yang lebih besar satu posisi ke kanan untuk memberi ruang, lalu sisipkan key di tempat yang benar.

```
INSERTION-SORT(A)
  for j = 2 to A.length              // mulai dari elemen ke-2
      key = A[j]                     // ambil kartu
      i = j - 1                      // mulai membandingkan dari kiri
      while i > 0 and A[i] > key      // geser elemen lebih besar ke kanan
          A[i + 1] = A[i]
          i = i - 1
      A[i + 1] = key                  // sisipkan kartu
```

2.7.3 Penelusuran Melalui Contoh

Mari kita urutkan A = [5, 2, 4, 6, 1, 3] langkah demi langkah. Gambar 2.5 menunjukkan larik sebelum loop dimulai (key = –, belum ada yang terurut) dan setelah tiap iterasi j = 2, 3, 4, 5, 6 — prefiks berwarna adalah persis daerah terurut yang dijamin oleh invarian loop, dan sel emas adalah key yang baru disisipkan.



Gambar 2.5 — Penelusuran Insertion Sort pada A = [5, 2, 4, 6, 1, 3]. Larik sepenuhnya terurut setelah j = 6.

2.7.4 Membuktikan Kebenaran: Invarian Loop

> **Bagaimana kita MEMBUKTIKAN Insertion Sort selalu bekerja?**
> Bukan hanya untuk contoh ini — untuk setiap kemungkinan larik masukan, dengan panjang berapa pun.

Kita menggunakan invarian loop — properti yang benar sebelum dan sesudah setiap iterasi loop — dan membuktikannya persis seperti induksi matematis: kasus basis (Inisialisasi), langkah induktif (Pemeliharaan), dan kesimpulan (Terminasi).

> **Invarian Loop untuk Insertion Sort**
> Pada awal setiap iterasi loop for (untuk indeks j), sub-larik A[1..j−1] terdiri atas elemen-elemen yang semula ada di A[1..j−1], tetapi dalam urutan terurut.

- Inisialisasi (j = 2): A[1..1] berisi hanya satu elemen. Satu elemen tunggal secara trivial sudah terurut. ✓
- Pemeliharaan: asumsikan A[1..j−1] terurut. Loop while bagian dalam menggeser setiap elemen yang lebih besar dari key satu posisi ke kanan, lalu menempatkan key di tempat yang benar. Setelah ini, A[1..j] terurut, mempertahankan invarian untuk iterasi berikutnya. ✓
- Terminasi (j = n + 1): loop berakhir ketika j > n. Pada titik itu invarian memberi tahu kita A[1..n] terdiri atas elemen-elemen asli dalam urutan terurut — persis definisi dari pengurutan yang benar. ✓

> **Wawasan kunci**
> Invarian loop adalah teknik bukti standar untuk algoritma iteratif. Anggaplah sebagai: “Apa yang selalu benar tentang keadaan komputasi pada titik ini?” Anda akan menggunakan teknik ini sepanjang mata kuliah.

2.7.5 Menganalisis Waktu Eksekusi

Dalam model RAM, setiap baris memiliki biaya konstan. Misalkan t_j adalah banyaknya badan loop while bagian dalam dieksekusi untuk j tertentu. Total waktu eksekusi adalah $T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4\Sigma t_j + c_5\Sigma(t_j-1) + c_6\Sigma(t_j-1) + c_7(n-1)$. Variabel kuncinya adalah t_j — berapa banyak perbandingan yang dilakukan loop bagian dalam — dan ini sepenuhnya bergantung pada MASUKAN, yang persis mengapa kita membutuhkan pembedaan kasus terbaik/terburuk/rata-rata pada bagian berikutnya.

Fase 5 — Implementasi

```cpp
/*
 * Chapter 2 - Case Study: Insertion Sort
 * Paradigm : Incremental
 *
 * Pseudocode (INSERTION-SORT, 1-indexed in the textbook; 0-indexed below):
 *   for j = 1 to n-1:
 *       key <- A[j]
 *       i <- j - 1
 *       while i >= 0 and A[i] > key:
 *           A[i+1] <- A[i]
 *           i <- i - 1
 *       A[i+1] <- key
 *
 * Loop invariant: at the start of each iteration of the outer for loop,
 * the subarray A[0..j-1] consists of the elements originally in A[0..j-1],
 * in sorted order. (See textbook Section 2.7.4.)
 *
 * Complexity: Best case O(n) (already sorted), worst/average case O(n^2).
 */
#include <cstdio>
using namespace std;

void insertionSort(int A[], int n) {
    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;
    }
}

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    int A[100005];
    for (int i = 0; i < n; i++) scanf("%d", &A[i]);
    insertionSort(A, n);
    for (int i = 0; i < n; i++)
        printf("%d%c", A[i], i + 1 < n ? ' ' : '\n');
    return 0;
}
```

2.8 Kasus Terbaik, Terburuk, dan Rata-Rata

Algoritma yang sama dapat memakan waktu yang sangat berbeda tergantung pada masukan spesifiknya. Kita membedakan tiga kasus.

2.8.1 Kasus Terbaik: Sudah Terurut

Jika A sudah terurut, maka untuk setiap j, A[j−1] ≤ key, sehingga badan loop while tidak pernah dieksekusi: $t_j = 1$ untuk semua j. Ini menghasilkan $T(n) = c_1n + (c_2+c_3+c_4+c_7)(n-1) = an + b$ — sebuah fungsi LINEAR dari n.

> **Wawasan kunci**
> Kasus terbaik Insertion Sort: T(n) adalah LINEAR — O(n). Ia hanya memindai, mengonfirmasi semuanya sudah pada tempatnya.

2.8.2 Kasus Terburuk: Terurut Terbalik

Jika A dalam urutan menurun ketat (misalnya, [6, 5, 4, 3, 2, 1]), setiap elemen baru harus dibandingkan dengan SEMUA elemen yang sudah terurut sebelumnya: $t_j = j$ untuk setiap j. Karena $\Sigma(j=2 \text{ hingga } n)\ j = n(n+1)/2 - 1$, ini menghasilkan $T(n) = an^2 + bn + c$ — sebuah fungsi KUADRATIK dari n.

> **Wawasan kunci**
> Kasus terburuk Insertion Sort: T(n) adalah KUADRATIK — $O(n^2)$. Mengurutkan larik terbalik-urut sebanyak 1.000.000 elemen membutuhkan sekitar 10^{12} operasi — jauh terlalu lambat untuk batas waktu manapun yang realistis.

2.8.3 Kasus Rata-Rata

Secara rata-rata, dengan asumsi permutasi acak, setiap elemen dibandingkan dengan sekitar setengah elemen yang sudah terurut sebelumnya: $t_j \approx j/2$. Ini menghasilkan $T(n) = an^2/2 + \ldots$ — tetap $O(n^2)$. Kasus rata-rata memiliki ORDE PERTUMBUHAN YANG SAMA dengan kasus terburuk.

2.8.4 Mengapa Mata Kuliah Ini Fokus pada Kasus Terburuk

Kita hampir selalu menganalisis kasus terburuk, karena empat alasan: ia memberikan batas atas terjamin — algoritma TIDAK AKAN PERNAH memakan waktu lebih lama dari ini; sistem safety-critical (pengendalian lalu lintas udara, robot bedah) HARUS mengetahui kasus terburuk; untuk banyak algoritma, kasus terburuk sering terjadi dalam praktik; dan kasus rata-rata sering memiliki orde asimtotik yang sama dengan kasus terburuk, sementara menghitung rata-rata SEBENARNYA membutuhkan pengetahuan tentang distribusi masukan, yang sering tidak diketahui.

Analogi perusahaan pengiriman

"Paket Anda tiba dalam paling lambat 5 hari" (jaminan kasus terburuk) lebih berguna untuk perencanaan daripada "biasanya tiba dalam 3 hari" (klaim kasus rata-rata). Anda dapat merencanakan secara andal berdasarkan jaminan; Anda tidak dapat merencanakan secara andal berdasarkan rata-rata.

2.9 Pertumbuhan Fungsi: Mengapa Asimtotik Penting

Tidak semua fungsi tumbuh dengan laju yang sama, dan seiring n membesar, perbedaannya menjadi dramatis. Tabel 2.1 mengonkretkan ini untuk n = 1.000.000, dengan asumsi mesin modern yang mengeksekusi sekitar 10^9 operasi sederhana per detik.

Kompleksitas	Operasi pada n = 10^6	Waktu pada 10^9 ops/detik
O(log n)	≈ 20	instan
O(n)	1.000.000	0,001 dtk
O(n log n)	≈ 19.930.000	0,02 dtk
$O(n^2)$	1.000.000.000.000	≈ 17 menit
$O(2^n)$	≈ 10^{301030}	jauh melampaui ~10^{80} atom di alam semesta teramati

Wawasan kunci

Seiring n membesar, LAJU PERTUMBUHAN mendominasi segalanya. Konstanta dan suku berorde lebih rendah menjadi tidak relevan — inilah persis mengapa kita memakai notasi asimtotik (Big-O): ia menangkap laju pertumbuhan dan membuang derau.

2.9.1 Hierarki Laju Pertumbuhan

Dari yang tumbuh paling cepat ke paling lambat (terbaik ke terburuk untuk waktu eksekusi): $O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$.

Gambar 2.6 — Bagaimana jumlah operasi bertumbuh terhadap ukuran masukan n (direproduksi dari Gambar 1.1). Perhatikan bagaimana $O(n^2)$ dan $O(2^n)$ meledak sementara $O(\log n)$ dan $O(n)$ tetap nyaris datar.

> **Aturan praktis**
> Jika algoritma Anda $O(n^2)$ dan n = 1.000.000, ia membutuhkan sekitar 10^{12} operasi — sekitar 1.000 detik, atau 16 menit 40 detik — jauh melampaui hampir semua batas waktu. Anda butuh algoritma yang lebih baik, bukan komputer yang lebih cepat.

2.10 Pencarian: Masalah Fundamental

2.10.1 Definisi Masalah

Masukan: sebuah barisan A = [a_1, a_2, ..., a_n] dan nilai query q. Keluaran: sebuah indeks j sedemikian sehingga A[j] = q, atau NIL jika q tidak ada di A.

2.10.2 Linear Search (Larik Tak Terurut)

Jika larik tidak terurut, kita tidak punya pilihan selain memeriksa setiap elemen secara berurutan.

```
LINEAR-SEARCH(A, q)
  j = 1
  while j <= A.length and A[j] != q
      j = j + 1
  if j <= A.length then return j
  else return NIL
```

Kasus terburuk (q tidak ditemukan): kita memeriksa semua n elemen, O(n). Kasus rata-rata (q pada posisi acak): sekitar n/2 elemen, tetap O(n). Tidak ada cara untuk melakukan lebih baik pada larik tak terurut — ini adalah BATAS BAWAH, bukan sekadar batas atas.

> **Wawasan kunci**
> Linear search pada larik tak terurut adalah O(n), dan ini optimal — Anda tidak bisa menemukan elemen lebih cepat tanpa struktur tambahan, seperti pengurutan.

Fase 5 — Implementasi

```cpp
/*
 * Chapter 2 - Searching: Linear Search
 * Paradigm : Brute Force (no structure assumed)
 *
 * Pseudocode:
 *   j <- 0
 *   while j < n and A[j] != q: j <- j + 1
 *   return (j < n) ? j : NIL
 *
 * Complexity: Worst/average case O(n). Optimal for an UNSORTED array —
 * this is a lower bound, not just an upper bound (see textbook Section 2.10.2).
 */
#include <cstdio>
using namespace std;

int linearSearch(int A[], int n, int q) {
    int j = 0;
    while (j < n && A[j] != q) j++;
    return (j < n) ? j : -1;
}

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    int A[100005];
    for (int i = 0; i < n; i++) scanf("%d", &A[i]);
    int t;
    scanf("%d", &t);
    while (t--) {
        int q;
        scanf("%d", &q);
        int idx = linearSearch(A, n, q);
        if (idx == -1) printf("NIL\n"); else printf("%d\n", idx);
    }
    return 0;
}
```

2.10.3 Binary Search: Divide and Conquer untuk Pencarian

> **Bisakah kita lebih baik?**
> Bagaimana jika larik TERURUT? Ya — urutan terurut memungkinkan kita memanfaatkan struktur untuk pencarian yang jauh lebih cepat.

Masukan: barisan tidak-menurun yang TERURUT A = [$a_1 \le a_2 \le \ldots \le a_n$] dan query q. Bandingkan q dengan elemen TENGAH: jika A[mid] = q, selesai; jika A[mid] > q, q pasti berada di setengah kiri, sehingga buang setengah kanan; jika A[mid] < q, buang setengah kiri. Setiap perbandingan MEMBAGI DUA ruang pencarian — esensi dari divide and conquer.

```
BINARY-SEARCH(A, q)
  left = 1
  right = A.length
  do
      mid = (left + right) / 2        // pembagian bilangan bulat
      if A[mid] == q then return mid
      else if A[mid] > q then right = mid - 1
      else left = mid + 1
  while left <= right
  return NIL
```

Menelusuri pencarian q = 7 dalam A = [2, 4, 5, 7, 10]: Gambar 2.7 menunjukkan kedua langkah. Binary search menemukan jawaban hanya dalam 2 perbandingan, dibandingkan 4 untuk linear search pada larik yang sama.

Gambar 2.7 — Binary Search pada A = [2, 4, 5, 7, 10] mencari q = 7. Setiap langkah membuang setengah dari yang tersisa.

Berapa kali Anda bisa membelah n menjadi dua sebelum mencapai 1?
Mulai dengan n elemen. Setelah 1 perbandingan: n/2. Setelah 2: n/4. Setelah k: $n/2^k$. Kita berhenti ketika $n/2^k \le 1$, yaitu $2^k \ge n$, yaitu $k \ge \log_2(n)$.

Wawasan kunci
Binary search berjalan dalam waktu O(log n). Untuk n = 1.000.000, ia membutuhkan paling banyak 20 perbandingan — bandingkan dengan 1.000.000 perbandingan untuk linear search. Struktur terurut dari data memberikan percepatan eksponensial.

Trade-off-nya adalah pra-pemrosesan: binary search membutuhkan larik terurut, dan pengurutan berbiaya O(n log n). Biaya totalnya adalah O(n log n) untuk mengurutkan ditambah O(log n) per pencarian — sepadan setiap kali Anda mencari berkali-kali: untuk satu kali pencarian, linear lebih murah; untuk 100 kali pencarian, binary mulai menang untuk n besar; untuk 10.000 kali pencarian, binary search jauh lebih unggul.

Fase 5 — Implementasi

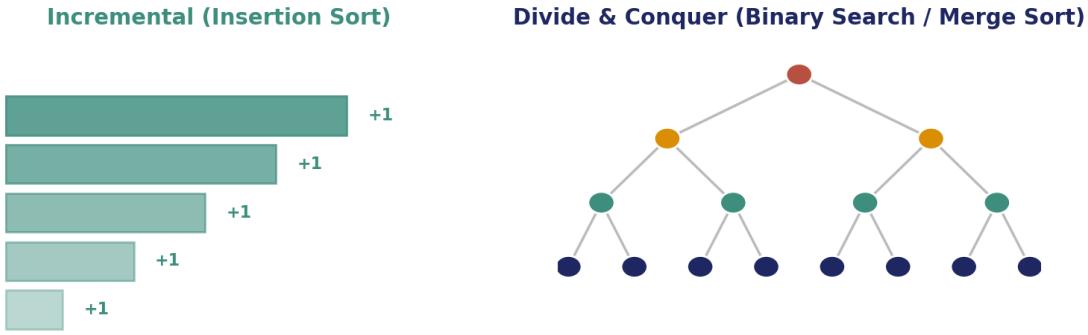
```cpp
/*
 * Chapter 2 - Searching: Binary Search
 * Paradigm : Divide and Conquer
 *
 * Precondition: A is sorted in non-decreasing order.
 * Pseudocode:
 *   left <- 0, right <- n-1
 *   while left <= right:
 *       mid <- (left + right) / 2
 *       if A[mid] == q: return mid
 *       else if A[mid] > q: right <- mid - 1
 *       else: left <- mid + 1
 *   return NIL
 *
 * Complexity: O(log n) — each comparison halves the search space
 * (see textbook Section 2.11.5).
 */
#include <cstdio>
using namespace std;

int binarySearch(int A[], int n, int q) {
    int left = 0, right = n - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (A[mid] == q) return mid;
        else if (A[mid] > q) right = mid - 1;
        else left = mid + 1;
    }
    return -1;
}

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    int A[100005];
    for (int i = 0; i < n; i++) scanf("%d", &A[i]);
    int t;
    scanf("%d", &t);
    while (t--) {
        int q;
        scanf("%d", &q);
        int idx = binarySearch(A, n, q);
        if (idx == -1) printf("NIL\n"); else printf("%d\n", idx);
    }
    return 0;
}
```

2.11 Paradigma Perancangan, Ditinjau Ulang

Kita telah melihat contoh konkret dari dua paradigma fundamental yang disebutkan di Bab 1.

Incremental vs. Divide-and-Conquer

Dua cara membangun solusi: satu elemen setiap kali, atau membelah masalah jadi dua dan menggabungkan hasilnya.



Gambar 2.8 — Algoritma incremental membangun solusi satu elemen setiap kali; algoritma divide-and-conquer membelah masalah dan menggabungkan sub-hasil.

2.11.1 Pendekatan Incremental

Bangun solusi sepotong demi sepotong, memproses satu elemen per iterasi. Insertion Sort adalah contoh klasik: pada setiap langkah, kita “menyisipkan” satu elemen lagi ke bagian yang sudah terurut. Kekuatan: sederhana, mudah dipahami dan diimplementasikan, dan bekerja baik untuk masukan kecil atau yang hampir terurut. Kelemahan: sering mengarah ke algoritma $O(n^2)$, karena setiap penyisipan mungkin membutuhkan pemindaian seluruh bagian yang terurut.

2.11.2 Divide and Conquer

Pecah masalah menjadi sub-masalah yang lebih kecil dengan tipe yang sama, selesaikan secara rekursif, lalu gabungkan hasilnya. Binary Search adalah contoh sempurna: setiap perbandingan membelah dua ruang pencarian. Kekuatan: sering mengarah ke algoritma $O(n \log n)$ atau $O(\log n)$, secara alami cocok untuk rekursi. Anda akan segera mempelajari Merge Sort, yang membelah larik menjadi dua, mengurutkan tiap bagian secara rekursif, lalu menggabungkan dua bagian terurut dalam $O(n \log n)$ — jauh lebih cepat daripada $O(n^2)$ milik Insertion Sort untuk n besar.

2.11.3 Algoritma Ad Hoc

Frasa Latin ad hoc berarti “untuk ini” — solusi yang dirancang untuk satu masalah spesifik dan tidak dapat digeneralisasi. Soal Columns di Kuliah 1 (jawaban = $\max(n, \max(a_i))$) dan Interesting Sequence (jawaban = popcount(m−1) mod 3) keduanya ad hoc: kuat, tetapi sulit diajarkan secara sistematis. Paradigma — incremental, divide-and-conquer, dynamic programming, greedy — adalah yang memberi Anda alat pemecahan masalah UMUM yang dapat dipakai untuk masalah-masalah baru.

2.12 Profitina dalam Praktik: Di Mana Bab Ini Muncul dalam Produk Nyata

> **Tentang kotak ini**
> Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) asal Indonesia yang dibangun oleh penulis, tanpa membocorkan detail implementasi yang proprietary. Lihat dokumen pendamping 'Profitina: Tinjauan Teknis Non-Rahasia' untuk gambaran lengkapnya.

Rekayasa Profitina sendiri harus membuat keputusan kasus terbaik/terburuk/rata-rata yang sama seperti yang diajarkan bab ini. Ketika Profitina mencari bisnis yang sudah pernah dipindai sebelumnya berdasarkan kunci domain ternormalisasi di dalam indeks dalam-memori, pencarian tersebut berperilaku seperti binary search atas kunci-kunci terurut dalam kasus terburuk — O(log n) — bukan pemindaian linear atas setiap catatan. Namun ketika sinyal baru harus digabungkan ke dalam kumpulan hasil yang sudah diproses satu bisnis setiap kali (misalnya, menggabungkan secara bertahap sitasi yang baru ditemukan untuk sebuah bisnis seiring ditemukannya), pola pembaruan tersebut secara struktural identik dengan pendekatan incremental Insertion Sort: setiap item baru "disisipkan" ke tempatnya yang benar di antara item yang sudah diproses, dan para insinyur Profitina memikirkan kasus terburuk langkah ini persis seperti Bagian 2.8 di sini — dengan bertanya apa yang terjadi jika setiap item baru harus dibandingkan dengan semua yang sudah tersimpan.

2.13 Ringkasan Bab dan Poin-Poin Kunci

- Sebuah ALGORITMA adalah prosedur langkah demi langkah; sebuah PROGRAM adalah implementasinya; sebuah STRUKTUR DATA adalah cara data diorganisasikan.
- Kita menganalisis algoritma menggunakan model RAM, menghitung operasi sebagai fungsi dari ukuran masukan n.
- Insertion Sort adalah O(n) pada kasus terbaik (sudah terurut) dan $O(n^2)$ pada kasus terburuk/rata-rata (terbalik-urut / acak).
- Invarian loop membuktikan kebenaran untuk algoritma iteratif: Inisialisasi, Pemeliharaan, Terminasi.
- Kita fokus pada analisis KASUS TERBURUK karena ia memberikan batas atas yang terjamin.
- Hierarki laju pertumbuhan: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$ — perbedaannya menjadi dramatis untuk n besar.
- Linear search adalah O(n) dan optimal untuk larik tak terurut; binary search adalah O(log n) untuk larik terurut — peningkatan eksponensial dari memanfaatkan struktur.
- Paradigma perancangan (incremental, divide-and-conquer) adalah strategi umum; solusi ad hoc bersifat spesifik-masalah.

> ***“Sebuah algoritma harus DILIHAT untuk bisa DIPERCAYA.”***
>
> **— Donald E. Knuth**
>
> *Telusuri setiap algoritma dengan tangan sendiri. Pemahaman datang dari proses melakukannya.*

Kuliah berikutnya, kita akan memformalkan semua yang dipakai secara informal di bab ini: definisi presisi dari O, Θ, dan Ω; cara membuktikan secara formal bahwa f(n) = O(g(n)); dan cara menyelesaikan rekurensi (Teorema Master dan metode substitusi) — perangkat yang dibutuhkan untuk menganalisis T(n) = 2T(n/2) + O(n) = O(n log n) milik Merge Sort.

2.14 Pertanyaan Tinjauan

Pertanyaan-pertanyaan ini untuk studi mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Kuliah 3.

1. Buktikan invarian loop untuk Insertion Sort dengan kata-kata Anda sendiri, tanpa melihat Bagian 2.7.4, menggunakan contoh larik baru pilihan Anda.
2. Buat larik masukan berukuran 5 yang menghasilkan kasus TERBURUK untuk Insertion Sort, dan satu lagi yang menghasilkan kasus TERBAIK. Hitung t_j untuk setiap j pada kedua kasus.
3. Jelaskan, dalam istilah model RAM, mengapa akses larik A[i] berbiaya sama (O(1)) terlepas dari seberapa besar i atau di mana dalam memori larik tersebut berada.
4. Untuk n = 1.000, kira-kira berapa banyak operasi yang dilakukan algoritma O(n log n), dibandingkan dengan algoritma $O(n^2)$? Gunakan metode Tabel 2.1 untuk membenarkan estimasi Anda.
5. Binary search membutuhkan larik terurut. Jika Anda hanya perlu melakukan 3 kali pencarian pada larik 10.000 elemen, apakah sepadan untuk mengurutkan terlebih dahulu? Justifikasi menggunakan model biaya O(n log n) sort + O(log n) per pencarian dari Bagian 2.10.3.

Lampiran 2.A — Log Verifikasi untuk Kode Sumber Bab 2

Seperti di Bab 1, setiap program yang disertakan buku ini dikompilasi dan dijalankan terhadap contoh yang diverifikasi secara manual sebelum disertakan. Ketiga program di bawah ini dikompilasi dengan `g++ -O2 -std=c++17 -Wall` (hanya peringatan scanf yang tidak berbahaya, nol error) dan menghasilkan keluaran persis seperti yang diprediksi pada teks di atas.

```
$ printf "6\n5 2 4 6 1 3\n" | ./01_insertion_sort
1 2 3 4 5 6                          # sesuai penelusuran Bagian 2.7.3 (Gambar
2.5)

$ printf "5\n2 5 4 10 7\n2\n5\n9\n" | ./02_linear_search
1                                    # q=5 ditemukan pada posisi indeks-0 = 1
NIL                                  # q=9 tidak ada

$ printf "5\n2 4 5 7 10\n3\n7\n2\n6\n" | ./03_binary_search
3                                    # q=7 ditemukan pada posisi indeks-0 = 3,
dalam 2 perbandingan
0                                    # q=2 ditemukan pada posisi indeks-0 = 0
NIL                                  # q=6 tidak ada
```

Latihan SPOJ — buku pendamping
Teknik bab ini diuji sungguhan di juri daring SPOJ. Buka Bab 2–3 buku pendamping “SPOJ Competitive Programming” (folder SPOJ, BI & EN) untuk insertion sort, pencarian linier & biner di bawah batasan nyata — lengkap dengan tips, trik, dan hint agar solusi Anda Accepted, tanpa membocorkan jawaban.

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Bab 1–2).

[2] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, edisi ke-4. Lulu Press, 2020–2023.

[3] D. E. Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, edisi ke-2. Addison-Wesley, 1998.

[4] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem.” Proceedings of the London Mathematical Society, 1936.

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 3

Notasi Asimtotik, Rekurensi, dan Merge Sort

Menjadikan "O(n log n)" klaim yang presisi dan terbukti, bukan sekadar perasaan — sekaligus berkenalan dengan algoritma divide-and-conquer pertama yang benar-benar mengalahkan Insertion Sort.

Tujuan Pembelajaran — setelah bab ini Anda mampu:
(1) Menyatakan definisi presisi (dengan konstanta dan ambang batas) notasi O, Ω, dan Θ, serta menggunakannya untuk membuktikan klaim asimtotik sederhana secara langsung. (2) Mengenali relasi rekurensi yang muncul dari algoritma rekursif, dan menyelesaikannya dengan metode pohon rekursi, metode substitusi, atau Master Theorem. (3) Menelusuri, mengimplementasikan, dan membuktikan kebenaran Merge Sort, termasuk subrutin MERGE. (4) Menurunkan rekurensi Merge Sort $T(n) = 2T(n/2) + \Theta(n)$ dan menyelesaikannya sehingga $T(n) = \Theta(n \log n)$. (5) Menjelaskan, dengan angka konkret, mengapa algoritma $O(n \log n)$ bukan sekadar "sedikit lebih cepat" daripada $O(n^2)$, melainkan berbeda secara kategoris pada skala besar.

3.1 Rekap: Dari Kuliah 2 ke Kuliah 3

Kuliah 2 memakai notasi Big-O terus-menerus — "$O(n)$", "$O(n^2)$", "$O(\log n)$" — tetapi selalu secara informal, bersandar pada intuisi tentang laju pertumbuhan. Ketidakformalan itu cukup baik untuk membangun intuisi, tetapi tidak dapat menopang bukti yang ketat, dan tidak dapat memberi tahu kita cara menyelesaikan persamaan aneh yang dihasilkan algoritma rekursif, seperti $T(n) = 2T(n/2) + \Theta(n)$. Bab ini memperbaiki kedua celah tersebut. Pertama kita jadikan O, Ω, dan Θ sebagai definisi matematis presisi dengan konstanta dan ambang batas, jenis definisi yang dapat dipakai dalam bukti. Kemudian kita bangun tiga alat untuk menyelesaikan rekurensi — pohon rekursi, substitusi, dan Master Theorem — dan memakai ketiganya pada Merge Sort, algoritma pengurutan divide-and-conquer pertama kita yang sesungguhnya.

Gambar 3.1 — Posisi bab ini dalam peta jalan 16 kuliah DAA. Kuliah 5 akan menerapkan resep divide-and-conquer yang dibangun di sini pada Quicksort; Kuliah 6 dan 7 lalu beralih ke wilayah yang sama sekali baru — pohon pencarian seimbang, dan program dinamis.

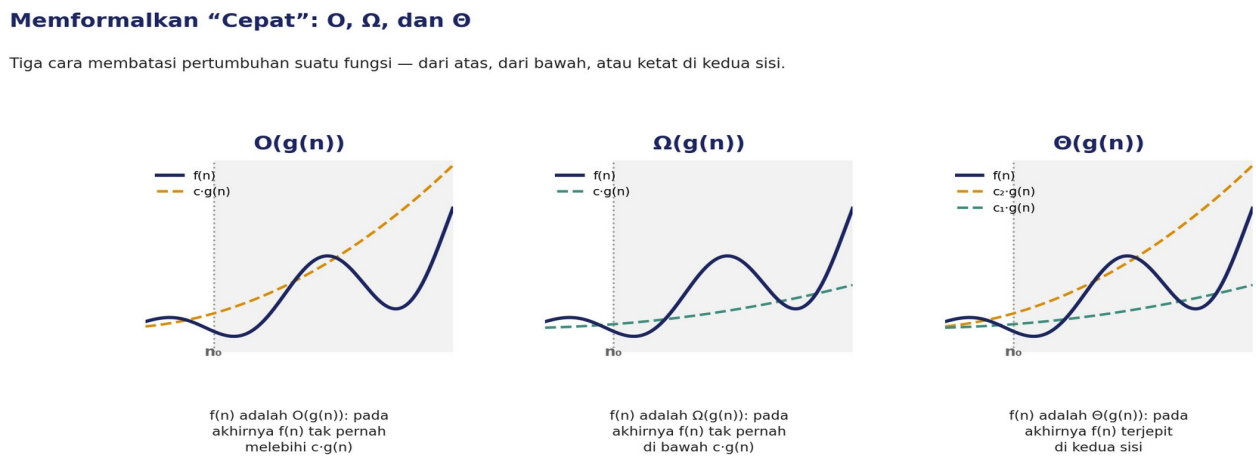
3.2 Memformalkan Notasi Asimtotik: O, Ω, dan Θ

Kuliah 2 memakai frasa seperti "T(n) kira-kira n^2" tanpa pernah menetapkan arti "kira-kira". Berikut versi presisinya, untuk fungsi f(n) yang secara asimtotik taknegatif (taknegatif untuk semua n yang cukup besar).

Big-O (batas atas)
f(n) = O(g(n)) jika terdapat konstanta positif c dan n_0 sedemikian sehingga $0 \le f(n) \le c \cdot g(n)$ untuk semua $n \ge n_0$. Dengan kata lain: mulai suatu titik, f tak pernah melebihi kelipatan konstan dari g. O(g(n)) adalah batas atas — boleh jadi ketat, boleh jadi tidak.

Big-Omega (batas bawah)
f(n) = Ω(g(n)) jika terdapat konstanta positif c dan n_0 sedemikian sehingga $0 \le c \cdot g(n) \le f(n)$ untuk semua $n \ge n_0$. Dengan kata lain: mulai suatu titik, f tak pernah lebih kecil dari kelipatan konstan dari g. Ω(g(n)) adalah batas bawah.

Big-Theta (batas ketat)
f(n) = Θ(g(n)) jika terdapat konstanta positif c_1, c_2, dan n_0 sedemikian sehingga $0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ untuk semua $n \ge n_0$. Setara dengan: f(n) = O(g(n)) DAN f(n) = Ω(g(n)). Θ adalah batas ketat — yang sesungguhnya kita inginkan kapan pun bisa diperoleh.



Gambar 3.2 — Ketiga notasi hanya membuat klaim mulai dari n_0; tidak ada syarat untuk n kecil. O membatasi dari atas, Ω dari bawah, Θ menjepit f(n) di antara dua salinan g(n) yang diskalakan.

Wawasan kunci
Setiap definisi di atas punya bentuk yang sama: “pada akhirnya ($n \ge n_0$), sampai faktor konstan (c), f berperilaku seperti g.” Konstanta dan ambang batas itulah yang membuat kita boleh mengabaikan suku berorde rendah dan konstanta yang bergantung mesin — persis hal yang dilakukan Kuliah 2 secara informal dengan mata.

3.3 Membuktikan Batas Asimtotik dari Definisi

Mari kita buktikan klaim dari Kuliah 2 yang sebelumnya hanya kita nyatakan tanpa bukti: $3n^2 + 2n + 5 = O(n^2)$.

Bukti terpandu

Klaim: $3n^2 + 2n + 5 = O(n^2)$. Bukti: untuk semua $n \geq 1$, $2n \leq 2n^2$ dan $5 \leq 5n^2$, sehingga $3n^2 + 2n + 5 \leq 3n^2 + 2n^2 + 5n^2 = 10n^2$. Dengan memilih $c = 10$ dan $n_0 = 1$, kita punya $0 \leq 3n^2 + 2n + 5 \leq 10n^2$ untuk semua $n \geq n_0 = 1$. Berdasarkan definisi O, $3n^2 + 2n + 5 = O(n^2)$. ∎

Fungsi yang sama juga $\Omega(n^2)$ — untuk semua $n \geq 1$, $3n^2 + 2n + 5 \geq 3n^2$, sehingga $c = 3$ dan $n_0 = 1$ berlaku — dan karena itu $\Theta(n^2)$. Perhatikan bahwa konstanta c dan n_0 tidak tunggal; $c \geq 10$ dan $n_0 \geq 1$ mana pun juga berlaku pada bukti O di atas. Sebuah bukti hanya perlu menunjukkan SATU pasangan (c, n_0) yang valid, bukan yang terbaik.

Kesalahan umum mahasiswa

"$f(n) = O(n^2)$" TIDAK berarti f tumbuh persis seperti n^2. Artinya f tumbuh PALING BANYAK seperti n^2, sampai suatu konstanta. Baik $f(n) = n$ maupun $f(n) = 0{,}001n^2$ keduanya $O(n^2)$ — Big-O adalah batas atas untuk seluruh keluarga fungsi, bukan satu kurva tunggal. Inilah persis mengapa kita lebih menyukai Θ kapan pun bisa dibuktikan: Θ menetapkan laju pertumbuhan secara pasti.

3.4 Sifat-Sifat Notasi Asimtotik

Daftar singkat fakta, semuanya dapat dibuktikan langsung dari definisi di atas, yang menjadikan penalaran asimtotik sehari-hari sah secara formal, bukan sekadar intuitif:

- Transitivitas: jika $f(n) = O(g(n))$ dan $g(n) = O(h(n))$, maka $f(n) = O(h(n))$. (Berlaku juga untuk Ω dan Θ.)
- Refleksivitas: $f(n) = \Theta(f(n))$ untuk f mana pun.
- Simetri transpos: $f(n) = O(g(n))$ jika dan hanya jika $g(n) = \Omega(f(n))$.
- Jumlah: jika $f_1(n) = O(g_1(n))$ dan $f_2(n) = O(g_2(n))$, maka $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$. Inilah mengapa, dalam $T(n) = c_1n + c_2n^2 + c_3$, suku n^2 sendirian menentukan kelas-Θ — suku berorde rendah dan konstanta terserap.
- Konstanta tidak penting, laju pertumbuhan yang penting: untuk konstanta $k > 0$ mana pun, $k \cdot f(n) = \Theta(f(n))$.

Wawasan kunci

Sifat-sifat ini persis mengapa Kuliah 2 boleh dengan aman membuang suku berorde rendah dan faktor konstan ketika membandingkan algoritma — operasi seperti “tambah sejumlah konstan akses larik ekstra” tak pernah mengubah kelas-Θ dari analisis waktu-eksekusi yang benar.

3.5 Rekurensi: Dari Mana Asalnya

Sebuah rekurensi adalah persamaan yang mendeskripsikan suatu fungsi dalam istilah nilainya sendiri pada masukan yang lebih kecil, bersama sebuah kasus basis. Rekurensi muncul langsung dari algoritma rekursif: untuk menganalisis algoritma rekursif, pertama tuliskan rekurensi untuk waktu eksekusinya T(n), lalu selesaikan rekurensi itu untuk mendapatkan bentuk tertutup seperti Θ(n log n).

Untuk algoritma divide-and-conquer yang membagi masalah berukuran n menjadi a sub-masalah, masing-masing berukuran n/b, dan menghabiskan f(n) waktu untuk membagi dan menggabungkan (di luar panggilan rekursif), waktu eksekusinya memenuhi:

> **Rekurensi umum divide-and-conquer**
> $T(n) = a \cdot T(n/b) + f(n)$, untuk n lebih besar dari suatu ambang batas konstan, dengan $T(n) = \Theta(1)$ untuk kasus basis (n kecil). Di sini $a \geq 1$ adalah banyaknya sub-masalah, $b > 1$ adalah faktor pengecilan masukan, dan f(n) adalah biaya membagi masalah dan menggabungkan solusi sub-masalah.

Tiga teknik penyelesaian bab ini — pohon rekursi, substitusi, dan Master Theorem — adalah tiga cara berbeda untuk mengubah persamaan ini menjadi batas-Θ bentuk tertutup.

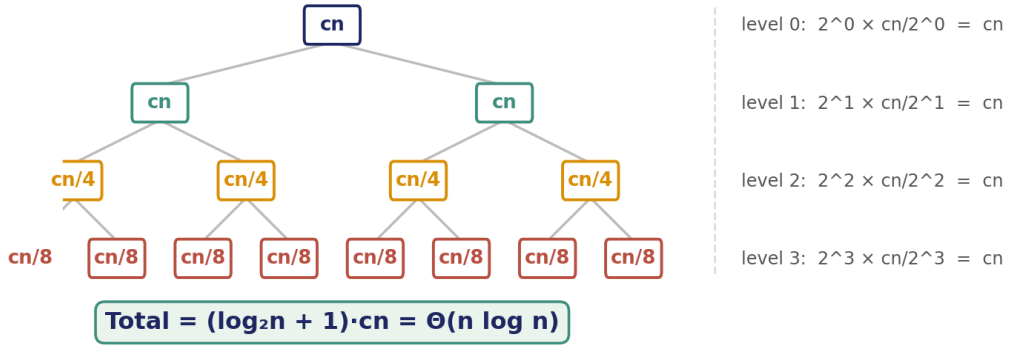
3.6 Menyelesaikan Rekurensi I: Metode Pohon Rekursi

Pohon rekursi membuat total biaya suatu rekurensi terlihat dengan menggambar satu simpul per panggilan rekursif, diberi label biaya panggilan itu TANPA menghitung anak-anaknya, lalu menjumlahkan setiap simpul dalam pohon. Perhatikan $T(n) = 2T(n/2) + cn$ — bentuk yang akan ternyata dimiliki Merge Sort.

Pada akar, biaya non-rekursif adalah cn. Akar memiliki 2 anak, masing-masing menyelesaikan sub-masalah berukuran n/2, masing-masing dengan biaya non-rekursif c(n/2) — sehingga kedua anak bersama-sama juga berbiaya cn. Anak-anak mereka (4 buah, berukuran n/4 masing-masing) kembali berbiaya cn secara total. Pola ini — setiap level berbiaya persis cn — berlanjut hingga sub-masalah mencapai ukuran 1, yang membutuhkan $\log_2 n$ level di bawah akar, sehingga $\log_2 n + 1$ level secara total (termasuk akar).

Menyelesaikan T(n) = 2T(n/2) + cn dengan Pohon Rekursi

Setiap level berbiaya persis cn. Ada $\log_2 n + 1$ level → total = Θ(n log n).



Gambar 3.3 — Pohon rekursi untuk T(n) = 2T(n/2) + cn. Setiap level berbiaya persis cn terlepas dari kedalamannya, karena setiap level punya dua kali lebih banyak simpul dari level di atasnya, masing-masing mengerjakan separuh pekerjaan.

Kesimpulan pohon rekursi

Total biaya = (banyak level) × (biaya per level) = $(\log_2 n + 1) \times cn = cn \cdot \log_2 n + cn = \Theta(n \log n)$.

Wawasan kunci

Metode pohon rekursi intuitif dan sulit salah begitu digambar dengan benar, tetapi bersifat informal — bukan dengan sendirinya sebuah bukti. Dalam praktik ini dipakai untuk MENEBAK jawaban, yang kemudian diverifikasi secara ketat lewat substitusi (Bagian 3.7) atau langsung dibaca dari Master Theorem (Bagian 3.8).

3.7 Menyelesaikan Rekurensi II: Metode Substitusi

Metode substitusi membuktikan sebuah tebakan batas dengan induksi matematis: tebak bentuk tertutupnya, lalu verifikasi bahwa ia memenuhi rekurensi dengan substitusi langsung ke kedua ruas.

Bukti terpandu dengan substitusi

Klaim: T(n) = 2T(n/2) + n adalah O(n log n). Tebakan: $T(n) \le cn \log_2 n$ untuk suatu konstanta c yang akan ditentukan, untuk $n \ge 2$. Langkah induktif: asumsikan $T(n/2) \le c(n/2)\log_2(n/2)$ berlaku untuk sub-masalah yang lebih kecil. Maka $T(n) = 2T(n/2) + n \le 2 \cdot c(n/2)\log_2(n/2) + n = cn \cdot \log_2(n/2) + n = cn \cdot (\log_2 n - 1) + n = cn \cdot \log_2 n - cn + n = cn \cdot \log_2 n - (c-1)n$. Ini $\le cn \cdot \log_2 n$ persis ketika $(c-1)n \ge 0$, yaitu ketika $c \ge 1$. Dengan memilih $c = 2$ (memeriksa kasus basis $T(1) = \Theta(1) \le c \cdot 1 \cdot \log_2 1 = 0$ secara terpisah mengharuskan induksi dimulai di $n = 2$, di mana pemeriksaan langsung mengonfirmasi batas berlaku untuk c yang cukup besar), langkah induktif berlaku untuk semua $n \ge 2$. Karena itu $T(n) = O(n \log n)$. ∎

Kehalusan yang sering menjebak

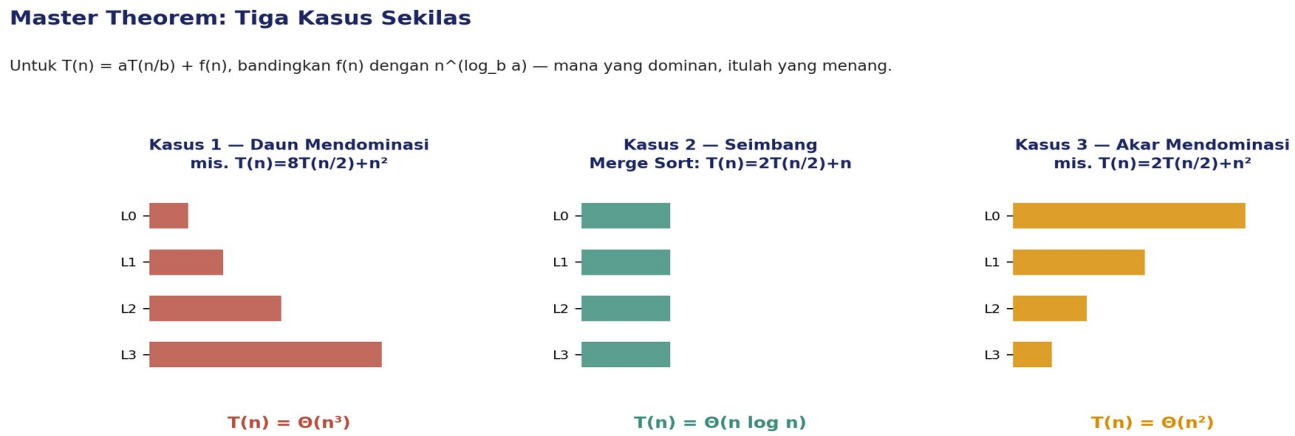
Langkah induktif di atas membutuhkan (c−1)n ≥ 0 — suku '−cn' PENTING untuk menyerap '+n' dan menjaga ketidaksamaan tetap ketat. Kesalahan umum adalah menebak T(n) ≤ cn (linear) untuk rekurensi yang sama ini: induksi kemudian menghasilkan T(n) ≤ cn + n, yang TIDAK ≤ cn untuk c tetap mana pun — tebakannya sekadar salah, dan tidak ada aljabar yang bisa memaksanya berhasil. Substitusi membuktikan sebuah tebakan benar atau mengungkap bahwa ia salah; ia tidak menghasilkan tebakan untuk Anda (itulah fungsi pohon rekursi).

3.8 Menyelesaikan Rekurensi III: Master Theorem

Metode pohon rekursi dan substitusi sama-sama berhasil, tetapi keduanya membutuhkan sedikit kecerdikan. Untuk kasus umum T(n) = aT(n/b) + f(n), Master Theorem memberi jawaban langsung dengan membandingkan f(n) — biaya membagi dan menggabungkan — terhadap n^(log_b a) — biaya yang akan dihasilkan jika daun-daun pohon rekursi yang mengerjakan semua pekerjaan.

Master Theorem (bentuk CLRS)

Misalkan a ≥ 1 dan b > 1 adalah konstanta dan f(n) adalah suatu fungsi, dan T(n) = aT(n/b) + f(n) (dengan T(n) = Θ(1) untuk n di bawah suatu ambang batas konstan). Maka: Kasus 1: jika f(n) = O(n^(log_b a − ε)) untuk suatu konstanta ε > 0 (f tumbuh secara POLINOMIAL lebih lambat dari n^(log_b a)), maka T(n) = Θ(n^(log_b a)). Kasus 2: jika f(n) = Θ(n^(log_b a)) (f tumbuh dengan LAJU YANG SAMA), maka T(n) = Θ(n^(log_b a) · log n). Kasus 3: jika f(n) = Ω(n^(log_b a + ε)) untuk suatu konstanta ε > 0 (f tumbuh secara POLINOMIAL lebih cepat), DAN syarat keteraturan a·f(n/b) ≤ c·f(n) berlaku untuk suatu konstanta c < 1 dan semua n yang cukup besar, maka T(n) = Θ(f(n)).



Gambar 3.4 — Ketiga kasus berkorespondensi dengan di mana biaya pohon rekursi terkonsentrasi: di daun-daun (Kasus 1), tersebar merata di semua level (Kasus 2), atau di akar (Kasus 3).

Intuisi di balik ketiga kasus persis seperti yang ditunjukkan Gambar 3.4. Pada Kasus 1, f(n) kecil relatif terhadap n^(log_b a), sehingga daun-daun — tempat a^(kedalaman) panggilan rekursif berakhir — menyumbang total biaya terbesar, dan biaya daun itulah yang mendominasi. Pada Kasus 2, setiap level pohon rekursi menyumbang jumlah yang sama (seperti pada contoh Merge Sort, Bagian 3.6), sehingga totalnya adalah (biaya per level) × (banyak level) = Θ(n^(log_b a) log n). Pada Kasus 3, f(n) cukup besar

sehingga panggilan pertama saja sudah mendominasi apa pun yang mungkin ditambahkan anak-anaknya, asalkan syarat keteraturan menyingkirkan kemungkinan f(n) berosilasi dengan cara yang merusak ini — syarat teknis yang dipenuhi oleh setiap fungsi polinomial f(n) yang akan Anda jumpai di mata kuliah ini.

> **Master Theorem tidak selalu berlaku**
> Ada celah antara Kasus 1 dan 2 (f(n) lebih kecil secara polinomial adalah Kasus 1, tetapi hanya lebih kecil secara asimtotik tanpa celah polinomial bukan keduanya) dan celah serupa antara 2 dan 3. Rekurensi seperti T(n) = 2T(n/2) + n/log n tidak masuk ke satu pun dari tiga kasus — Master Theorem sekadar tidak memberi jawaban, dan Anda harus kembali ke metode pohon rekursi atau substitusi.

Menerapkan ini pada rekurensi Merge Sort, diturunkan di Bagian 3.9.5: T(n) = 2T(n/2) + Θ(n). Di sini a = 2, b = 2, sehingga n^(log_b a) = n^(log$_2$2) = n^1 = n. Karena f(n) = Θ(n) = Θ(n^(log_b a)), kita berada di Kasus 2, memberikan T(n) = Θ(n log n) — cocok persis dengan yang ditemukan pohon rekursi di Bagian 3.6 lewat penjumlahan langsung.

3.9 Studi Kasus: Merge Sort — Analisis Lengkap

Merge Sort adalah algoritma pengurutan divide-and-conquer sesungguhnya yang pertama kita jumpai, dan algoritma yang menjadikan setiap alat yang dibangun bab ini konkret.

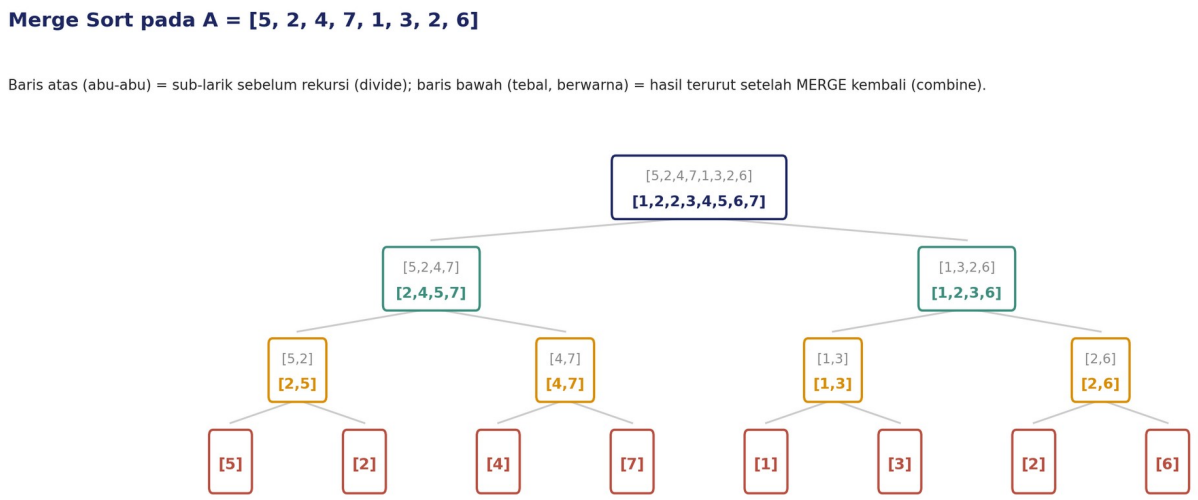
3.9.1 Strateginya

Divide: bagi larik n-elemen menjadi dua separuh berukuran sekitar n/2 elemen masing-masing. Conquer: urutkan setiap separuh secara rekursif. Combine: gabungkan kedua separuh yang kini terurut dengan menggabungkannya menjadi satu larik terurut. Kasus basis adalah larik berisi 0 atau 1 elemen, yang trivial sudah terurut.

```
MERGE-SORT(A, p, r)
  if p < r                          // lebih dari satu elemen?
      q = floor((p + r) / 2)        // titik tengah
      MERGE-SORT(A, p, q)           // conquer separuh kiri
      MERGE-SORT(A, q + 1, r)       // conquer separuh kanan
      MERGE(A, p, q, r)             // combine
```

3.9.2 Menelusuri Proses Divide-and-Combine

Mari kita urutkan A = [5, 2, 4, 7, 1, 3, 2, 6]. Gambar 3.5 menunjukkan rekursi lengkapnya: label atas (abu-abu) tiap simpul adalah sub-larik sebelum rekursi, dan label bawah (tebal, berwarna) adalah hasil terurut setelah MERGE kembali. Membaca pohon dari atas ke bawah menunjukkan fase DIVIDE membelah larik hingga elemen tunggal; membacanya dari bawah ke atas menunjukkan fase COMBINE menggabungkan run-run terurut kembali menjadi larik yang sepenuhnya terurut.



Gambar 3.5 — Merge Sort pada A = [5, 2, 4, 7, 1, 3, 2, 6]. Delapan daun tunggal digabung berpasangan, lalu berpasangan-pasangan, hingga akar memegang larik yang sepenuhnya terurut [1, 2, 2, 3, 4, 5, 6, 7].

3.9.3 Subrutin MERGE

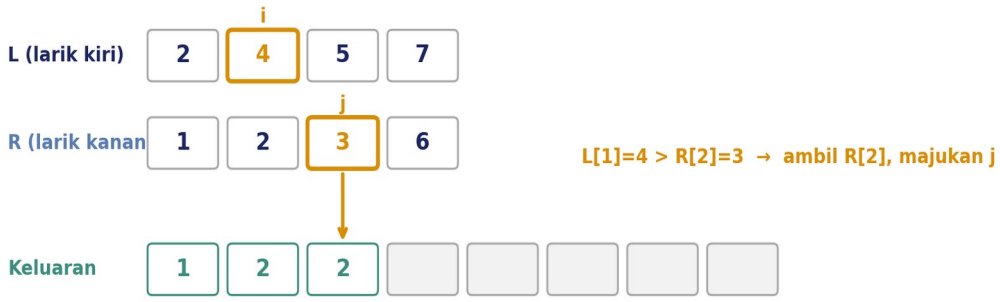
MERGE(A, p, q, r) mengasumsikan A[p..q] dan A[q+1..r] masing-masing sudah terurut, dan menggabungkan keduanya menjadi satu run terurut yang menempati A[p..r]. Idenya: salin kedua run ke larik sementara L dan R, lalu berulang kali bandingkan elemen terdepan keduanya, salin yang lebih kecil ke keluaran dan majukan pointer run tersebut, hingga salah satu run habis — pada titik itu sisa run lainnya disalin langsung, karena sudah terurut.

```
MERGE(A, p, q, r)
  n1 = q - p + 1                    // panjang run kiri
  n2 = r - q                        // panjang run kanan
  misalkan L[1..n1] dan R[1..n2] larik baru
  for i = 1 to n1:  L[i] = A[p + i - 1]
  for j = 1 to n2:  R[j] = A[q + j]
  i = 1, j = 1
  for k = p to r:                   // isi A[p..r] secara berurutan
      if j > n2 or (i <= n1 and L[i] <= R[j])
          A[k] = L[i];  i = i + 1
      else
          A[k] = R[j];  j = j + 1
```

Gambar 3.6 menunjukkan snapshot tengah-merge, diambil dari merge level-atas [2,4,5,7] dan [1,2,3,6] pada penelusuran di atas. Tiga elemen telah ditempatkan di keluaran (1, 2, 2); pointer kini berada di L[1]=4 dan R[2]=3. Karena elemen terdepan R lebih kecil, ia disalin berikutnya dan pointernya maju — persis aturan yang dikodekan pseudokode sebagai perbandingan if/else di dalam loop for.

Anatomi MERGE: Dua Larik Terurut, Satu Pointer Masing-Masing

Snapshot tengah-merge: 3 elemen sudah ditempatkan. Bandingkan kedua pointer, salin yang lebih kecil, majukan pointer itu.



Gambar 3.6 — Anatomi satu langkah MERGE: bandingkan elemen terkini kedua pointer, salin yang lebih kecil ke keluaran, majukan pointer itu. Ulangi hingga salah satu run habis.

3.9.4 Membuktikan MERGE Benar: Invarian Loop

Apa yang harus selalu benar setiap kali melalui loop for-k?

Persis gaya pertanyaan yang sama yang ditanyakan Kuliah 2 tentang Insertion Sort — dan jawabannya bergaya persis sama.

Invarian Loop untuk MERGE

Pada awal setiap iterasi loop for k, sub-larik A[p..k−1] berisi k−p elemen terkecil dari L dan R gabungan, dalam urutan terurut, dan L[i] serta R[j] adalah elemen terkecil dari larik masing-masing yang belum disalin ke A (atau, jika i > n1 atau j > n2, run itu telah habis).

- Inisialisasi (k = p): A[p..p−1] kosong, trivial berisi 0 elemen terkecil dalam urutan terurut, dan i = j = 1 secara benar menunjuk ke depan setiap run yang baru disalin. ✓
- Pemeliharaan: setiap iterasi membandingkan L[i] dan R[j] (memperlakukan run yang habis sebagai efektif tak hingga) dan menyalin mana pun yang lebih kecil ke A[k], lalu memajukan pointer itu. Karena L dan R masing-masing terurut secara individual, elemen yang disalin adalah elemen terkecil yang tersisa secara keseluruhan, sehingga A[p..k] kini berisi k−p+1 elemen terkecil dalam urutan terurut, mempertahankan invarian untuk k+1. ✓
- Terminasi (k = r + 1): invarian memberi tahu kita A[p..r] berisi seluruh r−p+1 elemen dari L dan R gabungan, dalam urutan terurut — persis postkondisi yang harus ditetapkan MERGE. ✓

Wawasan kunci

Bukti kebenaran MERGE memiliki bentuk tiga-bagian yang identik (Inisialisasi, Pemeliharaan, Terminasi) seperti Insertion Sort di Kuliah 2 — karena MERGE, meski terasa berbeda, tetap pada dasarnya adalah algoritma iteratif dengan loop for, dan setiap algoritma iteratif yang benar mengizinkan gaya bukti ini.

3.9.5 Menurunkan dan Menyelesaikan Rekurensi

MERGE-SORT(A, p, r) pada n = r − p + 1 elemen melakukan 2 panggilan rekursif, masing-masing pada sub-masalah berukuran n/2, ditambah satu panggilan ke MERGE, yang melakukan Θ(n) pekerjaan (menyalin n elemen ke L dan R, lalu n perbandingan/salinan lagi kembali ke A). Ini memberikan rekurensi:

> **Rekurensi Merge Sort**
> T(n) = 2T(n/2) + Θ(n), dengan T(1) = Θ(1).

Pohon rekursi Bagian 3.6 sudah menyelesaikan persis rekurensi ini (dengan Θ(n) dituliskan sebagai cn) dan menemukan Θ(n log n). Master Theorem Bagian 3.8 memberikan jawaban yang sama secara langsung: a = 2, b = 2, n^(log_b a) = n, dan f(n) = Θ(n) cocok dengan n^(log_b a) persis, sehingga Kasus 2 berlaku dan T(n) = Θ(n log n).

Wawasan kunci

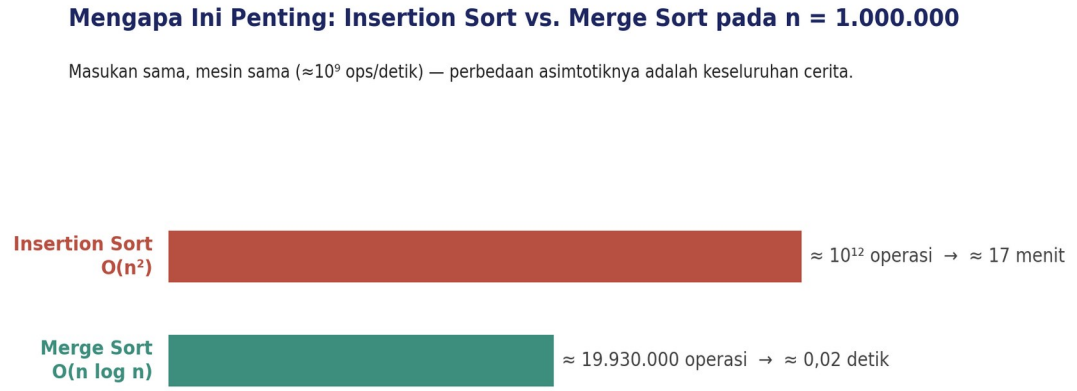
Tiga metode independen — pohon rekursi, substitusi, dan Master Theorem — semuanya sepakat: Merge Sort berjalan dalam waktu Θ(n log n), pada kasus TERBAIK, TERBURUK, dan RATA-RATA sekaligus. Berbeda dengan Insertion Sort, Merge Sort tidak punya masukan buruk — langkah divide selalu membagi rata terlepas dari datanya, sehingga tidak ada larik terbalik-urut yang bisa menjebaknya ke perilaku kuadratik.

Fase 5 — Implementasi

```cpp
/*
 * Chapter 3 - Case Study: Merge Sort
 * Paradigm : Divide and Conquer
 *
 * Pseudocode (MERGE-SORT, 1-indexed in the textbook; 0-indexed below):
 *   MERGE-SORT(A, p, r):
 *     if p < r:
 *         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): merges the two sorted runs A[p..q] and A[q+1..r]
 *     using two temporary arrays and a sentinel-free two-pointer sweep.
 *
 * Recurrence: T(n) = 2T(n/2) + Theta(n)  =>  T(n) = Theta(n log n)
 * (see textbook Section 3.9, solved via recursion tree and Master Theorem).
 */
#include <cstdio>
#include <vector>
using namespace std;

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);
    int i = 0, j = 0, k = p;
    while (i < (int)L.size() && j < (int)R.size()) {
        if (L[i] <= R[j]) A[k++] = L[i++];
        else A[k++] = R[j++];
    }
    while (i < (int)L.size()) A[k++] = L[i++];
    while (j < (int)R.size()) A[k++] = R[j++];
}

void mergeSort(vector<int>& A, int p, int r) {
    if (p < r) {
        int q = (p + r) / 2;
        mergeSort(A, p, q);
        mergeSort(A, q + 1, r);
        merge(A, p, q, r);
    }
}

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    vector<int> A(n);
    for (int i = 0; i < n; i++) scanf("%d", &A[i]);
    if (n > 0) mergeSort(A, 0, n - 1);
    for (int i = 0; i < n; i++)
        printf("%d%c", A[i], i + 1 < n ? ' ' : '\n');
    return 0;
}
```

3.10 Mengapa Ini Penting: Merge Sort vs. Insertion Sort

Θ(n log n) berbanding Θ(n^2) terdengar seperti perbedaan sedang hingga Anda memasukkan angka nyata. Gambar 3.7 memakai angka persis dari Tabel 2.1 Kuliah 2, pada n = 1.000.000 di mesin yang mengeksekusi sekitar 10^9 operasi sederhana per detik.



Gambar 3.7 — Pada satu juta elemen, Θ(n²) Insertion Sort menjadi penantian 17 menit; Θ(n log n) Merge Sort selesai sebelum Anda sempat berkedip. Ini bukan perbaikan faktor-konstan — ini perbedaan jenis.

Harga yang dibayar Merge Sort untuk percepatan ini adalah ruang: berbeda dengan Insertion Sort, yang mengurutkan di tempat memakai O(1) memori ekstra, MERGE membutuhkan Θ(n) ruang tambahan untuk larik sementara L dan R. Ini adalah trade-off yang nyata dan jujur — waktu Θ(n log n) dengan harga ruang Θ(n) — dan ini adalah kali pertama mata kuliah ini harus menimbang waktu terhadap ruang secara eksplisit. Anda akan menjumpai trade-off ini berulang-ulang sepanjang sisa mata kuliah.

3.11 Paradigma Perancangan Ditinjau Ulang: Divide and Conquer, Diformalkan

Kuliah 2 memperkenalkan divide-and-conquer secara informal lewat Binary Search. Merge Sort memungkinkan kita menyatakan resep tiga-langkah paradigma ini secara presisi, kini setelah kita punya kosakata untuk menganalisis tiap langkah:

- DIVIDE masalah menjadi a sub-masalah, masing-masing berukuran n/b (untuk Merge Sort: a = 2, b = 2 — dibelah menjadi dua separuh sama besar).
- CONQUER setiap sub-masalah dengan menyelesaikannya secara rekursif. Jika ukuran sub-masalah cukup kecil, selesaikan langsung (kasus basis).
- COMBINE solusi sub-masalah menjadi solusi untuk masalah asli (untuk Merge Sort: langkah MERGE berwaktu Θ(n)).

Waktu eksekusi algoritma divide-and-conquer mana pun yang dibangun dengan cara ini persis rekurensi umum dari Bagian 3.5: T(n) = a·T(n/b) + f(n), di mana f(n) adalah biaya langkah divide dan combine. Inilah persis mengapa Master Theorem dirumuskan dalam istilah a, b, dan f(n) — ia dibangun untuk menganalisis resep persis ini.

Wawasan kunci

Divide-and-conquer menukar argumen global yang lebih sulit dilihat (mengapa seluruh larik terurut?) dengan argumen lokal yang lebih mudah dilihat (jika kedua separuh terurut, dan MERGE benar, seluruh larik terurut). Gaya argumen kebenaran rekursif ini — asumsikan panggilan rekursif benar, lalu buktikan langkah combine mempertahankan kebenaran — akan berulang untuk setiap algoritma divide-and-conquer berikutnya, termasuk Quicksort di Kuliah 5.

3.12 Profitina dalam Praktik: Di Mana Bab Ini Muncul dalam Produk Nyata

Tentang kotak ini

Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) nyata buatan Indonesia oleh penulis, tanpa membocorkan detail implementasi proprietary.

Profitina secara berkala perlu menggabungkan dua daftar sinyal bisnis yang sudah terperingkat — misalnya, sebuah batch bisnis yang baru dipindai ulang (sudah terurut berdasarkan skor visibilitas dalam batch tersebut) yang harus dilipat ke dalam leaderboard suatu wilayah yang sudah terurut. Alih-alih mengurutkan ulang daftar gabungan dari awal dalam waktu Θ(n log n), para insinyur Profitina menerapkan persis langkah MERGE dari Bagian 3.9.3: karena kedua daftar sudah terurut secara individual, satu kali lintasan linear Θ(n) dengan dua pointer menghasilkan hasil gabungan yang sepenuhnya terurut — kemenangan asimtotik yang sama yang dieksploitasi pengurutan divide-and-conquer di setiap level rekursinya, diterapkan sekali di level teratas pipeline yang jauh lebih besar. Mengenali “kedua separuh sudah terurut” sebagai prasyarat untuk merge murah Θ(n), alih-alih pengurutan ulang mahal Θ(n log n), adalah persis jenis pemikiran asimtotik yang dilatih bab ini.

3.13 Ringkasan Bab dan Poin-Poin Kunci

- O(g(n)), Ω(g(n)), dan Θ(g(n)) adalah definisi presisi yang melibatkan konstanta c dan ambang batas n_0 — bukan deskripsi kabur tentang "seberapa cepat" suatu fungsi tumbuh.
- Rekurensi T(n) = aT(n/b) + f(n) mendeskripsikan waktu eksekusi algoritma divide-and-conquer yang membelah menjadi a sub-masalah berukuran n/b, menghabiskan f(n) waktu di luar panggilan rekursif.
- Tiga cara menyelesaikan rekurensi: pohon rekursi (jumlahkan setiap level — baik untuk menebak), substitusi (buktikan tebakan dengan induksi — ketat tetapi butuh tebakan), dan Master Theorem (tiga kasus berdasarkan membandingkan f(n) dengan n^(log_b a) — cepat, tetapi tidak selalu berlaku).
- Merge Sort membelah larik menjadi dua, mengurutkan tiap separuh secara rekursif, dan menggabungkan hasilnya dalam waktu Θ(n) lewat sapuan dua-pointer — memberikan T(n) = 2T(n/2) + Θ(n) = Θ(n log n) di setiap kasus, terbaik, terburuk, dan rata-rata sekaligus.

- Kebenaran MERGE mengikuti resep invarian loop Inisialisasi/Pemeliharaan/Terminasi yang sama seperti Insertion Sort di Kuliah 2.
- Θ(n log n) berbanding Θ(n^2) bukan perbedaan kecepatan kecil — pada n = 1.000.000 ini adalah perbedaan antara milidetik dan 17 menit.
- Harga kecepatan Merge Sort adalah ruang ekstra Θ(n) — trade-off waktu/ruang eksplisit pertama pada mata kuliah ini.

> ***"Untuk memahami rekursi, Anda harus lebih dulu memahami rekursi."***
>
> **— lelucon lama ilmu komputer, tetap tidak lucu pada pengulangan ke-2^n**
>
> *(Kalau leluconnya tidak lucu, lihat lagi: kutipan ini.)*

Kuliah 5 akan menerapkan semua yang dibangun di bab ini — rekurensi, Master Theorem, resep divide-and-conquer — pada Quicksort, algoritma yang membelah secara tidak merata namun tetap mencapai O(n log n) rata-rata, secara in-place. Kuliah 6 dan 7 lalu meninggalkan pengurutan menuju wilayah baru: pohon pencarian biner seimbang, dan program dinamis.

3.14 Pertanyaan Tinjauan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Kuliah 4 (Kuis 1).

1. Dengan memakai definisi O secara langsung (tunjukkan c dan n_0), buktikan bahwa $5n^3 + 100n^2 + 1 = O(n^3)$.
2. Apakah $2^{n+1} = O(2^n)$? Apakah $n^2 = O(2^n)$? Justifikasi kedua jawaban memakai definisi dari Bagian 3.2.
3. Selesaikan T(n) = 4T(n/2) + n memakai baik metode pohon rekursi maupun Master Theorem, dan konfirmasi kedua jawaban sepakat. (Petunjuk: ini Kasus 1.)
4. Selesaikan T(n) = T(n/2) + 1 memakai Master Theorem. Algoritma familiar apa dari Kuliah 2 yang memiliki persis rekurensi ini?
5. Telusuri MERGE-SORT dengan tangan pada A = [8, 3, 5, 1, 9, 2], gambar pohon rekursi lengkap dengan gaya Gambar 3.5, dan verifikasi jawaban akhir Anda terurut.
6. Jelaskan mengapa waktu eksekusi Θ(n log n) Merge Sort berlaku bahkan pada kasus TERBURUK, berbeda dengan Insertion Sort, yang Θ(n) pada kasus terbaik tetapi Θ(n^2) pada kasus terburuk.

Lampiran 3.A — Log Verifikasi untuk Kode Sumber Bab 3

Seperti pada bab-bab sebelumnya, setiap program yang disertakan dalam buku ini dikompilasi dan dijalankan terhadap contoh yang diverifikasi dengan tangan sebelum disertakan. Program di bawah dikompilasi dengan `g++ -O2 -std=c++17 -Wall` (hanya peringatan scanf yang jinak, nol galat) dan menghasilkan persis keluaran yang diprediksi teks di atas.

```
$ printf "8\n5 2 4 7 1 3 2 6\n" | ./01_merge_sort
1 2 2 3 4 5 6 7                     # cocok dengan penelusuran Bagian 3.9.2
(Gambar 3.5)

$ printf "6\n5 2 4 6 1 3\n" | ./01_merge_sort
1 2 3 4 5 6                         # larik yang sama yang diurutkan Kuliah 2
dengan Insertion Sort — hasil sama, algoritma berbeda
```

Latihan SPOJ — buku pendamping
Teknik bab ini diuji sungguhan di juri daring SPOJ. Buka Bab 3 & 9 buku pendamping “SPOJ Competitive Programming” (folder SPOJ, BI & EN) untuk pola pikir bagi-dan-taklukkan serta penganggaran kompleksitas berbasis rekurens — lengkap dengan tips, trik, dan hint agar solusi Anda Accepted, tanpa membocorkan jawaban.

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, edisi ke-4. MIT Press, 2022. (Bab 3–4).

[2] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, edisi ke-4. Lulu Press, 2020–2023.

[3] D. E. Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, edisi ke-2. Addison-Wesley, 1998.

[4] J. von Neumann. "First Draft of a Report on the EDVAC" (1945) — deskripsi awal prosedur pengurutan bergaya merge-sort yang diimplementasikan untuk EDVAC, banyak dikutip sebagai deskripsi tertulis pertama Merge Sort.

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 4 · BAB LATIHAN

Soal Latihan: Fondasi Hingga Merge Sort

Tidak ada materi baru di sini — hanya dua belas soal yang dirancang untuk memastikan Kuliah 1 sampai 3 benar-benar melekat, sebelum Kuis 1.

Tujuan Pembelajaran — setelah mengerjakan bab ini Anda mampu:
(1) Mengidentifikasi properti mana dari lima properti pendefinisi algoritma yang dilanggar sebuah prosedur cacat, dan memperbaikinya. (2) Menghitung operasi model RAM secara tepat, menurunkan waktu eksekusi kasus terbaik/terburuk/rata-rata, dan membuktikan invarian loop untuk varian Insertion Sort. (3) Mengurutkan fungsi berdasarkan pertumbuhan asimtotik dan mengadaptasi Binary Search ke spesifikasi baru. (4) Membuktikan batas asimtotik langsung dari definisi O, termasuk membuktikan suatu batas TIDAK berlaku. (5) Menyelesaikan rekurensi baru dengan pohon rekursi, substitusi, dan Master Theorem — serta mengenali kapan Master Theorem tidak berlaku. (6) Memperluas langkah MERGE dari Merge Sort untuk menyelesaikan soal terkait, dan mengklasifikasikan algoritma berdasarkan paradigma perancangan.

4.1 Rekap: Kuliah 1–3 Secara Singkat

Kuliah 1 mendefinisikan apa sesungguhnya sebuah algoritma — prosedur yang finite, definite, dan efektif dengan masukan dan keluaran yang terspesifikasi jelas — dan menelusuri enam soal yang menunjukkan bahwa tugas yang sama bisa punya banyak solusi benar dengan kualitas yang sangat berbeda. Kuliah 2 memberi kita cara mengukur kualitas itu yang bebas dari mesin: model RAM, yang menghitung operasi alih-alih detik, diterapkan penuh pada Insertion Sort (dengan bukti kebenaran invarian loop yang ketat) serta pada Linear dan Binary Search. Kuliah 3 kemudian membuat bahasa Big-O informal dari Kuliah 2 menjadi presisi secara matematis — definisi eksak O, Ω, dan Θ dengan konstanta dan ambang batas — dan membangun tiga alat (pohon rekursi, substitusi, dan Master Theorem) untuk menyelesaikan rekurensi yang dihasilkan algoritma rekursif seperti Merge Sort.

Gambar 4.1 — Bab ini berada di Kuliah 4 dalam peta jalan 16-kuliah DAA: sebuah checkpoint, bukan topik baru, tepat sebelum Kuis 1.

4.2 Cara Menggunakan Bab Latihan Ini

Apa bab ini — dan apa yang BUKAN

Ini adalah Bab Latihan: bab praktik, bukan bab konten baru. Setiap soal di bawah berakar pada materi spesifik dari Kuliah 1–3 (lihat Gambar 4.2). Setiap soal disertai PETUNJUK — dorongan ke arah cara berpikir yang benar — tetapi sengaja TIDAK disertai solusi lengkap atau source code jawaban. Mengerjakan soal itu sendiri, meski tidak sempurna, mengajarkan jauh lebih banyak daripada membaca jawaban jadi.

Gambar 4.2 — Setiap soal di Bagian 4.3 berakar pada salah satu dari tiga kuliah sebelumnya.

Sebelum Anda mulai

Cobalah setiap soal sungguh-sungguh sebelum membaca petunjuknya. Jika buntu, baca HANYA petunjuknya — bukan solusi — lalu coba lagi. Inilah persis disiplin yang akan dihargai oleh Kuis 1 open-book di Bagian 4.4: ujian mengizinkan referensi, tetapi bukan pengganti penalaran Anda sendiri.

4.3 Soal Latihan

4.3.1 Fondasi & Model RAM

Soal 1 *[Mudah]*. Anda diberi sebuah "resep" langkah-demi-langkah yang mengandung instruksi "ulangi langkah 3 sampai Anda puas." Properti algoritma mana yang paling jelas dilanggar prosedur ini? Jelaskan dengan tepat mengapa, dan tuliskan ulang instruksi tersebut agar memenuhi definisi presisi sebuah algoritma.

Petunjuk

Pikirkan kembali properti Finiteness dan Definiteness dari Bab 1 — properti mana yang secara eksplisit menuntut kondisi berhenti yang OBJEKTIF, bukan subjektif?

Soal 2 *[Mudah]*. Untuk potongan pseudokode "for i = 1 to n: for j = 1 to i: A[j] = A[j] + 1", hitung secara TEPAT berapa kali assignment A[j] = A[j] + 1 dijalankan, sebagai fungsi dari n — bukan cuma Big-O-nya. Berikan bentuk tertutup.

> **Petunjuk**
> Ini adalah deret aritmetika 1 + 2 + … + n yang sama persis yang sudah Anda pakai untuk biaya kasus terburuk Insertion Sort di Bab 2 — rumus penjumlahannya identik.

4.3.2 Insertion Sort & Analisis Kasus

Soal 3 *[Sedang]*. INSERTION-SORT biasa menyisipkan tiap elemen ke KIRI. Tuliskan invarian loop untuk versi yang discan dari KANAN ke KIRI (menyisipkan dari elemen terakhir menuju elemen pertama, tetap membangun hasil terurut menaik). Buktikan Inisialisasi dan Terminasi secara eksplisit (Pemeliharaan boleh diringkas).

> **Petunjuk**
> Invariannya nyaris identik dengan Bab 2 — hanya rentang indeks yang sudah terurut yang berjalan ke arah berlawanan, bukan logika yang mendasarinya.

Soal 4 *[Sedang]*. Untuk larik masukan yang HAMPIR terurut — tepat satu pasang elemen tertukar posisinya, sisanya sudah urut — apakah Insertion Sort berperilaku lebih dekat ke kasus TERBAIK, TERBURUK, atau di antara keduanya? Berikan alasan berbasis definisi t_j dari Bab 2, bukan sekadar intuisi.

> **Petunjuk**
> Hitung t_j untuk SETIAP posisi j — berapa banyak nilai j yang butuh pergeseran lebih dari satu posisi, dan seberapa jauh sesungguhnya pergeserannya?

4.3.3 Pertumbuhan Fungsi & Pencarian

Soal 5 *[Sedang]*. Urutkan lima fungsi berikut dari yang tumbuh PALING LAMBAT ke PALING CEPAT secara asimtotik, dan untuk SATU pasang yang menurut Anda paling mudah disalahpahami, jelaskan mengapa urutannya begitu: n log n, 2^n, n^2, log n, n^1.5.

> **Petunjuk**
> Bandingkan kelimanya pada dasar yang sama — misalnya, ambil log dari masing-masing, atau amati rasio f(n)/g(n) saat $n \to \infty$ — alih-alih menebak dari "bentuknya".

Soal 6 *[Sedang]*. Modifikasi Binary Search standar agar, jika target q muncul berulang kali dalam larik terurut, fungsi mengembalikan indeks KEMUNCULAN PERTAMA (bukan sembarang kemunculan). Jelaskan perubahan pada kondisi pencarian, dan argumentasikan mengapa kompleksitasnya tetap O(log n).

> **Petunjuk**
> Alih-alih berhenti begitu A[mid] == q ditemukan, pikirkan bagaimana caranya tetap "mempersempit ke kiri" bahkan setelah sebuah kecocokan ditemukan.

4.3.4 Notasi Asimtotik Formal

Soal 7 *[Sulit]*. Buktikan dari DEFINISI (tunjukkan c dan n_0 eksplisit) bahwa $5n^3 + 100n^2 + 1 = O(n^3)$, TETAPI $5n^3 + 100n^2 + 1 \neq O(n^2)$.

> **Petunjuk**
> Bagian $O(n^3)$ mengikuti pola bukti Bagian 3.3 secara langsung. Untuk bagian "$\neq O(n^2)$", coba bukti dengan KONTRADIKSI — asumsikan ada c dan n_0 yang berlaku, lalu tunjukkan pertidaksamaannya pasti gagal untuk n yang cukup besar.

4.3.5 Rekurensi

Soal 8 *[Sulit]*. Selesaikan T(n) = 3T(n/2) + n menggunakan metode pohon rekursi (Bagian 3.6). Berapa total biaya di level ke-i? Berapa banyak level? Mana yang mendominasi jumlah keseluruhan — level atas atau level bawah?

> **Petunjuk**
> Berbeda dari T(n) = 2T(n/2) + cn di buku, di mana setiap level berbiaya sama, di sini jumlah simpul bertumbuh 3× tiap level sementara biaya per simpul hanya menyusut 2× — perhatikan baik-baik rasio itu.

Soal 9 *[Sulit]*. Buktikan dengan induksi bahwa T(n) = T(n/2) + 1 adalah O(log n). Tuliskan tebakan Anda, langkah induktif lengkap, dan syarat kasus basisnya.

> **Petunjuk**
> Tebakan berbentuk $T(n) \leq c \cdot \log_2 n$. Perhatikan baik-baik apa yang terjadi pada suku "+1" di sini dibanding suku "+n" di Bagian 3.7 — konstanta c yang dibutuhkan di sini ternyata jauh lebih longgar.

Soal 10 *[Sulit]*. Untuk MASING-MASING rekurensi berikut, tentukan apakah Master Theorem berlaku, dan jika ya kasus mana serta hasil Θ-nya: (a) T(n) = 4T(n/2) + n, (b) T(n) = 4T(n/2) + n^2, (c) T(n) = 4T(n/2) + n^3, (d) T(n) = 2T(n/2) + n/log n.

> **Petunjuk**
> Hitung n^(log_b a) = n^2 untuk keempat rekurensi ini (a = 4, b = 2). Salah satu dari keempat soal ini sengaja dirancang agar Master Theorem TIDAK berlaku — Bagian 3.8 sudah menjelaskan tepat celah semacam apa yang harus dicari.

4.3.6 Merge Sort & Paradigma Perancangan

Soal 11 *[Sulit]*. Sepasang indeks (i, j) dengan i < j disebut INVERSI jika A[i] > A[j] — ukuran seberapa "tidak terurut" sebuah larik. Jelaskan, tanpa menuliskan kode penuh, bagaimana MERGE dari Bagian 3.9.3 dapat dimodifikasi untuk MENGHITUNG jumlah total inversi sambil tetap mengurutkan larik, dalam waktu total Θ(n log n) yang sama seperti Merge Sort biasa.

> **Petunjuk**
> Setiap kali algoritma mengambil elemen dari R (larik kanan) karena ia lebih kecil dari elemen yang sedang ditunjuk L, berapa banyak inversi baru yang baru saja "ditemukan" dalam satu operasi itu?

Soal 12 *[Sedang]*. Untuk masing-masing dari tiga algoritma berikut, klasifikasikan sebagai INCREMENTAL atau DIVIDE-AND-CONQUER, dan berikan SATU kalimat justifikasi: (a) Insertion Sort, (b) Merge Sort, (c) Binary Search.

> **Petunjuk**
> Paradigma incremental membangun solusi satu elemen setiap kali sambil mempertahankan invarian; D&C membelah masalah menjadi sub-masalah independen lalu menggabungkannya. Binary Search agak menjebak — ia "membelah" tapi hanya pernah mengerjakan SATU sisi, bukan dua-duanya.

4.4 Format Kuis 1: Esai Open-Book Take-Home

Kuis 1 adalah tugas esai open-book, take-home yang mencakup persis materi yang diulas di bab ini — tidak lebih. Anda punya waktu satu minggu untuk mengumpulkan jawaban tertulis dengan penalaran lengkap; jawaban akhir yang benar tanpa justifikasi hanya mendapat sedikit nilai, karena inti dari kuis berformat esai adalah melihat BAGAIMANA Anda berpikir, bukan sekadar apa kesimpulannya.

> **Open-book berarti referensi, bukan rekan penulis**
> Anda boleh membuka buku teks, catatan, dan slide kuliah saat menjawab. Anda TIDAK BOLEH berkolaborasi dengan teman sekelas atau mengumpulkan penalaran yang bukan sungguh-sungguh milik Anda sendiri — kuis open-book menguji apakah Anda bisa MENERAPKAN apa yang telah dipelajari, tanpa pengawasan, yang persis itulah makna "merancang dan menganalisis algoritma" dalam praktik. Jika Anda mengutip sumber secara langsung, cantumkan sitasinya.

Kriteria	Yang dinilai	Bobot
Ketepatan penalaran	Apakah logika, bukti, atau turunannya benar-benar valid — bukan cuma jawaban akhirnya?	40%
Kejelasan bukti/turunan	Bisakah pembaca mengikuti tiap langkah tanpa harus menebak maksud Anda?	25%
Penggunaan notasi yang tepat	Apakah O/Ω/Θ, rekurensi, dan invarian dinyatakan secara presisi, sesuai definisi Bab 3?	20%

Presentasi	Apakah jawaban tersusun rapi, terbaca, dan mudah dinilai?	15%

4.5 Profitina dalam Praktik: Self-Review Sebelum Submit

> **Tentang kotak ini**
> Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) nyata buatan Indonesia oleh penulis, tanpa membocorkan detail implementasi proprietary.

Sebelum sebuah perubahan algoritma baru di-deploy ke produksi di Profitina, para insinyur menjalankannya melalui checklist self-review singkat — mirip sekali dengan Lampiran 4.A di bawah — sebelum meminta pasang mata kedua: apakah klaim kompleksitasnya benar-benar diturunkan, bukan sekadar ditebak? Apakah kasus tepi (masukan kosong, elemen semua sama, masukan yang sudah terurut) sudah dipertimbangkan? Kebiasaan self-checking yang adversarial sebelum review eksternal inilah yang persis membuat format ujian open-book take-home berhasil: disiplin memverifikasi penalaran Anda sendiri, alih-alih mempercayai naluri pertama, adalah keterampilan yang dilatih bab ini.

4.6 Ringkasan Bab

- Bab ini tidak memperkenalkan materi baru — ia adalah checkpoint yang mencakup Kuliah 1 sampai 3.
- Dua belas soal mencakup seluruh rentang yang diulas: definisi algoritma, model RAM, Insertion Sort dan invariannya, pertumbuhan fungsi, pencarian, bukti asimtotik formal, rekurensi (pohon rekursi, substitusi, Master Theorem), Merge Sort, dan paradigma perancangan.
- Setiap soal menyertakan petunjuk, bukan solusi — mengerjakan penalarannya sendiri adalah intinya.
- Kuis 1 adalah esai open-book, take-home: referensi diperbolehkan, tetapi penalarannya harus milik Anda sendiri, dan ketepatan PENALARAN (bukan cuma jawaban akhir) adalah mayoritas nilai.

“Sebuah algoritma harus DILIHAT untuk bisa DIPERCAYA.”

— Donald E. Knuth

(Membaca hint tidak terhitung “melihat” — mengerjakan soalnya baru terhitung.)

Lampiran 4.A — Daftar Periksa Mandiri (Tidak Dinilai)

Sebelum mengumpulkan jawaban apa pun — pada soal-soal bab ini maupun pada Kuis 1 itu sendiri — jalankan melalui daftar periksa ini. Hanya butuh beberapa menit dan menangkap sebagian besar kesalahan yang paling sering dilihat penilai.

- Apakah saya menyatakan definisi formal (bukan cuma intuisi) ketika diminta membuktikan sesuatu?
- Dalam bukti O/Ω/Θ apa pun, apakah saya menunjukkan konstanta c dan n_0 eksplisit — bukan sekadar menyatakan batasnya?
- Dalam bukti substitusi, apakah saya memeriksa kasus basis DAN langkah induktif secara terpisah, dan apakah saya memeriksa bahwa langkah induktif itu benar-benar berjalan secara aljabar (bukan cuma "terlihat benar")?
- Jika saya memakai Master Theorem, apakah saya menyatakan dengan jelas kasus mana yang saya pakai dan MENGAPA f(n) masuk ke kasus itu?
- Apakah saya membedakan klaim tentang kasus TERBAIK, TERBURUK, dan RATA-RATA secara eksplisit, alih-alih memperlakukan "waktu eksekusi" sebagai satu angka tunggal?
- Apakah saya membaca ulang jawaban saya sendiri seolah-olah saya adalah penilai skeptis yang mencari celah dalam argumen?

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, edisi ke-4. MIT Press, 2022. (Bab 1–4).

[2] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, edisi ke-4. Lulu Press, 2020–2023.

[3] D. E. Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, edisi ke-2. Addison-Wesley, 1998.

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 5

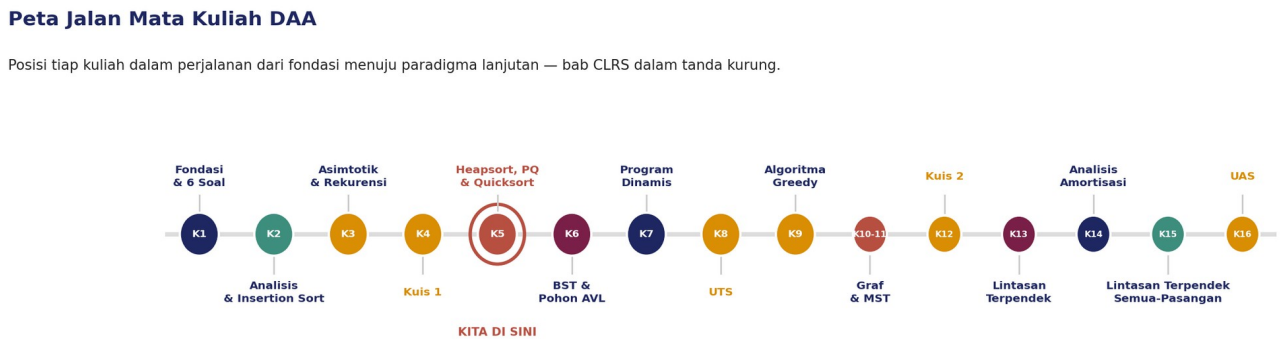
Heapsort, Antrian Prioritas, dan Quicksort

Menutup celah antara "di tempat" (in-place) dan "cepat" — dua jawaban berbeda untuk pertanyaan yang sama, dan yang mana yang sesungguhnya menang dalam praktik.

Tujuan Pembelajaran — setelah bab ini Anda mampu:
(1) Menjelaskan bagaimana sebuah binary heap mengkodekan pohon biner nyaris lengkap di dalam sebuah larik biasa, hanya dengan aritmetika indeks. (2) Menelusuri, mengimplementasikan, dan menganalisis HEAPIFY, BUILD-HEAP, dan HEAPSORT, termasuk bukti yang tidak jelas begitu saja bahwa BUILD-HEAP berjalan dalam O(n), bukan O(n log n). (3) Memakai heap untuk mengimplementasikan antrian prioritas, dan menyatakan waktu eksekusi setiap operasinya. (4) Menelusuri, mengimplementasikan, dan menganalisis prosedur PARTITION dan Quicksort, pada kasus terbaik, terburuk, dan rata-ratanya. (5) Menjelaskan mengapa pengacakan (randomization) mengalahkan masukan kasus-terburuk yang disengaja, dan mengapa Quicksort tetap menjadi pengurutan default di sebagian besar sistem nyata meskipun jaminan kasus terburuknya lebih buruk daripada Heapsort atau Merge Sort.

5.1 Rekap: Dari Kuliah 3 ke Kuliah 5

Kuliah 4 adalah Kuis 1 — sebuah checkpoint untuk Kuliah 1 hingga 3, tanpa materi baru. Kuliah ini melanjutkan cerita persis dari titik Kuliah 3 berhenti. Kuliah 3 memberi kita Merge Sort: algoritma divide-and-conquer yang menjamin waktu Θ(n log n) di setiap kasus, dengan harga ruang ekstra Θ(n) untuk langkah MERGE-nya. Itu meninggalkan satu pertanyaan terbuka, yang terlihat begitu kita jajarkan semua yang sudah dipelajari sejauh ini dalam satu tabel.



Gambar 5.1 — Posisi bab ini dalam peta jalan 16 kuliah DAA. Kuliah 6 beralih ke keluarga struktur yang berbeda (pohon pencarian biner seimbang); Kuliah 7 beralih ke paradigma perancangan yang sama sekali berbeda (pemrograman dinamis). Dua algoritma bab ini — Heapsort dan Quicksort — adalah pemberhentian terakhir dari cerita pengurutan yang dimulai di Kuliah 2 dan 3.

Algoritma	Kasus Terburuk	Kasus Terbaik	In-Place?	Ruang Ekstra	Ciri Kunci
Insertion Sort	$\Theta(n^2)$	$\Theta(n)$	Ya	$O(1)$	Bagus untuk data hampir terurut

Algoritma	Kasus Terburuk	Kasus Terbaik	In-Place?	Ruang Ekstra	Ciri Kunci
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	Ya	O(1)	Selalu tepat n−1 pertukaran
Bubble Sort	$\Theta(n^2)$	$\Theta(n)$	Ya	O(1)	Sederhana tapi lambat
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	Tidak	O(n)	Cepat tetapi butuh memori ekstra

Perhatikan celah yang menjengkelkan pada tabel ini: setiap pengurutan in-place sejauh ini adalah $\Theta(n^2)$, dan satu-satunya pengurutan $\Theta(n \log n)$ sejauh ini bukan in-place. Apakah mungkin ada pengurutan yang SEKALIGUS cepat ($O(n \log n)$) DAN in-place (ruang ekstra $O(1)$)?

> **Impian itu, dan dua jawaban berbeda**
> Bab ini menghadirkan DUA solusi independen untuk celah itu. Heapsort (Bagian 5.2–5.5) menjamin $O(n \log n)$ di setiap kasus, in-place, memakai struktur data baru — binary heap. Quicksort (Bagian 5.6–5.9) mencapai $O(n \log n)$ rata-rata, juga in-place, tanpa struktur data ekstra sama sekali — hanya langkah partisi in-place yang cerdik. Keduanya sampai di tujuan yang sama lewat jalan yang sama sekali berbeda, dan, mengejutkan, yang secara teoretis lebih lemah di antara keduanya (Quicksort) justru yang dipakai hampir semua sistem nyata.

5.2 Wawasan Selection Sort: Percepat Titik Sumbatan, Bukan Seluruh Algoritma

Ingat kembali Selection Sort dari Kuliah 2: setiap iterasi melakukan dua hal. Langkah A memindai A[1..i] untuk menemukan maksimum, berbiaya $\Theta(i)$. Langkah B menukar maksimum itu ke posisi i, berbiaya $\Theta(1)$. Diulang n kali, totalnya $\Theta(n^2)$ — dan Langkah A sepenuhnya menjadi titik sumbatan; Langkah B sudah pada dasarnya gratis.

> **Pertanyaan yang layak direnungkan**
> Jika Langkah A bisa dilakukan dalam $O(\log n)$, bukan $O(n)$ — menemukan maksimum dalam waktu logaritmik — apa yang terjadi pada total waktu eksekusi? (n iterasi) × $O(\log n)$ per iterasi = $O(n \log n)$. Seluruh algoritma menjadi lebih cepat hanya dengan mempercepat langkahnya yang lambat, tanpa mengubah struktur luarnya sama sekali.

Itulah persis strategi bagian ini: pertahankan bentuk dasar Selection Sort — berulang kali ekstrak maksimum, tempatkan, perkecil himpunan yang tersisa — tetapi ganti pemindaian larik biasa dengan struktur data yang lebih cerdas yang menemukan (dan kemudian menghapus) maksimum jauh lebih cepat. Struktur data itu adalah binary heap, dan Selection-Sort-plus-heap punya namanya sendiri: Heapsort.

Wawasan kunci

Kadang cara tercepat mempercepat algoritma yang benar tetapi lambat bukanlah merancang ulang algoritmanya sama sekali — melainkan mengubah STRUKTUR DATA di baliknya. Ini tema yang akan berulang sepanjang sisa mata kuliah ini.

5.3 Binary Heap: Pohon yang Tersembunyi di Dalam Larik

Binary heap tampak menipu sederhana di permukaan — hanya sebuah larik — tetapi diam-diam mengkodekan pohon biner nyaris lengkap, tanpa pointer sama sekali.

Binary Max-Heap

Binary max-heap adalah larik A[1..n] yang dipandang sebagai pohon biner nyaris lengkap di mana (1) setiap level kecuali mungkin level terakhir terisi penuh, (2) level terakhir terisi dari kiri ke kanan, dan (3) setiap induk sekurang-kurangnya sebesar kedua anaknya: A[Parent(i)] ≥ A[i] untuk semua i > 1. Sifat (3) adalah sifat max-heap. Min-heap membalik ketidaksamaan: setiap induk ≤ kedua anaknya.

5.3.1 Korespondensi Larik-ke-Pohon: Tanpa Pointer

Kejeniusan sebuah heap adalah bahwa struktur pohonnya sepenuhnya hidup dalam aritmetika indeks — tiga rumus, untuk simpul pada posisi ber-indeks-1 i:

Relasi	Rumus	Contoh (i = 3)	Trik perangkat keras
Induk dari simpul i	⌊i / 2⌋	Parent(3) = 1	Geser kanan 1 bit
Anak kiri dari simpul i	2i	Left(3) = 6	Geser kiri 1 bit
Anak kanan dari simpul i	2i + 1	Right(3) = 7	Geser kiri, set bit rendah

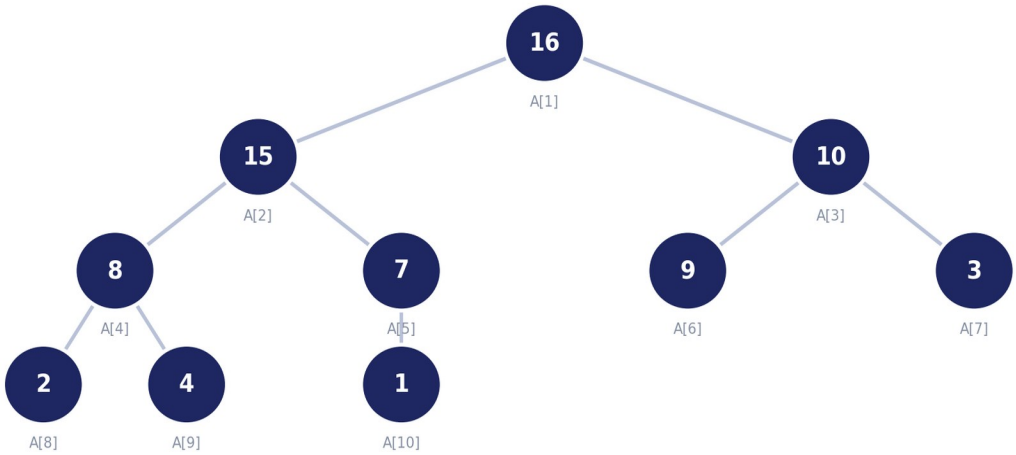
Mengalikan atau membagi dengan 2 dalam biner adalah satu pergeseran bit — salah satu operasi tercepat yang bisa dilakukan CPU mana pun. Tidak ada penelusuran pointer dan tidak ada alokasi memori yang tersebar; seluruh pohon hidup berdampingan dalam satu larik, yang juga sangat baik untuk lokalitas cache (tema yang akan kembali dibahas pada jebakan Bagian 5.5 tentang Heapsort).

5.3.2 Contoh Konkret (Gambar 6.1 CLRS)

Ambil A = [16, 15, 10, 8, 7, 9, 3, 2, 4, 1]. Gambar 5.2 menggambar persis larik ini sebagai pohon, dengan indeks larik setiap simpul ditampilkan di bawahnya.

Binary Max-Heap: Pohon yang Tersembunyi di Dalam Larik

CLRS Gambar 6.1 — larik A = [16,15,10,8,7,9,3,2,4,1], dipandang sebagai pohon biner nyaris lengkap. Setiap induk ≥ kedua anaknya.



Gambar 5.2 — Gambar 6.1 CLRS: larik A = [16,15,10,8,7,9,3,2,4,1] dipandang sebagai pohon biner nyaris lengkap. Periksa setiap pasangan induk-anak: 16≥15, 16≥10, 15≥8, 15≥7, 10≥9, 10≥3, 8≥2, 8≥4, 7≥1 — semuanya berlaku.

Sebuah heap BUKAN larik terurut

Mahasiswa sangat sering mengira heap pasti terurut. Tidak. Pada Gambar 5.2, simpul 9 (di A[6]) datang SETELAH simpul 8 (di A[4]) dalam urutan larik, padahal 9 > 8. Sifat heap hanya membatasi relasi induk-versus-anak, tidak pernah saudara atau sepupu. SATU-SATUNYA jaminan yang diberikan max-heap tentang seluruh larik adalah bahwa A[1] menyimpan maksimum keseluruhan.

5.3.3 Sifat-Sifat Struktural Kunci

- Akar A[1] selalu menyimpan elemen maksimum dari seluruh heap.
- Tinggi = ⌊lg n⌋. Heap dengan 10 elemen, seperti pada Gambar 5.2, memiliki tinggi 3 (akar pada tinggi 3, daun pada tinggi 0).
- Daun menempati posisi ⌊n/2⌋+1 hingga n — separuh kedua larik. Setiap daun secara trivial adalah heap 1-elemen yang valid dengan sendirinya.
- Sebuah heap dengan tegas TIDAK terurut — hanya menjamin induk ≥ anak, tidak pernah urutan kiri-ke-kanan global mana pun.

Koneksi CP4

std::priority_queue milik C++ adalah max-heap secara default. Modul heapq milik Python adalah min-heap. PriorityQueue milik Java juga min-heap secara default. Untuk membalik min-heap menjadi max-heap (atau sebaliknya), negasikan kunci atau sediakan comparator kustom — trik yang terus-menerus dipakai dalam pemrograman kompetitif untuk soal-soal yang dibangun di sekitar algoritma Dijkstra atau algoritma Prim.

5.4 HEAPIFY: Mesin yang Menjaga Kejujuran Heap

Bayangkan sebuah simpul yang kedua subpohonnya masing-masing sudah menjadi max-heap yang valid, tetapi simpul itu sendiri mungkin terlalu kecil untuk posisinya. HEAPIFY (kadang disebut MAX-HEAPIFY) memperbaiki persis situasi ini dengan menenggelamkan nilai yang melanggar itu ke bawah hingga sifat heap terpulihkan.

HEAPIFY(A, i)

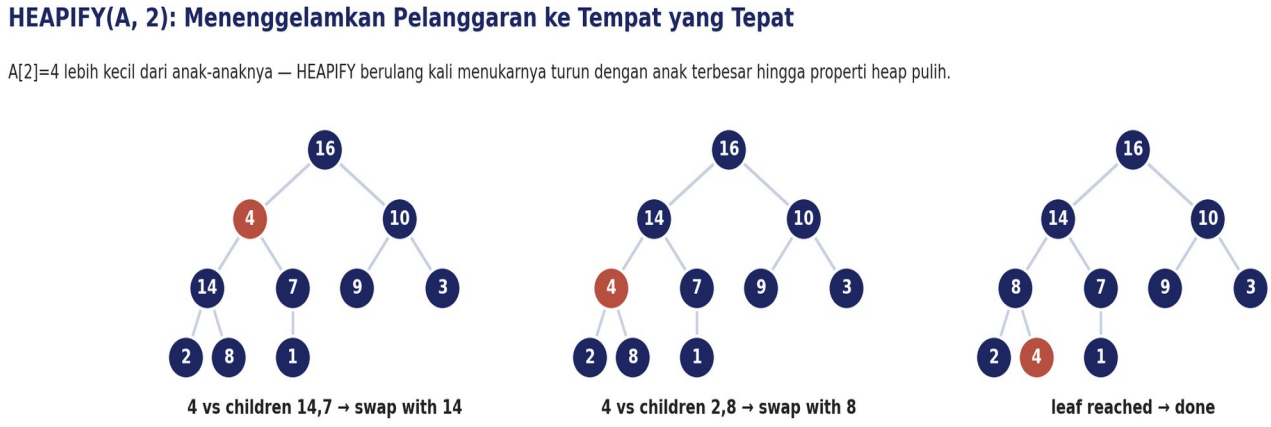
Prasyarat: pohon biner yang berakar di LEFT(i) dan RIGHT(i) masing-masing adalah max-heap yang valid, tetapi A[i] mungkin melanggar sifat max-heap terhadap anak-anaknya. HEAPIFY berulang kali menukar A[i] dengan anaknya yang terbesar hingga A[i] sekurang-kurangnya sebesar kedua anaknya, atau i tidak punya anak (daun).

```
HEAPIFY(A, i):                          // CLRS hal.165
  l = LEFT(i)                           // l = 2i
  r = RIGHT(i)                          // r = 2i + 1
  largest = i
  if l <= heap-size and A[l] > A[i]
      largest = l
  if r <= heap-size and A[r] > A[largest]
      largest = r
  if largest != i
      swap A[i] <-> A[largest]
      HEAPIFY(A, largest)               // rekursi turun ke subpohon yang terdampak
```

Dalam bahasa sederhana: di antara simpul i dan (hingga) kedua anaknya, temukan mana pun yang terbesar. Jika i sendiri sudah terbesar, HEAPIFY selesai — subpohon sudah menjadi heap yang valid. Jika tidak, tukar A[i] dengan anaknya yang terbesar, dan rekursi ke subpohon anak itu yang kini mungkin melanggar sifat heap.

5.4.1 Penelusuran: HEAPIFY(A, 2)

Mulai dari A = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1], di mana A[2] = 4 melanggar sifat heap terhadap anak-anaknya A[4] = 14 dan A[5] = 7. Gambar 5.3 menelusuri ketiga langkahnya.



Gambar 5.3 — HEAPIFY(A, 2) menenggelamkan nilai pelanggar 4 turun dua level: pertama ditukar dengan 14 (anak yang lebih besar), lalu dengan 8 (yang lebih besar dari anak-anak barunya), hingga mencapai posisi daun tanpa pelanggaran lebih lanjut.

5.4.2 Waktu Eksekusi

> **Waktu eksekusi HEAPIFY**
> HEAPIFY berjalan dalam O(log n). Setiap panggilan melakukan kerja Θ(1) (membandingkan paling banyak 3 nilai, paling banyak 1 pertukaran) ditambah satu panggilan rekursif ke subpohon berukuran paling banyak 2n/3. Ini memberi rekurensi T(n) ≤ T(2n/3) + Θ(1), yang diselesaikan Master Theorem (Kuliah 3, Kasus 2 dengan a=1, b=3/2) menjadi T(n) = O(log n). Setara: HEAPIFY melakukan paling banyak satu pertukaran per level pohon, dan sebuah heap punya ⌊lg n⌋ level.

5.5 BUILD-HEAP: Konstruksi O(n) yang Mengejutkan

Diberikan larik A[1..n] yang sembarang dan tak terurut, BUILD-HEAP menyusunnya ulang menjadi max-heap yang valid hanya dengan panggilan berulang ke HEAPIFY.

```
BUILD-HEAP(A):
  for i = floor(n/2) downto 1
      HEAPIFY(A, i)
```

Itulah seluruh algoritmanya: panggil HEAPIFY pada setiap simpul non-daun, bekerja dari dasar pohon menuju akar. Daun-daun — posisi ⌊n/2⌋+1 hingga n — sama sekali tidak butuh kerja, karena satu simpul secara trivial sudah menjadi heap 1-elemen yang valid.

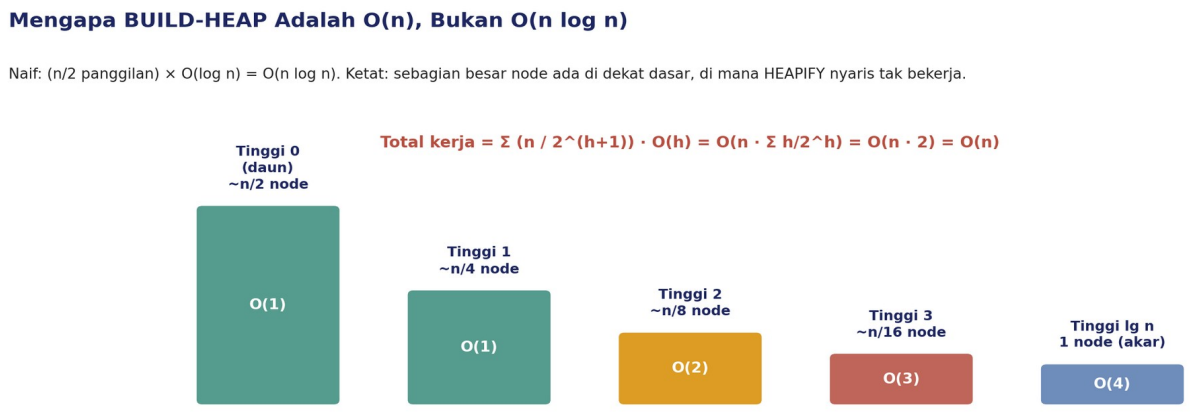
5.5.1 Kebenaran: Sebuah Invarian Loop

> **Invarian loop untuk BUILD-HEAP**
> Pada awal setiap iterasi loop for, setiap simpul i+1, i+2, ..., n adalah akar dari max-heap yang valid.

- Inisialisasi (i = ⌊n/2⌋): simpul ⌊n/2⌋+1 hingga n semuanya daun, masing-masing secara trivial max-heap 1-elemen yang valid. ✓
- Pemeliharaan: anak-anak simpul i punya indeks yang lebih besar secara ketat daripada i, sehingga menurut invarian mereka sudah menjadi akar max-heap yang valid — persis prasyarat HEAPIFY. Memanggil HEAPIFY(A, i) karenanya secara benar menjadikan simpul i juga akar max-heap yang valid, memperluas invarian ke i, i+1, ..., n. ✓
- Terminasi (i = 0): invarian kini mencakup simpul 1 hingga n — seluruh larik adalah max-heap yang valid. ✓

5.5.2 Mengapa Ini O(n), Bukan O(n log n)

Analisis naif mengatakan: n/2 panggilan ke HEAPIFY, masing-masing O(log n), untuk total O(n log n). Batas itu benar tetapi tidak ketat. Analisis ketat memperhatikan bahwa sebagian besar panggilan BUILD-HEAP terjadi pada subpohon kecil dekat dasar pohon, di mana HEAPIFY hampir tidak punya apa-apa untuk dikerjakan.



Gambar 5.4 — Sekitar n/2 simpul berada pada tinggi 0 (daun — bahkan tidak pernah dipanggil), n/4 pada tinggi 1, n/8 pada tinggi 2, dan seterusnya, dengan hanya 1 simpul (akar) pada tinggi maksimum lg n. Menjumlahkan (banyak simpul pada tinggi h) × (biaya O(h)) atas semua tinggi menghasilkan deret bergaya geometris yang konvergen ke O(n).

Analisis ketat

T(n) = Σ (n / 2^(h+1)) · O(h) = O(n · Σ h/2^h), dijumlahkan atas h dari 0 hingga lg n. Deret takhingga Σ h/2^h (untuk h = 0, 1, 2, …) konvergen ke tepat 2 — fakta baku yang diturunkan dari mendiferensialkan deret geometri Σ x^h = 1/(1−x). Jadi T(n) = O(n · 2) = O(n).

Wawasan kunci

BUILD-HEAP hanya membutuhkan waktu O(n), bukan O(n log n) — salah satu hasil yang sungguh mengejutkan dalam analisis algoritma pengantar, dan hasil langsung dari kebiasaan analisis-ketat yang dibangun Kuliah 3 (pohon rekursi, Master Theorem): batas per-panggilan yang naif itu benar tetapi jauh dari ketat, dan hanya dengan menjumlahkan biaya per-level kebenarannya terungkap.

5.6 HEAPSORT: O(n log n) dan In-Place

Dengan BUILD-HEAP dan HEAPIFY di tangan, Heapsort sendiri singkat: bangun max-heap, lalu berulang kali pindahkan maksimum saat ini (selalu di akar) ke posisi akhirnya yang terurut di ujung larik, perkecil heap, dan pulihkan sifat heap pada akar (yang kini lebih kecil).

```
HEAPSORT(A):                          // CLRS hal.170
  BUILD-HEAP(A)                       // O(n)
  for i = n downto 2                  // n-1 kali
      swap A[1] <-> A[i]               // pindahkan maks saat ini ke slot
terurutnya
      heap-size = heap-size - 1         // perkecil heap, kecualikan A[i]
      HEAPIFY(A, 1)                  // O(log n): pulihkan sifat heap
```

Setelah BUILD-HEAP, A[1] menyimpan maksimum keseluruhan. Menukarnya ke posisi n menempatkannya pada tempat akhirnya yang sudah benar-terurut. Memperkecil heap-size sebesar 1 berarti HEAPIFY(A,1) berikutnya tidak pernah lagi melihat elemen yang sudah ditempatkan itu. HEAPIFY kemudian memulihkan sifat heap memakai nilai apa pun yang baru ditukar ke akar, dan A[1] sekali lagi menyimpan maksimum dari heap yang TERSISA (lebih kecil). Mengulang ini n−1 kali mengurutkan seluruh larik.

Gambar 5.5 — Dua iterasi pertama ekstrak-dan-perkecil pada A = [16,14,10,8,7,9,3,2,4,1]. Setiap iterasi menukar akar (emas) ke batas antara heap yang menyusut dan sufiks terurut yang tumbuh, lalu meng-heapify ulang heap yang lebih kecil.

Langkah	Biaya	Berapa Kali
BUILD-HEAP	O(n)	1
Loop: swap + HEAPIFY	O(log n)	n − 1
Total	O(n log n)	—

Jaminan Heapsort

Heapsort berjalan dalam Θ(n log n) pada kasus terburuk — selalu, tanpa masukan buruk — dan mengurutkan in-place hanya memakai ruang ekstra O(1). Ini BUKAN pengurutan stabil (elemen yang sama bisa berpindah urutan relatif satu sama lain).

Lalu mengapa tidak semua orang memakai Heapsort?

Meskipun menjamin O(n log n) tanpa kasus terburuk, Heapsort punya lokalitas cache yang buruk: HEAPIFY terus-menerus melompat antara induk dan anak yang bisa jauh terpisah dalam memori (indeks i versus indeks 2i), menyebabkan cache miss yang sering. Quicksort, sebaliknya, mengakses memori dalam larik yang sebagian besar berurutan dan sekitar 2–3× lebih cepat dalam praktik meskipun kasus terburuk teoretisnya lebih buruk. Inilah persis mengapa sebagian besar pustaka standar secara default memakai algoritma keluarga Quicksort, bukan Heapsort — Bagian 5.7 dan seterusnya menjelaskan alasannya.

5.7 Antrian Prioritas: Aplikasi Heap Paling Terkenal

Antrian prioritas (priority queue, PQ) adalah tipe data abstrak untuk memelihara himpunan elemen dinamis, masing-masing dengan kunci terkait (prioritasnya), yang selalu melayani elemen berprioritas tertinggi lebih dulu — berbeda dari antrian FIFO, yang melayani menurut urutan kedatangan.

Operasi Max-PQ
INSERT(S, x) — tambahkan elemen x ke himpunan. MAXIMUM(S) — kembalikan kunci terbesar tanpa menghapusnya. EXTRACT-MAX(S) — hapus dan kembalikan kunci terbesar. INCREASE-KEY(S, x, k) — naikkan kunci elemen x menjadi nilai baru k (diasumsikan ≥ kunci saat ini).

5.7.1 Aplikasi

- Penjadwalan pekerjaan sistem operasi: pekerjaan berprioritas tertinggi berjalan berikutnya.
- Simulasi berbasis peristiwa: proses peristiwa dalam urutan kronologis memakai min-PQ berkunci waktu peristiwa.
- Algoritma graf (dipratinjau di sini, dikembangkan lebih lanjut nanti dalam mata kuliah): lintasan terpendek Dijkstra dan pohon rentang minimum Prim keduanya berulang kali mengekstrak simpul berjarak-minimum dari min-PQ.
- Pengodean Huffman: berulang kali mengekstrak dua simpul berfrekuensi terendah dari min-PQ membangun kode bebas-prefiks optimal untuk kompresi data.

5.7.2 Implementasi Berbasis Heap

- MAXIMUM(A): kembalikan A[1] langsung. Waktu: O(1).
- EXTRACT-MAX(A): simpan A[1], pindahkan A[heap-size] ke akar, perkecil heap sebesar satu, HEAPIFY(A,1) untuk memulihkan sifatnya, lalu kembalikan maksimum yang disimpan. Waktu: O(log n).
- HEAP-INSERT(A, key): perbesar heap sebesar satu slot, tempatkan key pada posisi dasar baru, lalu "gelembungkan naik" — berulang kali tukar dengan induknya selama induk lebih kecil. Waktu: O(log n).

Operasi	PQ Berbasis Heap	PQ Larik Tak-Terurut
INSERT	O(log n)	O(1)
MAXIMUM	O(1)	O(n)
EXTRACT-MAX	O(log n)	O(n)
INCREASE-KEY	O(log n)	O(1)

Tip CP4
Dalam lomba, jangan pernah mengimplementasikan heap sendiri dari nol — pakai antrian prioritas bawaan bahasa pemrograman. Tetapi memahami internal di atas esensial untuk menganalisis dengan benar kompleksitas waktu algoritma graf mana pun yang MEMAKAI antrian prioritas secara internal — persis yang akan dibutuhkan kuliah-kuliah berikutnya tentang algoritma Dijkstra dan Prim.

5.8 Quicksort: Pengurutan Tercepat dalam Praktik

Ditemukan oleh Tony Hoare pada 1959, Quicksort tetap menjadi pengurutan tujuan-umum default di sebagian besar sistem nyata dan pustaka standar. Meskipun kasus terburuknya $\Theta(n^2)$ — secara teoretis lebih buruk daripada Merge Sort maupun Heapsort — kasus rata-ratanya $\Theta(n \log n)$ membawa faktor konstan yang sangat kecil dan perilaku cache yang sangat baik, itulah mengapa ia menang dalam praktik lebih sering daripada yang disarankan jaminan kasus terburuknya saja.

> **Quicksort**
> Algoritma pengurutan divide-and-conquer yang (1) mempartisi larik di sekitar pivot terpilih, menempatkan setiap elemen ≤ pivot di kirinya dan setiap elemen ≥ pivot di kanannya, (2) secara rekursif mengurutkan tiap sisi, dan (3) sama sekali tidak butuh langkah gabung — begitu kedua sisi masing-masing terurut, seluruh larik sudah terurut, in-place.

5.8.1 Quicksort sebagai Bagi, Taklukkan, dan Gabung (Trivial)

1. Bagi: pilih pivot x. Susun ulang larik in-place sehingga elemen ≤ x berada sebelum elemen ≥ x.
2. Taklukkan: secara rekursif urutkan partisi kiri dan kanan.
3. Gabung: sama sekali tidak ada apa pun — larik sudah sepenuhnya terurut in-place begitu kedua panggilan rekursif kembali.

> **Dibandingkan dengan Merge Sort**
> Merge Sort punya langkah bagi yang trivial (belah larik tepat dua berdasarkan indeks) tetapi langkah gabung yang mahal $\Theta(n)$ (MERGE). Quicksort membalik ini sepenuhnya: langkah bagi yang mahal (PARTITION melakukan semua kerja sesungguhnya) tetapi langkah gabung yang trivial (tidak ada apa-apa). Kerja berat hanya berpindah dari fase gabung ke fase bagi — resep divide-and-conquer yang sama dari Kuliah 3, diterapkan dengan biaya yang terdistribusi berbeda di antara ketiga langkahnya.

5.9 Prosedur PARTITION

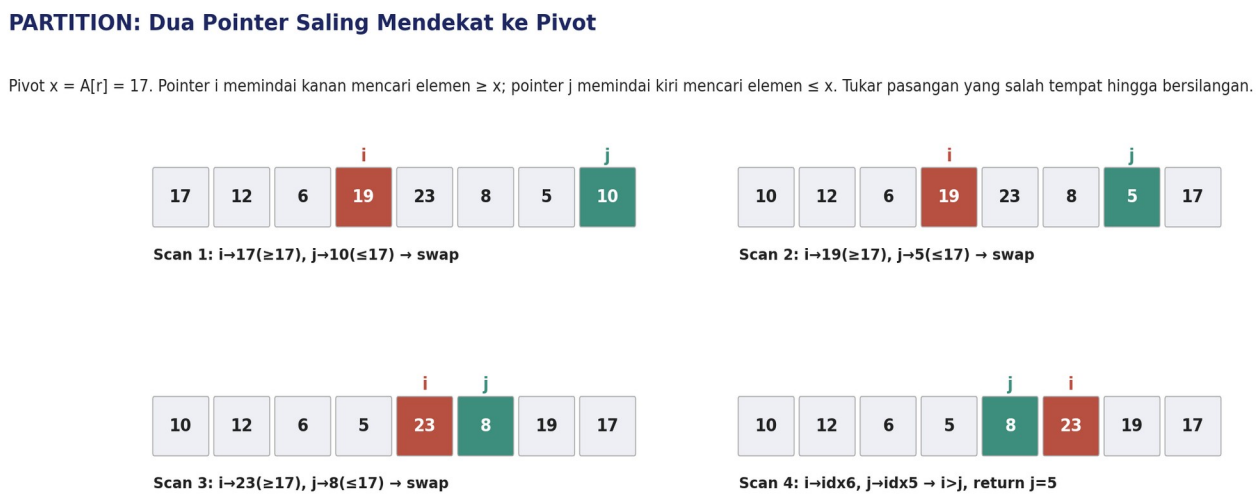
```
PARTITION(A, p, r):                    // Skema Hoare, CLRS hal.183
  x = A[r]                             // pivot = elemen terakhir rentang
  i = p - 1
  j = r + 1
  while true
      repeat j = j - 1 until A[j] <= x
      repeat i = i + 1 until A[i] >= x
      if i < j
          swap A[i] <-> A[j]
      else
          return j
```

Pointer i memindai ke kanan mencari elemen yang TIDAK termasuk di kiri (yang ≥ pivot). Pointer j memindai ke kiri mencari elemen yang TIDAK termasuk di kanan (yang ≤ pivot). Kapan pun kedua pointer menemukan pasangan yang salah tempat sebelum bersilangan, mereka bertukar —

memindahkan tiap elemen ke separuh tempat ia seharusnya berada. Ketika pointer akhirnya bersilangan, partisi selesai dan PARTITION mengembalikan titik pemisah j.

5.9.1 Penelusuran

Telusuri mekanisme pemindaian PARTITION pada larik [17, 12, 6, 19, 23, 8, 5, 10] memakai nilai pivot x = 17 sebagai ambang perbandingan. (Penelusuran ini mengisolasi perilaku pindai-dan-tukar dua-pointer dengan sendirinya; Bagian 5.10 menunjukkan bagaimana QUICKSORT memasok x = A[r] secara otomatis dari elemen terakhir rentang saat ini, apa pun nilainya.) Gambar 5.6 menunjukkan keempat langkah pindai-dan-tukar.



Gambar 5.6 — Empat pemindaian PARTITION dengan pivot x = 17. Pointer i (terracotta) berburu ke kanan mencari elemen ≥ 17; pointer j (teal) berburu ke kiri mencari elemen ≤ 17. Pemindaian 1–3 masing-masing menemukan pasangan salah tempat dan bertukar; pemindaian 4 menemukan i > j dan mengembalikan titik pemisah j = 5 — segala sesuatu pada atau sebelum indeks 5 adalah ≤ 17, segala sesuatu setelahnya ≥ 17.

> **Wawasan kunci**
>
> PARTITION melakukan kerja Θ(n) — setiap elemen diperiksa oleh tepat satu dari kedua pointer sebelum mereka bersilangan — hanya memakai ruang ekstra O(1). Langkah partisi in-place Θ(n) inilah persis biaya langkah bagi QUICKSORT; seluruh rekurensi yang dibangun di Bagian 5.10 mengalir langsung dari fakta tunggal ini.

5.10 Algoritma QUICKSORT

```
QUICKSORT(A, p, r):
  if p < r
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q)
      QUICKSORT(A, q + 1, r)
```

Panggilan awal adalah QUICKSORT(A, 1, n). Perhatikan panggilan rekursifnya adalah QUICKSORT(A, p, q) dan QUICKSORT(A, q+1, r) — bukan q−1 dan q — karena skema partisi Hoare (Bagian 5.9) tidak menjamin pivot itu sendiri mendarat di indeks q; ia hanya menjamin segala sesuatu hingga dan termasuk q adalah ≤ segala sesuatu setelah q.

5.11 Menganalisis Quicksort: Kasus Terbaik, Terburuk, dan Rata-Rata

Waktu eksekusi Quicksort sepenuhnya bergantung pada seberapa seimbang pembelahan PARTITION kebetulan terjadi — inilah satu hal yang membedakannya dari Merge Sort, yang pembelahannya selalu tepat seimbang berdasarkan konstruksi.

5.11.1 Kasus Terbaik: Pembelahan Sempurna

Jika PARTITION selalu membelah larik tepat dua, rekurensinya adalah $T(n) = 2T(n/2) + \Theta(n)$ — persis rekurensi Merge Sort dari Kuliah 3 — yang diselesaikan Kasus 2 Master Theorem menjadi $\Theta(n \log n)$.

5.11.2 Kasus Terburuk: Pembelahan Setaksimal Mungkin Tak-Seimbang

Jika satu sisi setiap partisi hanya mendapat 1 elemen dan sisi lainnya $n-1$ (terjadi, misalnya, pada masukan yang sudah terurut atau terurut-terbalik dengan pivot elemen-terakhir naif), rekurensinya menjadi $T(n) = T(1) + T(n-1) + \Theta(n) = \Theta(n) + \Theta(n-1) + \ldots + \Theta(1) = \Theta(n^2)$.

> **Pembalikan yang ironis**
> Masukan terurut adalah kasus TERBAIK Insertion Sort ($\Theta(n)$, Kuliah 2) tetapi kasus TERBURUK Quicksort naif ($\Theta(n^2)$)! Inilah persis mengapa Quicksort naif berpivot-tetap tidak boleh pernah dipakai pada data yang mungkin sudah terurut atau hampir terurut tanpa suatu bentuk pengacakan (Bagian 5.12) — musuh, atau sekadar nasib buruk, dapat memicu persis kasus terburuk ini.

5.11.3 Pembelahan 1-dari-10 — Tetap O(n log n)!

Pertimbangkan pembelahan yang jauh kurang seimbang tetapi tetap berperbandingan-konstan: $T(n) = T(n/10) + T(9n/10) + \Theta(n)$. Gambar 5.7 membandingkan kedalaman rekursi pembelahan ini terhadap kasus seimbang sempurna dan kasus setaksimal mungkin tak-seimbang.

Gambar 5.7 — Bentuk rekursi untuk $n = 16$ pada tiga kualitas partisi. Kasus seimbang dangkal dan lebar (kedalaman $\lg n$); kasus degeneratif $1:(n-1)$ adalah irisan tipis dan dalam (kedalaman $n-1$); pembelahan 1:9 lebih dalam daripada seimbang tetapi tetap dangkal relatif terhadap n, memberikan $\Theta(n \log n)$ pada kedua kasus.

Mengapa pembelahan timpang tetap berhasil

Sisi pendek pembelahan 1:9 habis setelah $\log_{10}$ n level; sisi panjang bertahan sekitar $\log_{10/9}$ n level. Tetapi pada SETIAP level, total kerja yang dijumlahkan di seluruh subpersoalan tetap paling banyak Θ(n) — persis seperti kasus seimbang, hanya tersebar pada lebih banyak level. Jadi $T(n) = \Theta(n \cdot \log_{10/9} n) = \Theta(n \log n)$.

Generalisasi yang layak diingat

SETIAP pembelahan berperbandingan-konstan — 50:50, 10:90, bahkan 1:99 — memberikan O(n log n) secara keseluruhan. Hanya satu kasus degeneratif tunggal pembelahan 1:(n−1), di mana satu sisi tak pernah mengecilkan persoalan dengan pecahan konstan mana pun, yang menghasilkan $\Theta(n^2)$. Inilah persis mengapa Quicksort begitu tangguh dalam praktik: pivot yang benar-benar seimbang jarang, tetapi yang benar-benar degeneratif jauh lebih jarang lagi.

5.11.4 Kasus Rata-Rata: Pembelahan Baik dan Buruk Berselang-Seling

Misalkan pembelahan berselang-seling antara beruntung (seimbang sempurna) dan sial (1 versus n−1). Misalkan L(n) menyatakan biaya yang dimulai dari pembelahan beruntung dan U(n) dari yang sial:

```
L(n) = 2·U(n/2) + Θ(n)              // pembelahan beruntung, lalu dua subpersoalan
U(n) = L(n-1) + Θ(n)                // satu pembelahan sial, lalu satu subpersoalan
// Substitusikan U ke L:
L(n) = 2·(L(n/2 - 1) + Θ(n/2)) + Θ(n) = 2·L(n/2 - 1) + Θ(n) = Θ(n log n)
```

Wawasan kunci

Biaya ekstra pembelahan sial terserap oleh pembelahan beruntung yang mengikutinya — nasib buruk saja, selama berselang-seling dengan nasib baik, tidak dapat menyeret kasus rata-rata turun ke $\Theta(n^2)$. Argumen probabilistik lengkap (di luar cakupan mata kuliah ini) mengonfirmasi ini: kasus rata-rata SESUNGGUHNYA Quicksort atas semua permutasi masukan adalah Θ(n log n) yang solid, dengan konstanta harapan mendekati 1,39·n·lg n perbandingan.

5.12 Randomized Quicksort: Menjinakkan Kasus Terburuk

Dengan aturan pemilihan pivot yang tetap (seperti "selalu elemen terakhir"), musuh yang mengetahui aturan itu selalu dapat merancang masukan kasus-terburuk di muka — seperti yang ditunjukkan Bagian 5.11.2 untuk data yang sudah terurut. Perbaikannya adalah menghilangkan kemampuan musuh untuk memprediksi pivot sama sekali: pilih secara acak.

```
RANDOMIZED-PARTITION(A, p, r):
  i = RANDOM(p, r)                  // indeks acak seragam dalam [p, r]
  swap A[r] <-> A[i]                // pindahkan pilihan acak ke posisi pivot
  return PARTITION(A, p, r)         // lalu partisi persis seperti sebelumnya

RANDOMIZED-QUICKSORT(A, p, r):
  if p < r
      q = RANDOMIZED-PARTITION(A, p, r)
      RANDOMIZED-QUICKSORT(A, p, q)
      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Sifat-sifat Randomized Quicksort

(1) Waktu eksekusinya tidak lagi bergantung pada urutan awal masukan — setiap masukan, secara harapan, adalah masukan kasus-rata-rata. (2) Tidak ada satu masukan pun yang bisa dirancang di muka untuk memicu kasus terburuk, karena pilihan pivot itu sendiri acak, bukan fungsi dari data. (3) Urutan lemparan koin yang sial BISA, pada prinsipnya, tetap menghasilkan kasus terburuk $\Theta(n^2)$ — tetapi probabilitasnya secara eksponensial kecil, pada dasarnya tak pernah teramati dalam praktik. (4) Waktu eksekusi harapan: O(n log n), dengan sekitar 1,39·n·lg n perbandingan yang diharapkan.

5.12.1 Strategi Pemilihan Pivot Lainnya

- Median-of-three: pakai median dari A[p], A[⌊(p+r)/2⌋], dan A[r] sebagai pivot — murah menghindari kasus terburuk pada masukan yang sudah terurut tanpa pengacakan penuh.
- Dual-pivot Quicksort (Yaroslavskiy): dipakai Arrays.sort milik Java untuk tipe primitif — dua pivot mempartisi larik menjadi tiga wilayah, bukan dua.
- Introsort (dipakai std::sort milik C++): menjalankan Quicksort, tetapi beralih ke Heapsort jika kedalaman rekursi tumbuh mencurigakan besar (tanda pembelahan mendekati kasus-terburuk), dan beralih ke Insertion Sort untuk sub-larik kecil di mana overhead-nya yang rendah menang.

Perspektif CP4

Dalam pemrograman kompetitif, selalu pakai pengurutan bawaan bahasa daripada menulis ulang Quicksort sendiri: std::sort milik C++ memakai Introsort, Java memakai dual-pivot Quicksort untuk primitif dan TimSort untuk objek, dan sorted() milik Python memakai TimSort. Memahami analisis acak Quicksort di sini adalah yang nantinya memungkinkan Anda bernalar dengan benar tentang waktu eksekusi harapan dan argumen probabilistik di tempat lain dalam mata kuliah ini.

5.13 Lanskap Pengurutan Lengkap Sejauh Ini

Algoritma	Terburuk	Rata-Rata	Terbaik	Ruang	Stabil?	In-Place?	Diperkenalkan
Insertion Sort	n^2	n^2	n	1	Ya	Ya	Kuliah 2
Selection Sort	n^2	n^2	n^2	1	Tidak	Ya	Kuliah 2

Algoritma	Terburuk	Rata-Rata	Terbaik	Ruang	Stabil?	In-Place?	Diperkenalkan
Bubble Sort	n^2	n^2	n	1	Ya	Ya	Kuliah 2
Merge Sort	n lg n	n lg n	n lg n	n	Ya	Tidak	Kuliah 3
Heapsort	n lg n	n lg n	n lg n	1	Tidak	Ya	Kuliah 5
Quicksort	n^2	n lg n	n lg n	lg n	Tidak	Ya	Kuliah 5

5.13.1 Kapan Memakai Apa?

Situasi	Pilihan Terbaik & Alasannya
Larik kecil (n < 50)	Insertion Sort — overhead rendah, cepat untuk masukan kecil
Data hampir terurut	Insertion Sort — kasus terbaik Θ(n) untuk masukan hampir terurut
Tujuan umum, kecepatan diutamakan	Randomized Quicksort — tercepat dalam praktik, perilaku cache sangat baik
Harus menjamin O(n log n)	Heapsort atau Merge Sort — tanpa kasus terburuk $\Theta(n^2)$
Stabilitas dibutuhkan	Merge Sort — satu-satunya pengurutan stabil Θ(n log n) yang sudah dipelajari
Memori sangat terbatas	Heapsort — ruang ekstra O(1) dengan jaminan O(n log n)
Insert + extract-max dinamis	Binary heap / antrian prioritas — O(log n) untuk kedua operasi
Pemrograman kompetitif	Pengurutan bawaan bahasa: std::sort (Introsort), Arrays.sort, sorted() (TimSort)

5.14 Profitina dalam Praktik: Di Mana Bab Ini Muncul dalam Produk Nyata

> **Tentang kotak ini**
> Seperti di setiap bab, kotak ini menghubungkan materi mata kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) asal Indonesia yang nyata, dibangun oleh penulis, tanpa mengungkap detail implementasi milik perusahaan.

Profitina terus-menerus memantau kumpulan besar dan selalu berubah dari bisnis yang dilacak, dan berulang kali perlu menjawab "bisnis mana yang saat ini memiliki skor AI-visibility terendah, dan karena itu paling perlu perhatian lebih dulu?" — persis operasi EXTRACT-MIN dari Bagian 5.7, diterapkan pada min-heap berkunci skor visibilitas. Skor diperbarui saat pemindaian jawaban-AI baru selesai, yang persis adalah operasi bergaya INCREASE-KEY / DECREASE-KEY, dan antrian prioritas berbasis heap menjaga setiap operasi tetap O(log n) bahkan saat himpunan bisnis yang dilacak tumbuh hingga ribuan, alih-alih O(n) yang dibutuhkan daftar tak-terurut untuk setiap pencarian tunggal. Secara terpisah, ketika Profitina butuh pengurutan tujuan-umum satu kali — memberi peringkat pada satu batch sebutan pesaing yang baru diambil berdasarkan skor relevansi, misalnya, tanpa struktur tertentu yang bisa dimanfaatkan — insinyur memakai pengurutan bawaan platform (yang berupa

suatu varian Introsort atau TimSort, sesuai perspektif CP4 Bagian 5.12.1) daripada menulis ulang apa pun sendiri, persis karena jebakan lokalitas-cache Bagian 5.6 dan argumen pengacakan Bagian 5.12 adalah persis pertimbangan yang sudah dirancang di sekitarnya oleh implementasi pustaka standar yang teruji medan.

5.15 Ringkasan Bab dan Poin-Poin Kunci

- Binary heap menyimpan pohon biner nyaris lengkap secara implisit di dalam larik biasa, hanya memakai aritmetika indeks (Parent, Left, Right) — tanpa pointer, dan lokalitas cache yang sangat baik untuk pola penelusuran pohon yang dilakukan HEAPIFY.
- HEAPIFY menenggelamkan satu elemen pelanggar dalam waktu O(log n); ia adalah satu operasi primitif di balik setiap operasi heap lain dalam bab ini.
- BUILD-HEAP berjalan dalam O(n), bukan O(n log n) yang secara naif diharapkan — sebagian besar panggilannya beroperasi pada subpohon kecil dekat dasar pohon, di mana HEAPIFY hampir tidak melakukan apa-apa.
- Heapsort = BUILD-HEAP + (n−1) putaran ekstrak-maks-dan-perkecil = Θ(n log n) di setiap kasus, in-place — tetapi dengan lokalitas cache yang buruk yang membuatnya lebih lambat dalam praktik daripada Quicksort.
- Antrian prioritas berbasis heap mendukung INSERT dan EXTRACT-MAX (atau -MIN) dalam O(log n), dan MAXIMUM (atau MINIMUM) dalam O(1) — fondasi bagi algoritma Dijkstra dan Prim nanti dalam mata kuliah ini.
- Quicksort mempartisi in-place di sekitar pivot dalam waktu Θ(n); kasus terbaik dan rata-rata Θ(n log n), kasus terburuk $\Theta(n^2)$.
- SETIAP pembelahan berperbandingan-konstan — bukan hanya yang seimbang sempurna — memberikan Θ(n log n). Hanya satu kasus degeneratif tunggal 1:(n−1) yang menghasilkan $\Theta(n^2)$.
- Randomized Quicksort menghilangkan kemampuan musuh memprediksi pivot, membuat kasus terburuk $\Theta(n^2)$ secara eksponensial tidak mungkin, bukan sekadar bisa dihindari lewat rancangan masukan yang cermat.
- Dalam praktik, Quicksort biasanya mengalahkan Heapsort meskipun jaminan kasus terburuknya lebih buruk, karena akses memori berurutan dan faktor konstan yang kecil — teori dan praktik tidak selalu menunjuk arah yang sama, dan keduanya penting.

“Struktur data yang cerdas dan kode yang bodoh bekerja jauh lebih baik daripada sebaliknya.”

— Eric S. Raymond

(Selection Sort + binary heap = Heapsort. Loop “bodoh” yang sama, struktur data yang lebih cerdas di baliknya.)

Kuliah 6 beralih sepenuhnya dari pengurutan menuju keluarga struktur data baru: pohon pencarian biner seimbang, yang menjaga pencarian, penyisipan, dan penghapusan semuanya pada O(log n) bahkan di bawah urutan penyisipan yang disengaja jahat — memecahkan, untuk beban kerja yang berat pada pencarian, versi masalah keseimbangan yang sama yang dipecahkan bab ini untuk pengurutan.

5.16 Pertanyaan Tinjauan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Gambar binary heap yang berkorespondensi dengan A = [23, 17, 14, 6, 13, 10, 1, 5, 7, 12] sebagai pohon, dengan gaya Gambar 5.2, dan verifikasi sifat max-heap berlaku pada setiap pasangan induk-anak.
2. Telusuri HEAPIFY(A, 1) secara manual pada A = [4, 14, 10, 8, 7, 9, 3, 2, 1] (heap yang akarnya melanggar sifat terhadap anak-anaknya), dengan gaya Gambar 5.3.
3. Jelaskan, dengan kata-kata Anda sendiri dan tanpa melihat kembali Bagian 5.5.2, mengapa BUILD-HEAP adalah O(n), bukan O(n log n).
4. Telusuri PARTITION pada [8, 3, 15, 2, 9, 5, 1, 12] memakai pivot x = A[r] = 12, dengan gaya Gambar 5.6, dan nyatakan indeks pemisah yang dikembalikan.
5. Konstruksikan larik 6-elemen yang memaksa Quicksort naif (pivot elemen-terakhir) ke kasus terburuk $\Theta(n^2)$-nya, dan jelaskan mengapa Randomized Quicksort tidak akan secara andal mengulang kasus terburuk yang sama pada larik yang sama itu.
6. Sebuah antrian prioritas diimplementasikan sebagai larik TAK-TERURUT (bukan heap). Nyatakan waktu eksekusi INSERT, MAXIMUM, dan EXTRACT-MAX untuk implementasi ini, dan bandingkan dengan tabel Bagian 5.7.2 untuk versi berbasis heap.

Lampiran 5.A — Log Verifikasi untuk Kode Sumber Bab 5

Seperti pada bab-bab sebelumnya, setiap program yang disertakan buku ini dikompilasi dan dijalankan terhadap contoh yang diverifikasi manual sebelum dimasukkan. Kedua program di bawah ini dikompilasi dengan `g++ -O2 -std=c++17 -Wall` (hanya peringatan jinak nilai-kembali scanf, nol kesalahan) dan menghasilkan persis keluaran yang diprediksi dalam teks di atas.

```
$ printf "10\n16 15 10 8 7 9 3 2 4 1\n" | ./01_heapsort
1 2 3 4 7 8 9 10 15 16               # mengurutkan persis heap dari Gambar 5.2

$ printf "8\n5 2 4 7 1 3 2 6\n" | ./01_heapsort
1 2 2 3 4 5 6 7               # larik sama yang diurutkan Kuliah 3 dengan Merge
Sort

$ printf "8\n17 12 6 19 23 8 5 10\n" | ./02_quicksort
5 6 8 10 12 17 19 23                  # Quicksort pivot-acak, terurut dengan
benar

$ printf "10\n10 9 8 7 6 5 4 3 2 1\n" | ./02_quicksort
1 2 3 4 5 6 7 8 9 10          # terurut-terbalik: kasus terburuk naif, aman di
sini
```

Latihan SPOJ — buku pendamping
Teknik bab ini diuji sungguhan di juri daring SPOJ. Buka Bab 6 & 11 buku pendamping “SPOJ Competitive Programming” (folder SPOJ, BI & EN) untuk heap dan priority queue sebagai kuda beban struktur data kontes — lengkap dengan tips, trik, dan hint agar solusi Anda Accepted, tanpa membocorkan jawaban.

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Bab 6–7).

[2] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, 4th ed. Lulu Press, 2020–2023.

[3] C. A. R. Hoare. "Quicksort." The Computer Journal, 5(1), 1962, 10–16. Publikasi asli yang menjelaskan algoritma yang dirancang Hoare pada 1959.

[4] E. S. Raymond. The Cathedral and the Bazaar. O'Reilly Media, 1999 — sumber kutipan penutup bab ini tentang struktur data dan kode.

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 6

Pohon Pencarian Biner dan Pohon AVL

Meninggalkan pengurutan: struktur yang selalu menjaga datanya tetap urut, dan trik penyeimbangan yang membuatnya tetap cepat apa pun urutan kedatangan data.

Tujuan Pembelajaran — setelah bab ini Anda mampu:
(1) Menjelaskan mengapa kamus dinamis yang menjaga urutan butuh lebih dari sekadar larik biasa atau larik terurut, dan menyatakan sifat binary-search-tree secara presisi. (2) Menelusuri dan mengimplementasikan SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, dan PREDECESSOR pada BST, serta menyatakan waktu eksekusi tiap operasi dalam istilah tinggi pohon. (3) Menelusuri dan mengimplementasikan BST-INSERT serta ketiga kasus BST-DELETE, termasuk penyambungan in-order-successor untuk penghapusan bersimpul-dua-anak. (4) Menjelaskan mengapa urutan penyisipan yang adversarial atau sudah terurut dapat menurunkan tinggi BST menjadi Θ(n), dan mengapa itu membuat batas O(log n) setiap operasi bersyarat, bukan terjamin. (5) Menyatakan invarian faktor-keseimbangan AVL, menelusuri keempat kasus rotasi (LL, RR, LR, RL), dan menjelaskan mengapa semuanya memulihkan keseimbangan dalam waktu O(1). (6) Membuktikan, lewat argumen pohon-Fibonacci, bahwa tinggi pohon AVL selalu O(log n) — mengubah batas bersyarat BST biasa menjadi jaminan tanpa syarat.

6.1 Rekap: Dari Kuliah 5 ke Kuliah 6

Kuliah 5 menutup kisah pengurutan: Heapsort dan Quicksort sama-sama mengubah larik tak-terurut menjadi larik yang sepenuhnya terurut, dan keduanya melakukannya dengan baik — tetapi tidak satu pun dirancang untuk dikueri, diperbarui, dan dikueri ulang berkali-kali seiring data datang dari waktu ke waktu. Begitu Quicksort selesai, larik itu adalah cuplikan beku; menyisipkan satu elemen baru berarti, secara umum, mengurutkan ulang hampir semuanya di sekitarnya. Bab ini mengajukan pertanyaan yang sama sekali berbeda: bagaimana jika data terus berubah — rekaman baru datang, yang lama dihapus — dan pada setiap saat Anda perlu menjawab "apakah x ada?", "apa elemen terkecil/terbesar?", atau "apa yang datang tepat setelah x?" dengan cepat, tanpa pernah mengurutkan ulang dari awal?

Gambar 6.1 — Posisi bab ini dalam peta jalan enam belas kuliah DAA. Kuliah 6 meninggalkan pengurutan sepenuhnya untuk keluarga struktur baru — pohon yang menjaga isinya tetap urut setiap saat, bukan hanya di akhir satu kali jalannya algoritma. Kuliah 7 beralih ke paradigma yang sama sekali berbeda: pemrograman dinamis.

Jenis persoalan yang sama sekali berbeda

Kuliah 2, 3, dan 5 semuanya bertanya "diberi seluruh data di muka, hasilkan larik terurut." Bab ini bertanya "diberi data yang datang dan pergi seiring waktu, selalu siap menjawab kueri berbasis urutan dengan cepat." Alat untuk pekerjaan pertama adalah algoritma; alat untuk pekerjaan kedua adalah struktur data — satu yang tidak pernah "selesai" seperti Quicksort selesai, dan justru harus tetap benar dan efisien setelah setiap satu pembaruan.

6.2 Mengapa Larik Terurut Tidak Cukup: ADT Kamus

Kamus (juga disebut himpunan dinamis atau struktur pencarian) adalah tipe data abstrak yang harus secara efisien mendukung, minimal: SEARCH(k) — apakah kunci k ada, dan jika ya, di mana; INSERT(x) — tambahkan elemen x; dan DELETE(x) — hapus elemen x. Banyak aplikasi juga butuh MINIMUM, MAXIMUM, SUCCESSOR(x), dan PREDECESSOR(x) — elemen terkecil, terbesar, berikutnya-lebih-besar, dan berikutnya-lebih-kecil yang saat ini ada.

Larik terurut mendukung SEARCH dengan indah — Binary Search dari Kuliah 2 menemukan kunci mana pun dalam O(log n). MINIMUM dan MAXIMUM adalah O(1) — keduanya hanyalah slot pertama dan terakhir. Tetapi INSERT dan DELETE sangat buruk: menjaga larik tetap terurut setelah menyisipkan di tengah membutuhkan pergeseran, rata-rata, Θ(n) elemen untuk membuat (atau menutup) celah. Larik tak-terurut membalik trade-off ini — penyisipan O(1) di akhir, tetapi pencarian O(n), karena tidak ada apa pun yang memberi tahu Anda ke mana harus mencari.

Struktur	SEARCH	INSERT	DELETE	MIN / MAX	SUCCESSOR
Larik tak-terurut	Θ(n)	Θ(1)	Θ(n)	Θ(n)	Θ(n)
Larik terurut	Θ(log n)	Θ(n)	Θ(n)	Θ(1)	Θ(1)
Binary search tree (bab ini)	O(h)	O(h)	O(h)	O(h)	O(h)

Setiap baris di atas punya operasi titik-sumbatannya sendiri. Binary search tree, yang diperkenalkan berikutnya, adalah struktur pertama dalam mata kuliah ini yang mendukung KELIMA operasi dalam waktu yang sebanding hanya dengan h, tinggi pohon — tidak ada operasi yang dipaksa menjadi Θ(n) hanya karena operasi lain butuh cepat. Jebakannya, yang dikembangkan sepanjang bab ini, adalah h sendiri tidak otomatis kecil; Bagian 6.8–6.9 menunjukkan ia bisa memburuk menjadi Θ(n), dan Bagian 6.10 dan seterusnya menunjukkan bagaimana pohon AVL memperbaiki itu.

6.3 Binary Search Tree: Definisi dan Sifat BST

Binary Search Tree (BST)

Binary search tree adalah pohon biner di mana setiap simpul menyimpan sebuah kunci (dan, biasanya, data terkait), dan memenuhi sifat BST: untuk setiap simpul x, setiap kunci di subpohon kiri x adalah ≤ kunci x, dan setiap kunci di subpohon kanan x adalah ≥ kunci x. Berbeda dengan binary max-heap (Bab 5), urutan BST bersifat GLOBAL di sepanjang jalur akar-ke-daun mana pun — tidak terbatas pada perbandingan induk-versus-anak.

Gambar 6.2 — Binary search tree yang menyimpan kunci {1, 3, 4, 6, 7, 8, 10, 13, 14}. Periksa sifat BST di akar: setiap kunci di subpohon kiri (1, 3, 4, 6, 7) adalah ≤ 8, dan setiap kunci di subpohon kanan (10, 13, 14) adalah ≥ 8. Sifat yang sama berlaku secara rekursif di setiap simpul lainnya.

BST bukan heap, dan sebaliknya

Binary max-heap Bab 5 hanya menjamin induk ≥ anak — sebuah batasan yang murni LOKAL yang tidak mengatakan apa pun tentang bagaimana sebuah simpul dibandingkan dengan pamannya, sepupunya, atau simpul mana pun di luar pasangan induk/anaknya sendiri. Sifat BST bersifat GLOBAL: simpul 6 pada Gambar 6.2 bukan sekadar ≥ anak-anaknya sendiri — ia dijamin lebih kecil dari setiap kunci di mana pun dalam subpohon kanan simpul 8, bahkan simpul yang 6 tidak punya sisi langsung ke sana. Mengacaukan kedua sifat ini adalah salah satu kesalahan konseptual paling umum saat berpindah dari Bab 5 ke bab ini.

6.3.1 Traversal In-Order Memulihkan Urutan Terurut

Karena sifat BST, mengunjungi simpul-simpul pohon secara in-order (secara rekursif: subpohon kiri, lalu simpul itu sendiri, lalu subpohon kanan) selalu menghasilkan setiap kunci dalam urutan tak-menurun. Pohon Gambar 6.2, jika dikunjungi in-order, menghasilkan tepat 1, 3, 4, 6, 7, 8, 10, 13, 14 — terurut, tanpa kerja tambahan di luar traversal Θ(n) itu sendiri. Ini adalah pemeriksaan kewarasan paling berguna untuk implementasi BST mana pun: setelah urutan penyisipan dan penghapusan apa pun, traversal in-order dari BST yang benar harus selalu keluar terurut.

```
INORDER-WALK(x):
  if x != NIL
      INORDER-WALK(x.left)
      print x.key
      INORDER-WALK(x.right)
```

6.4 Kueri Statistik-Urutan: MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR

Tiga dari operasi kamus pada tabel Bagian 6.2 muncul dari sifat BST hampir secara gratis, begitu Anda memperhatikan di mana kunci terkecil dan terbesar pasti berada.

- MINIMUM(x): kunci terkecil di subpohon berakar x selalu ditemukan dengan mengikuti pointer anak-kiri hingga mencapai simpul tanpa anak kiri — simpul itu menyimpan minimum. Secara simetris, MAXIMUM(x) mengikuti pointer anak-kanan hingga ke ujung.
- SUCCESSOR(x): kunci berikutnya-lebih-besar setelah x. Jika x punya subpohon kanan, successor adalah MINIMUM(x.right) — kunci terkecil yang secara ketat lebih besar dari x. Jika x tidak punya subpohon kanan, successor adalah leluhur terendah dari x yang anak kirinya juga leluhur x (dengan kata lain: naik hingga Anda berpindah dari anak-kiri ke induknya).
- PREDECESSOR(x): bayangan cermin dari SUCCESSOR — MAXIMUM(x.left) jika x punya subpohon kiri, jika tidak, leluhur terendah yang dicapai dengan naik dari anak-kanan.

> **Setiap operasi ini adalah O(h)**
> MINIMUM dan MAXIMUM melakukan paling banyak h penelusuran pointer. SUCCESSOR dan PREDECESSOR melakukan paling banyak satu panggilan MINIMUM/MAXIMUM (yang sendiri O(h)) ditambah paling banyak h langkah ke atas — tetap O(h) secara keseluruhan. Ini adalah pola yang berulang untuk setiap operasi BST dalam bab ini: biayanya selalu dibatasi oleh tinggi pohon, bukan langsung oleh jumlah kunci.

6.5 BST-SEARCH: Menyusuri Satu Jalur dari Akar

Mencari BST untuk kunci k dimulai di akar dan, di setiap simpul, membandingkan k dengan kunci simpul saat ini: jika sama, selesai; jika lebih kecil, rekursi ke kiri; jika lebih besar, rekursi ke kanan. Karena sifat BST, satu perbandingan ini di setiap langkah cukup untuk mengeliminasi seluruh subpohon dari pertimbangan — tidak pernah ada kebutuhan untuk mundur (backtrack).

```
BST-SEARCH(x, k):
  if x == NIL or k == x.key
      return x
  if k < x.key
      return BST-SEARCH(x.left, k)
  else
      return BST-SEARCH(x.right, k)
```

Telusuri SEARCH(root, 4) pada pohon Gambar 6.2: 4 < 8, jadi ke kiri ke 3; 4 > 3, jadi ke kanan ke 6; 4 < 6, jadi ke kiri ke 4 — ditemukan, setelah 3 perbandingan. Gambar 6.3 menggambar ketiga langkah tersebut.

Gambar 6.3 — SEARCH(root, 4) pada pohon dari Gambar 6.2. Tepat satu jalur diikuti dari akar ke target — tidak ada bagian pohon lain yang pernah diperiksa, karena di setiap simpul sifat BST memberi tahu Anda dengan tidak ambigu subpohon tunggal mana yang mungkin berisi kunci tersebut.

> **Waktu eksekusi SEARCH**
> BST-SEARCH berjalan dalam O(h), di mana h adalah tinggi pohon — satu perbandingan per level, mengikuti tepat satu jalur akar-ke-simpul. Pada pohon Gambar 6.2 (tinggi 3), tidak ada pencarian yang pernah menghabiskan lebih dari 4 perbandingan (akar plus 3 level ke bawah).

6.6 BST-INSERT: Jatuh ke Tempat Kosong

Menyisipkan kunci baru z bekerja hampir identik dengan SEARCH: turun dari akar membandingkan z dengan kunci setiap simpul, pergi ke kiri atau kanan persis seperti SEARCH — kecuali bahwa ketika Anda mencapai pointer NIL (kosong) alih-alih menemukan z sudah ada, Anda menempelkan simpul baru yang menyimpan z tepat di sana.

```
BST-INSERT(T, z):                    // z adalah simpul baru dengan z.key sudah
diisi
  y = NIL
  x = T.root
  while x != NIL                     // temukan induk y tempat z seharusnya
berada
      y = x
      if z.key < x.key
          x = x.left
      else
          x = x.right
  z.parent = y
  if y == NIL
      T.root = z                     // pohon sebelumnya kosong
  elif z.key < y.key
      y.left = z
  else
      y.right = z
```

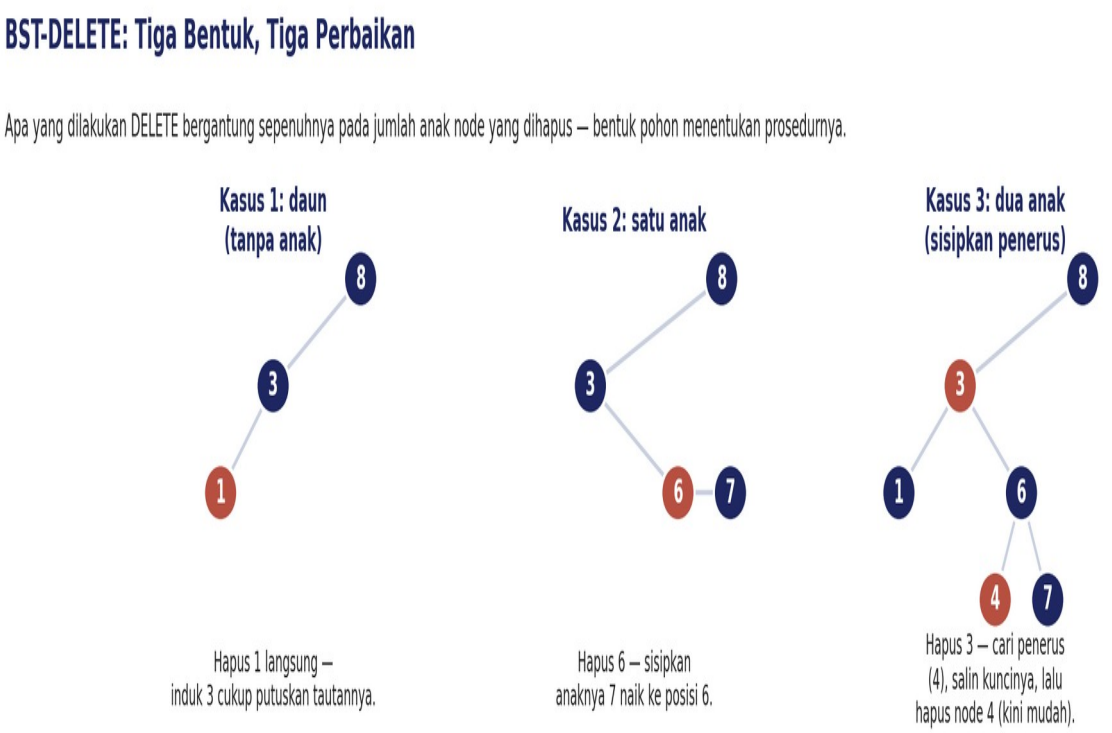
Wawasan kunci

BST-INSERT tidak pernah perlu menyusun ulang simpul yang SUDAH ADA — ia hanya pernah menambahkan satu daun baru. Ini membuat penyisipan murah (O(h), batas yang sama dengan SEARCH) tetapi juga persis mengapa bentuk BST adalah rekaman fosil langsung dari urutan penyisipannya: dua BST yang menyimpan himpunan kunci yang identik bisa punya tinggi yang sangat berbeda tergantung urutan kedatangan kuncinya. Bagian 6.9 mengubah pengamatan ini menjadi persoalan sentral bab ini.

6.7 BST-DELETE: Tiga Kasus, Tiga Perbaikan

Penghapusan adalah operasi BST yang paling rumit, karena menghapus sebuah simpul tidak boleh memutus atau mengacaukan urutan sisa pohon. Perbaikannya bergantung sepenuhnya pada berapa banyak anak yang dimiliki simpul yang dihapus.

- Kasus 1 — simpul adalah daun (tanpa anak): cukup hapus; perbarui pointer induknya menjadi NIL.
- Kasus 2 — simpul punya tepat satu anak: sambungkan anak itu naik ke posisi bekas simpul yang dihapus, menghubungkannya langsung ke induk simpul yang dihapus.
- Kasus 3 — simpul punya dua anak: temukan in-order successor-nya (MINIMUM(x.right) dari Bagian 6.4 — dijamin TIDAK punya anak kiri), salin kunci successor itu ke simpul yang "dihapus", lalu hapus simpul successor itu sebagai gantinya. Karena successor punya paling banyak satu anak (anak kanan, mungkin tidak ada), menghapus successor itu selalu Kasus 1 atau Kasus 2 — tidak pernah Kasus 3 lagi.



Gambar 6.4 — Ketiga kasus BST-DELETE pada variasi pohon Gambar 6.2. Kasus 3 (menghapus simpul 3) adalah satu-satunya yang butuh langkah kedua: temukan successor (4), salin kuncinya ke atas, lalu hapus simpul 4 yang kini mudah.

Mengapa tidak cari predecessor saja?

Baik in-order successor (MINIMUM dari subpohon kanan) maupun in-order predecessor (MAXIMUM dari subpohon kiri) sama-sama benar untuk Kasus 3 — keduanya menjaga sifat BST. Beberapa implementasi berselang-seling antara keduanya untuk menjaga pohon sedikit lebih seimbang selama banyak penghapusan, tetapi BST biasa tidak memberi jaminan seperti itu; Bagian 6.10 dan seterusnya persis merupakan perbaikan untuk celah tersebut.

Waktu eksekusi DELETE

Setiap kasus BST-DELETE melakukan kerja O(1) di luar paling banyak satu panggilan MINIMUM dan penelusuran awal seperti-SEARCH untuk menemukan simpul — jadi biaya totalnya adalah O(h), batas yang sama dengan setiap operasi BST lain dalam bab ini.

6.8 Menganalisis Operasi BST: Semuanya Bergantung pada Tinggi

Lihat kembali Bagian 6.4 hingga 6.7: SEARCH adalah O(h). INSERT adalah O(h). DELETE adalah O(h). MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR semuanya O(h). Setiap operasi kamus tunggal yang dibangun bab ini memakan biaya persis sama — tinggi h pohon — dan tidak ada yang lain. Ini adalah kabar yang sangat baik atau nyaris tidak ada kabar sama sekali, sepenuhnya bergantung pada seberapa besar h bisa menjadi relatif terhadap n, jumlah kunci yang disimpan.

Bentuk pohon	Tinggi h	Biaya setiap operasi
Sempurna seimbang (kasus terbaik)	Θ(log n)	Θ(log n)
Sembarang / rata-rata acak	O(log n) diharapkan, tidak terjamin	diharapkan O(log n)
Sepenuhnya timpang (kasus terburuk)	Θ(n)	Θ(n)

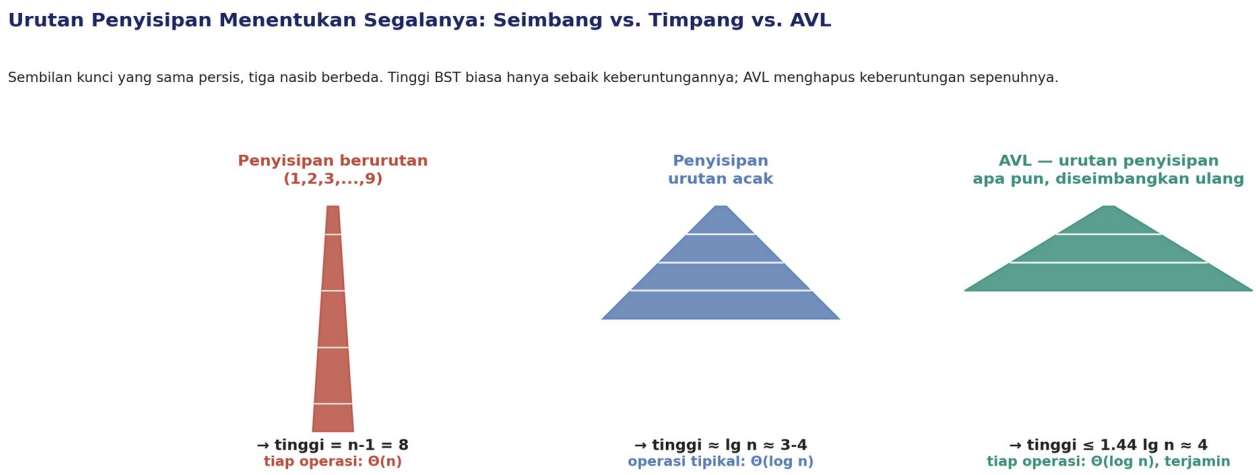
Batasnya nyata, tetapi BERSYARAT

"O(h)" adalah batas yang sepenuhnya benar dan dapat dibuktikan secara ketat untuk setiap operasi BST — tetapi ia diam-diam menyembunyikan rentang nilai yang sangat besar, dari Θ(log n) hingga Θ(n), sepenuhnya bergantung pada h. Batas yang bergantung pada variabel yang sendirinya tak terkendali bukanlah hal yang sama dengan jaminan. Bagian 6.9 menunjukkan persis bagaimana h berakhir di ujung buruk rentang itu, dan Bagian 6.10–6.13 menunjukkan cara membuat ujung baiknya tanpa syarat.

6.9 Persoalan Keseimbangan: Bagaimana Urutan Penyisipan Menciptakan Pohon Degeneratif

Ingat kembali wawasan Bagian 6.6: bentuk BST adalah rekaman langsung dari urutan penyisipannya. Sisipkan sembilan kunci yang sama {1,2,…,9} dalam urutan naik ketat, dan logika turun-dan-

tempelkan-daun milik BST-INSERT tidak punya pilihan selain menempelkan setiap kunci baru sebagai anak-kanan dari kunci sebelumnya — karena setiap kunci baru lebih besar dari segala sesuatu yang sudah ada. Hasilnya bukan pohon dalam pengertian yang berguna; ia adalah rantai condong-kanan, secara struktural identik dengan linked list, dengan tinggi n − 1.



Gambar 6.5 — Sembilan kunci yang sama persis, tiga nasib berbeda. Penyisipan urutan-terurut menghasilkan rantai degeneratif (tinggi 8 — setiap operasi menjadi Θ(n)); urutan penyisipan acak yang menguntungkan menghasilkan pohon yang jauh lebih dangkal (tinggi ≈ lg n); pohon AVL (Bagian 6.10–6.13) menjamin tinggi yang dangkal apa pun urutan kedatangan kuncinya.

Ini bukan kasus tepi yang langka

Masukan yang sudah terurut atau hampir terurut umum dalam praktik — cap waktu log, nama yang dialfabetkan, ID yang naik otomatis, pembacaan sensor. BST biasa tidak sekadar rentan terhadap masukan adversarial yang direkayasa; ia memburuk pada persis jenis masukan yang mungkin dilihat sistem nyata. Ini persis analog dengan penemuan Bagian 5.11.2 bahwa kasus terburuk Quicksort naif JUGA masukan yang sudah terurut — tema yang berulang di seluruh mata kuliah ini: algoritma atau struktur yang performanya bergantung pada urutan masukan butuh argumen eksplisit mengapa masukan realistis tidak akan memicu kasus buruk, dan BST biasa sama sekali tidak punya argumen semacam itu.

Pertanyaan yang layak direnungkan

Setiap operasi dalam bab ini terbukti O(h). Jika h bisa seburuk Θ(n), dalam pengertian apa bab ini memperbaiki larik terurut dari Bagian 6.2 sama sekali? (Jawaban, dikembangkan sepanjang empat bagian berikutnya: belum — BST biasa saja TIDAK menyelesaikan persoalan kamus secara memuaskan. Perbaikannya adalah menambahkan invarian kedua, ditegakkan setelah setiap penyisipan, yang menjaga h kecil secara terbukti.)

6.10 Pohon AVL: Invarian Faktor-Keseimbangan

Dinamai dari penemunya, Georgy Adelson-Velsky dan Evgenii Landis (1962), pohon AVL adalah binary search tree yang ditambah dengan tepat satu aturan ekstra, diperiksa di setiap simpul.

Kondisi keseimbangan AVL

Untuk setiap simpul x dalam pohon AVL, definisikan faktor keseimbangannya BF(x) = height(x.left) − height(x.right) (memperlakukan subpohon kosong sebagai tinggi 0, atau setara tinggi −1 tergantung konvensi — bab ini memakai height(NIL) = 0). Pohon AVL mensyaratkan BF(x) ∈ {−1, 0, +1} untuk setiap simpul x, setiap saat.

Setiap operasi yang sudah didefinisikan — SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR — bekerja sepenuhnya tanpa perubahan pada pohon AVL, karena pohon AVL ADALAH BST; ia hanya memenuhi satu invarian tambahan. INSERT dan DELETE, bagaimanapun, harus diperluas: setelah penyisipan atau penghapusan bergaya-BST biasa, faktor keseimbangan setiap simpul pada jalur kembali naik ke akar harus diperiksa ulang, dan diperbaiki jika sudah bergeser ke +2 atau −2.

6.10.1 Mengapa {−1, 0, +1} Cukup untuk Menjamin O(log n)

Bagian 6.13 membuktikan ini secara ketat lewat argumen pohon-Fibonacci, tetapi intuisinya sederhana: faktor keseimbangan yang dibatasi paling banyak 1 melarang kedua subpohon sebuah simpul berbeda tinggi lebih dari satu level. Batasan tunggal itu, ditegakkan di SETIAP simpul secara bersamaan, cukup kuat untuk membatasi tinggi seluruh pohon pada kira-kira 1,44·lg(n), apa pun urutan penyisipannya — mengubah batas O(h) bersyarat Bagian 6.9 menjadi jaminan O(log n) tanpa syarat.

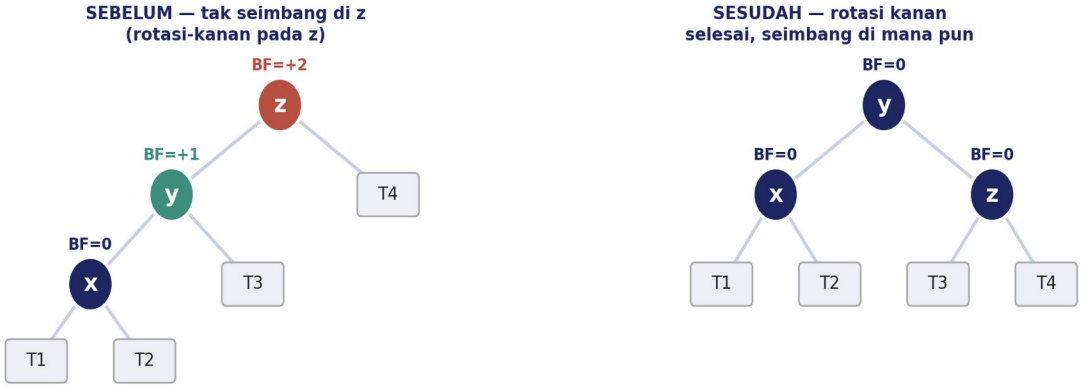
6.11 Memulihkan Keseimbangan: Empat Kasus Rotasi

Ketika sebuah penyisipan (Bagian 6.12) mendorong faktor keseimbangan suatu simpul z menjadi +2 atau −2, sebuah rotasi — penyusunan ulang lokal O(1) atas sejumlah kecil pointer — memulihkan invarian tersebut. Ada tepat empat kasus, ditentukan oleh subpohon z mana yang lebih berat dan subpohon dari subpohon ITU mana yang lebih berat.

- Kasus LL (z berat-kiri, dan anak-kiri z yaitu y sendiri berat-kiri atau seimbang): satu rotasi KANAN tunggal di z.
- Kasus RR (z berat-kanan, dan anak-kanan z yaitu y sendiri berat-kanan atau seimbang): satu rotasi KIRI tunggal di z — bayangan cermin dari LL.
- Kasus LR (z berat-kiri, tetapi anak-kiri z yaitu y berat-kanan): rotasi KIRI di y, segera diikuti rotasi KANAN di z — rotasi ganda.
- Kasus RL (z berat-kanan, tetapi anak-kanan z yaitu y berat-kiri): rotasi KANAN di y, segera diikuti rotasi KIRI di z — bayangan cermin dari LR.

Memulihkan Keseimbangan: Rotasi Kanan (Kasus LL)

Node z punya faktor keseimbangan +2 (berat kiri) dengan anak kiri y yang juga berat kiri — satu rotasi kanan di z memulihkan keseimbangan dalam O(1).



Gambar 6.6 — Kasus LL: simpul z punya faktor keseimbangan +2 dengan anak kirinya y juga berat-kiri. Satu rotasi kanan tunggal di z — y naik mengambil tempat z, z menjadi anak-kanan y, dan bekas subpohon-kanan y yaitu T3 menjadi subpohon-kiri baru z — memulihkan setiap faktor keseimbangan menjadi 0 dalam O(1).

```
ROTATE-RIGHT(z):                          // z berat-kiri (kasus LL)
  y = z.left
  z.left = y.right                        // bekas subpohon-kanan y pindah ke bawah
z
  y.right = z                             // z turun menjadi anak-kanan y
  hitung-ulang height(z), lalu height(y)  // tinggi z dulu: y kini bergantung
padanya
  return y                                // y adalah akar baru subpohon ini
```

Rotasi menyentuh O(1) pointer, bukan O(n)

Meski "menyusun ulang pohon" terdengar mahal, satu rotasi tunggal hanya menugaskan-ulang tepat 3 pointer (dan rotasi ganda, 6) dan menghitung-ulang tepat 2 tinggi (atau 3, untuk rotasi ganda) — apa pun jumlah total simpul yang dimiliki pohon. Biaya perbaikan O(1) ini, diterapkan paling banyak sekali per leluhur pada jalur kembali ke akar setelah sebuah penyisipan, adalah yang menjaga biaya total AVL-INSERT tetap O(log n), bukan O(n).

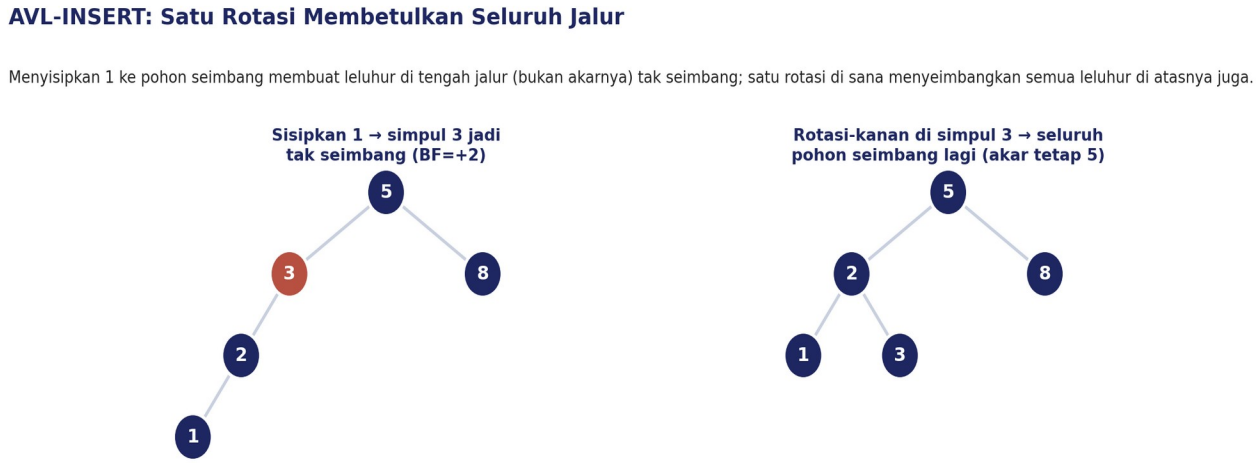
Mengapa kasus LR butuh DUA rotasi, bukan satu?

Satu rotasi kanan tunggal di z saja, diterapkan langsung pada ketidakseimbangan berbentuk-LR, TIDAK memulihkan keseimbangan — coba telusuri dengan tangan dan Anda akan menemukan subpohon z masih tak seimbang setelah hanya satu rotasi, hanya kini condong ke arah lain. Rotasi kiri ekstra di y terlebih dahulu membentuk-ulang kasus LR menjadi kasus LL biasa, yang kemudian diselesaikan dengan benar oleh rotasi kanan tunggal di z. Inilah persis mengapa LR dan RL disebut rotasi ganda.

6.12 AVL-INSERT: Satu Rotasi Membetulkan Seluruh Jalur

AVL-INSERT dimulai dengan BST-INSERT biasa (Bagian 6.6) untuk menempelkan kunci baru sebagai daun, lalu berjalan naik kembali sepanjang jalur dari daun baru itu ke akar, memperbarui tinggi dan faktor keseimbangan setiap leluhur. Saat faktor keseimbangan sebuah leluhur ditemukan menjadi +2 atau −2, tepat SATU dari empat kasus rotasi Bagian 6.11 diterapkan pada leluhur itu — dan yang

mengejutkan, rotasi tunggal itu dijamin memulihkan setiap leluhur lain lebih jauh di sepanjang jalur ke faktor keseimbangan yang valid juga, sehingga tidak lebih dari satu rotasi (tunggal atau ganda) yang pernah dibutuhkan per penyisipan.



Gambar 6.7 — Menyisipkan 1 ke pohon 4-simpul seimbang {5, 3, 8, 2} membuat simpul 3 tak seimbang (faktor keseimbangan menjadi +2) — dua level di atas daun baru, bukan akarnya. Satu rotasi kanan tunggal di simpul 3 memulihkan keseimbangan ke seluruh pohon dalam satu langkah O(1); akar (5) dan subpohon kanannya (8) tidak pernah berpindah, dan faktor keseimbangan akar sendiri turut pulih sebagai efek samping.

```
AVL-INSERT(node, key):
  if node == NIL
      return new Node(key)            // kasus dasar: di sinilah key menempel
  if key < node.key
      node.left = AVL-INSERT(node.left, key)
  else
      node.right = AVL-INSERT(node.right, key)
  hitung-ulang node.height
  bf = balance-factor(node)
  if bf > 1 and key < node.left.key:
      return ROTATE-RIGHT(node)             // LL
  if bf > 1 and key >= node.left.key:
      node.left = ROTATE-LEFT(node.left)
      return ROTATE-RIGHT(node)             // LR
  if bf < -1 and key > node.right.key:
      return ROTATE-LEFT(node)              // RR
  if bf < -1 and key <= node.right.key:
      node.right = ROTATE-RIGHT(node.right)
      return ROTATE-LEFT(node)              // RL
  return node                                // sudah seimbang
```

Waktu eksekusi AVL-INSERT

Penurunan awal untuk menemukan titik penyisipan adalah O(h) = O(log n) (Bagian 6.13). Perjalanan kembali naik, menghitung-ulang tinggi dan memeriksa faktor keseimbangan, juga O(h). Paling banyak satu rotasi (tunggal atau ganda) dilakukan, dengan biaya O(1). Total: O(log n) — dan karena Bagian 6.13 membuktikan h = O(log n) TANPA SYARAT untuk pohon AVL mana pun, batas ini berlaku untuk setiap urutan penyisipan, bukan hanya yang menguntungkan.

- AVL-DELETE mengikuti bentuk yang sama: BST-DELETE biasa (Bagian 6.7), lalu berjalan kembali naik sepanjang jalur yang terpengaruh menyeimbangkan ulang sesuai kebutuhan — kecuali penghapusan bisa membutuhkan hingga O(log n) rotasi di sepanjang jalur itu (bukan

hanya satu), karena menghapus sebuah simpul bisa memicu rantai penyeimbangan-ulang naik ke banyak leluhur. Biaya per-rotasi tetap O(1), sehingga totalnya tetap O(log n).

6.13 Mengapa AVL Menjamin O(log n): Batas Tinggi

Bagian ini membuktikan klaim yang dipakai secara informal sepanjang Bagian 6.10–6.12: pohon AVL dengan n simpul punya tinggi O(log n), tanpa pengecualian untuk urutan penyisipan mana pun.

N(h): jumlah simpul paling sedikit yang bisa dimiliki pohon AVL bertinggi h
Misalkan N(h) adalah jumlah simpul minimum dalam pohon AVL mana pun bertinggi h. Untuk membuat pohon bertinggi h dengan simpul SESEDIKIT mungkin sambil tetap valid-AVL, satu subpohon harus bertinggi h−1 (sekecil yang diizinkan pohon bertinggi h) dan yang lain sekecil yang diizinkan aturan keseimbangan, yaitu tinggi h−2 — memberikan rekurensi N(h) = N(h−1) + N(h−2) + 1, dengan N(0) = 1 dan N(−1) = 0.

Rekurensi ini berbentuk-Fibonacci (hanya berbeda pada "+1"-nya), dan dapat ditunjukkan bahwa N(h) tumbuh paling tidak secepat deret Fibonacci itu sendiri: N(h) ≥ Fib(h+2) − 1, yang tumbuh secara eksponensial dalam h pada laju mendekati rasio emas φ ≈ 1,618. Membalik hubungan ini — menyelesaikan h dalam istilah n alih-alih n dalam istilah h — memberikan batas tinggi secara langsung.

Batas tinggi AVL
Untuk pohon AVL dengan n simpul, tinggi h memenuhi $h \le \log_{\varphi}(\sqrt{5}\cdot(n+2)) - 2 \approx 1{,}44 \cdot \log_2(n+2)$. Bandingkan ini dengan tinggi pohon sempurna-seimbang $\lfloor \lg n \rfloor$: tinggi pohon AVL tidak pernah lebih dari sekitar 44% lebih tinggi dari pohon seimbang terbaik yang mungkin — dan, yang krusial, batas ini berlaku untuk SETIAP pohon AVL dengan n simpul, bukan hanya yang tipikal atau rata-rata.

Wawasan kunci — mengubah "O(h)" menjadi "O(log n)"
Setiap operasi dalam Bagian 6.4–6.7 terbukti O(h) untuk BST sembarang. Bagian 6.9 menunjukkan h bisa seburuk Θ(n). Bagian ini membuktikan bahwa begitu invarian AVL dijaga, h = O(log n) bukan sekadar tipikal — itu dipaksakan secara matematis, untuk setiap kemungkinan urutan penyisipan, tanpa masukan adversarial yang bisa berbuat lebih baik dari batas Fibonacci yang sudah ketat di atas. Itulah seluruh hasil dari Bagian 6.10–6.13: operasi O(h) yang sama dari Bagian 6.4–6.7, kini dengan h sendiri dijamin kecil.

6.14 Profitina dalam Praktik: Di Mana Bab Ini Muncul dalam Produk Nyata

Tentang kotak ini
Seperti di setiap bab, kotak ini menghubungkan materi mata kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan dibangun oleh penulis, tanpa mengungkap detail implementasi yang bersifat kepemilikan.

Profitina memelihara katalog bisnis yang dilacak, entri pesaing, dan kata kunci yang dipantau yang terus berubah dan harus selalu dapat dikueri berdasarkan nama atau identifier — bisnis baru dionboard, yang usang diarsipkan, dan platform harus menjawab "apakah bisnis ini sudah ada di katalog kita?" dan "siapa pesaing terlacak berikutnya secara alfabetis?" kapan saja, persis operasi SEARCH dan SUCCESSOR dari Bagian 6.4–6.5. Karena urutan onboarding secara efektif sewenang-wenang — didorong oleh kapan pelanggan mendaftar, bukan oleh kunci terurut apa pun — para insinyur mengandalkan struktur pencarian seimbang (dalam praktik, indeks balanced-tree atau keluarga B-tree milik pustaka bahasa atau basis data, dibangun di atas persis gagasan AVL/rotasi pada Bagian 6.10–6.12) alih-alih BST biasa, persis karena risiko rantai-degeneratif Bagian 6.9 adalah moda kegagalan yang nyata, bukan hipotetis, terhadap data onboarding yang bisa datang dalam batch yang hampir-terurut (mis., bisnis yang diimpor secara alfabetis dari direktori mitra). Jaminan keseimbangan dari Bagian 6.13 adalah yang memungkinkan Profitina menjanjikan latensi pencarian yang konsisten apa pun urutan pelanggan mendaftar.

6.15 Ringkasan Bab dan Poin-Poin Kunci

- Binary search tree (BST) menjaga sifat BST secara GLOBAL di sepanjang setiap jalur akar-ke-daun: subpohon kiri ≤ simpul ≤ subpohon kanan, secara rekursif — berbeda dengan batasan induk-anak heap yang murni lokal.
- Traversal in-order dari BST selalu memulihkan kunci dalam urutan terurut — pemeriksaan kewarasan terbaik untuk implementasi BST mana pun.
- SEARCH, INSERT, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, dan DELETE semuanya O(h), di mana h adalah tinggi pohon — tidak pernah lebih, tetapi juga tidak pernah otomatis lebih kecil.
- BST-DELETE punya tiga kasus (daun, satu anak, dua anak); kasus dua-anak direduksi menjadi kasus yang lebih mudah dengan menyambungkan kunci in-order successor lalu menghapus simpul successor yang lebih sederhana itu.
- Bentuk BST adalah rekaman langsung dari urutan penyisipannya — urutan penyisipan yang sudah terurut atau hampir terurut (pola dunia-nyata yang umum, bukan kasus tepi yang langka) menghasilkan pohon degeneratif berbentuk-linked-list dengan tinggi Θ(n).
- Pohon AVL menambahkan satu invarian — faktor keseimbangan setiap simpul tetap di {−1, 0, +1} — dipulihkan setelah setiap penyisipan oleh paling banyak satu rotasi (tunggal: LL/RR, atau ganda: LR/RL), masing-masing perbaikan pointer lokal O(1).
- Argumen pohon-Fibonacci membuktikan tinggi pohon AVL selalu O(log n) (khususnya, paling banyak ≈1,44·lg(n+2)) — jaminan TANPA SYARAT, berbeda dengan O(h) bersyarat BST biasa.
- Hasil akhirnya: setiap operasi kamus pada pohon AVL berjalan dalam O(log n), terjamin, apa pun urutan kedatangan kuncinya — persoalan yang sama yang tidak bisa diselesaikan sama sekali oleh trade-off larik-terurut-versus-larik-tak-terurut di Bagian 6.2.

"Agar cukup sederhana untuk dipahami, struktur data butuh bentuk yang baik lebih daripada algoritma yang baik."

— diadaptasi dari prinsip D. Knuth dalam The Art of Computer Programming

(Setiap operasi BST menjadi "jelas" begitu pohonnya berbentuk baik — seluruh tugas AVL adalah menjaga bentuk itu tetap benar.)

Kuliah 7 meninggalkan struktur pencarian untuk paradigma perancangan yang sama sekali berbeda: pemrograman dinamis, yang menyelesaikan persoalan optimisasi dengan menggabungkan solusi subpersoalan yang tumpang tindih — strategi yang secara sepintas terlihat seperti dekomposisi rekursif divide-and-conquer (Kuliah 3) tetapi mengeksploitasi TUMPANG TINDIH antar subpersoalan alih-alih pembelahan bersih tanpa-tumpang-tindih yang membuat rekurensi Merge Sort dan Quicksort begitu bersih dianalisis.

6.16 Pertanyaan Tinjauan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Sisipkan kunci 50, 30, 70, 20, 40, 60, 80 ke BST biasa yang awalnya kosong, dalam urutan itu. Gambar pohon yang dihasilkan dengan gaya Gambar 6.2, dan nyatakan tingginya.
2. Memakai pohon yang Anda bangun pada Pertanyaan 1, telusuri BST-DELETE(30) dengan tangan, identifikasi kasus mana dari ketiga kasus Bagian 6.7 yang berlaku, dengan gaya Gambar 6.4.
3. Konstruksikan urutan penyisipan 6-kunci yang memaksa BST biasa ke tinggi kasus-terburuknya Θ(n), dan jelaskan — dengan kata-kata Anda sendiri, tanpa melihat kembali Bagian 6.9 — mengapa urutan spesifik itu bermasalah.
4. Mulai dari pohon AVL kosong, sisipkan 10, 20, 30 dalam urutan itu. Identifikasi kasus rotasi mana (LL, RR, LR, atau RL) yang terpicu, dan gambar pohon sebelum dan sesudah rotasi, dengan gaya Gambar 6.6.
5. Memakai batas tinggi AVL dari Bagian 6.13, hitung tinggi maksimum yang mungkin dari pohon AVL dengan n = 1.000.000 simpul, dan bandingkan dengan tinggi kasus-terburuk Θ(n) yang bisa dicapai BST biasa dengan n yang sama.
6. Jelaskan, dengan kata-kata Anda sendiri dan tanpa melihat kembali Bagian 6.11, mengapa kasus LR membutuhkan dua rotasi (rotasi kiri di y diikuti rotasi kanan di z) sedangkan kasus LL hanya butuh satu.

Lampiran 6.A — Log Verifikasi untuk Kode Sumber Bab 6

Seperti pada bab-bab sebelumnya, setiap program yang disertakan dengan buku ini dikompilasi dan dijalankan terhadap contoh yang diverifikasi dengan tangan sebelum disertakan. Kedua program di bawah dikompilasi dengan `g++ -O2 -std=c++17 -Wall` (hanya peringatan nilai-kembali scanf yang tidak berbahaya, nol kesalahan) dan menghasilkan persis keluaran yang diprediksi teks di atas.

```
$ printf "9\n8 3 10 1 6 14 4 7 13\n4\n3\n" | ./01_bst
inorder: 1 3 4 6 7 8 10 13 14        # persis pohon Gambar 6.2, terkonfirmasi
terurut
search 4: found at depth 3          # cocok dengan penelusuran 3-langkah Gambar
6.3
inorder after delete 3: 1 4 6 7 8 10 13 14   # Kasus 3: successor 4 disambungkan

$ printf "9\n1 2 3 4 5 6 7 8 9\n5\n5\n" | ./01_bst
inorder: 1 2 3 4 5 6 7 8 9          # penyisipan urutan-terurut
search 5: found at depth 4          # mengonfirmasi bentuk rantai degeneratif

$ printf "9\n1 2 3 4 5 6 7 8 9\n" | ./02_avl
inorder: 1 2 3 4 5 6 7 8 9
height: 4                            # vs. tinggi 8 untuk BST biasa di atas
bound: 1.44 * lg(n+2) = 4.98         # nyaman di dalam batas yang terbukti
```

Latihan SPOJ — buku pendamping
Teknik bab ini diuji sungguhan di juri daring SPOJ. Buka Bab 6 buku pendamping “SPOJ Competitive Programming” (folder SPOJ, BI & EN) untuk memilih struktur pohon/set di bawah anggaran byte — lengkap dengan tips, trik, dan hint agar solusi Anda Accepted, tanpa membocorkan jawaban.

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Bab 12, Binary Search Trees).

[2] G. M. Adelson-Velskii, E. M. Landis. "An algorithm for the organization of information." Proceedings of the USSR Academy of Sciences, 146, 1962, 263–266. Makalah asli yang memperkenalkan apa yang sekarang disebut pohon AVL.

[3] M. A. Weiss. Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2014. (Bab 4, rotasi pohon AVL dan penurunan batas tinggi).

[4] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, 4th ed. Lulu Press, 2020–2023.

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 7

Program Dinamis (Dynamic Programming)

Sebuah paradigma perancangan untuk soal optimisasi: selesaikan tiap subpersoalan berbeda tepat sekali, ingat jawabannya, dan jangan pernah mengulang kerja yang akan diulangi oleh panggilan rekursif yang tumpang tindih.

Tujuan Pembelajaran — setelah bab ini Anda mampu:
(1) Menyatakan dua syarat yang harus dipenuhi suatu soal agar program dinamis bisa diterapkan — substruktur optimal dan subpersoalan tumpang-tindih — dan memeriksa apakah suatu soal memenuhinya. (2) Menjelaskan, memakai Fibonacci rekursif naif, mengapa panggilan rekursif yang eksponensial bisa menyusut jadi subpersoalan BERBEDA yang polinomial. (3) Membandingkan memoization (top-down) dan tabulasi (bottom-up), dan mengimplementasikan salah satunya dengan benar dari sebuah rekurensi. (4) Menurunkan, menelusuri, dan mengimplementasikan solusi program dinamis untuk tiga soal klasik: Rod-Cutting, Perkalian Rantai Matriks, dan Longest Common Subsequence. (5) Merekonstruksi solusi optimal sesungguhnya (bukan cuma nilainya) dengan menelusuri ulang pilihan yang dicatat saat fase pengisian tabel. (6) Menganalisis mengapa waktu eksekusi solusi program dinamis ditentukan oleh JUMLAH subpersoalan berbeda dikali KERJA per subpersoalan, bukan oleh ukuran pohon rekursi naif.

7.1 Recap: Dari Kuliah 6 ke Kuliah 7

Kuliah 6 membahas struktur data yang tetap benar dan efisien di tengah aliran pembaruan yang tak terduga — BST atau pohon AVL tidak pernah "selesai", ia terus menjawab kueri. Bab ini beralih ke jenis soal yang sama sekali berbeda: diberikan satu pertanyaan optimisasi yang terdefinisi jelas — cara termurah memotong batang, jumlah perkalian tersedikit untuk menghitung hasil kali matriks, subsequence terpanjang yang sama dari dua string — temukan jawaban terbaik yang mungkin, sekali, dan mampu membuktikan bahwa itu memang yang terbaik.

Gambar 7.1 — Posisi bab ini dalam peta jalan enam belas kuliah DAA. Kuliah 7 memperkenalkan paradigma perancangan yang sama sekali baru — program dinamis — berbeda dari fokus struktur-pencarian Kuliah 6 maupun paradigma divide-and-conquer Kuliah 3.

Sebuah paradigma, bukan satu algoritma

Divide-and-conquer (Kuliah 3) adalah POLA rekursif spesifik: bagi jadi potongan tak-tumpang-tindih, selesaikan tiap potongan, gabungkan. Program dinamis lebih dekat ke FILOSOFI yang bisa dilapiskan di atas solusi rekursif benar apa pun: jika subpersoalan rekursi itu tumpang tindih, simpan-cache tiap subpersoalan berbeda alih-alih menghitungnya ulang. Setiap algoritma di bab ini lahir sebagai definisi rekursif biasa — bagian program-dinamisnya sepenuhnya soal BAGAIMANA rekursi itu dievaluasi.

7.2 Apa Itu Program Dinamis? Substruktur Optimal dan Subpersoalan Tumpang-Tindih

Program dinamis berlaku untuk soal optimisasi — soal yang meminta nilai terbaik di antara banyak solusi yang mungkin — yang memenuhi dua syarat struktural sekaligus.

Substruktur optimal

Suatu soal memiliki substruktur optimal jika solusi optimal untuk keseluruhan soal dibangun dari solusi optimal untuk subpersoalannya. Substruktur optimal rod-cutting: cara optimal memotong batang panjang-n terdiri dari potongan pertama sepanjang i, ditambah cara OPTIMAL memotong sisa panjang n − i. Jika sisa potongan itu dipotong secara sub-optimal, keseluruhan solusi juga tak bisa optimal — Anda selalu bisa berbuat lebih baik dengan memperbaiki bagian itu saja.

Subpersoalan tumpang-tindih

Suatu soal memiliki subpersoalan tumpang-tindih jika algoritma rekursif naifnya mengunjungi ulang subpersoalan yang persis sama berkali-kali, alih-alih menghasilkan subpersoalan yang sepenuhnya baru di tiap panggilan rekursif (seperti pembagian tak-tumpang-tindih divide-and-conquer). Ketika jumlah subpersoalan BERBEDA kecil — biasanya polinomial terhadap ukuran input — tapi pohon rekursi naif mengunjunginya berulang secara eksponensial, program dinamis mengubah ledakan eksponensial itu jadi biaya yang hanya sebanding dengan jumlah (kecil) subpersoalan berbeda.

Algoritma divide-and-conquer seperti Merge Sort (Kuliah 3) juga mengandalkan substruktur optimal, tapi subpersoalannya tak pernah tumpang tindih — separuh kiri dan separuh kanan larik tak berbagi elemen apa pun, jadi tak ada yang perlu di-cache. Seluruh manfaat program dinamis datang dari syarat KEDUA, subpersoalan tumpang-tindih, yang tak dimiliki soal divide-and-conquer biasa. Bagian 7.3 mengonkretkan tumpang-tindih ini dengan contoh paling sederhana yang mungkin.

7.3 Kisah Peringatan: Fibonacci Rekursif Naif

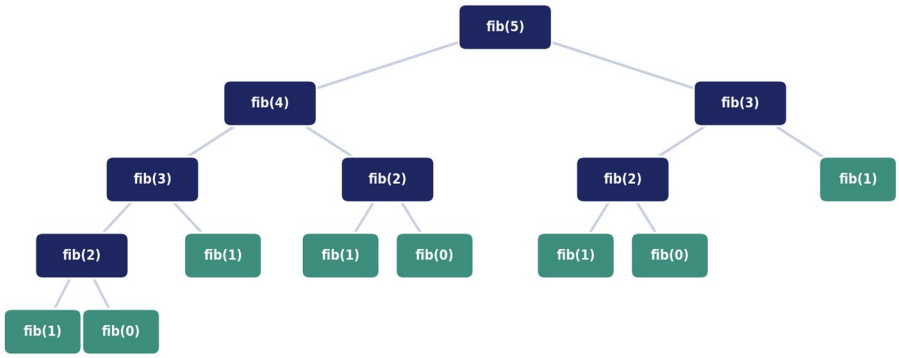
Bilangan Fibonacci didefinisikan oleh rekurensi fib(0) = 0, fib(1) = 1, dan fib(n) = fib(n − 1) + fib(n − 2) untuk n ≥ 2. Menerjemahkan rekurensi ini langsung jadi kode rekursif itu benar — tapi lambat luar biasa, karena alasan yang tak ada hubungannya dengan besarnya angka dan semuanya soal berapa kali panggilan yang SAMA diulang.

```
NAIVE-FIB(n):
  if n == 0 or n == 1
      return n
  return NAIVE-FIB(n-1) + NAIVE-FIB(n-2)
```

Fibonacci Rekursif Naif: Panggilan yang Sama, Berulang-Ulang

Menghitung fib(5) via rekursi buku-teks fib(n)=fib(n-1)+fib(n-2) menurunkan ulang fib(3) DUA KALI dan fib(2) TIGA KALI — dari nol, tiap kali.

15 panggilan total untuk menghitung fib(5): hanya 6 subpersoalan BERBEDA (fib(0)..fib(5)) — 9 panggilan lainnya murni kerja berulang.



Gambar 7.2 — Pohon rekursi lengkap untuk NAIVE-FIB(5). Hanya 6 subpersoalan berbeda yang ada — fib(0) hingga fib(5) — tapi pohon ini membuat 15 panggilan total, menurunkan ulang fib(3) dua kali dan fib(2) tiga kali, tiap kali dari nol.

Ledakannya eksponensial, bukan sekadar "agak boros"

NAIVE-FIB(n) membuat Θ(φⁿ) panggilan, dengan φ ≈ 1,618 adalah rasio emas — pertumbuhan yang benar-benar eksponensial, bukan inefisiensi faktor-konstan. NAIVE-FIB(30) saja membuat sekitar 2,7 juta panggilan untuk menghitung nilai dengan hanya 31 kemungkinan input berbeda (fib(0) hingga fib(30)). NAIVE-FIB(50) akan memakan waktu beberapa menit di CPU modern; komputasi setara dengan tiap subpersoalan berbeda diselesaikan sekali hanya makan mikrodetik. Celah antara eksponensial dan linear inilah yang program dinamis hadir untuk menutupnya.

Fibonacci secara trivial memenuhi kedua syarat Bagian 7.2: substruktur optimal otomatis (hanya ada satu kemungkinan "solusi" per n, jadi optimal secara vakum), dan subpersoalan tumpang-tindih persis seperti yang ditunjukkan Gambar 7.2. Dua bagian berikut menyajikan dua cara standar memanfaatkan tumpang-tindih itu.

7.4 Memoization: Program Dinamis Top-Down

Memoization mempertahankan struktur rekursif ASLI algoritma naif sepenuhnya utuh, dan menambahkan tepat satu gagasan: sebelum melakukan kerja rekursif apa pun untuk suatu input, periksa dulu apakah input persis itu sudah pernah diselesaikan. Jika sudah, kembalikan jawaban ter-cache segera; jika belum, hitung seperti biasa, tapi simpan hasilnya sebelum kembali.

```
MEMO-FIB(n, memo):                  // memo adalah tabel, awalnya kosong
  if n in memo
      return memo[n]                // sudah selesai -- tanpa rekursi sama sekali
  if n == 0 or n == 1
      result = n
  else
      result = MEMO-FIB(n-1, memo) + MEMO-FIB(n-2, memo)
  memo[n] = result                  // simpan sebelum kembali
  return result
```

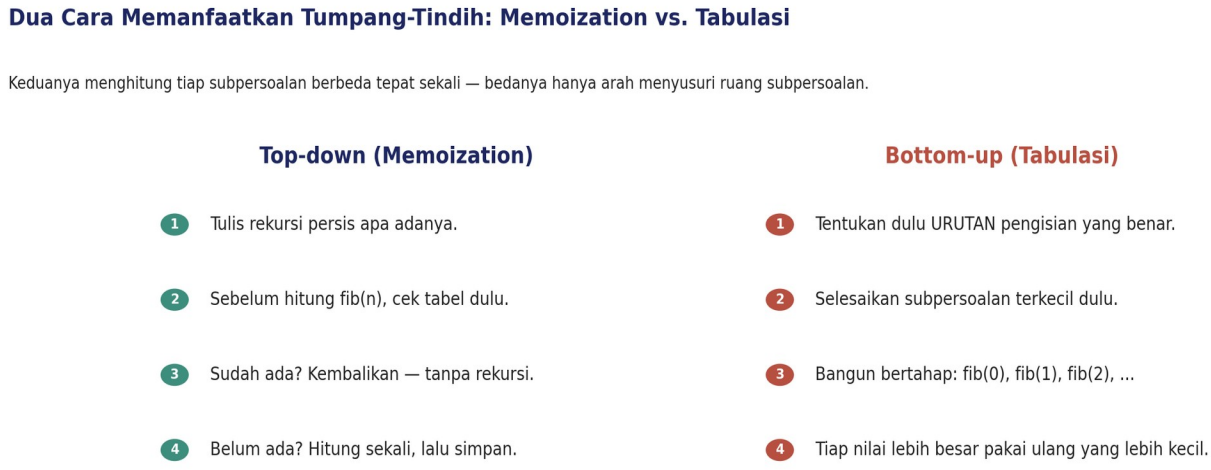
Memoization mengubah pohon rekursi jadi DAG

MEMO-FIB(5) tetap membuat panggilan awal yang SAMA MEMO-FIB(4) dan MEMO-FIB(3) seperti NAIVE-FIB(5) — tapi tiap panggilan setelah yang pertama untuk suatu n kini mengenai tabel memo dan kembali dalam O(1), alih-alih membuka ulang seluruh subpohon. Graf komputasi efektifnya menyusut dari POHON berukuran eksponensial (Gambar 7.2) jadi rantai subpersoalan berbeda berukuran linear — sebuah directed acyclic graph (DAG) dengan tepat n+1 simpul berbeda.

7.5 Tabulasi: Program Dinamis Bottom-Up

Tabulasi menyelesaikan persis subpersoalan yang sama dengan memoization, tapi dari arah berlawanan: alih-alih mulai dari atas (fib(n)) dan berekursi turun sampai mengenai kasus dasar, ia mulai dari kasus dasar dan membangun NAIK, mengisi tabel dalam urutan yang menjamin tiap nilai yang dibutuhkan suatu langkah sudah dihitung pada saat langkah itu berjalan.

```
TAB-FIB(n):
  let f[0..n] be a new array
  f[0] = 0
  f[1] = 1
  for i = 2 to n
      f[i] = f[i-1] + f[i-2]        // keduanya sudah dihitung -- tanpa rekursi
  return f[n]
```



Gambar 7.3 — Memoization dan tabulasi menyelesaikan himpunan subpersoalan yang identik dan mencapai waktu eksekusi yang identik — bedanya hanya apakah ruang subpersoalan dijelajahi top-down (mengikuti rekursi alami, meng-cache sambil jalan) atau bottom-up (menyelesaikan yang terkecil dulu, dalam urutan aman yang telah dihitung sebelumnya).

Memilih di antara keduanya

Tabulasi biasanya lebih cepat dengan faktor konstan (tanpa overhead panggilan rekursif) dan lebih mudah dinalar untuk optimisasi ruang (Bagian 7.13 membahas ini lagi). Memoization sering lebih mudah DITULIS dengan benar untuk rekurensi yang rumit, karena hanya perlu mengidentifikasi struktur rekursifnya — urutan pengisian yang benar ditangani otomatis oleh rekursi itu sendiri, sedangkan tabulasi mengharuskan programmer menentukan urutan itu secara eksplisit di muka. Setiap algoritma yang dikembangkan di sisa bab ini disajikan secara bottom-up (tabulasi), mengikuti penyajian CLRS yang biasa dan source code yang dikirim bersama mata kuliah ini.

7.6 Soal Rod-Cutting dan Solusi Program Dinamisnya

Soal Rod-Cutting

Diberikan batang sepanjang n dan tabel harga p[1..n], dengan p[i] adalah harga jual potongan sepanjang i, tentukan pendapatan maksimum r[n] yang bisa diperoleh dengan memotong batang jadi nol atau lebih potongan (panjang bilangan bulat, kombinasi apa pun) dan menjual tiap potongan terpisah. Memotong sendiri gratis; satu-satunya pertanyaan adalah panjang mana yang dipotong.

Argumen substruktur optimal: untuk batang panjang n, bayangkan potongan pertama yang dipotong punya panjang i (1 ≤ i ≤ n, dengan i = n berarti "tanpa potongan sama sekali"). Apa pun nilai i itu, SISA batang (panjang n − i) itu sendiri harus dipotong secara optimal — jika tidak, mengganti sisa yang sub-optimal itu dengan potongan optimal untuk panjang n − i yang sama akan secara ketat meningkatkan total pendapatan, bertentangan dengan optimalitas keseluruhan solusi. Ini memberi rekurensi r[n] = maks untuk 1 ≤ i ≤ n dari (p[i] + r[n − i]), dengan r[0] = 0.

Rod-Cutting: Semua Cara Memotong Batang Panjang-4

Harga p[1..4] = 1, 5, 8, 9. Memotong gratis, tapi tak semua potongan menguntungkan — pembagian terbaik tak jelas hanya dari harga saja.

r[i]: pendapatan terbaik untuk batang panjang i, dan s[i]: panjang potongan pertama

i	0	1	2	3	4
p[i]	–	1	5	8	9
r[i]	0	1	5	8	**10**
s[i]	–	1	2	3	2

r[4]=10 dicapai dengan memotong batang panjang-4 jadi dua potongan panjang-2 (s[4]=2, lalu s[2]=2) — bukan menjual utuh (p[4]=9) atau pembagian lain.

Gambar 7.4 — Tabel r[] dan s[] lengkap untuk batang panjang-4 dengan harga p = [1, 5, 8, 9]. r[4] = 10 (lebih baik daripada menjual batang utuh seharga p[4] = 9) dicapai dengan memotong jadi dua potongan panjang-2 — tak jelas hanya dari tabel harga saja.

Fase 5 — Implementasi: rod-cutting bottom-up dengan rekonstruksi

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n):
  let r[0..n] and s[0..n] be new arrays
  r[0] = 0
  for j = 1 to n                          // selesaikan tiap panjang 1 s/d n
      best = -infinity
      for i = 1 to j                       // coba tiap panjang potongan-pertama
          if p[i] + r[j-i] > best
              best = p[i] + r[j-i]
              best_cut = i
      r[j] = best
      s[j] = best_cut                      // ingat pilihannya, bukan cuma
nilainya
  return r, s
```

s[] yang membuat rekonstruksi mungkin

r[] saja menjawab "berapa pendapatan terbaik", tapi membuang informasi BAGAIMANA mencapainya. Mencatat s[j] — panjang potongan pertama yang dipotong solusi optimal untuk panjang j — pada saat r[j] dihitung tak menambah biaya (memang sudah dihitung untuk mencari maksimum), tapi persis yang dibutuhkan Bagian 7.7 untuk mencetak instruksi pemotongan sesungguhnya, bukan cuma angka.

Waktu eksekusi: loop ganda mempertimbangkan tiap pasangan (i, j) dengan $1 \le i \le j \le n$, memberi $\Theta(n^2)$ — peningkatan drastis dari solusi rekursif naif $\Theta(2^n)$ (yang mencoba tiap satu dari 2^(n-1) cara menyusun batang panjang-n dari potongan berurutan, tanpa memoization sama sekali).

7.7 Merekonstruksi Solusi Rod-Cutting Optimal

Dari larik s[] Bagian 7.6, mencetak urutan potongan sesungguhnya untuk panjang n adalah loop sederhana: lihat s[n], cetak, lalu ulangi untuk panjang SISA n − s[n], sampai panjang sisa mencapai 0.

```
PRINT-CUT-ROD-SOLUTION(s, n):
  while n > 0
      print s[n]                // panjang potongan berikutnya
      n = n - s[n]              // lanjut dengan sisanya
```

Telusuri ini pada tabel Gambar 7.4 untuk n = 4: s[4] = 2, cetak 2, sisa panjang 4 − 2 = 2; s[2] = 2, cetak 2, sisa panjang 2 − 2 = 0, berhenti. Keluaran: 2, 2 — persis cocok dengan keluaran terverifikasi program C++ di Lampiran 7.A.

> **Waktu eksekusi rekonstruksi**
> PRINT-CUT-ROD-SOLUTION melakukan kerja O(1) per iterasi dan panjang sisa selalu berkurang tiap iterasi, jadi ia berhenti dalam paling banyak n iterasi — O(n) total, dapat diabaikan dibanding fase pengisian tabel $O(n^2)$ yang harus berjalan lebih dulu.

7.8 Perkalian Rantai Matriks: Mengapa Pengelompokan Penting

> **Soal Perkalian Rantai Matriks**
> Diberikan rantai matriks $A_1, A_2, \ldots, A_k$ dengan matriks berurutan yang kompatibel untuk perkalian, tentukan urutan perkalian (pengelompokan/parenthesization) yang meminimalkan TOTAL jumlah perkalian skalar yang dibutuhkan untuk menghitung hasil kali penuh $A_1A_2\cdots A_k$. Perkalian matriks bersifat asosiatif, jadi setiap pengelompokan menghasilkan matriks akhir yang SAMA — tapi tidak biaya yang sama.

Perkalian matriks itu sendiri, untuk matriks p×q dikali q×r, berbiaya tepat p·q·r perkalian skalar dan menghasilkan hasil p×r. Contoh di Gambar 7.5 mengonkretkan taruhannya: tiga matriks A_1 (10×100), A_2 (100×5), A_3 (5×50), dikalikan dalam dua urutan berbeda namun sama-sama valid.

Gambar 7.5 — Tiga matriks yang persis sama, hasil kali akhir yang persis sama — tapi pengelompokan $(A_1A_2)A_3$ berbiaya 7.500 perkalian skalar sementara $A_1(A_2A_3)$ berbiaya 75.000 — satu orde besaran lebih banyak, hanya dari pengelompokan.

Ini bukan perbedaan sebesar galat pembulatan

Perbedaan biaya 10× HANYA dari pengelompokan, dengan tiga matriks kecil yang persis sama, adalah hal NORMAL untuk soal ini, bukan kasus terburuk yang dipilih khusus — dan celahnya melebar lagi seiring rantai memanjang dan dimensi makin bervariasi. Mengalikan matriks dalam urutan kemunculannya di suatu ekspresi, tanpa mempertimbangkan biaya, adalah bug performa nyata dan umum di kode numerik.

7.9 Solusi Program Dinamis untuk Perkalian Rantai Matriks

Misalkan m[i][j] menyatakan jumlah minimum perkalian skalar yang dibutuhkan untuk menghitung hasil kali $A_iA_{i+1}\cdots A_j$, dan dimensi diberikan oleh larik p[0..k] dengan matriks A_i berdimensi p[i-1] × p[i]. Argumen substruktur optimal: untuk pengelompokan optimal $A_iA_{i+1}\cdots A_j$, ADA titik pisah k ($i \le k < j$) yang menjadi perkalian TERAKHIR yang dilakukan — pertama menghitung $(A_i\cdots A_k)$ dan $(A_{k+1}\cdots A_j)$ secara optimal, lalu mengalikan kedua hasil itu.

```
MATRIX-CHAIN-ORDER(p):                      // p[0..k], k matriks
  let m[1..k][1..k] and s[1..k][1..k] be new tables
  for i = 1 to k
      m[i][i] = 0                           // satu matriks sendiri tak berbiaya
  for len = 2 to k                          // panjang rantai, terpendek dulu
      for i = 1 to k - len + 1
          j = i + len - 1
          m[i][j] = infinity
          for split = i to j - 1
              cost = m[i][split] + m[split+1][j] + p[i-1]*p[split]*p[j]
              if cost < m[i][j]
                  m[i][j] = cost
                  s[i][j] = split           // ingat DI MANA pisahan terakhir
  return m, s
```

Verifikasi terhadap angka Gambar 7.5 dengan k = 3 matriks, p = [10, 100, 5, 50] (jadi A_1 adalah p[0]×p[1] = 10×100, A_2 adalah p[1]×p[2] = 100×5, A_3 adalah p[2]×p[3] = 5×50): m[1][1] = m[2][2] = m[3][3] = 0. Untuk panjang 2: m[1][2] = p[0]·p[1]·p[2] = 10·100·5 = 5.000; m[2][3] = p[1]·p[2]·p[3] = 100·5·50 = 25.000. Untuk panjang 3, m[1][3] = min untuk split ∈ {1,2}: split=1 memberi m[1][1]+m[2][3]+p[0]p[1]p[3] = 0+25.000+10·100·50 = 75.000; split=2 memberi m[1][2]+m[3][3]+p[0]p[2]p[3] = 5.000+0+10·5·50 = 7.500. Minimumnya, 7.500 di split=2, persis cocok dengan pengelompokan murah $(A_1A_2)A_3$ di Gambar 7.5.

Waktu $\Theta(k^3)$ dari tabel berukuran $\Theta(k^2)$

Ada $\Theta(k^2)$ subpersoalan (i, j) berbeda (satu per pasangan dengan $i \le j$), dan menghitung tiap satu mempertimbangkan hingga k kemungkinan titik pisah — memberi $\Theta(k^3)$ total waktu. Ini contoh langsung prinsip Bagian 7.2: soal yang solusi rekursif naifnya akan mencoba setiap kemungkinan pengelompokan penuh (jumlah eksponensial bilangan Catalan) menyusut jadi jumlah subpersoalan berbeda yang polinomial, tiap satu diselesaikan sekali.

7.10 Longest Common Subsequence: Soal dan Rekurensi

> **Soal Longest Common Subsequence (LCS)**
> Diberikan dua sekuens X = ⟨x_1,...,x_m⟩ dan Y = ⟨y_1,...,y_n⟩, temukan panjang (dan, idealnya, contoh) sekuens terpanjang yang merupakan SUBSEQUENCE dari X maupun Y. Subsequence tak harus kontigu — didapat dengan menghapus nol atau lebih elemen dari sekuens asli tanpa mengubah urutan elemen yang tersisa.

Misalkan c[i][j] menyatakan panjang LCS dari prefiks-panjang-i X dan prefiks-panjang-j Y. Argumen substruktur optimal terbagi dua kasus: jika x_i = y_j, karakter yang cocok itu PASTI bagian dari suatu longest common subsequence dari kedua prefiks penuh (argumen cut-and-paste standar menunjukkan LCS mana pun yang tak memakainya bisa diperpanjang), jadi c[i][j] = c[i-1][j-1] + 1. Jika x_i ≠ y_j, LCS dari kedua prefiks tak bisa memakai x_i DAN y_j sekaligus, jadi sama dengan yang lebih baik antara membuang salah satunya: c[i][j] = max(c[i-1][j], c[i][j-1]).

```
LCS-LENGTH(X, Y):                    // X panjang m, Y panjang n
  let c[0..m][0..n] be a new table, initialized to 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
          else
              c[i][j] = max(c[i-1][j], c[i][j-1])
  return c
```

7.11 Solusi Program Dinamis untuk LCS: Mengisi Tabel

Telusuri LCS-LENGTH pada contoh asli CLRS, X = ⟨A,B,C,B,D,A,B⟩ dan Y = ⟨B,D,C,A,B,A⟩. Gambar 7.6 menunjukkan tabel 8×7 lengkap (baris 0..7 untuk prefiks X, kolom 0..6 untuk prefiks Y).

Longest Common Subsequence: Mengisi — dan Membaca Ulang — Tabel

X = A B C B D A B, Y = B D C A B A (contoh asli CLRS). c[i][j] = panjang LCS dari i huruf pertama X dan j huruf pertama Y.

		Y					
i↓ j→		B	D	C	A	B	A
	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

X

Menelusuri balik dari c[7][6] sepanjang jalur bersorot mengembalikan LCS “BCBA”, panjang 4 — baca langkah diagonal (cocok) secara terbalik.

Gambar 7.6 — Tabel LCS lengkap untuk X = ABCBDAB, Y = BDCABA. c[7][6] = 4 adalah jawabannya. Sel emas menandai empat kecocokan karakter yang ditemukan saat menelusuri balik (Bagian 7.12); sel hijau-muda menandai sel antara yang dilalui tanpa kecocokan.

> **Membaca bentuk tabel**
> Tiap langkah diagonal (kiri-atas ke kanan-bawah) dalam tabel berkorespondensi dengan karakter YANG COCOK antara X dan Y yang dimasukkan ke LCS. Tiap langkah horizontal atau vertikal berkorespondensi dengan melewati satu karakter dari Y atau X, karena karakter itu tak ikut serta dalam longest common subsequence khusus ini. c[i][j] tak pernah berkurang seiring i atau j bertambah — monotonisitas ini pemeriksaan-kewarasan yang berguna untuk implementasi apa pun.

Waktu eksekusi: loop ganda mengisi Θ(mn) sel tabel, tiap sel dalam waktu O(1) memakai dua tetangga yang sudah dihitung — Θ(mn) total, dibanding solusi rekursif naif Θ(2^(m+n)) (yang mencoba setiap kemungkinan cara menyelaraskan karakter, dengan tumpang-tindih masif antar subpersoalan).

7.12 Merekonstruksi LCS dengan Backtracking

Seperti larik s[] Bagian 7.7, tabel c[][] yang sudah selesai hanya memberi PANJANG LCS. Memulihkan longest common subsequence sesungguhnya berarti menelusuri balik dari c[m][n], di tiap langkah memeriksa kasus rekurensi mana yang menghasilkan nilai sel saat ini, dan mengikuti kasus itu secara terbalik.

Fase 5 — Implementasi: LCS dengan rekonstruksi backtracking

```
PRINT-LCS(X, Y, c):                    // telusuri balik dari (m, n)
  i = length(X); j = length(Y)
  result = empty sequence
  while i > 0 and j > 0
      if X[i] == Y[j]
          prepend X[i] to result       // karakter ini bagian dari LCS
          i = i - 1; j = j - 1          // pindah diagonal
      elif c[i-1][j] >= c[i][j-1]
          i = i - 1                     // karakter baris ini tak dipakai
      else
          j = j - 1                     // karakter kolom ini tak dipakai
  return result
```

Telusuri ini pada tabel Gambar 7.6 mulai dari c[7][6] = 4: X[7]='B' ≠ Y[6]='A', dan c[6][6]=4 ≥ c[7][5]=4, jadi pindah naik ke (6,6). X[6]='A' = Y[6]='A' — cocok, tambahkan 'A' di depan, pindah diagonal ke (5,5). X[5]='D' ≠ Y[5]='B', c[4][5]=3 ≥ c[5][4]=2, pindah naik ke (4,5). X[4]='B' = Y[5]='B' — cocok, tambahkan 'B' di depan, pindah diagonal ke (3,4). X[3]='C' ≠ Y[4]='A', c[3][3]=2 ≥ c[2][4]=1, pindah kiri ke (3,3). X[3]='C' = Y[3]='C' — cocok, tambahkan 'C' di depan, pindah diagonal ke (2,2). X[2]='B' ≠ Y[2]='D', c[2][1]=1 ≥ c[1][2]=0, pindah kiri ke (2,1). X[2]='B' = Y[1]='B' — cocok, tambahkan 'B' di depan. Hasil, dibangun dengan menambahkan di depan dalam urutan ini: B, C, B, A → string "BCBA", panjang 4 — persis cocok dengan keluaran kompilasi Lampiran 7.A.

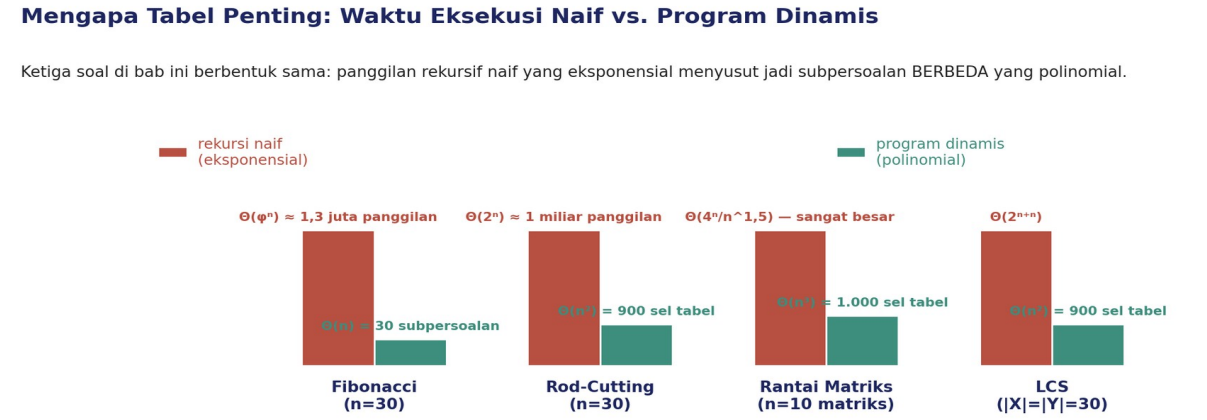
Backtracking berbiaya O(m+n), bukan O(mn)

Tiap langkah PRINT-LCS mengurangi i, mengurangi j, atau mengurangi keduanya — jadi paling banyak m+n langkah pernah diambil, tak peduli sebesar apa tabelnya. Rekonstruksi selalu murah begitu tabel sudah ada; bagian yang mahal, untuk ketiga algoritma di bab ini, adalah mengisi tabel di tempat pertama.

7.13 Menganalisis Program Dinamis: Kapan Ia Menguntungkan?

Setiap algoritma di bab ini berbagi bentuk yang persis sama: formulasi rekursif naif yang subpersoalannya tumpang tindih secara masif, disusutkan oleh memoization atau tabulasi jadi biaya (jumlah subpersoalan berbeda) × (kerja per subpersoalan).

Soal	Subpersoalan berbeda	Kerja per subpersoalan	Waktu total	Waktu rekursif naif
Fibonacci	Θ(n)	O(1)	Θ(n)	$\Theta(\varphi^n)$
Rod-Cutting	Θ(n)	O(n)	$\Theta(n^2)$	$\Theta(2^n)$
Rantai Matriks	$\Theta(k^2)$	O(k)	$\Theta(k^3)$	$\Theta(4^k/k\text{^}1,5)$ (Catalan)
LCS	Θ(mn)	O(1)	Θ(mn)	Θ(2^(m+n))



Gambar 7.7 — Setiap soal di bab ini berbentuk sama: panggilan rekursif naif yang eksponensial menyusut jadi subpersoalan berbeda yang polinomial begitu tiap satu diselesaikan tepat sekali.

Resepnya, dalam tiga pertanyaan

Diberikan soal optimisasi baru, program dinamis layak dicoba ketika jawaban untuk ketiga pertanyaan ini adalah ya: (1) Bisakah solusi optimal dinyatakan secara rekursif dalam solusi optimal subpersoalan yang lebih kecil (substruktur optimal)? (2) Apakah formulasi rekursif itu mengunjungi ulang subpersoalan yang sama lebih dari sekali (subpersoalan tumpang-tindih)? (3) Apakah total jumlah subpersoalan BERBEDA kecil — biasanya polinomial terhadap ukuran input? Jika (1) berlaku tapi (2) tidak, divide-and-conquer biasa (Kuliah 3) sudah menanganinya secara efisien, tanpa perlu meng-cache apa pun.

Pertanyaan optimisasi-ruang yang layak direnungkan

Tabel LCS-LENGTH berukuran Θ(mn) sel, tapi PRINT-LCS Bagian 7.12 hanya pernah melihat O(1) sel tetangga sekaligus saat backtracking — dan menghitung c[i][j] hanya pernah butuh BARIS i-1 dan baris i saat ini, tak pernah baris sebelumnya. Bisakah PANJANG LCS dihitung hanya dengan O(n) ruang alih-alih O(mn)? (Bisa — simpan hanya dua baris sekaligus. Tangkapannya: melakukan itu membuang informasi yang dibutuhkan PRINT-LCS untuk merekonstruksi subsequence sesungguhnya, yang butuh tabel PENUH, atau trik divide-and-conquer yang lebih cerdik atas komputasi ber-ruang-tereduksi, untuk memulihkannya.)

7.14 Profitina dalam Praktik: Di Mana Bab Ini Muncul dalam Produk Nyata

Tentang kotak ini

Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) asal Indonesia yang nyata, dibangun oleh penulis, tanpa membocorkan detail implementasi yang proprietary.

Profitina menilai seberapa terlihat suatu bisnis yang dilacak di berbagai jawaban buatan AI, varian prompt, dan perbandingan pesaing — dan banyak dari perhitungan penilaian itu berbagi sub-hasil yang tumpang tindih antar prompt dan jendela waktu yang berdekatan (mis. dua prompt yang berbeda satu klausa sering berbagi sebagian besar sub-komputasi peringkat lanjutannya). Alih-alih menghitung ulang setiap skor dari nol untuk tiap varian prompt, pipeline penilaian meng-cache sub-hasil antara yang diberi kunci berdasarkan input sub-komputasi itu sendiri — persis gagasan memoization Bagian 7.4 diterapkan di luar Fibonacci buku-teks: sub-komputasi yang identik, diminta berulang kali di seluruh matriks prompt-dan-pesaing yang besar, diselesaikan sekali dan dipakai ulang. Di mana struktur ketergantungan antar sub-skor sudah dipahami di muka (perhitungan-ulang batch tiap malam di seluruh katalog yang dilacak), lintasan tabulasi bottom-up — mengisi skor dalam urutan aman yang tetap dan telah dihitung di muka, persis seperti dijelaskan Bagian 7.5 — lebih disukai daripada memoization per-permintaan, untuk alasan prediktabilitas dan overhead yang sama yang dibahas Bagian 7.5.

7.15 Rangkuman Bab dan Poin-Poin Kunci

- Program dinamis berlaku untuk soal optimisasi dengan KEDUA substruktur optimal (solusi optimal dibangun dari sub-solusi optimal) DAN subpersoalan tumpang-tindih (rekursi naif mengunjungi ulang subpersoalan yang sama berkali-kali).
- Fibonacci rekursif naif membuat $\Theta(\varphi^n)$ panggilan untuk menyelesaikan soal dengan hanya $\Theta(n)$ subpersoalan berbeda — ilustrasi kanonik tumpang-tindih, dan betapa banyak yang terbuang kalau tak dimanfaatkan.
- Memoization (top-down) mempertahankan struktur rekursif alami dan menambahkan cache; tabulasi (bottom-up) menyelesaikan subpersoalan terkecil dulu dalam urutan aman yang telah dihitung. Keduanya mengunjungi tiap subpersoalan berbeda tepat sekali.
- Rod-Cutting: r[j] = maks untuk $1 \le i \le j$ dari (p[i] + r[j−i]), diselesaikan dalam $\Theta(n^2)$; larik pendamping s[] mencatat pilihan potongan-pertama yang dibutuhkan untuk merekonstruksi urutan pemotongan optimal sesungguhnya.
- Perkalian Rantai Matriks: m[i][j] dibangun dari titik pisah terbaik atas $\Theta(k^2)$ subpersoalan, tiap satu mempertimbangkan hingga k pisahan, untuk $\Theta(k^3)$ total — dibanding jumlah pengelompokan (bilangan Catalan) eksponensial yang dicoba secara naif.
- Longest Common Subsequence: c[i][j] = c[i-1][j-1]+1 saat cocok, selainnya maks(c[i-1][j], c[i][j-1]) — $\Theta(mn)$ untuk mengisi tabel, $O(m+n)$ untuk backtracking dan memulihkan subsequence sesungguhnya.
- Merekonstruksi SOLUSI optimal sesungguhnya (bukan cuma nilai numeriknya) selalu berbiaya asimtotik lebih murah daripada membangun tabelnya di tempat pertama — langkah yang mahal selalu fase pengisian tabel.

"Mereka yang tak ingat masa lalu ditakdirkan mengulanginya — dan dalam program dinamis, begitu juga tiap subpersoalan yang lupa kau simpan."

— aforisme mata kuliah ini, memelesetkan dalil George Santayana 1905

(Bellman sendiri menamai bidang ini "dynamic programming" pada 1950-an sebagian karena kata "programming" terdengar meyakinkan secara birokratis bagi penyandang dana riset era Perang Dingin — bukan karena berarti menulis kode dalam arti modern.)

Kuliah 8 adalah pos pemeriksaan kedua mata kuliah ini — Bab/Deck Latihan yang mereview Kuliah 5 hingga 7 (Heapsort/Quicksort, BST/Pohon AVL, dan Program Dinamis) menjelang UTS. Kuliah 9 kemudian memperkenalkan Algoritma Greedy — paradigma yang, berbeda dari program dinamis di bab ini, berkomitmen pada satu pilihan terbaik-lokal di tiap langkah dan tak pernah meninjaunya ulang, menukar penyimpanan-cache subpersoalan yang menyeluruh di bab ini dengan prinsip perancangan yang jauh lebih sederhana (tapi tak selalu berlaku).

7.16 Soal Latihan

Soal-soal ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Gambar pohon rekursi naif lengkap untuk NAIVE-FIB(6), mengikuti gaya Gambar 7.2, dan hitung persis berapa kali fib(2) dan fib(1) masing-masing dihitung ulang dari nol.
2. Memakai tabel harga rod-cutting p = [1, 5, 8, 9, 10] (kini panjang 5), hitung r[5] dan s[5] dengan tangan, mengikuti gaya Gambar 7.4, dan nyatakan urutan pemotongan optimal sesungguhnya.
3. Jelaskan, dengan kata-kata Anda sendiri dan tanpa melihat kembali Bagian 7.9, mengapa argumen substruktur optimal perkalian rantai matriks perlu mempertimbangkan SETIAP titik pisah k yang mungkin, bukan hanya yang di tengah.
4. Telusuri LCS-LENGTH dengan tangan untuk X = "AGCAT" dan Y = "GAC", membangun tabel c[][] lengkap mengikuti gaya Gambar 7.6, dan nyatakan panjang LCS serta satu string LCS yang valid.
5. Jelaskan, dengan kata-kata Anda sendiri, mengapa Fibonacci ber-memoization dan Fibonacci ber-tabulasi memiliki waktu eksekusi asimtotik Θ(n) yang SAMA meski menjelajahi ruang subpersoalan dari arah berlawanan.
6. Suatu soal memiliki substruktur optimal tapi solusi rekursif naifnya tak pernah mengunjungi ulang subpersoalan yang sama. Apakah program dinamis masih berguna untuk soal ini? Justifikasi jawaban Anda memakai dua syarat Bagian 7.2.

Lampiran 7.A — Log Verifikasi untuk Source Code Bab 7

Seperti pada bab-bab sebelumnya, setiap program yang dikirim bersama buku ini dikompilasi dan dijalankan terhadap contoh yang telah diverifikasi tangan sebelum disertakan. Kedua program di bawah dikompilasi dengan `g++ -O2 -std=c++17 -Wall` (nol peringatan, nol galat) dan menghasilkan persis keluaran yang diprediksi dalam teks di atas.

```
$ printf "4\n1 5 8 9\n" | ./01_rod_cutting
r[4] = 10                       # persis cocok dengan tabel Gambar 7.4
s[] = 1 2 3 2
cuts: 2 + 2                     # persis cocok dengan telusuran Bagian 7.7

$ printf "ABCBDAB\nBDCABA\n" | ./02_lcs
table:
0 0 0 0 0 0 0
0 0 0 0 1 1 1
0 1 1 1 1 2 2
0 1 1 2 2 2 2
0 1 1 2 2 3 3
0 1 2 2 2 3 3
0 1 2 2 3 3 4
0 1 2 2 3 4 4
LCS length: 4                   # persis cocok dengan Gambar 7.6
LCS: BCBA                       # persis cocok dengan telusuran Bagian 7.12
```

Latihan SPOJ — buku pendamping
Teknik bab ini diuji sungguhan di juri daring SPOJ. Buka Bab 9 buku pendamping “SPOJ Competitive Programming” (folder SPOJ, BI & EN) untuk katalog lengkap pola dynamic programming SPOJ — lengkap dengan tips, trik, dan hint agar solusi Anda Accepted, tanpa membocorkan jawaban.

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Bab 14, Dynamic Programming).

[2] R. Bellman. Eye of the Hurricane: An Autobiography. World Scientific, 1984. (Catatan Bellman sendiri tentang penamaan "dynamic programming" pada 1950-an.)

[3] M. A. Weiss. Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2014. (Bab 10, Dynamic Programming.)

[4] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, 4th ed. Lulu Press, 2020–2023.

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 8 · BAB LATIHAN

Soal Latihan: Heapsort Hingga Program Dinamis

Tidak ada materi baru di sini — hanya dua belas soal yang dirancang untuk memastikan Kuliah 5 hingga 7 benar-benar melekat, sebelum UTS.

Tujuan Pembelajaran — setelah bab ini Anda mampu:
(1) Menelusuri BUILD-MAX-HEAP dan HEAPSORT pada larik konkret, dan bernalar tentang pelanggaran sifat heap serta operasi Priority Queue. (2) Menelusuri PARTITION Quicksort pada larik konkret, memilih pivot, dan bernalar kapan pengacakan menghindari kasus terburuk $\Theta(n^2)$. (3) Menentukan apakah larik tertentu bisa muncul dari BST yang valid, dan bernalar tentang biaya SEARCH/INSERT/DELETE dalam istilah tinggi. (4) Menunjukkan bagaimana urutan penyisipan bisa (atau tidak bisa) menciptakan BST degeneratif, dan menghitung faktor keseimbangan AVL setelah penyisipan. (5) Mengidentifikasi kasus rotasi (LL/RR/LR/RL) yang dibutuhkan untuk memulihkan keseimbangan AVL setelah penyisipan tertentu, dan menerapkannya dengan tangan. (6) Mengenali substruktur optimal dan subpersoalan tumpang-tindih pada soal yang belum dikenal, dan menelusuri dengan tangan tabel memoization, tabel tabulasi, atau backtrack LCS.

8.1 Rekap: Kuliah 5–7 Secara Singkat

Kuliah 5 menutup kisah pengurutan dengan dua algoritma lagi dari keluarga $\Theta(n \log n)$: Heapsort, dibangun di atas struktur data binary max-heap dan operasi BUILD-MAX-HEAP/MAX-HEAPIFY-nya (serta abstraksi Priority Queue yang diimplementasikan langsung oleh heap), dan Quicksort, yang langkah PARTITION-nya memberikan pengurutan in-place, ramah-cache dengan waktu eksekusi harapan $\Theta(n \log n)$ — tapi kasus terburuk $\Theta(n^2)$ yang pengacakan memang dirancang khusus untuk dihindari. Kuliah 6 kemudian meninggalkan pengurutan untuk struktur data yang dibangun menjawab kueri berbasis urutan selamanya: Binary Search Tree, yang operasi SEARCH/INSERT/DELETE/MINIMUM/MAXIMUM/SUCCESSOR/PREDECESSOR-nya semua berbiaya $O(h)$ — batas yang bisa memburuk jadi $\Theta(n)$ pada urutan penyisipan yang sial, yang persis menjadi persoalan yang invarian faktor-keseimbangan pohon AVL dan empat kasus rotasinya (LL/RR/LR/RL) dibangun untuk memperbaiki, menjamin $h = O(\log n)$ tanpa syarat. Kuliah 7 memperkenalkan paradigma perancangan yang sama sekali berbeda: program dinamis, yang mengenali ketika solusi rekursif suatu soal mengunjungi ulang subpersoalan yang sama berkali-kali (subpersoalan tumpang-tindih) dan memiliki solusi optimal yang dibangun dari sub-solusi optimal (substruktur optimal), dan memanfaatkan itu dengan memoization (top-down) atau tabulasi (bottom-up) — mengubah rekursi naif eksponensial jadi waktu eksekusi polinomial untuk Rod-Cutting, Perkalian Rantai Matriks, dan Longest Common Subsequence.

Gambar 8.1 — Bab ini berada di Kuliah 8 dalam peta jalan enam belas kuliah DAA: sebuah pos pemeriksaan, bukan topik baru, tepat sebelum UTS.

8.2 Cara Memakai Bab Latihan Ini

Apa bab ini — dan bukan apa

Ini adalah Bab Latihan: bab praktik, bukan bab konten baru. Setiap soal di bawah berakar pada materi spesifik dari Kuliah 5–7 (lihat Gambar 8.2). Tiap soal disertai PETUNJUK — dorongan ke arah cara berpikir yang benar — tapi sengaja BUKAN solusi lengkap atau source code. Mengerjakan soal sendiri, sekalipun tidak sempurna, mengajarkan jauh lebih banyak daripada membaca jawaban jadi.

Gambar 8.2 — Setiap soal di Bagian 8.3 berakar pada salah satu dari tiga kuliah sebelumnya.

Sebelum mulai

Coba tiap soal sungguh-sungguh sebelum membaca petunjuknya. Kalau macet, baca hanya petunjuknya — bukan solusi — lalu coba lagi. Ini persis disiplin yang akan dihargai UTS open-book Bagian 8.4: ujian mengizinkan referensi, tapi bukan pengganti penalaran Anda sendiri.

8.3 Soal Latihan

8.3.1 Heapsort, Priority Queue & Quicksort

Soal 1 *[Mudah]*. Apakah larik [16, 4, 10, 14, 7, 9, 3, 2, 8, 1] adalah binary max-heap yang valid (representasi larik biasa, 1-indexed)? Justifikasi jawaban Anda dengan memeriksa sifat heap di setiap simpul internal, dan jika gagal, sebutkan simpul PERTAMA (berdasarkan indeks) yang gagal.

> **Petunjuk**
> Untuk indeks larik i, anak-anaknya berada di 2i dan 2i+1 (1-indexed). Anda hanya perlu memeriksa indeks yang benar-benar punya setidaknya satu anak.

Soal 2 *[Sedang]*. Telusuri HEAPSORT pada max-heap [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]: tunjukkan larik setelah SETIAP langkah extract-max-dan-heapify, bukan cuma hasil akhir terurut. Berapa total panggilan MAX-HEAPIFY yang dibuat ini, dan berapa biaya terburuk masing-masing dalam istilah ukuran heap saat itu?

> **Petunjuk**
> Tiap iterasi HEAPSORT menukar A[1] dengan A[n], mengecilkan heap satu, lalu meng-heapify ulang dari akar — persis MAX-HEAPIFY yang sama yang Anda pakai membangun heap di awal.

Soal 3 *[Sedang]*. Telusuri PARTITION Lomuto pada larik [13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11] memakai elemen TERAKHIR sebagai pivot. Tunjukkan keadaan larik setelah setiap penukaran, dan nyatakan indeks akhir pivot setelah partisi selesai.

> **Petunjuk**
> Lacak invarian dengan cermat: indeks i menandai batas wilayah "diketahui ≤ pivot", dan indeks j memindai maju — penukaran terjadi hanya ketika A[j] ≤ pivot.

Soal 4 *[Sulit]*. Susun sebuah larik spesifik sepanjang 8 yang akan membuat Quicksort DETERMINISTIK (pivot = elemen terakhir, tanpa pengacakan) mengalami kasus terburuk $\Theta(n^2)$-nya. Lalu jelaskan, dengan kata-kata Anda sendiri, MENGAPA memilih pivot secara seragam-acak mengalahkan lawan yang tahu skema partisi Anda tapi tidak tahu pilihan acak Anda.

> **Petunjuk**
> Kasus terburuk untuk Quicksort pivot-elemen-terakhir terjadi ketika larik sudah terurut (atau terurut-terbalik) — tiap panggilan partisi kemudian membagi n elemen jadi (n−1, 0) alih-alih apa pun yang seimbang.

8.3.2 Binary Search Tree & Pohon AVL

Soal 5 *[Mudah]*. Anda diberi keluaran traversal in-order dari suatu pohon biner: 1, 3, 4, 6, 7, 8, 10, 13, 14. Apakah sekuens ini, DENGAN SENDIRINYA, membuktikan pohon itu BST yang valid? Jika tidak, informasi tambahan apa tentang BENTUK pohon yang Anda butuhkan untuk memastikannya?

> **Petunjuk**
> Traversal in-order dari POHON BINER APA PUN yang kebetulan memenuhi sifat BST menghasilkan keluaran terurut — tapi periksa apakah arah sebaliknya (keluaran terurut ⟹ BST valid) benar-benar dijamin oleh sekuens saja, atau butuh mengetahui struktur pohonnya juga.

Soal 6 *[Sedang]*. Sisipkan kunci 10, 20, 30, 40, 50 ke BST yang awalnya kosong, DALAM URUTAN ITU, memakai BST-INSERT biasa. Gambar pohon hasilnya dan nyatakan tingginya. Lalu temukan urutan penyisipan BERBEDA dari lima kunci yang sama yang menghasilkan pohon setinggi 2 (minimum yang mungkin untuk 5 kunci).

> **Petunjuk**
> BST-INSERT selalu menambah daun baru di tempat pencarian kunci itu akan berakhir — jadi menyisipkan kunci dalam urutan sudah-terurut menghasilkan bentuk yang sangat spesifik, sangat buruk. Untuk urutan yang seimbang, pikirkan kunci mana yang harus disisipkan PERTAMA agar berakhir di akar.

Soal 7 *[Sedang]*. Mulai dari pohon 5-simpul seimbang yang Anda bangun di Soal 6 (atau BST seimbang apa pun dari 5 kunci berbeda dengan tinggi 2), sisipkan satu kunci lagi yang menciptakan pelanggaran faktor-keseimbangan (BF = +2 atau −2) di suatu simpul. Nyatakan dengan tepat simpul mana yang menjadi tak seimbang dan faktor keseimbangannya sebelum rotasi apa pun diterapkan.

> **Petunjuk**
> BF(x) = height(x.left) − height(x.right). Hitung ulang tinggi dari bawah ke atas setelah penyisipan, persis seperti AVL-INSERT lakukan di Bagian 6.12, sebelum memutuskan simpul mana (jika ada) yang bergeser keluar dari {−1, 0, +1}.

Soal 8 *[Sulit]*. Untuk simpul tak-seimbang yang Anda temukan di Soal 7, identifikasi kasus rotasi MANA dari keempatnya (LL, RR, LR, RL) yang berlaku, dan lakukan rotasi tersebut dengan tangan — tunjukkan pohon persis sebelum dan persis sesudah. Verifikasi bahwa faktor keseimbangan setiap simpul kembali ke {−1, 0, +1} setelah rotasi Anda.

> **Petunjuk**
> Kasusnya bergantung pada DUA faktor keseimbangan: BF simpul tak-seimbang z sendiri, dan BF anak z yang berada di sisi "berat". Diagram empat-kasus Bagian 6.11 memberi tahu persis rotasi tunggal atau ganda mana yang dipanggil tiap kombinasi.

8.3.3 Program Dinamis

Soal 9 *[Sedang]*. Soal baru: diberikan sekuens n harga saham, temukan panjang rentetan KONTIGU harga naik-ketat terpanjang. Apakah soal ini punya substruktur optimal? Apakah punya subpersoalan tumpang-tindih dalam arti yang didefinisikan Bagian 7.2? Justifikasi kedua jawaban, dan nyatakan apakah program dinamis alat yang tepat di sini atau pendekatan lebih sederhana sudah cukup.

> **Petunjuk**
> Coba definisikan hubungan rekursif dalam istilah "jawaban terbaik yang berakhir tepat di posisi i." Lalu tanyakan: apakah mengevaluasi hubungan itu di setiap posisi membutuhkan menyelesaikan subpersoalan apa pun LEBIH dari sekali?

Soal 10 ***[Sedang]***. Telusuri dengan tangan MEMO-FIB(6) (Bagian 7.4) panggilan demi panggilan: daftar setiap panggilan rekursif yang dibuat, dalam urutan pertama kali dipanggil, dan tandai mana yang HIT cache versus mana yang melakukan kerja sungguhan. Berapa banyak komputasi subpersoalan sungguhan (non-cache) yang dilakukan seluruhnya?

> **Petunjuk**
> Gambar seperti pohon rekursi Gambar 7.2 untuk fib(5), tapi kini coret setiap subpohon yang akan dicegah dieksplorasi oleh hit tabel-memo — hitung hanya komputasi pertama-kali yang bertahan.

Soal 11 ***[Sulit]***. Untuk tabel harga rod-cutting p = [1, 5, 8, 9, 10] (panjang 1 hingga 5), hitung tabel r[] dan s[] LENGKAP dengan tangan, mengikuti EXTENDED-BOTTOM-UP-CUT-ROD (Bagian 7.6–7.7). Nyatakan r[5] dan pemotongan optimal sesungguhnya (sekuens panjang potongan) yang mencapainya.

> **Petunjuk**
> Bangun tabel secara ketat kiri ke kanan: r[0] = 0, lalu untuk tiap j dari 1 hingga 5, coba tiap panjang potongan-pertama i dari 1 hingga j memakai r[j−i], yang sudah dihitung. Ikuti contoh terverifikasi p=[1,5,8,9] Gambar 7.4 sebagai templat Anda, diperpanjang satu kolom.

Soal 12 ***[Sulit]***. Untuk X = "AGCAT" dan Y = "GAC", bangun tabel LCS-LENGTH c[i][j] LENGKAP dengan tangan (Bagian 7.10–7.11), lalu pakai PRINT-LCS (Bagian 7.12) untuk menelusuri balik dan nyatakan satu longest common subsequence yang valid, beserta panjangnya.

> **Petunjuk**
> Ikuti contoh terverifikasi X=ABCBDAB/Y=BDCABA Gambar 7.6 sebagai templat Anda: c[i][j] = c[i−1][j−1] + 1 saat karakter cocok, selainnya maks(c[i−1][j], c[i][j−1]) — lalu telusuri balik dari c[m][n] persis seperti PRINT-LCS lakukan.

8.4 Format UTS: Open-Book, Take-Home Essay

UTS adalah tugas esai open-book, take-home yang mencakup persis materi yang direview di bab ini (Kuliah 5 hingga 7) ditambah semua sebelumnya — tidak lebih dari itu. Anda punya satu minggu untuk mengumpulkan jawaban tertulis dengan penalaran lengkap; jawaban akhir yang benar tanpa justifikasi hanya mendapat sedikit nilai, karena inti ujian format-esai adalah melihat BAGAIMANA Anda berpikir, bukan cuma apa kesimpulan Anda.

Open-book berarti referensi, bukan co-author

Anda boleh berkonsultasi dengan buku teks, catatan, dan slide kuliah saat menjawab. Anda TIDAK BOLEH berkolaborasi dengan teman sekelas atau menyerahkan penalaran yang bukan benar-benar milik Anda — ujian open-book menguji apakah Anda bisa MENERAPKAN apa yang telah Anda pelajari, tanpa pengawasan, yang persis apa arti "merancang dan menganalisis algoritma" dalam praktik. Jika Anda mengutip sumber langsung, sertakan sitasinya.

Kriteria	Apa yang dinilai	Bobot
Kebenaran penalaran	Apakah logika, bukti, telusuran, atau derivasi benar-benar valid — bukan cuma jawaban akhir?	40%
Kejelasan bukti / derivasi	Bisakah pembaca mengikuti tiap langkah tanpa harus menebak maksud Anda?	25%
Penggunaan notasi yang tepat	Apakah O/Ω/Θ, diagram heap/pohon, dan tabel DP dinyatakan presisi, sesuai konvensi bab-bab sebelumnya?	20%
Penyajian	Apakah jawaban terorganisir, terbaca, dan mudah dinilai?	15%

8.5 Profitina dalam Praktik: Self-Review Sebelum Pengumpulan

Tentang kotak ini

Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) asal Indonesia yang nyata, dibangun oleh penulis, tanpa membocorkan detail implementasi yang proprietary.

Sebelum perubahan algoritma baru dirilis ke produksi di Profitina, para insinyur menjalankannya lewat checklist self-review singkat — sangat mirip Lampiran 8.A di bawah — sebelum pernah meminta sepasang mata kedua: apakah invarian struktur-data yang diklaim (sifat heap, urutan BST, faktor keseimbangan AVL) benar-benar diperiksa pada contoh konkret, bukan cuma diasumsikan? Apakah kasus dasar dan urutan pengisian suatu tabel program-dinamis benar-benar ditelusuri dengan tangan setidaknya sekali, seperti yang dituntut soal-soal Bagian 8.3.3? Kebiasaan pemeriksaan-diri adversarial sebelum review eksternal ini — disiplin yang sama di balik pelajaran perburuan-bug yang diulang di seluruh buku teks ini, bahwa hasil yang rapi secara visual tetap bisa salah secara matematis — persis yang dilatih oleh format ujian open-book take-home.

8.6 Rangkuman Bab

- Bab ini tidak memperkenalkan materi baru — ini adalah pos pemeriksaan yang mencakup Kuliah 5 hingga 7.
- Dua belas soal mencakup seluruh rentang yang direview: sifat heap dan HEAPSORT, partisi Quicksort dan kasus terburuknya, bentuk BST dan urutan penyisipan, faktor keseimbangan

AVL dan kasus rotasi, serta dua syarat struktural program dinamis diterapkan pada memoization, tabulasi, Rod-Cutting, dan Longest Common Subsequence.

- Tiap soal disertai petunjuk, bukan solusi — mengerjakan penalarannya sendiri adalah intinya.
- UTS adalah esai open-book, take-home: referensi diperbolehkan, tapi penalarannya harus milik Anda sendiri, dan kebenaran PENALARAN (bukan cuma jawaban akhir) adalah mayoritas nilai.

“Pengurutan adalah yang dikerjakan komputer saat tak ada yang lebih menarik — sampai Anda butuh itu cepat.”

— aforisme mata kuliah ini

(Membaca ulang bukti tidak terhitung menguasainya — menurunkannya ulang dari nol baru terhitung.)

Lampiran 8.A — Checklist Self-Check (Tidak Dinilai)

Sebelum menyerahkan jawaban apa pun — dalam soal bab ini atau dalam UTS itu sendiri — jalankan lewat checklist ini. Ini memakan beberapa menit dan menangkap sebagian besar kesalahan yang paling sering dilihat penilai.

- Apakah saya benar-benar memverifikasi invarian yang diklaim (sifat heap, urutan BST, faktor keseimbangan AVL) pada contoh konkret yang diberikan, bukan cuma menegaskan itu berlaku?
- Dalam tabel yang ditelusuri dengan tangan (larik heap, tabel DP, pohon BST/AVL), apakah saya mengisinya dalam urutan yang benar dan menunjukkan SETIAP keadaan antara, bukan cuma hasil akhir?
- Jika saya mengklaim suatu kasus rotasi (LL/RR/LR/RL), apakah saya menyebutkan dua faktor keseimbangan yang menentukannya, bukan cuma bentuk pohon akhir?
- Jika saya mengklaim suatu soal punya (atau tidak punya) subpersoalan tumpang-tindih, apakah saya benar-benar menunjuk subpersoalan spesifik yang dihitung lebih dari sekali — atau bukti spesifik bahwa tidak ada?
- Apakah saya membedakan kasus TERBURUK dari kasus HARAPAN secara eksplisit di mana pun Quicksort atau pengacakan terlibat?
- Apakah saya membaca ulang jawaban saya sendiri seolah saya penilai skeptis yang mencari celah dalam argumen?

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Bab 6–7, 12, 14).

[2] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, 4th ed. Lulu Press, 2020–2023.

[3] M. A. Weiss. Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2014. (Bab 4, 10).

[4] G. M. Adelson-Velskii, E. M. Landis. "An algorithm for the organization of information." Proceedings of the USSR Academy of Sciences, 146, 1962, 263–266.

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 9

Algoritma Greedy

Paradigma perancangan yang membuat satu pilihan terbaik secara lokal di tiap langkah, tak pernah meninjaunya ulang, dan — untuk kelas soal tertentu yang bisa dibuktikan punya struktur yang tepat — tetap berakhir pada jawaban optimal global.

Tujuan Pembelajaran — setelah bab ini Anda mampu:
(1) Menyatakan dua sifat yang harus dipenuhi suatu soal agar algoritma greedy terbukti benar — sifat greedy-choice dan substruktur optimal — dan membedakannya dari syarat program dinamis. (2) Menurunkan, menelusuri, dan mengimplementasikan solusi greedy untuk soal Activity Selection, serta menjelaskan mengapa mengurutkan berdasarkan waktu SELESAI (bukan waktu mulai atau durasi) yang membuat pilihan greedy aman. (3) Menyusun bukti argumen pertukaran (exchange argument) bahwa suatu pilihan greedy tak pernah membuat jawaban akhir lebih buruk — teknik standar untuk membuktikan kebenaran algoritma greedy apa pun. (4) Membangun pohon Huffman dari sekumpulan frekuensi karakter, menurunkan kode prefix-free yang dihasilkan, dan menghitung total panjang enkodenya. (5) Menjelaskan mengapa greedy-berdasar-rasio optimal untuk soal Fractional Knapsack tapi terbukti SALAH untuk soal 0/1 Knapsack — dan mengidentifikasi, untuk soal baru, di sisi mana dari batas itu soal tersebut berada. (6) Menganalisis waktu eksekusi tiap algoritma greedy di bab ini dan menjelaskan mengapa algoritma greedy biasanya didominasi oleh satu langkah pengurutan, bukan oleh fase pengisian tabel.

9.1 Rekap: Dari Kuliah 8 ke Kuliah 9

Kuliah 8 adalah pos pemeriksaan — tanpa materi baru, hanya kesempatan mengonsolidasi Heapsort, Priority Queue, Quicksort, BST/Pohon AVL, dan Program Dinamis menjelang UTS. Bab ini membuka paradigma perancangan yang sama sekali baru: algoritma greedy. Jika Bab 7 (program dinamis) menyelesaikan setiap subpersoalan berbeda tepat sekali lalu menggabungkan hasilnya, algoritma greedy membuat satu pilihan tak-terbatalkan di tiap langkah, berdasarkan hanya pada apa yang terlihat terbaik SAAT ITU, dan tak pernah menoleh ke belakang untuk meninjaunya ulang.

Gambar 9.1 — Posisi bab ini dalam peta jalan enam belas kuliah DAA. Kuliah 9 memperkenalkan paradigma greedy, tepat setelah pos pemeriksaan UTS dan tepat sebelum algoritma graf — beberapa di antaranya (algoritma MST Kruskal dan Prim, Kuliah 10–11) sebenarnya adalah algoritma greedy yang menyamar.

Mengapa "greedy" bukan hinaan di sini

Dalam bahasa sehari-hari, "greedy" (rakus) menyiratkan kepicikan yang berbalik merugikan. Dalam perancangan algoritma, algoritma greedy hanya disebut benar ketika bisa DIBUKTIKAN bahwa kepicikan itu tidak berbalik merugikan — bahwa berkomitmen pada pilihan terbaik lokal di tiap langkah dijamin mencapai jawaban terbaik global. Bagian 9.2 memastikan dengan tepat apa yang harus benar agar jaminan itu berlaku, dan contoh kerja terakhir bab ini (Bagian 9.9) menunjukkan kasus di mana kelihatannya jaminan itu berlaku, tapi terbukti tidak.

9.2 Apa yang Membuat Algoritma Greedy Benar? Dua Syarat Struktural

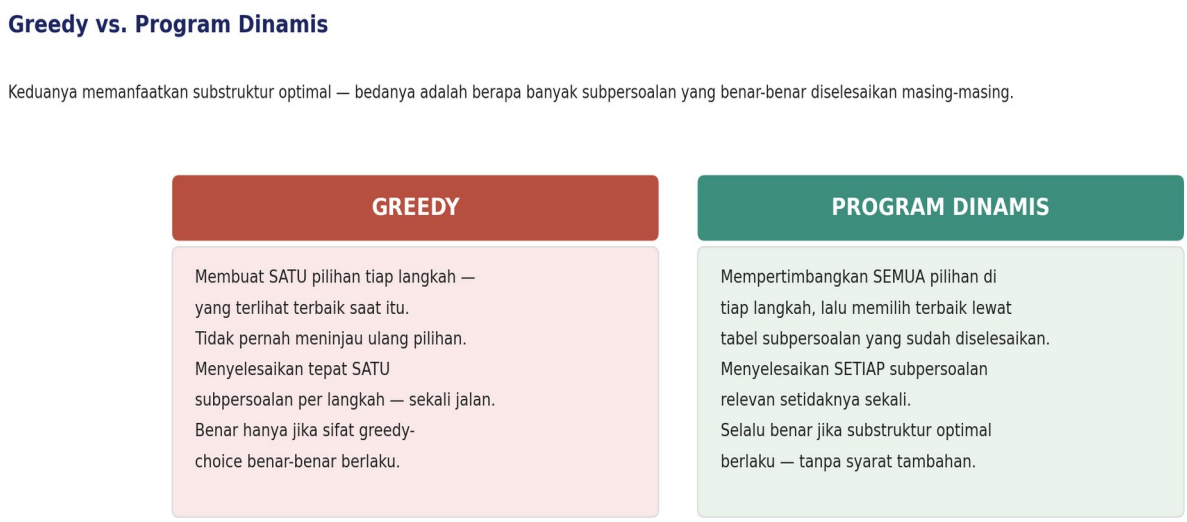
Seperti program dinamis, algoritma greedy berlaku untuk soal optimisasi, dan kebenarannya bertumpu pada sifat struktural soal tersebut — tapi syarat yang lebih ketat dibanding program dinamis.

Sifat greedy-choice

Suatu soal memiliki sifat greedy-choice jika solusi optimal global selalu bisa dicapai dengan membuat satu pilihan LOKAL optimal (greedy) lebih dulu, lalu menyelesaikan secara optimal subpersoalan sisa yang ditinggalkan pilihan itu. Yang krusial, ini harus berlaku apa pun bentuk sisa input — pilihan greedy tak boleh bergantung pada penyelesaian sebagian subpersoalan sisa terlebih dahulu, atau itu bukan lagi pilihan greedy sejati (sekali-jalan, tak-ditinjau-ulang).

Substruktur optimal (lagi)

Persis seperti di Bab 7: solusi optimal untuk keseluruhan soal harus dibangun dari solusi optimal subpersoalan yang ditinggalkan pilihan pertama. Algoritma greedy dan program dinamis SAMA-SAMA membutuhkan substruktur optimal — bedanya, program dinamis, karena tak punya cara mengetahui pilihan mana yang aman di muka, mencoba semua pilihan yang diisyaratkan substruktur optimal lalu menggabungkan yang terbaik lewat tabel, sementara algoritma greedy memotong seluruh pencarian itu karena sifat greedy-choice menjamin pilihan tunggal mana yang harus dicoba.



Gambar 9.2 — Kedua paradigma bertumpu pada substruktur optimal. Program dinamis menyelesaikan setiap subpersoalan yang diisyaratkan substruktur bisa relevan; algoritma greedy, setiap kali sifat greedy-choice juga berlaku, hanya menyelesaikan satu.

Substruktur optimal saja TIDAK cukup untuk greedy

Suatu soal bisa memiliki substruktur optimal tanpa memiliki sifat greedy-choice — 0/1 Knapsack (Bagian 9.9) persis kasus semacam itu. Menerapkan algoritma greedy pada soal yang tak memiliki sifat greedy-choice menghasilkan kode yang berjalan cepat dan mengembalikan jawaban yang percaya diri, terlihat masuk akal, tapi SALAH — tanpa pesan error apa pun, karena tak ada yang tampak tak biasa dari eksekusi algoritmanya. Sifat ini harus dibuktikan, bukan diasumsikan hanya karena suatu soal "terasa greedy".

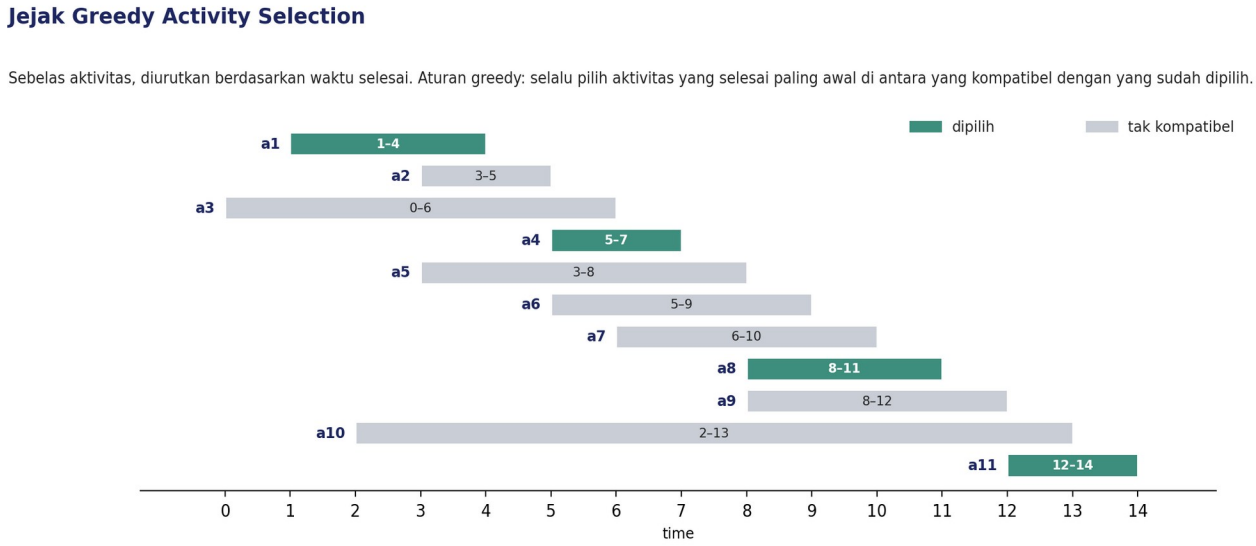
9.3 Soal Activity Selection

Soal Activity Selection

Diberikan sekumpulan n aktivitas, masing-masing dengan waktu mulai dan waktu selesai, serta satu sumber daya bersama (ruangan, mesin, slot rapat) yang hanya bisa menampung satu aktivitas dalam satu waktu, pilih subset TERBESAR yang mungkin dari aktivitas-aktivitas yang saling kompatibel — aktivitas yang interval waktunya tidak tumpang tindih.

Strategi greedy untuk soal ini butuh aturan aktivitas mana yang dipilih lebih dulu. Beberapa aturan yang terdengar masuk akal ternyata salah: memilih aktivitas TERPENDEK lebih dulu, atau yang MULAI paling awal, keduanya bisa ditunjukkan lewat contoh-tandingan kadang menghasilkan jumlah akhir yang tidak optimal. Aturan yang terbukti benar adalah mengurutkan berdasarkan waktu SELESAI dan selalu memilih aktivitas kompatibel yang selesai paling awal.

```
GREEDY-ACTIVITY-SELECTOR(s, f):   // s = waktu mulai, f = waktu selesai, terurut
f
  n = length(f)
  A = { 1 }                        // selalu ambil aktivitas pertama yang
selesai
  last_finish = f[1]
  for i = 2 to n
      if s[i] >= last_finish        // kompatibel dgn semua yg sudah dipilih
          A = A union { i }
          last_finish = f[i]
  return A
```



Gambar 9.3 — Contoh klasik CLRS dengan 11 aktivitas, diurutkan berdasarkan waktu selesai. Aturan greedy memilih a1, lalu melewati setiap aktivitas yang tak kompatibel dengan a1 hingga mencapai a4 (aktivitas selesai-paling-awal yang mulai pada atau setelah waktu selesai a1), lalu berulang — mencapai a8, kemudian a11.

Menelusuri GREEDY-ACTIVITY-SELECTOR pada data Gambar 9.3: mulai dengan a1 (selesai 4). a2 (mulai 3) dan a3 (mulai 0) keduanya mulai sebelum waktu 4 — tak kompatibel, lewati keduanya. a4 (mulai 5) mulai pada atau setelah waktu 4 — kompatibel, pilih, perbarui last_finish jadi 7. a5 (mulai 3), a6 (mulai 5), dan a7 (mulai 6) semuanya mulai sebelum waktu 7 — lewati ketiganya. a8 (mulai 8) kompatibel — pilih, perbarui last_finish jadi 11. a9 (mulai 8) dan a10 (mulai 2) keduanya mulai sebelum 11 — lewati keduanya. a11 (mulai 12) kompatibel — pilih. Jawaban akhir: {a1, a4, a8, a11} — empat aktivitas, persis cocok dengan keluaran terkompilasi Lampiran 9.A.

9.4 Membuktikan Greedy Benar: Argumen Pertukaran

Mengurutkan berdasarkan waktu selesai dan memilih secara greedy mudah dinyatakan — tapi mengapa itu benar-benar optimal? Teknik standar membuktikan algoritma greedy benar adalah ARGUMEN PERTUKARAN (exchange argument): ambil solusi optimal sembarang, lalu tunjukkan itu selalu bisa dimodifikasi ("dipertukarkan") untuk menyertakan pilihan greedy, tanpa membuatnya lebih buruk. Jika pilihan greedy selalu bisa ditukar masuk tanpa biaya, greedy tidak kehilangan apa pun dengan memilihnya lebih dulu.

Argumen pertukaran untuk Activity Selection, garis besarnya

Misalkan a1 adalah aktivitas dengan waktu selesai paling awal, dan misalkan A adalah solusi optimal APA PUN. Jika a1 ∈ A, pilihan greedy sudah menjadi bagian solusi optimal — selesai. Jika a1 ∉ A, misalkan a_k adalah aktivitas di A dengan waktu selesai paling awal. Karena a1 selesai tidak lebih lambat dari a_k (a1 punya waktu selesai paling awal secara global), mengganti a_k dengan a1 di A tak bisa menciptakan tumpang-tindih baru — setiap aktivitas di A yang kompatibel dengan a_k tetap kompatibel dengan a1 yang selesai lebih awal atau sama. Ini menghasilkan solusi BERBEDA A′ = A − {a_k} + {a1}, berukuran SAMA dengan A, yang MEMUAT pilihan greedy. Karena A diasumsikan optimal dan A′ berukuran sama, A′ juga optimal. Bagaimanapun, ada solusi optimal yang memuat a1 — persis yang disyaratkan sifat greedy-choice (Bagian 9.2).

Substruktur optimal menutup argumen ini: begitu a1 ditetapkan sebagai bagian solusi optimal, soal sisanya — memilih subset kompatibel berukuran maksimum dari aktivitas yang mulai pada atau setelah waktu selesai a1 — adalah PERSIS SOAL YANG SAMA (Activity Selection) pada input yang lebih kecil. Menyelesaikan soal sisa itu secara greedy, dan mengulang argumen pertukaran yang sama di tiap langkah, menunjukkan seluruh algoritma greedy menghasilkan solusi optimal global melalui induksi.

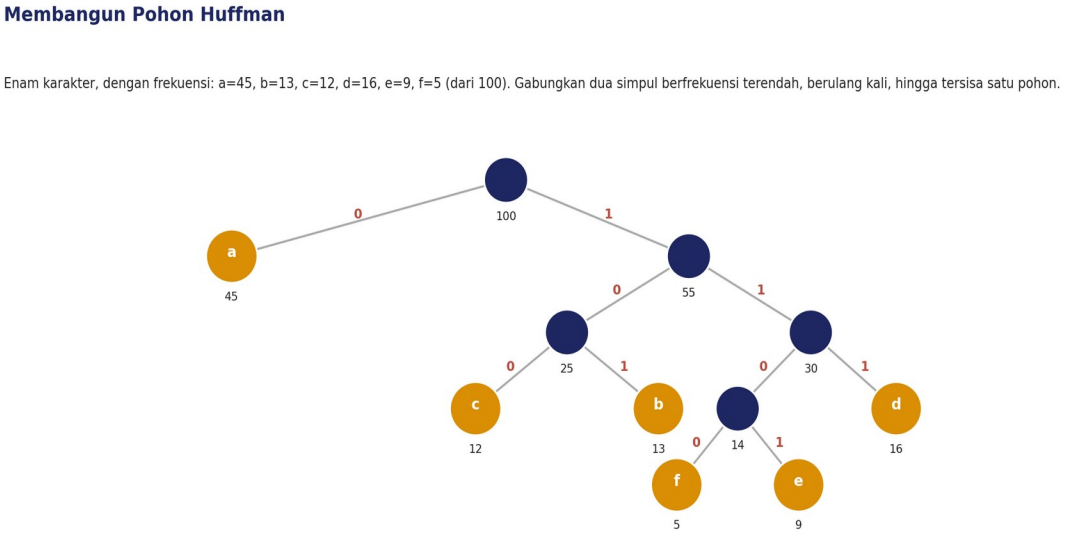
9.5 Huffman Coding: Kompresi Prefix-Free Optimal

Soal Huffman Coding

Diberikan sekumpulan karakter dan frekuensinya (atau probabilitasnya) dalam suatu teks sumber, bangun sebuah KODE BINER panjang-variabel — rangkaian bit berbeda untuk tiap karakter — yang (a) prefix-free (kode tak ada karakter yang merupakan prefiks kode karakter lain, sehingga aliran kode yang digabung bisa didekode tanpa ambiguitas tanpa pemisah) dan (b) meminimumkan jumlah TOTAL bit yang dibutuhkan mengenkode seluruh teks sumber, mengingat frekuensi tersebut.

Gagasan greedy-nya: karakter berfrekuensi TINGGI harus mendapat kode PENDEK, dan karakter berfrekuensi RENDAH bisa memakai kode PANJANG, karena kontribusi panjang kodenya ke total jauh lebih jarang. Algoritma Huffman membangun kode optimal dari bawah ke atas sebagai pohon biner: berulang kali ambil dua simpul berfrekuensi-terendah saat ini (mulanya, karakter individual; kemudian, subpohon gabungan) dan gabungkan menjadi simpul baru berfrekuensi jumlah keduanya, berlanjut hingga tersisa satu pohon. Kode tiap karakter kemudian dibaca sebagai rangkaian cabang kiri/kanan (0/1) dari akar ke daun karakter tersebut.

```
HUFFMAN(C):                          // C = himpunan karakter dgn frekuensinya
  n = |C|
  Q = min-priority-queue dari C, berdasarkan frekuensi
  for i = 1 to n - 1
      lo = EXTRACT-MIN(Q)            // dua simpul berfrekuensi terendah saat ini
      hi = EXTRACT-MIN(Q)
      z = simpul baru
      z.left = lo; z.right = hi
      z.freq = lo.freq + hi.freq
      INSERT(Q, z)
  return EXTRACT-MIN(Q)              // satu simpul tersisa adalah akarnya
```



Gambar 9.4 — Pohon Huffman dibangun dari enam karakter dengan frekuensi a=45, b=13, c=12, d=16, e=9, f=5 (dari 100). Dua simpul berfrekuensi terendah digabung di tiap langkah: mula-mula f+e=14, lalu c+b=25, lalu (f+e)+d=30, lalu (c+b)+((f+e)+d)=55, akhirnya a+55=100.

9.6 Membaca Kode dan Menghitung Penghematannya

Membaca kode tiap karakter dari Gambar 9.4 (0 = cabang kiri, 1 = cabang kanan, dari akar): a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100. Perhatikan sifat yang memang dirancang untuk dihasilkan konstruksi greedy ini: a, karakter paling sering (45 dari 100), mendapat kode terpendek (1 bit); e dan f, yang paling jarang (9 dan 5), mendapat kode terpanjang (4 bit masing-masing).

Phase 5 — Implement: membaca kode dari pohon dan menjumlahkan panjangnya

```
COLLECT-CODES(node, path, codes):        // path mulai sebagai string kosong
  if node.isLeaf
      codes[node.character] = path
      return
  COLLECT-CODES(node.left,  path + '0', codes)
  COLLECT-CODES(node.right, path + '1', codes)
```

Total panjang enkode = Σ (frekuensi × panjang kode) = 45(1) + 13(3) + 12(3) + 16(3) + 9(4) + 5(4) = 45 + 39 + 36 + 48 + 36 + 20 = 224 bit, untuk 100 karakter teks sumber — persis cocok dengan keluaran terkompilasi Lampiran 9.A. Kode PANJANG-TETAP butuh $\lceil \log_2 6 \rceil = 3$ bit per karakter tanpa memandang frekuensi, jadi 100 × 3 = 300 bit total. Kode panjang-variabel Huffman menghemat 76 bit — pengurangan 25,3% — murni dengan memanfaatkan ketimpangan frekuensi, tanpa kehilangan informasi sama sekali.

Mengapa pilihan greedy ini terbukti aman

Argumen pertukaran di sini (bukti lengkap di CLRS Teorema 16.2, dihilangkan demi ringkas) menunjukkan bahwa dua karakter berfrekuensi terendah secara global selalu bisa ditempatkan sebagai SAUDARA (sibling) pada level TERDALAM dari SUATU pohon kode optimal — persis yang dilakukan langkah penggabungan pertama. Substruktur optimal kemudian berlaku: begitu dua karakter itu digabung menjadi satu simbol gabungan, mencari kode optimal untuk alfabet ASLI berkurang menjadi mencari kode optimal untuk alfabet LEBIH KECIL (satu simbol lebih sedikit), yang persis soal yang sama, diselesaikan secara rekursif oleh setiap penggabungan berikutnya.

9.7 Fractional Knapsack: Keberhasilan Greedy Kedua

Soal Fractional Knapsack

Diberikan n item, masing-masing dengan nilai dan berat, serta sebuah ransel berkapasitas W, pilih berapa banyak dari SETIAP item yang diambil (fraksi berapa pun antara 0 dan 100% dari berat item diperbolehkan) untuk memaksimumkan total nilai, dengan syarat total berat yang diambil tak melebihi W.

Aturan greedy-nya: hitung RASIO nilai-terhadap-berat tiap item, urutkan item berdasarkan rasio itu menurun, dan ambil sebanyak mungkin item dengan rasio tertinggi lebih dulu, lalu item rasio tertinggi berikutnya, dan seterusnya, hingga ransel penuh (item terakhir yang diambil boleh hanya sebagian, persis mengisi sisa kapasitas).

Contoh kerja: kapasitas W = 50, dengan tiga item — item 1 (nilai 60, berat 10, rasio 6), item 2 (nilai 100, berat 20, rasio 5), item 3 (nilai 120, berat 30, rasio 4). Terurut berdasarkan rasio: item 1, lalu item 2, lalu item 3. Ambil semua item 1 (berat 10, sisa kapasitas 40, nilai sejauh ini 60). Ambil semua item 2 (berat 20, sisa kapasitas 20, nilai sejauh ini 160). Hanya 20 dari 30 berat item 3 yang muat — ambil fraksi 20/30 = 2/3, menyumbang 120 × 2/3 = 80. Total nilai: 60 + 100 + 80 = 240, memakai penuh kapasitas 50 — persis cocok dengan batang pertama Gambar 9.5.

Mengapa fraksi yang memungkinkan-lah yang membuat greedy aman di sini

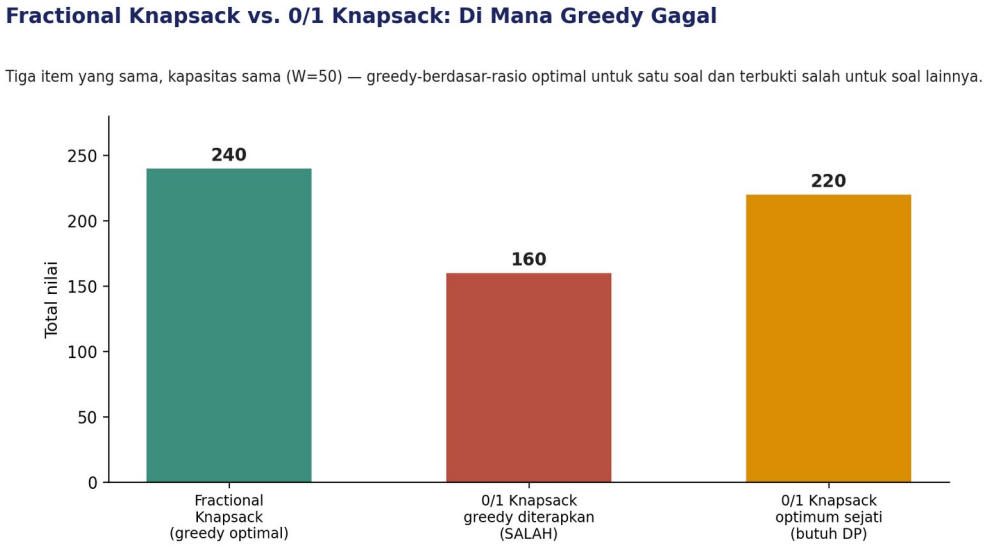
Karena fraksi APA PUN dari item APA PUN bisa diambil, tak pernah ada skenario di mana mengambil rasio terbaik yang tersedia lebih dulu "memboroskan" kapasitas yang bisa dipakai lebih cerdik oleh pilihan pertama yang berbeda — sisa kapasitas apa pun selalu bisa diisi dengan fraksi dari apa pun yang diambil berikutnya, pada rasio item berikutnya itu sendiri. Fleksibilitas ini persis yang dihilangkan Bagian 9.8, dan persis mengapa menghilangkannya merusak sifat greedy-choice.

9.8 0/1 Knapsack: Di Mana Aturan Greedy yang Sama Gagal

Soal 0/1 Knapsack

Setup yang sama dengan Fractional Knapsack — n item dengan nilai dan berat, kapasitas W — kecuali setiap item harus diambil SELURUHNYA atau tidak sama sekali; tak ada fraksi yang diperbolehkan.

Menerapkan persis aturan greedy-berdasar-rasio yang sama pada tiga item YANG SAMA dari Bagian 9.7, tapi kini melarang fraksi: ambil item 1 (berat 10, sisa kapasitas 40, nilai 60). Ambil item 2 (berat 20, sisa kapasitas 20, nilai 160). Item 3 butuh 30 satuan berat, tapi hanya 20 tersisa — karena fraksi dilarang, item 3 harus dilewati SELURUHNYA. Jawaban akhir greedy-berdasar-rasio: 160.



Gambar 9.5 — Tiga item yang sama, kapasitas yang sama, dua soal berbeda. Greedy-berdasar-rasio mencapai optimum sejati (240) untuk Fractional Knapsack, tapi hanya 160 ketika aturan yang sama (secara keliru) diterapkan pada 0/1 Knapsack — kurang penuh 60 dari optimum sejati soal itu, 220 (item 2 dan 3 bersama, tanpa item 1 sama sekali).

160 terlihat seperti jawaban yang sepenuhnya masuk akal — itulah bahayanya

Tak ada apa pun dari eksekusi greedy-berdasar-rasio pada 0/1 Knapsack yang menandakan kegagalan: ia berjalan hingga selesai, memakai langkah-langkah valid, dan mengembalikan angka spesifik yang percaya diri. Hanya dengan memeriksa terhadap optimum SEJATI (220, dicapai dengan mengambil item 2 dan 3 bersama dan tanpa item 1 sama sekali — diverifikasi di Lampiran 9.A lewat pencarian brute-force menyeluruh atas semua $2^3 = 8$ subset yang mungkin dari tiga item) terungkap celah nilai sebesar 60. 0/1 Knapsack memang memiliki substruktur optimal, tapi TIDAK memiliki sifat greedy-choice — algoritma yang benar untuknya adalah program dinamis (tabel berindeks item dan sisa kapasitas), soal yang teknik Bab 7 pada mata kuliah ini berlaku langsung tapi teknik greedy bab ini terbukti tak bisa menyelesaikannya dengan benar.

9.9 Menganalisis Algoritma Greedy: Ke Mana Sebenarnya Waktu Dihabiskan

Setiap algoritma di bab ini berbagi bentuk yang sama, cukup berbeda dari algoritma program-dinamis Bab 7: hampir seluruh waktu eksekusi dihabiskan pada satu PENGURUTAN, dengan langkah greedy itu sendiri hanya memakan waktu linear sesudahnya.

Algoritma	Langkah pengurutan	Langkah greedy	Total waktu
Activity Selection	Θ(n log n) — urutkan berdasar waktu selesai	Θ(n)	Θ(n log n)
Huffman Coding	Θ(n log n) — n operasi EXTRACT-MIN/INSERT pada heap	—	Θ(n log n)
Fractional Knapsack	Θ(n log n) — urutkan berdasar rasio nilai/berat	Θ(n)	Θ(n log n)

Kontras yang berguna dengan Bab 7

Algoritma program-dinamis Bab 7 menghabiskan waktu mengisi sebuah TABEL berukuran proporsional dengan jumlah subpersoalan berbeda (Θ(n^2) untuk Rod-Cutting, Θ(mn) untuk LCS). Algoritma greedy bab ini menghabiskan waktu pada satu PENGURUTAN (atau, ekuivalen, rangkaian operasi priority-queue — EXTRACT-MIN berulang Huffman persis pengurutan yang menyamar), karena sifat greedy-choice menghilangkan kebutuhan untuk pernah meninjau ulang suatu pilihan yang sudah dibuat. Ketika suatu soal bisa diselesaikan secara greedy sama sekali, algoritma yang dihasilkan biasanya lebih sederhana diimplementasikan DAN secara asimtotik tidak lebih buruk dari alternatif program-dinamis — seluruh kesulitannya ada pada MEMBUKTIKAN sifat greedy-choice berlaku sejak awal.

9.10 Algoritma Greedy dalam Praktik: Di Mana Bab Ini Muncul di Produk Nyata

Tentang kotak ini

Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) asal Indonesia yang nyata dibangun oleh penulis, tanpa membocorkan detail implementasi proprietary.

Profitina secara berkala perlu menjadwalkan sekumpulan pekerja penilai (scoring worker) terbatas terhadap antrean pekerjaan evaluasi-prompt yang tertunda, masing-masing dengan waktu siap paling awal dan tenggat — secara struktural mirip interval mulai/selesai Activity Selection, dan dijadwalkan dengan cara yang sama: urutkan berdasarkan tenggat, secara greedy tugaskan pekerjaan kompatibel berikutnya ke pekerja yang bebas. Berkas laporan yang diekspor Profitina (ringkasan visibilitas per-klien, dikirim sebagai lampiran terkompresi) juga mendapat manfaat persis dari wawasan Bagian 9.5: field teks yang sangat sering berulang di suatu laporan (nama kompetitor umum, templat prompt umum) diberi kode internal lebih pendek dibanding yang jarang, gagasan berbasis-frekuensi yang

sama yang diformalkan Huffman coding, bahkan ketika kompresi byte-di-jaringan sesungguhnya ditangani pustaka serbaguna alih-alih pohon Huffman buatan tangan. Dan ketika Profitina harus memuat sekumpulan batch panggilan API penyedia-AI ke dalam anggaran rate-limit per-menit yang tetap, bentuk alokasi-sumber-daya yang mendasarinya mirip Knapsack — mengenali variasi knapsack mana yang sebenarnya berlaku (bisakah biaya satu panggilan dipecah secara fraksional lintas dua menit, atau harus selesai secara atomik dalam satu menit?) menentukan apakah heuristik greedy yang cepat terbukti benar atau sekadar aproksimasi yang terlihat masuk akal, persis perbedaan yang diajarkan Bagian 9.7–9.8.

9.11 Rangkuman Bab dan Poin Kunci

- Algoritma greedy membuat satu pilihan optimal lokal di tiap langkah dan tak pernah meninjaunya ulang — benar hanya jika soal terbukti memiliki KEDUANYA, sifat greedy-choice dan substruktur optimal.
- Activity Selection: urutkan berdasarkan waktu selesai, secara greedy ambil setiap aktivitas kompatibel dalam urutan itu. Terbukti benar lewat argumen pertukaran — aktivitas selesai-paling-awal selalu bisa ditukar masuk ke solusi optimal apa pun tanpa biaya.
- Huffman Coding: berulang kali gabungkan dua simpul berfrekuensi terendah menjadi sebuah pohon; kode akar-ke-daun yang dihasilkan prefix-free dan meminimumkan total panjang enkode — karakter yang sering berakhir dengan kode pendek, yang jarang dengan kode panjang.
- Fractional Knapsack: urutkan berdasarkan rasio nilai-terhadap-berat, ambil sebanyak mungkin item berasio terbaik, lalu berikutnya, dan seterusnya — optimal persis karena jumlah fraksional menghilangkan skenario di mana urutan berbeda bisa memakai kapasitas lebih cerdik.
- 0/1 Knapsack: aturan greedy-berdasar-rasio YANG SAMA terbukti SALAH begitu item tak bisa dipecah — ilustrasi penting bahwa substruktur optimal saja tak menyiratkan sifat greedy-choice, dan bahwa program dinamis (Bab 7), bukan greedy, adalah alat yang benar di sini.
- Setiap algoritma greedy di bab ini didominasi oleh satu pengurutan Θ(n log n) (atau operasi priority-queue setara), dengan langkah greedy sesungguhnya sesudahnya hanya memakan waktu linear — kontras tajam dengan waktu eksekusi pengisian-tabel program dinamis.

"Algoritma greedy tak pernah menoleh ke belakang — itu bisa jadi kekuatan terbesarnya, atau alasan ia baru saja salah menjawab."

— aforisme mata kuliah ini

(Argumen pertukaran (exchange argument) yang memberi tahu Anda mana yang benar, sebelum Anda tahu dengan cara yang sulit.)

Kuliah 10 melanjutkan langsung dari wawasan terakhir bab ini: beberapa algoritma graf terpenting — algoritma Kruskal dan Prim untuk Minimum Spanning Tree, dibahas di Kuliah 10 dan 11 — ADALAH algoritma greedy, masing-masing terbukti benar lewat argumen pertukarannya sendiri, menerapkan persis perkakas yang dikembangkan Bagian 9.2 dan 9.4 ke ranah soal yang sama sekali baru.

9.12 Pertanyaan Ulasan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Menggunakan data aktivitas s = [1, 3, 0, 5, 3, 5, 6, 8, 8, 2, 12], f = [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] dari Gambar 9.3, tunjukkan bahwa mengurutkan berdasarkan waktu MULAI alih-alih waktu selesai dan menerapkan aturan greedy yang sama bisa memilih kurang dari 4 aktivitas. (Petunjuk: coba mulai dengan a3, yang punya waktu mulai paling awal.)
2. Bangun pohon Huffman dengan tangan untuk empat karakter dengan frekuensi w=10, x=15, y=30, z=45 (dari 100), dengan gaya Gambar 9.4, nyatakan kode tiap karakter, dan hitung total panjang enkode untuk 100 karakter.
3. Jelaskan, dengan kata-kata Anda sendiri dan tanpa melihat kembali Bagian 9.4, mengapa argumen pertukaran untuk Activity Selection bergantung khusus pada a1 memiliki waktu selesai PALING AWAL, bukan (misalnya) waktu mulai PALING AKHIR.
4. Untuk instans Fractional Knapsack di Bagian 9.7 (kapasitas 50, item (60,10), (100,20), (120,30)), tunjukkan bahwa greedy-berdasar-rasio tetap optimal bahkan jika kapasitas diubah menjadi 35 alih-alih 50, dan nyatakan total nilai barunya.
5. Sebuah algoritma greedy untuk soal optimisasi BARU diusulkan, dan kebetulan menghasilkan jawaban benar pada setiap contoh yang dicoba pengusulnya. Apakah ini cukup untuk menyimpulkan sifat greedy-choice berlaku? Justifikasi jawaban Anda memakai Bagian 9.2 dan contoh-tandingan 0/1 Knapsack di Bagian 9.8.
6. Jelaskan mengapa waktu eksekusi Huffman coding adalah Θ(n log n) padahal pohon yang dibangunnya hanya punya n daun — dari mana sebenarnya faktor log n itu berasal?

Lampiran 9.A — Log Verifikasi untuk Source Code Bab 9

Seperti di bab-bab sebelumnya, setiap program yang disertakan buku ini dikompilasi dan dijalankan terhadap contoh yang telah diverifikasi dengan tangan sebelum disertakan. Kedua program di bawah dikompilasi dengan `g++ -O2 -std=c++17 -Wall` (nol peringatan, nol error) dan menghasilkan persis keluaran yang diprediksi di teks di atas. Optimum 0/1 Knapsack yang disebutkan di Bagian 9.8 diverifikasi secara independen lewat pencarian brute-force menyeluruh atas semua $2^3 = 8$ subset dari tiga item tersebut.

```
$ ./01_activity_selection
Selected activities (by original id): a1 a4 a8 a11
Detail:
  a1: [1, 4)
  a4: [5, 7)
  a8: [8, 11)
  a11: [12, 14)
Total activities selected: 4          # persis cocok Gambar 9.3

$ ./02_huffman_coding
Huffman codes:
  a (freq 45): 0  (1 bits)
  b (freq 13): 101  (3 bits)
  c (freq 12): 100  (3 bits)
  d (freq 16): 111  (3 bits)
  e (freq 9): 1101  (4 bits)
  f (freq 5): 1100  (4 bits)
Total encoded length: 224 bits        # persis cocok Bagian 9.6
Fixed-length (3 bits/char): 300 bits
Savings: 76 bits (25.3%)
```

Latihan SPOJ — buku pendamping
Teknik bab ini diuji sungguhan di juri daring SPOJ. Buka Bab 10 buku pendamping “SPOJ Competitive Programming” (folder SPOJ, BI & EN) untuk soal greedy dan argumen-pertukaran, lengkap dengan sketsa bukti — lengkap dengan tips, trik, dan hint agar solusi Anda Accepted, tanpa membocorkan jawaban.

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Bab 15, Algoritma Greedy).

[2] D. A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes." Proceedings of the IRE, 40(9), 1952, 1098–1101. (Makalah asli — Huffman merancang ini sebagai solusi tugas akhir mata kuliah saat menjadi mahasiswa di MIT.)

[3] M. A. Weiss. Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2014. (Bab 10, Algoritma Greedy.)

[4] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, 4th ed. Lulu Press, 2020–2023.

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 10

Graf dan Minimum Spanning Tree: Algoritma Kruskal

Algoritma graf pertama dalam mata kuliah ini — dan domain kedua yang sama sekali berbeda di mana paradigma greedy dari Bab 9 ternyata menjadi alat yang tepat, setelah kebenarannya dibuktikan dengan argumen baru: sifat cut (cut property).

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Merepresentasikan graf menggunakan adjacency list dan adjacency matrix, serta menyatakan trade-off ruang/waktu di antara keduanya. (2) Menyatakan masalah Minimum Spanning Tree (MST) secara presisi, dan menjelaskan mengapa spanning tree memiliki tepat |V| − 1 sisi. (3) Menyatakan dan menerapkan SIFAT CUT (CUT PROPERTY) — fakta struktural yang membuat algoritma MST greedy terbukti benar. (4) Menelusuri, mengimplementasikan, dan menganalisis algoritma Kruskal, termasuk penggunaan struktur data union-find (disjoint-set) untuk mendeteksi siklus dalam waktu O(α(n)) teramortisasi per operasi. (5) Membuktikan algoritma Kruskal benar menggunakan sifat cut, mengikuti gaya argumen-pertukaran yang sama yang diperkenalkan di Bab 9. (6) Menganalisis total waktu jalan algoritma Kruskal dan mengidentifikasi langkah mana yang mendominasinya.

10.1 Rekap: Dari Algoritma Greedy ke Graf

Bab 9 memperkenalkan paradigma greedy melalui tiga masalah — Activity Selection, Huffman Coding, Fractional Knapsack — masing-masing berasal dari sudut komputasi yang berbeda, disatukan hanya oleh dua sifat struktural: sifat greedy-choice dan optimal substructure. Bab ini dan bab berikutnya menerapkan paradigma yang sama ke domain yang sama sekali baru: GRAF, struktur simpul dan sisi yang memodelkan jaringan — peta jalan, jaringan komputer, koneksi sosial, rantai ketergantungan, dan (seperti diilustrasikan contoh kerja bab ini) jenis penyebaran multi-region yang suatu saat bisa dijalankan platform seperti Profitina. Kuliah 10 membahas representasi graf dan algoritma Kruskal untuk masalah Minimum Spanning Tree; Kuliah 11 membahas algoritma MST kedua, Prim, dibangun di atas struktur data berbeda namun bertumpu pada argumen kebenaran yang persis sama.

Gambar 10.1 — Posisi bab ini dalam peta jalan enam belas kuliah DAA. Kuliah 10 dan 11 bersama-sama membahas graf dan Minimum Spanning Tree — topik pertama dalam mata kuliah ini yang dibangun di atas algoritma greedy yang diterapkan ke jenis struktur yang benar-benar baru.

Algoritma Kruskal dan Prim ADALAH algoritma greedy

Bab 9 ditutup dengan forward-reference persis ke fakta ini: kedua algoritma bab ini membuat serangkaian pilihan terbaik secara lokal — selalu sisi termurah yang tersedia yang tidak menciptakan masalah — dan tidak pernah meninjau ulang pilihan yang telah dibuat. Yang membuatnya benar bukanlah paradigma baru, melainkan TEKNIK PEMBUKTIAN baru (sifat cut, Bagian 10.3) yang diterapkan pada struktur masalah baru (graf) — persyaratan sifat-greedy-choice-plus-optimal-substructure yang sama dari Bab 9 tetap menentukan apakah pendekatan ini valid sama sekali.

10.2 Dasar-Dasar Graf: Simpul, Sisi, dan Representasi

Graf, simpul, sisi

Sebuah graf G = (V, E) terdiri dari himpunan SIMPUL V (juga disebut node) dan himpunan SISI E, di mana setiap sisi menghubungkan dua simpul. Bab ini bekerja dengan graf TAK BERARAH dan BERBOBOT: setiap sisi {u, v} dapat dilalui ke arah mana pun, dan membawa BOBOT numerik (biaya, jarak, atau kapasitas, tergantung apa yang dimodelkan graf tersebut).

Graf dapat direpresentasikan dengan dua cara standar. ADJACENCY LIST menyimpan, untuk setiap simpul, daftar tetangganya (dan bobot sisi penghubungnya) — menggunakan ruang $\Theta(V + E)$, dan memungkinkan enumerasi tetangga suatu simpul dalam waktu sebanding dengan derajatnya. ADJACENCY MATRIX menyimpan grid $|V| \times |V|$ di mana entri (u, v) menyimpan bobot sisi {u, v} (atau ∞/0 jika tidak ada sisi tersebut) — menggunakan ruang $\Theta(V^2)$ terlepas dari berapa banyak sisi yang benar-benar ada, tetapi menjawab "apakah ada sisi antara u dan v?" dalam waktu O(1). Untuk graf jarang (sparse) yang umum pada jaringan nyata (di mana $|E|$ jauh lebih kecil dari $|V|^2$), adjacency list hampir selalu menjadi pilihan yang lebih baik, dan itulah yang dipakai algoritma-algoritma bab ini.

Representasi	Ruang	"Apakah {u,v} sisi?"	Enumerasi tetangga v
Adjacency list	$\Theta(V + E)$	O(deg(v)) — scan daftar v	$\Theta(\deg(v))$
Adjacency matrix	$\Theta(V^2)$	O(1) — lookup langsung	$\Theta(V)$ — scan satu baris penuh

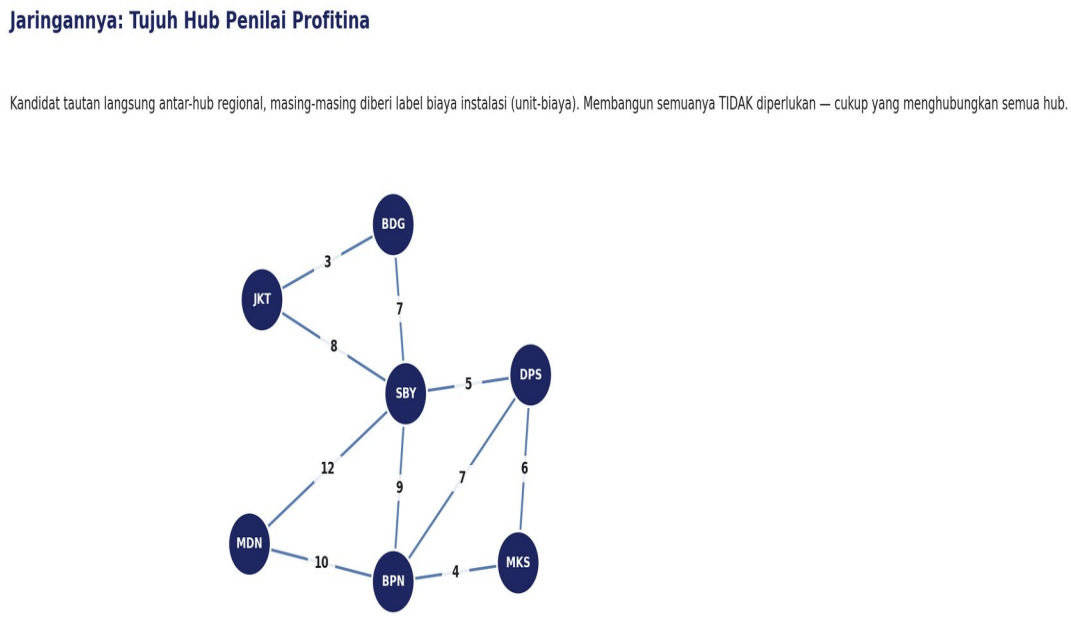
10.3 Masalah Minimum Spanning Tree dan Sifat Cut

Spanning tree, Minimum Spanning Tree

Diberikan graf terhubung, tak berarah, berbobot G = (V, E), sebuah SPANNING TREE adalah subset tepat $|V| - 1$ sisi yang menghubungkan semua $|V|$ simpul tanpa siklus. MINIMUM SPANNING TREE (MST) adalah spanning tree dengan total bobot sisi sekecil mungkin di antara semua spanning tree G.

Mengapa tepat $|V| - 1$ sisi? Pohon pada n simpul memiliki tepat $n - 1$ sisi — satu sisi lebih sedikit dari jumlah simpul, karena menambahkan sisi ke-n mana pun ke pohon $(n - 1)$-sisi pasti akan menghubungkan dua simpul yang sudah terhubung (menciptakan siklus) atau pohon tersebut sudah

mencakup semua simpul dengan sisi lebih sedikit (kontradiksi untuk struktur terhubung). Contoh kerja bab ini — tujuh hub penilai regional Profitina dan sepuluh kandidat tautan langsung di antaranya, masing-masing dengan biaya instalasi — membutuhkan tepat 7 − 1 = 6 tautan yang diterima untuk menghubungkan setiap hub dengan total biaya minimum.



Gambar 10.2 — Jaringan kandidat penuh: tujuh hub, sepuluh kemungkinan tautan langsung, masing-masing diberi label biaya instalasinya. Membangun ke sepuluhnya tidak diperlukan — masalah MST meminta SUBSET termurah yang tetap menghubungkan setiap hub.

Sifat cut (cut property)
Untuk cara apa pun mempartisi simpul-simpul graf terhubung menjadi dua kelompok tak-kosong (sebuah "cut"), sisi TERINGAN yang melintasi cut tersebut — sisi termurah dengan satu titik ujung di setiap kelompok — termasuk dalam SUATU minimum spanning tree dari graf tersebut. Inilah fakta struktural yang membuat algoritma MST greedy terbukti benar: di setiap langkah, cut spesifik apa pun yang sedang dipertimbangkan suatu algoritma, sisi termurah yang melintasinya selalu merupakan pilihan yang aman.

10.4 Algoritma Kruskal

Algoritma Kruskal menerapkan sifat cut dengan cara paling langsung: urutkan SEMUA sisi berdasarkan bobot, dan proses dari yang termurah, menerima suatu sisi jika dan hanya jika kedua titik ujungnya belum terhubung oleh sisi-sisi yang sudah diterima sebelumnya (menerimanya akan menciptakan siklus, yang tidak boleh ada dalam pohon). Mendeteksi "apakah kedua simpul ini sudah terhubung oleh sisi yang diterima?" secara efisien adalah persis fungsi struktur data UNION-FIND (disjoint-set) pada Bagian 10.6.

```
KRUSKAL(V, E, w):                    // V = simpul, E = sisi, w = fungsi bobot
  A = { }                             // sisi yang diterima, akan berukuran |V| - 1
  for each vertex v in V
      MAKE-SET(v)                     // tiap simpul mulai di komponennya sendiri
  sort E into non-decreasing order by weight w
  for each edge (u, v) in sorted E
      if FIND-SET(u) != FIND-SET(v)  // u dan v di komponen berbeda?
          A = A union { (u, v) }
          UNION(u, v)                 // gabungkan kedua komponen
  return A
```

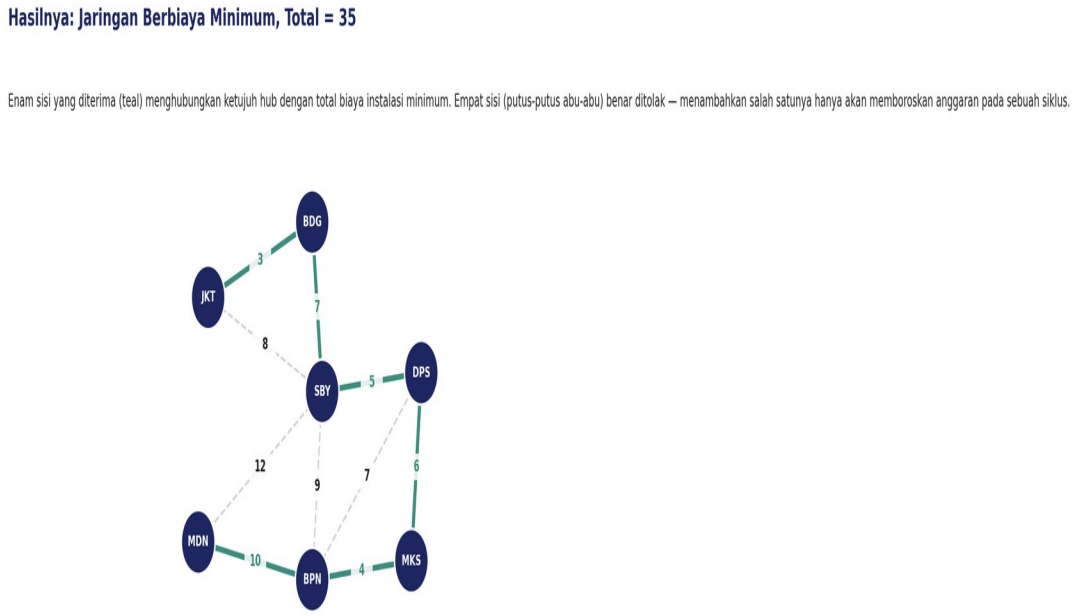
Jejak Kruskal: Mengurutkan Sisi Berdasarkan Bobot

Proses sisi dari yang termurah. Terima suatu sisi hanya jika kedua titik ujungnya masih berada di komponen berbeda (union-find) — tolak jika tidak, karena hanya akan menciptakan siklus.

w=3	JKT–BDG	TERIMA
w=4	BPN–MKS	TERIMA
w=5	SBY–DPS	TERIMA
w=6	MKS–DPS	TERIMA
w=7	BDG–SBY	TERIMA
w=7	BPN–DPS	TOLAK (siklus)
w=8	JKT–SBY	TOLAK (siklus)
w=9	SBY–BPN	TOLAK (siklus)
w=10	MDN–BPN	TERIMA
w=12	SBY–MDN	TOLAK (siklus)

Gambar 10.3 — Jejak Kruskal pada sepuluh kandidat tautan dari Gambar 10.2, diproses dari yang termurah. Enam sisi diterima (masing-masing menggabungkan dua komponen yang sebelumnya terpisah); empat ditolak karena titik-titik ujungnya sudah terhubung oleh sisi yang diterima sebelumnya.

Menelusuri data Gambar 10.3: JKT–BDG (w=3) diterima — sisi pertama selalu diterima, karena setiap simpul mulai di komponennya sendiri. BPN–MKS (w=4), SBY–DPS (w=5), dan MKS–DPS (w=6) masing-masing diterima secara berurutan, masing-masing menggabungkan dua komponen yang masih terpisah. BDG–SBY (w=7) diterima, menggabungkan komponen {JKT, BDG} dengan komponen {SBY, DPS, MKS, BPN} yang sedang membesar. Sisi BERIKUTNYA berdasarkan bobot, BPN–DPS (w=7), DITOLAK — kedua titik ujungnya sudah berada di komponen yang sama (penggabungan di atas menghubungkannya secara transitif) — menerimanya akan menciptakan siklus. JKT–SBY (w=8) dan SBY–BPN (w=9) ditolak dengan alasan yang sama. MDN–BPN (w=10) diterima, akhirnya menghubungkan MDN yang sebelumnya terisolasi. SBY–MDN (w=12), sisi termahal dan terakhir, ditolak — MDN sudah terhubung oleh sisi MDN–BPN yang lebih murah. Enam sisi diterima, cocok dengan $|V| - 1 = 7 - 1 = 6$ persis, untuk total biaya instalasi 35 — cocok persis dengan keluaran terkompilasi Lampiran 10.A.



Gambar 10.4 — Minimum spanning tree yang sudah jadi: enam sisi yang diterima (teal) menghubungkan ketujuh hub dengan total biaya 35; empat sisi yang ditolak (putus-putus abu-abu) masing-masing hanya akan menambahkan siklus yang berlebihan.

10.5 Membuktikan Kruskal Benar melalui Sifat Cut

Kebenaran Kruskal mengikuti dari sifat cut melalui induksi pada jumlah sisi yang sudah diterima sejauh ini. Misalkan sisi-sisi yang diterima sejauh ini membentuk subset dari SUATU minimum spanning tree (trivial benar ketika nol sisi telah diterima). Pertimbangkan sisi berikutnya (u, v) yang akan diterima Kruskal, karena FIND-SET(u) ≠ FIND-SET(v). Misalkan S adalah himpunan simpul dalam komponen u saat ini, dan V − S adalah semua yang lain — ini adalah cut yang valid, karena kedua sisi tidak kosong (v berada di sisi lain, menurut asumsi). Karena Kruskal memproses sisi dalam urutan bobot menaik dan belum menerima sisi apa pun yang melintasi cut khusus ini, (u, v) ADALAH sisi teringan yang melintasinya (sisi lintas yang lebih ringan mana pun sudah akan dipertimbangkan dan, oleh pemeriksaan penghindaran-siklus, sudah akan diterima terlebih dahulu). Berdasarkan sifat cut, (u, v) termasuk dalam suatu minimum spanning tree — sehingga menerimanya menjaga invarian induktif. Karena algoritma berhenti dengan tepat |V| − 1 sisi yang diterima membentuk struktur terhubung tanpa siklus, dan setiap sisi tersebut terbukti aman untuk ditambahkan, hasil akhirnya adalah minimum spanning tree yang sesungguhnya.

> **Paralel dengan argumen pertukaran Bab 9**
> Bukti ini memiliki bentuk yang persis sama dengan argumen pertukaran Bagian 9.4 untuk Activity Selection: tunjukkan bahwa pilihan greedy selalu aman (termasuk dalam SUATU solusi optimal), lalu tunjukkan bahwa yang tersisa setelah pilihan greedy adalah masalah YANG SAMA pada input yang lebih kecil (optimal substructure), dan gabungkan keduanya melalui induksi. Sifat cut memainkan peran yang dimainkan argumen waktu-selesai-paling-awal di Bab 9 — fakta spesifik-masalah yang membuat pilihan greedy terbukti aman.

10.6 Struktur Data Union-Find (Disjoint-Set)

Algoritma Kruskal membutuhkan satu operasi, diulang |E| kali: "apakah kedua simpul ini berada di komponen yang sama?", dan satu pembaruan, diulang |V| − 1 kali: "gabungkan kedua komponen ini." Struktur data UNION-FIND mendukung keduanya secara efisien. Setiap komponen direpresentasikan sebagai pohon elemen yang menunjuk ke akar bersama; FIND-SET(x) berjalan dari x ke akarnya (mengidentifikasi komponen mana x berada), dan UNION(x, y) melampirkan satu akar di bawah akar lainnya. Dua optimisasi standar membuat ini cepat: PATH COMPRESSION (selama FIND-SET, arahkan setiap node yang dikunjungi langsung ke akar, meratakan pohon untuk kueri di masa depan) dan UNION BY RANK (selalu lampirkan pohon yang lebih dangkal di bawah akar pohon yang lebih dalam, menjaga pohon tetap rata sejak awal). Bersama-sama, kedua optimisasi ini menurunkan biaya teramortisasi setiap operasi menjadi O(α(n)) — di mana α adalah fungsi invers Ackermann, kuantitas yang tumbuh begitu lambat sehingga nilainya kurang dari 5 untuk ukuran input mana pun yang dapat dikonstruksi dalam praktik, membuat operasi union-find secara efektif berwaktu konstan.

Fase 5 — Implementasi: union-find dengan path compression dan union by rank

```
MAKE-SET(x):  parent[x] = x;  rank[x] = 0

FIND-SET(x):
  if parent[x] != x
      parent[x] = FIND-SET(parent[x])   // path compression
  return parent[x]

UNION(x, y):
  rx = FIND-SET(x);  ry = FIND-SET(y)
  if rx == ry:  return                  // sudah di komponen yang sama
  if rank[rx] < rank[ry]:  swap(rx, ry)  // union by rank
  parent[ry] = rx
  if rank[rx] == rank[ry]:  rank[rx] = rank[rx] + 1
```

10.7 Menganalisis Waktu Jalan Kruskal

Mengurutkan |E| sisi membutuhkan waktu Θ(E log E). Loop utama kemudian berjalan |E| kali, dan setiap iterasi melakukan sejumlah konstan operasi union-find, masing-masing berbiaya O(α(V)) teramortisasi — secara efektif O(1). Karena graf sederhana memiliki $E = O(V^2)$, $\log E = O(2 \log V) = O(\log V)$, sehingga total waktu jalan adalah $\Theta(E \log E) = \Theta(E \log V)$, yang DIDOMINASI sepenuhnya oleh pengurutan awal — persis pola yang sama yang diidentifikasi Bagian 9.9 di seluruh algoritma greedy Bab 9.

Langkah	Biaya	Catatan
Urutkan |E| sisi berdasarkan bobot	Θ(E log E) = Θ(E log V)	Mendominasi total waktu jalan
|V| panggilan MAKE-SET	Θ(V)	Satu per simpul, sebelum loop utama
|E| panggilan FIND-SET / UNION	O(E · α(V)) ≈ O(E)	Teramortisasi hampir konstan per panggilan
Total	Θ(E log V)	Bentuk sama seperti setiap algoritma

Langkah	Biaya	Catatan
		Bab 9

10.8 Algoritma Graf dalam Praktik: Di Mana Bab Ini Muncul dalam Produk Nyata

> **Tentang kotak ini**
> Seperti di setiap bab, kotak ini menghubungkan materi mata kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) Indonesia nyata yang dibangun oleh penulis, tanpa mengungkapkan detail implementasi yang bersifat proprietary.

Profitina saat ini berjalan sebagai satu server self-hosted tunggal (FastAPI + PostgreSQL, di belakang Cloudflare Tunnel) — belum menjadi penyebaran terdistribusi multi-hub seperti yang digambarkan contoh kerja bab ini. Namun roadmap publik Profitina sendiri memang menyebut ekspansi semacam ini sebagai tahap berikutnya (Fase 4: skala regional, setelah dominasi domestik di Indonesia tercapai), sehingga skenario di bawah ini adalah pratinjau keputusan nyata di masa depan, bukan skenario yang dikarang tanpa dasar. Jika/saat ekspansi itu terjadi, memutuskan tautan langsung mana antar-hub yang benar-benar akan disewa (masing-masing dengan biaya bulanan nyata) untuk menjamin setiap hub dapat menjangkau setiap hub lain, dengan total biaya tautan sewa minimum, akan menjadi instans literal dari masalah Minimum Spanning Tree, dan algoritma Kruskal (atau algoritma Prim pada Kuliah 11) adalah alat yang sepenuhnya realistis untuk keputusan spesifik tersebut — meskipun jaringan produksi akan menambahkan batasan tambahan (persyaratan redundansi, batas latensi) di atas MST dasar yang sendirian tidak ditangkap oleh algoritma bab ini. Secara lebih abstrak, kapan pun platform seperti Profitina perlu mengidentifikasi cara termurah menjaga sekumpulan sumber daya (server, replika data, probe pemantauan) tetap saling terjangkau tanpa koneksi berlebihan yang memboroskan anggaran, bentuk dasar dari keputusan tersebut persis adalah masalah bab ini.

10.9 Ringkasan Bab dan Poin-Poin Kunci

- Graf G = (V, E) dapat direpresentasikan sebagai adjacency list (ruang $\Theta(V+E)$, enumerasi tetangga efisien) atau adjacency matrix (ruang $\Theta(V^2)$, lookup sisi $O(1)$) — adjacency list mendominasi untuk graf jarang yang umum pada jaringan nyata.
- Spanning tree menghubungkan semua $|V|$ simpul menggunakan tepat $|V| - 1$ sisi tanpa siklus; Minimum Spanning Tree (MST) adalah spanning tree dengan total bobot paling kecil.
- Sifat cut — sisi teringan yang melintasi cut mana pun termasuk dalam suatu MST — adalah fakta struktural yang membuat algoritma MST greedy terbukti benar, memainkan peran yang sama dengan argumen pertukaran Bab 9 untuk Activity Selection dan Huffman Coding.
- Algoritma Kruskal mengurutkan semua sisi berdasarkan bobot dan secara greedy menerima setiap sisi yang tidak menciptakan siklus, menggunakan struktur data union-find (dengan path compression dan union by rank) untuk menguji hal itu dalam waktu $O(1)$ teramortisasi secara efektif per operasi.

- Kebenaran Kruskal mengikuti dari sifat cut melalui induksi: di setiap langkah, sisi yang akan diterima terbukti sebagai sisi teringan yang melintasi suatu cut yang valid, sehingga aman berdasarkan sifat cut.
- Total waktu jalan Kruskal, Θ(E log V), didominasi oleh pengurutan awal sisi — bentuk "pengurutan mendominasi" yang sama yang ditemukan di seluruh algoritma greedy Bab 9.

"Spanning tree menyentuh setiap simpul; MINIMUM spanning tree melakukannya dengan biaya paling sedikit — dan greedy, secara mengagumkan, selalu menemukannya."

— aforisme mata kuliah ini

(Sifat cut (cut property) yang membuat fakta mengagumkan itu terbukti, bukan cuma teramati.)

Kuliah 11 melanjutkan langsung dari bab ini: algoritma MST kedua, Prim, membangun JENIS pohon yang SAMA melalui mekanisme berbeda (menumbuhkan satu komponen dari simpul awal, menggunakan min-priority-queue pada nilai key, bukan menggabungkan banyak komponen melalui union-find) — tetapi bertumpu pada sifat cut yang persis sama untuk membuktikannya benar, dan mencapai total bobot yang persis sama pada graf mana pun, karena Minimum Spanning Tree dari graf dengan semua bobot sisi berbeda adalah unik terlepas dari algoritma mana yang dipakai untuk menemukannya.

10.10 Pertanyaan Tinjauan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Menggunakan sepuluh kandidat tautan dari Gambar 10.2, tunjukkan apa yang terjadi jika algoritma Kruskal dijalankan TANPA union-find — yaitu, dengan memeriksa konektivitas melalui traversal graf penuh (BFS/DFS) setelah setiap kandidat sisi. Apakah MST akhirnya berbeda? Apakah waktu jalannya?
2. Buktikan, hanya menggunakan definisi pohon (terhubung, tanpa siklus), bahwa spanning tree mana pun pada |V| simpul memiliki tepat |V| − 1 sisi — jangan hanya mengutip fakta dari Bagian 10.3, turunkan sendiri.
3. Misalkan dua sisi dalam graf memiliki bobot SAMA, dan keduanya melintasi cut yang sama. Apakah sifat cut menjamin minimum spanning tree yang UNIK dalam kasus itu? Jelaskan, merujuk pada keterikatan antara BDG–SBY (w=7) dan BPN–DPS (w=7) dalam jejak Gambar 10.3.
4. Telusuri union-by-rank dan path compression secara manual untuk urutan MAKE-SET(a), MAKE-SET(b), MAKE-SET(c), MAKE-SET(d), UNION(a,b), UNION(c,d), UNION(a,c), FIND-SET(d) — gambar pohon yang dihasilkan setelah setiap langkah.

5. Jelaskan mengapa algoritma Kruskal, tidak seperti Activity Selection, membutuhkan struktur data bantu eksplisit (union-find) untuk menentukan apakah pilihan greedy-nya aman, bukan hanya satu variabel berjalan seperti last_finish.
6. Jika sebuah kandidat sisi baru ditambahkan ke jaringan Gambar 10.2 dengan bobot 1 antara MDN dan JKT, jalankan ulang algoritma Kruskal secara manual dan nyatakan MST baru serta total bobotnya.

Lampiran 10.A — Log Verifikasi untuk Kode Sumber Bab 10

Seperti pada bab-bab sebelumnya, setiap program yang disertakan dalam buku ini dikompilasi dan dijalankan terhadap contoh yang telah diverifikasi secara manual sebelum disertakan. Kedua program di bawah ini dikompilasi dengan `g++ -O2 -std=c++17 -Wall` (nol peringatan, nol kesalahan) dan menghasilkan keluaran persis seperti yang diprediksi dalam teks di atas. Bobot MST (35) juga diverifikasi silang secara independen oleh skrip Python yang melakukan pencarian brute-force menyeluruh atas seluruh C(10,6) = 210 subset enam-sisi dari jaringan kandidat — lihat /tmp/daa_pilot/verify_ch10.py.

```
$ ./01_kruskal_mst
...
Kruskal trace:
  ACCEPT JKT-BDG (w=3)
  ACCEPT BPN-MKS (w=4)
  ACCEPT SBY-DPS (w=5)
  ACCEPT MKS-DPS (w=6)
  ACCEPT BDG-SBY (w=7)
  REJECT BPN-DPS (w=7)  -- cycle
  REJECT JKT-SBY (w=8)  -- cycle
  REJECT SBY-BPN (w=9)  -- cycle
  ACCEPT MDN-BPN (w=10)
  REJECT SBY-MDN (w=12)  -- cycle
MST total weight: 35            # cocok persis dengan Gambar 10.3/10.4

$ ./02_prim_mst
Prim trace (start = JKT):
  START at JKT
  ADD BDG via edge JKT-BDG (w=3)
  ADD SBY via edge BDG-SBY (w=7)
  ADD DPS via edge SBY-DPS (w=5)
  ADD MKS via edge DPS-MKS (w=6)
  ADD BPN via edge MKS-BPN (w=4)
  ADD MDN via edge BPN-MDN (w=10)
MST total weight: 35            # cocok persis dgn Kruskal (detail Kuliah 11)
```

Latihan SPOJ — buku pendamping
Teknik bab ini diuji sungguhan di juri daring SPOJ. Buka Bab 11 buku pendamping “SPOJ Competitive Programming” (folder SPOJ, BI & EN) untuk toolkit graf: representasi, MST, dan penelusuran — lengkap dengan tips, trik, dan hint agar solusi Anda Accepted, tanpa membocorkan jawaban.

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Bab 21, Data Structures for Disjoint Sets; Bab 22, Elementary Graph Algorithms; Bab 23, Minimum Spanning Trees).

[2] J. B. Kruskal Jr. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem." Proceedings of the American Mathematical Society, 7(1), 1956, 48–50. (Makalah asli.)

[3] M. A. Weiss. Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2014. (Bab 9, Graph Algorithms; Bab 8, The Disjoint Set Class.)

[4] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, 4th ed. Lulu Press, 2020–2023.

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 11

Algoritma Prim untuk Minimum Spanning Tree

Algoritma MST kedua — masalah sama, contoh kerja sama, sifat cut sama, tapi mekanisme yang sama sekali berbeda: menumbuhkan satu pohon ke luar dari satu simpul, satu sisi termurah setiap kali, bukan menggabungkan banyak komponen berdasarkan bobot sisi terurut.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Menyatakan perbedaan mekanisme antara algoritma Kruskal (menggabungkan banyak komponen) dan algoritma Prim (menumbuhkan satu pohon), sambil menyadari bahwa keduanya menyelesaikan masalah MST yang identik. (2) Menelusuri algoritma Prim secara manual pada graf berbobot, termasuk array key[]/parent[] yang dijaganya untuk setiap simpul di luar pohon. (3) Membuktikan algoritma Prim benar menggunakan sifat cut, mengadaptasi bukti Bab 10 ke pilihan cut spesifik Prim di setiap langkah. (4) Mengimplementasikan algoritma Prim menggunakan min-priority queue berdasarkan key[], dan menyatakan waktu jalannya di bawah binary heap versus array sederhana. (5) Menjelaskan mengapa algoritma Kruskal dan Prim selalu sepakat pada total bobot MST (dan, ketika semua bobot sisi berbeda, pada set sisi yang persis sama) meskipun mekanismenya berbeda. (6) Memilih antara algoritma Kruskal dan Prim untuk graf tertentu berdasarkan kepadatannya (density).

11.1 Rekap: Dari Kruskal ke Prim

Bab 10 memperkenalkan masalah Minimum Spanning Tree (MST) dan sifat cut — fakta struktural bahwa sisi teringan yang melintasi cut mana pun dari graf terhubung termasuk dalam suatu MST — dan memakainya untuk membuktikan algoritma Kruskal benar. Mekanisme Kruskal adalah mengurutkan semua sisi sekali, lalu berulang kali mengajukan pertanyaan GLOBAL: "apakah sisi termurah berikutnya ini menghubungkan dua komponen berbeda?" Bab ini menyajikan algoritma MST klasik kedua yang sama pentingnya — algoritma Prim — yang menerapkan sifat cut yang persis sama tetapi mengajukan pertanyaan LOKAL yang sama sekali berbeda di setiap langkah: "dari semua sisi yang keluar dari satu pohon yang sudah saya bangun sejauh ini, mana yang termurah?" Kedua algoritma bersifat greedy; keduanya terbukti benar oleh sifat cut; dan pada graf mana pun, keduanya dijamin menemukan spanning tree dengan total bobot minimum yang sama.

Gambar 11.1 — Posisi bab ini dalam peta jalan enam belas kuliah DAA. Kuliah 11 menyelesaikan unit dua-kuliah tentang Graf dan Minimum Spanning Tree yang dimulai di Bab 10.

> **Masalah sama, contoh kerja sama, jawaban sama — mekanisme berbeda**
> Bab ini sengaja memakai ulang jaringan tujuh-hub Profitina yang persis sama dari Bab 10 (Gambar 10.2) alih-alih memperkenalkan jaringan baru. Pilihan ini bersifat pedagogis: menyaksikan dua algoritma berbeda mencapai total biaya 35 yang identik pada input yang identik adalah demonstrasi paling jelas bahwa algoritma MST benar karena sifat struktural bersama (sifat cut), bukan karena pembukuan spesifik satu algoritma tertentu.

11.2 Algoritma Prim: Menumbuhkan Satu Pohon

Algoritma Prim menjaga satu pohon yang terus tumbuh, dimulai dari simpul akar r yang sembarang, dan di setiap langkah menambahkan sisi termurah yang menghubungkan simpul yang SUDAH ada di pohon ke simpul yang belum. Untuk menemukan sisi termurah tersebut secara efisien tanpa memindai ulang seluruh graf di setiap langkah, algoritma Prim menjaga, untuk setiap simpul v yang belum ada di pohon, sebuah nilai key[v] — bobot sisi termurah yang pernah terlihat menghubungkan v ke pohon — dan sebuah nilai parent[v] — simpul pohon mana yang menjadi tujuan sisi termurah tersebut. Nilai-nilai ini disimpan dalam min-priority queue Q berdasarkan key[], sehingga simpul berikutnya yang akan ditambahkan selalu ditemukan dengan satu operasi EXTRACT-MIN.

```
PRIM(V, E, w, r):              // r = simpul awal, pohon tumbuh dari sana
  for each vertex v in V
      key[v] = infinity;  parent[v] = NIL
  key[r] = 0
  Q = V                          // min-priority queue berdasarkan key[]
  while Q is not empty
      u = EXTRACT-MIN(Q)         // simpul termurah yang belum di pohon
      for each v in Adj[u]
          if v in Q and w(u, v) < key[v]
              parent[v] = u
              key[v] = w(u, v)
              DECREASE-KEY(Q, v, key[v])
```

Setiap kali simpul u diekstrak, algoritma merelaksasi setiap sisi u: untuk tetangga v yang masih di Q, jika sisi (u, v) lebih murah dari koneksi terbaik v saat ini ke pohon, key[v] dan parent[v] diperbarui. Ini adalah ide RELAKSASI yang sama yang dipakai di seluruh algoritma lintasan terpendek (Bab 13) — satu-satunya perbedaan adalah key[v] milik Prim melacak bobot SISI termurah ke pohon, bukan JARAK LINTASAN termurah dari sebuah sumber.

11.3 Jejak Prim pada Jaringan Kandidat

Menjalankan algoritma Prim pada jaringan tujuh-hub Gambar 10.2, dimulai dari JKT, menghasilkan jejak pada Gambar 11.2 — diverifikasi secara independen di /tmp/daa_pilot/verify_ch11_prim_trace2.py dan diverifikasi silang terhadap keluaran terkompilasi Lampiran 11.A's 01_prim_mst.cpp sebelum disertakan di sini.

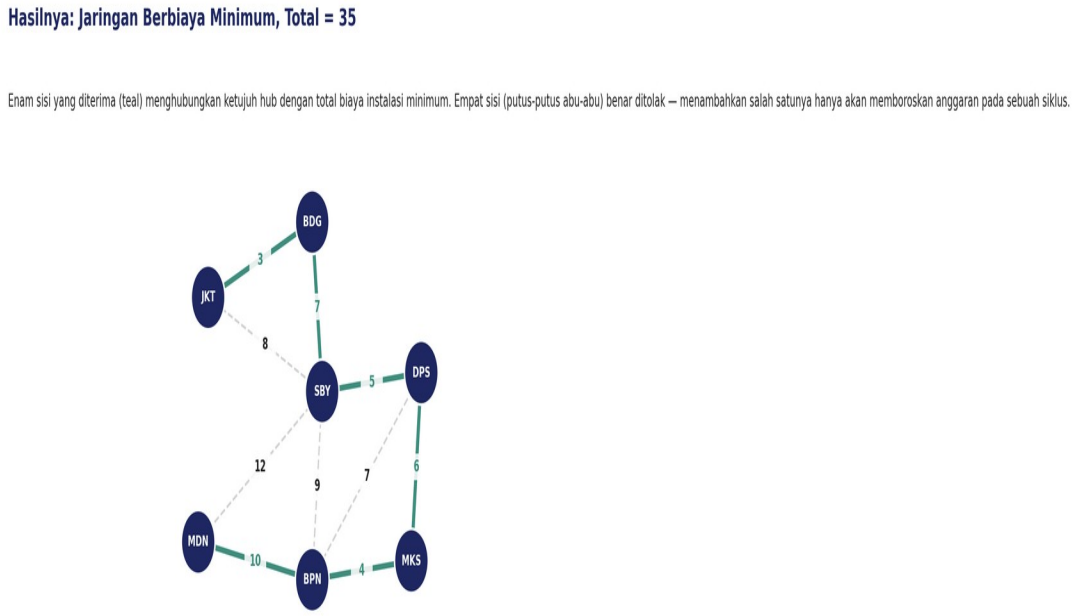
Jejak Prim: Menumbuhkan Satu Tree dari JKT

Berbeda dari Kruskal, Prim tidak pernah menolak sisi — ia menumbuhkan SATU tree satu simpul setiap kali, selalu memilih sisi termurah yang keluar dari tree saat ini. Diverifikasi terhadap /tmp/daa_pilot/code/chapter11/01_prim_mst.cpp.

Langkah	Simpul Ditambah		Sisi Dipakai	Yang Berubah di Frontier
1	MULAI	JKT	—	Mulai — key JKT diset 0, semua lainnya ∞
2	ADD	BDG	JKT–BDG (w=3)	Frontier semula {BDG:3, SBY:8} — BDG termurah
3	ADD	SBY	BDG–SBY (w=7)	BDG merelaksasi key SBY dari 8 turun ke 7
4	ADD	DPS	SBY–DPS (w=5)	SBY merelaksasi DPS:5, BPN:9, MDN:12 — DPS termurah
5	ADD	MKS	DPS–MKS (w=6)	DPS merelaksasi MKS:6, BPN:7→ tetap 7 s/d langkah berikut
6	ADD	BPN	MKS–BPN (w=4)	MKS merelaksasi key BPN dari 7 turun ke 4
7	ADD	MDN	BPN–MDN (w=10)	BPN merelaksasi key MDN dari 12 turun ke 10 — selesai

Gambar 11.2 — Jejak Prim dimulai dari JKT. Setiap baris menunjukkan simpul yang ditambahkan, sisi pohon yang dipakai untuk menambahkannya, dan bagaimana penambahan itu merelaksasi (menurunkan) nilai key[] tetangga-tetangganya yang masih di luar pohon.

Menelusuri data Gambar 11.2: pohon dimulai hanya {JKT}, dengan key[JKT]=0. Mengekstrak JKT merelaksasi BDG (key=3) dan SBY (key=8). BDG memiliki key lebih kecil, sehingga diekstrak berikutnya, melalui sisi JKT–BDG (w=3) — ini merelaksasi key SBY dari 8 turun ke 7, karena BDG–SBY (w=7) kini menjadi koneksi lebih murah dibanding JKT–SBY (w=8). SBY diekstrak berikutnya (key=7, via BDG), merelaksasi DPS (key=5), BPN (key=9), dan MDN (key=12). DPS memiliki key terkecil di antara semua simpul luar, sehingga diekstrak (via SBY–DPS, w=5), yang merelaksasi MKS (key=6) dan menurunkan key BPN dari 9 menjadi 7 (via BPN–DPS, w=7). MKS diekstrak berikutnya (key=6, via DPS), yang merelaksasi key BPN lebih jauh, dari 7 turun ke 4 (via BPN–MKS, w=4) — koneksi termurah yang akan pernah didapat BPN. BPN diekstrak berikutnya (key=4, via MKS), merelaksasi key MDN dari 12 turun ke 10 (via BPN–MDN, w=10). Akhirnya MDN diekstrak (key=10, via BPN), dan antrean kosong. Enam sisi ditambahkan — {JKT-BDG:3, BDG-SBY:7, SBY-DPS:5, DPS-MKS:6, MKS-BPN:4, BPN-MDN:10} — untuk total bobot 35, cocok persis dengan keluaran terkompilasi Lampiran 11.A dan hasil Kruskal Bab 10.



Gambar 11.3 — Minimum spanning tree yang sama yang ditemukan Bab 10 melalui algoritma Kruskal (Gambar 10.4), kini dicapai melalui algoritma Prim. Set sisi dan total biaya (35) identik — hanya URUTAN penemuan keenam sisi yang berbeda.

11.4 Mengapa Prim Mencapai MST yang Sama dengan Kruskal

Gambar 11.3 bukan kebetulan: kapan pun graf terhubung dan berbobot memiliki bobot sisi yang semuanya BERBEDA (seperti jaringan Gambar 10.2 — setiap satu dari sepuluh kandidat tautannya memiliki biaya instalasi yang berbeda), minimum spanning tree-nya bersifat UNIK, terlepas dari algoritma benar mana pun yang dipakai untuk menemukannya. Algoritma Kruskal dan Prim mempertimbangkan sisi dalam urutan yang sama sekali berbeda dan menjaga pembukuan yang sama sekali berbeda (daftar sisi terurut plus union-find, versus satu pohon plus priority queue pada key[]) — tetapi karena keduanya benar oleh sifat cut YANG SAMA, dan MST mendasar yang dicari keduanya adalah objek unik yang sama, mereka tidak bisa tidak sampai pada objek itu. (Ketika beberapa bobot sisi seri/sama, MST tidak harus unik — Pertanyaan Tinjauan 3 mengeksplorasi kasus ini secara langsung.)

> **Keunikan di bawah bobot yang berbeda**
> Jika graf terhubung dan berbobot memiliki semua bobot sisi berbeda, graf tersebut memiliki TEPAT SATU minimum spanning tree. Sketsa mengapa: jika dua spanning tree berbeda T1 dan T2 sama-sama memiliki bobot minimum yang sama, suatu sisi di T1 yang tidak ada di T2 dapat dipertukarkan dengan suatu sisi di T2 yang tidak ada di T1 melintasi cut yang diciptakan pertukaran tersebut — dan karena semua bobot berbeda, salah satu dari kedua pohon hasil tersebut pasti lebih murah secara ketat dibanding minimum yang diasumsikan, sebuah kontradiksi.

11.5 Membuktikan Prim Benar melalui Sifat Cut

Kebenaran Prim mengikuti dari sifat cut melalui induksi pada ukuran pohon yang terus tumbuh, dalam bukti dengan bentuk yang persis sama dengan bukti Kruskal Bab 10 — tetapi diterapkan pada cut YANG BERBEDA di setiap langkah. Misalkan sisi-sisi yang diterima sejauh ini membentuk subset dari suatu minimum spanning tree (trivial benar ketika pohon hanya berisi akar r). Misalkan S adalah himpunan simpul yang saat ini ada di pohon, dan V − S adalah semua yang lain — ini selalu merupakan cut yang valid, karena S tidak pernah kosong (setidaknya berisi r) dan, sampai algoritma selesai, V − S juga tidak pernah kosong. Sisi berikutnya yang akan diterima Prim adalah, berdasarkan konstruksi key[] dan EXTRACT-MIN, sisi TERMURAH dengan satu titik ujung di S dan satu titik ujung di V − S — dengan kata lain, persis sisi teringan yang melintasi cut khusus ini. Berdasarkan sifat cut, sisi tersebut termasuk dalam suatu minimum spanning tree, sehingga menerimanya menjaga invarian induktif. Karena algoritma berhenti dengan S = V dan tepat |V| − 1 sisi yang diterima membentuk struktur terhubung tanpa siklus (setiap simpul baru dilampirkan oleh tepat satu sisi), hasil akhirnya adalah minimum spanning tree yang sesungguhnya.

> **Satu kalimat yang memisahkan Kruskal dari Prim**
> Kruskal mempertimbangkan cut YANG BERBEDA di setiap langkah — secara implisit, cut antara dua komponen mana pun yang akan digabungkan sisi terurut berikutnya. Prim mempertimbangkan JENIS cut YANG SAMA di setiap langkah — selalu antara pohon yang sudah dibangun (S) dan semua yang lain (V − S), itulah sebabnya pohon Prim selalu terhubung di setiap tahap antara, sementara sisi-sisi yang diterima Kruskal dapat membentuk beberapa bagian terputus hingga mendekati akhir.

11.6 Mengimplementasikan Priority Queue

Waktu jalan Prim sepenuhnya bergantung pada bagaimana min-priority queue Q diimplementasikan, karena EXTRACT-MIN dipanggil tepat |V| kali (sekali per simpul yang ditambahkan ke pohon) dan DECREASE-KEY dipanggil paling banyak |E| kali (sekali per relaksasi sisi).

Fase 5 — Implementasi: algoritma Prim dengan binary min-heap

```
PRIM-HEAP(V, E, w, r):
  key[r] = 0;  for all other v: key[v] = infinity
  build a binary min-heap Q over V, keyed by key[]      // O(V)
  while Q is not empty
      u = HEAP-EXTRACT-MIN(Q)                          // O(log V)
      mark u as in the tree
      for each (v, w(u,v)) in Adj[u]
          if v in Q and w(u, v) < key[v]
              key[v] = w(u, v);  parent[v] = u
              HEAP-DECREASE-KEY(Q, v, key[v])           // O(log V)
```

Dengan binary heap, membangun heap awal berbiaya O(V), |V| pemanggilan EXTRACT-MIN berbiaya total O(V log V), dan pemanggilan DECREASE-KEY sebanyak O(E) berbiaya total O(E log V) — menghasilkan total waktu jalan O((V + E) log V), yang disederhanakan menjadi O(E log V) untuk graf terhubung mana pun (di mana E ≥ V − 1). Ini adalah batas asimtotik yang SAMA dengan Θ(E log V) milik

Kruskal — untuk graf jarang, kedua algoritma pada dasarnya seri. Priority queue berbasis array yang lebih sederhana, di mana EXTRACT-MIN memindai semua V entri dalam O(V) dan DECREASE-KEY adalah penulisan array O(1), memberikan total waktu $O(V^2)$ — lebih buruk untuk graf jarang, tetapi LEBIH BAIK untuk graf padat di mana E mendekati V^2, karena $O(V^2)$ kemudian mengalahkan O(E log V) = $O(V^2 \log V)$.

11.7 Menganalisis Waktu Jalan Prim

Implementasi priority queue	Biaya EXTRACT-MIN	Biaya DECREASE-KEY	Total waktu jalan
Array sederhana	O(V) tiap, $O(V^2)$ total	O(1) tiap, O(E) total	$O(V^2)$ — terbaik utk graf PADAT
Binary min-heap	O(log V) tiap, O(V log V) total	O(log V) tiap, O(E log V) total	O(E log V) — setara Kruskal
Fibonacci heap	O(log V) teramortisasi	O(1) teramortisasi	O(E + V log V) — terbaik secara teori

Kepadatan menentukan pemenang

Untuk graf JARANG (E mendekati V), versi binary-heap Prim dan Θ(E log V) milik Kruskal secara asimtotik seri — pilih mana pun yang lebih mudah diimplementasikan atau lebih cocok dengan struktur data yang sudah tersedia. Untuk graf PADAT (E mendekati V^2), versi array-sederhana Prim's $O(V^2)$ mengalahkan keduanya — menghindari membayar faktor logaritmik pada jumlah sisi yang sangat banyak. Algoritma Kruskal tidak memiliki percepatan setara untuk graf padat, karena tetap harus mengurutkan semua E sisi terlepas dari kepadatannya.

11.8 Profitina dalam Praktik: Memilih Algoritma MST

Tentang kotak ini

Seperti di setiap bab, kotak ini menghubungkan materi mata kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) Indonesia nyata yang dibangun oleh penulis, tanpa mengungkapkan detail implementasi yang bersifat proprietary.

Skenario hipotetis Fase-4 di Bab 10 memodelkan jaringan hub regional Profitina di masa depan sebagai graf jarang — segelintir hub dengan segelintir kandidat tautan, persis situasi di mana algoritma Kruskal (atau versi binary-heap Prim) adalah pilihan alami. Tetapi tidak semua keputusan berbentuk-MST dalam skenario itu bersifat jarang: saat mengevaluasi pasangan probe pemantauan redundan mana yang perlu dijaga saling terhubung penuh di dalam satu pusat data — jenis keputusan padat dan lokal yang sudah dihadapi deployment server tunggal Profitina yang sesungguhnya HARI INI — graf kandidat-tautan dapat menjadi hampir LENGKAP (setiap probe secara teknis dapat menjangkau setiap probe lain secara langsung), mendorong E mendekati V^2. Dalam rezim yang lebih padat itu, versi array-sederhana algoritma Prim — secara asimtotik lebih buruk di atas kertas untuk input jarang, tetapi $O(V^2)$ terlepas dari jumlah sisi — menjadi pilihan nyata yang lebih efisien, mengilustrasikan mengapa

perbandingan berbasis-kepadatan Bagian 11.7 bukan sekadar latihan buku teks tetapi keputusan rekayasa yang sesungguhnya.

11.9 Ringkasan Bab dan Poin-Poin Kunci

- Algoritma Prim menumbuhkan SATU pohon ke luar dari simpul awal, selalu menambahkan sisi termurah yang keluar dari pohon sejauh ini — mekanisme yang secara fundamental berbeda dari pendekatan urutkan-dan-gabung Kruskal, tetapi menyelesaikan masalah MST yang identik.
- Algoritma Prim menjaga key[v] dan parent[v] untuk setiap simpul di luar pohon, disimpan dalam min-priority queue, dan merelaksasi nilai-nilai ini setiap kali simpul yang baru ditambahkan menawarkan koneksi yang lebih murah.
- Pada jaringan tujuh-hub Bab 10, algoritma Prim (dimulai dari JKT) mencapai minimum spanning tree yang identik — enam sisi sama, total bobot sama 35 — dengan algoritma Kruskal, karena bobot sisi jaringan semuanya berbeda sehingga MST-nya unik.
- Kebenaran Prim mengikuti dari sifat cut yang sama dengan Kruskal, tetapi diterapkan pada cut yang berbeda di setiap langkah: selalu cut antara pohon yang sudah dibangun dan semua yang lain.
- Dengan binary min-heap, algoritma Prim berjalan dalam O(E log V) — setara dengan Θ(E log V) milik Kruskal untuk graf jarang. Dengan array sederhana, berjalan dalam $O(V^2)$ — pilihan lebih baik untuk graf padat, di mana pengurutan sisi wajib milik Kruskal tidak memiliki percepatan setara.
- Memilih antara algoritma Kruskal dan Prim dalam praktik adalah keputusan kepadatan, bukan keputusan kebenaran — keduanya selalu benar, oleh sifat cut mendasar yang sama.

"Kruskal meminta setiap sisi audisi menurut urutan harga. Prim hanya bertanya ke sisi yang menyentuh apa yang sudah dibangun — tujuan sama, perjalanan sama sekali berbeda."

— aforisme mata kuliah ini

(Keduanya aman lewat cut property yang sama — hanya cut mana yang dilihat yang berbeda.)

Kuliah 12 adalah sesi persiapan Kuis 2, meninjau Kuliah 9 hingga 11 (Algoritma Greedy, dan Graf serta Minimum Spanning Tree) dengan soal latihan alih-alih konten baru. Kuliah 13 kemudian beralih ke masalah graf yang terkait namun berbeda — LINTASAN TERPENDEK — di mana algoritma Dijkstra memakai ulang mekanisme priority-queue-plus-relaksasi Prim yang persis sama, tetapi melacak LINTASAN termurah dari sebuah sumber, bukan POHON termurah yang menghubungkan setiap simpul.

11.10 Pertanyaan Tinjauan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Telusuri algoritma Prim pada jaringan Gambar 10.2 dimulai dari simpul yang BERBEDA — misalnya, MKS alih-alih JKT. Apakah total bobot MST berubah? Apakah urutan pasti pembaruan key[] berubah?
2. Jelaskan, dengan kata-kata Anda sendiri, mengapa pohon Prim selalu terhubung di setiap langkah antara, sementara sisi-sisi yang diterima Kruskal dapat membentuk beberapa bagian terputus hingga mendekati akhir. Mengapa perbedaan ini tidak memengaruhi kebenaran kedua algoritma?
3. Bagian 11.4 mengklaim bahwa bobot sisi yang semuanya berbeda menjamin MST yang unik. Konstruksikan graf kecil (4–5 simpul) dengan dua sisi berbobot SAMA yang diposisikan sehingga algoritma Kruskal dan Prim menghasilkan dua minimum spanning tree berbeda yang sama-sama minimum.
4. Menggunakan jejak Gambar 11.2, identifikasi setiap titik di mana nilai key[] suatu simpul diperbarui LEBIH DARI SEKALI sebelum simpul tersebut akhirnya diekstrak. Apa yang direpresentasikan setiap pembaruan tersebut dalam istilah sifat cut?
5. Bandingkan kompleksitas ruang algoritma Kruskal (yang membutuhkan daftar sisi terurut dan struktur union-find) dengan algoritma Prim (yang membutuhkan priority queue dan array key[]/parent[]). Mana yang memakai lebih banyak ruang bantu, dan apakah bergantung pada kepadatan graf?
6. Seorang kolega mengklaim bahwa algoritma Prim dapat dimodifikasi untuk menemukan MAXIMUM spanning tree hanya dengan memakai EXTRACT-MAX alih-alih EXTRACT-MIN. Apakah klaim ini benar? Justifikasi jawaban Anda menggunakan sifat cut.

Lampiran 11.A — Log Verifikasi untuk Kode Sumber Bab 11

Seperti pada bab-bab sebelumnya, setiap program yang disertakan dalam buku ini dikompilasi dan dijalankan terhadap contoh yang telah diverifikasi secara manual sebelum disertakan. 01_prim_mst.cpp (identik di folder Code/Chapter11 bab ini dengan salinan Bab 10, karena kedua bab menelusuri algoritma yang sama pada jaringan yang sama) dikompilasi dengan `g++ -O2 -std=c++17 -Wall` (nol peringatan, nol kesalahan) dan menghasilkan keluaran persis seperti yang diprediksi dalam Gambar 11.2 dan narasi Bagian 11.3. Progres nilai key[] pada setiap ekstraksi juga diverifikasi silang secara independen oleh simulasi Python — lihat /tmp/daa_pilot/verify_ch11_prim_trace2.py — yang cocok persis dengan keluaran terkompilasi di setiap satu dari tujuh langkah.

```
$ ./01_prim_mst
Prim trace (start = JKT):
  START at JKT
  ADD BDG via edge JKT-BDG (w=3)
  ADD SBY via edge BDG-SBY (w=7)
  ADD DPS via edge SBY-DPS (w=5)
  ADD MKS via edge DPS-MKS (w=6)
  ADD BPN via edge MKS-BPN (w=4)
  ADD MDN via edge BPN-MDN (w=10)

MST edges (6):
  JKT-BDG (w=3)
  BDG-SBY (w=7)
  SBY-DPS (w=5)
  DPS-MKS (w=6)
  MKS-BPN (w=4)
  BPN-MDN (w=10)
MST total weight: 35     # cocok dgn Kruskal (Bab 10) dan Gambar 11.3
```

Latihan SPOJ — buku pendamping
Teknik bab ini diuji sungguhan di juri daring SPOJ. Buka Bab 11 buku pendamping “SPOJ Competitive Programming” (folder SPOJ, BI & EN) untuk implementasi MST yang lolos judge — lengkap dengan tips, trik, dan hint agar solusi Anda Accepted, tanpa membocorkan jawaban.

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Bab 23, Minimum Spanning Trees, Bagian 23.2, The algorithms of Kruskal and Prim).

[2] R. C. Prim. "Shortest Connection Networks and Some Generalizations." Bell System Technical Journal, 36(6), 1957, 1389–1401. (Makalah asli.)

[3] M. A. Weiss. Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2014. (Bab 9, Graph Algorithms — Prim's Algorithm.)

[4] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, 4th ed. Lulu Press, 2020–2023.

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 12 · BAB LATIHAN

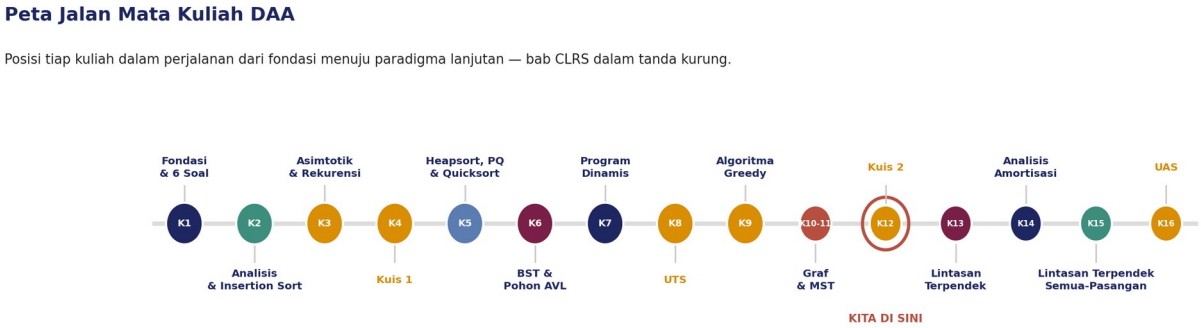
Soal Latihan: Algoritma Greedy Hingga Minimum Spanning Tree

Tidak ada materi baru di sini — hanya dua belas soal yang dirancang untuk memastikan Kuliah 9 sampai 11 benar-benar dikuasai, sebelum Kuis 2.

Tujuan Pembelajaran — setelah mengerjakan bab ini Anda akan mampu:
(1) Menerapkan strategi greedy pada instance baru activity selection dan menjustifikasi penalaran exchange-argument di baliknya. (2) Membangun pohon Huffman dari sekumpulan frekuensi secara manual dan menghitung penghematan bit-nya dibanding fixed-length encoding. (3) Menyelesaikan fractional knapsack secara manual, dan mengonstruksi instance konkret di mana greedy gagal pada varian 0/1. (4) Menelusuri algoritma Kruskal pada graf baru, termasuk operasi union-find yang mendeteksi siklus. (5) Menelusuri algoritma Prim pada graf baru dari simpul awal tertentu, dan mengonfirmasi ia mencapai total bobot yang sama dengan Kruskal. (6) Menentukan apakah minimum spanning tree unik untuk suatu graf tertentu, dan bernalar langsung lewat cut property.

12.1 Rekap: Kuliah 9–11 secara Ringkas

Kuliah 9 memperkenalkan paradigma desain yang sama sekali baru setelah tujuh kuliah divide-and-conquer dan program dinamis: algoritma greedy, yang membangun solusi lewat serangkaian pilihan lokal optimal, tidak pernah ditinjau ulang, dan membuktikan kebenarannya dengan exchange argument alih-alih rekurensi. Activity Selection (pilih berdasarkan waktu selesai tercepat), kode Huffman (berulang kali menggabungkan dua simpul berfrekuensi terendah menjadi kode biner prefix-free optimal), dan soal Fractional Knapsack (mengisi kapasitas berdasarkan rasio nilai-terhadap-bobot) semuanya tunduk pada paradigma ini — sementara soal 0/1 Knapsack, yang tampak mirip secara sekilas, justru mengalahkan pendekatan ini, memerlukan program dinamis sebagai gantinya. Kuliah 10 dan 11 kemudian menerapkan penalaran greedy pada graf: soal Minimum Spanning Tree, dibuktikan benar lewat cut property (sisi termurah yang melintasi cut mana pun termasuk dalam suatu MST), diselesaikan oleh dua algoritma greedy yang secara struktural berbeda namun selalu setuju pada jawaban yang sama kapan pun bobot sisi berbeda-beda. Algoritma Kruskal (Kuliah 10) mengurutkan setiap sisi sekali dan menggabungkan komponen dengan struktur data union-find, mempertimbangkan cut yang BERBEDA di setiap langkah; algoritma Prim (Kuliah 11) menumbuhkan satu tree ke luar dari satu simpul menggunakan array key[]/parent[] dan min-priority queue, mempertimbangkan cut yang SAMA — tree yang sudah dibangun versus semua yang lain — di setiap langkah.



Gambar 12.1 — Bab ini berada di Kuliah 12 dalam peta enam belas kuliah DAA: sebuah checkpoint, bukan topik baru, tepat sebelum Kuis 2.

12.2 Cara Memakai Bab Latihan Ini

Apa bab ini — dan apa yang bukan

Ini adalah Bab Latihan: bab praktik, bukan bab konten baru. Setiap soal di bawah berakar pada materi spesifik dari Kuliah 9–11 (lihat Gambar 12.2). Setiap soal disertai PETUNJUK — sebuah dorongan ke arah cara berpikir yang benar — tapi sengaja BUKAN solusi lengkap atau source code. Mengerjakan soal itu sendiri, bahkan tidak sempurna, mengajarkan jauh lebih banyak daripada membaca jawaban yang sudah jadi.

Gambar 12.2 — Setiap soal di Bagian 12.3 berakar pada salah satu dari tiga kuliah sebelumnya.

Sebelum mulai

Coba kerjakan setiap soal sungguh-sungguh sebelum membaca petunjuknya. Kalau macet, baca hanya petunjuknya — bukan solusi — lalu coba lagi. Ini persis disiplin yang akan dihargai Kuis 2 open-book di Bagian 12.4: kuis mengizinkan referensi, tapi bukan pengganti penalaran Anda sendiri.

12.3 Soal Latihan

12.3.1 Algoritma Greedy

Soal 1 *[Mudah]*. Enam kandidat aktivitas memiliki waktu (mulai, selesai): A(1,4), B(3,5), C(2,6), D(5,7), E(6,8), F(7,9). Terapkan algoritma greedy Activity Selection (urutkan berdasarkan waktu selesai, selalu pilih aktivitas berikutnya yang kompatibel dengan yang terakhir diterima) dan sebutkan HIMPUNAN LENGKAP aktivitas yang terpilih, secara berurutan.

> **Petunjuk**
> Urutkan ketat berdasarkan waktu selesai dulu — A sudah memiliki waktu selesai paling awal. Sebuah aktivitas kompatibel dengan yang terakhir diterima hanya jika waktu mulainya lebih besar atau sama dengan waktu selesai aktivitas terakhir yang diterima.

Soal 2 *[Sedang]*. Sebuah sumber memiliki enam simbol dengan frekuensi a=2, b=3, c=5, d=8, e=13, f=21 (sengaja menyerupai deret Fibonacci). Bangun pohon Huffman secara manual (berulang kali menggabungkan dua simpul berfrekuensi terendah) dan sebutkan panjang kata-kode yang dihasilkan untuk tiap simbol. Bandingkan total panjang terenkode terhadap kode tetap 3-bit untuk keenam simbol, dan jelaskan mengapa POLA FREKUENSI INI KHUSUSNYA menghasilkan pohon Huffman yang sangat tidak seimbang ("berbentuk ulat") alih-alih seimbang.

> **Petunjuk**
> Di setiap langkah penggabungan mungkin ada seri untuk "dua frekuensi terendah" — aturan pemutus-seri apa pun menghasilkan TOTAL biaya yang sama, meski bentuk pohon persisnya bisa berbeda. Perhatikan apa yang terjadi pada frekuensi terbesar (21): apakah ia pernah digabungkan sebelum langkah paling akhir?

Soal 3 *[Sedang]*. Sebuah Fractional Knapsack berkapasitas 15. Empat item (bobot, nilai) tersedia: I1(5,10), I2(3,9), I3(8,8), I4(6,12). Hitung rasio nilai-terhadap-bobot tiap item, tentukan urutan pengisian greedy, dan sebutkan nilai total maksimum yang dapat dicapai (termasuk item fraksional yang diambil).

> **Petunjuk**
> Urutkan ketat berdasarkan rasio nilai/bobot, menurun. Ambil item utuh secara greedy sampai item BERIKUTNYA akan melebihi kapasitas tersisa — lalu ambil hanya sebagian dari item itu yang muat, dan berhenti.

Soal 4 *[Sulit]*. Sebuah 0/1 Knapsack berkapasitas 10. Empat item (bobot, nilai) tersedia: X(6,30), Y(5,24), Z(5,24), W(2,9). Hitung solusi 0/1 greedy-berdasar-rasio (ambil item utuh berurutan rasio, lewati yang sudah tidak muat) dan bandingkan terhadap solusi optimal 0/1 yang SEBENARNYA. Tunjukkan bahwa greedy secara ketat lebih buruk di sini, dan jelaskan dengan kata-kata Anda sendiri MENGAPA mengambil item berrasio tertinggi tunggal dapat menghalangi kombinasi yang lebih baik.

> **Petunjuk**
> Urutkan keempat item berdasarkan rasio nilai/bobot dulu — X memiliki rasio tertinggi tunggal, tapi mengambilnya utuh menghabiskan 6 dari 10 unit kapasitas. Tanyakan kombinasi APA dari ketiga item LAIN yang bisa mengisi kapasitas yang sama itu, dan bandingkan total nilainya.

12.3.2 Graf & Minimum Spanning Tree Kruskal

Soal 5 ***[Mudah]***. Sebuah jaringan enam kandidat hub P, Q, R, S, T, U memiliki kandidat tautan (dengan biaya): P–Q(4), P–R(2), Q–R(1), Q–S(5), R–S(8), R–T(10), S–T(2), S–U(6), T–U(3). Seorang kolega mengusulkan spanning tree {P–Q, Q–R, Q–S, S–T, T–U} dengan total biaya 4+1+5+2+3=15. Apakah ini MST? Jika tidak, sebutkan SATU sisi spesifik yang akan Anda tukar keluar, dan sisi yang akan Anda tukar masuk, untuk secara ketat mengurangi total biaya — dan sebutkan total barunya.

> **Petunjuk**
> Sebuah spanning tree minimum hanya jika TIDAK ADA pertukaran satu sisi pun yang dapat menurunkan biayanya. Periksa apakah mengganti P–Q dengan sisi yang lebih murah yang tetap menghubungkan P ke sisa tree itu mungkin — cut property memberi tahu Anda persis sisi mana yang melintasi suatu cut yang paling aman untuk dipilih.

Soal 6 ***[Sedang]***. Menggunakan jaringan enam-hub yang sama dari Soal 5, telusuri algoritma Kruskal secara lengkap: sebutkan setiap kandidat sisi dalam urutan terurut, dan untuk masing-masing nyatakan TERIMA atau TOLAK (dengan alasan, jika ditolak). Sebutkan himpunan sisi MST akhir dan total bobotnya.

> **Petunjuk**
> Proses sisi ketat berdasarkan bobot menaik. Sebuah sisi ditolak jika dan hanya jika kedua ujungnya sudah berada dalam komponen union-find yang sama — menerimanya akan membentuk siklus, bukan tree.

Soal 7 ***[Sedang]***. Melanjutkan Soal 6: tunjukkan pointer parent hutan union-find setelah SETIAP operasi union yang diterima (asumsikan union-by-rank, dan setiap himpunan singleton awal dimulai dengan rank 0 untuk masing-masing P, Q, R, S, T, U). Sebutkan simpul mana yang berakhir sebagai akar komponen akhir tunggal.

> **Petunjuk**
> Union-by-rank melekatkan akar tree berrank LEBIH RENDAH di bawah akar tree berrank LEBIH TINGGI; ketika rank sama, akar mana pun bisa menjadi akar baru tapi rank-nya bertambah satu. Lacak keputusan ini secara eksplisit di setiap union yang diterima — akar spesifik yang bertahan bergantung pada konvensi pemutus-seri yang Anda pilih, jadi sebutkan konvensi Anda.

Soal 8 ***[Sulit]***. Menggunakan MST yang Anda temukan di Soal 6, buktikan — menggunakan cut property secara langsung, bukan lewat pencarian menyeluruh — bahwa sisi termurah tunggal di SELURUH jaringan (Q–R, bobot 1) HARUS termasuk dalam setiap minimum spanning tree graf ini, terlepas dari algoritma mana yang dipakai untuk menemukannya.

> **Petunjuk**
> Pertimbangkan cut yang memisahkan {Q} dari lima simpul lainnya. Di antara semua sisi yang melintasi cut ini, apakah Q–R secara ketat yang termurah? Cut property menyatakan bahwa sisi berbobot-minimum yang melintasi cut MANA PUN aman untuk disertakan dalam suatu MST — tapi di sini Anda perlu klaim yang lebih kuat bahwa ia termasuk dalam SETIAP MST. Pikirkan apa yang akan salah jika ada MST lain yang mengecualikannya.

12.3.3 Algoritma Minimum Spanning Tree Prim

Soal 9 ***[Sedang]***. Menggunakan jaringan enam-hub yang sama dari Soal 5, telusuri algoritma Prim dimulai dari simpul U (bukan P) — tunjukkan nilai key[] dan pointer parent[] untuk setiap simpul yang belum ditambahkan di setiap langkah, dengan gaya tabel jejak Bagian 11.3. Sebutkan himpunan sisi MST akhir dan total bobotnya, dan konfirmasi ia cocok dengan hasil Kruskal di Soal 6.

> **Petunjuk**
> Inisialisasi key[U]=0 dan semua lainnya tak terhingga. Di setiap langkah, ekstrak simpul belum-ditambahkan dengan key[] terkecil, tambahkan ke tree, lalu relaksasi setiap sisi yang keluar darinya — persis seperti yang dilakukan PRIM di Bagian 11.2, hanya dimulai dari simpul berbeda.

Soal 10 ***[Sedang]***. Sebuah jaringan empat-hub W, X, Y, Z memiliki kandidat tautan: W–X(2), X–Y(3), Y–Z(2), Z–W(3), W–Y(5). Temukan total bobot minimum spanning tree, lalu tentukan apakah MST-nya UNIK. Jika tidak, tunjukkan DUA himpunan sisi berbeda yang sama-sama mencapai total bobot minimum tersebut.

> **Petunjuk**
> Perhatikan bahwa ada dua pasang sisi yang seri bobotnya di sini (X–Y dan Z–W sama-sama berbobot 3). Bagian 11.4 menetapkan bahwa bobot sisi yang semuanya berbeda-beda menjamin MST unik — graf ini sengaja melanggar prasyarat itu. Coba jalankan algoritma Kruskal dengan seri diputus KEDUA cara dan bandingkan dua tree yang dihasilkan.

Soal 11 ***[Sulit]***. Dalam skenario hipotetis masa depan yang sesuai roadmap Fase-4 (ekspansi regional) yang memang dinyatakan publik oleh Profitina (BUKAN deskripsi deployment server-tunggalnya saat ini), Profitina sedang mempertimbangkan meng-upgrade jaringan probe pemantauannya dari 20 probe dengan 30 kandidat tautan (jarang, E mendekati V) menjadi topologi pusat data yang saling terhubung penuh dengan 20 probe dan 190 kandidat tautan (padat, E mendekati V(V−1)/2). Untuk MASING-MASING skenario, sebutkan implementasi Prim mana (array sederhana, $O(V^2)$, atau binary min-heap, O(E log V)) yang memiliki waktu eksekusi asimtotik lebih kecil, tunjukkan aritmatika yang mengarah ke jawaban Anda.

Petunjuk

Hitung V^2 untuk V=20, dan bandingkan terhadap E $\log_2 V$ untuk E sebenarnya di masing-masing skenario. Titik crossover antara kapan Prim berbasis-array dan berbasis-heap lebih cepat bergantung persis pada bagaimana E dibandingkan dengan V log V, bukan hanya pada apakah graf 'terasa' padat atau jarang.

Soal 12 ***[Sulit]***. Sebuah soal baru: Anda diberi graf tak-berarah, terhubung, berbobot dan diberi tahu bahwa graf itu memiliki SETIDAKNYA DUA sisi berbobot sama. Apakah fakta ini SAJA menjamin graf tersebut memiliki lebih dari satu minimum spanning tree? Konstruksikan contoh kecil (4 atau 5 simpul) di mana sebuah graf memiliki sepasang bobot sisi yang seri tapi TETAP memiliki MST yang unik, untuk mendukung jawaban Anda.

Petunjuk

Sebuah seri hanya mengancam keunikan jika KEDUA sisi yang seri secara bersamaan layak menjadi sisi aman untuk cut yang SAMA di suatu titik dalam eksekusi Kruskal. Jika sisi-sisi yang seri itu tidak pernah bersaing untuk cut yang sama — misalnya, jika salah satunya akan membentuk siklus pada saat ia dipertimbangkan terlepas dari yang lain — keunikan bisa bertahan dari seri itu. Contoh Soal 10 menunjukkan seri yang MEMANG merusak keunikan; Anda perlu satu yang tidak.

12.4 Format Kuis 2: Open-Book, Take-Home Essay

Kuis 2 adalah tugas esai open-book, take-home yang mencakup persis materi yang diulas di bab ini (Kuliah 9 sampai 11) plus semua yang sebelumnya — tidak lebih. Anda akan memiliki satu minggu untuk mengumpulkan jawaban tertulis dengan penalaran lengkap; jawaban akhir yang benar tanpa justifikasi hanya mendapat sedikit nilai, karena inti dari format ujian esai adalah melihat BAGAIMANA Anda berpikir, bukan hanya apa kesimpulan Anda.

Open-book berarti referensi, bukan co-author

Anda boleh berkonsultasi dengan buku teks, catatan, dan slide kuliah saat menjawab. Anda TIDAK BOLEH berkolaborasi dengan teman sekelas atau mengumpulkan penalaran yang bukan sungguh-sungguh milik Anda sendiri — ujian open-book menguji apakah Anda dapat MENERAPKAN apa yang telah Anda pelajari, tanpa pengawasan, yang persis merupakan makna "mendesain dan menganalisis algoritma" dalam praktik. Jika Anda mengutip sumber secara langsung, sertakan sitasinya.

Kriteria	Yang dinilai	Bobot
Kebenaran penalaran	Apakah logika, bukti, jejak, atau derivasi benar-benar valid — bukan cuma jawaban akhir?	40%
Kejelasan bukti/derivasi	Bisakah pembaca mengikuti setiap langkah tanpa harus menebak maksud Anda?	25%
Penggunaan notasi yang tepat	Apakah O/Ω/Θ, argumen cut-property, dan diagram graf/tree dinyatakan presisi, sesuai	20%

	konvensi bab-bab sebelumnya?	
Presentasi	Apakah jawaban terorganisir, terbaca, dan mudah dinilai?	15%

12.5 Profitina in Practice: Tinjau-Ulang Sendiri Sebelum Pengumpulan

> **Tentang kotak ini**
> Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) Indonesia nyata yang dibangun oleh penulis, tanpa membocorkan detail implementasi yang proprietary.

Sebelum perubahan algoritma baru dirilis ke produksi di Profitina, para insinyur menjalankannya lewat checklist tinjau-ulang-sendiri singkat — sangat mirip Lampiran 12.A di bawah — sebelum meminta sepasang mata kedua: apakah sifat pilihan-greedy yang diklaim benar-benar dijustifikasi dengan exchange argument, atau hanya diasumsikan? Apakah kandidat minimum spanning tree benar-benar diperiksa terhadap cut property pada sisi spesifik yang dipersoalkan, alih-alih diasumsikan benar karena algoritma yang menghasilkannya dipercaya? Kebiasaan tinjau-ulang-sendiri secara adversarial sebelum tinjauan eksternal ini — disiplin yang sama di balik pelajaran perburuan-bug yang diulang di seluruh buku teks ini, bahwa hasil yang rapi secara visual tetap bisa salah secara matematis — persis apa yang dilatih oleh format ujian take-home open-book.

12.6 Ringkasan Bab

- Bab ini tidak memperkenalkan materi baru — ia adalah checkpoint yang mencakup Kuliah 9 sampai 11.
- Dua belas soal mencakup seluruh rentang yang diulas: Activity Selection greedy, kode Huffman, Fractional versus 0/1 Knapsack, algoritma Kruskal dengan union-find, algoritma Prim dari simpul awal sembarang, dan keunikan MST di bawah bobot sisi yang seri.
- Setiap soal menyertakan petunjuk, bukan solusi — mengerjakan penalarannya sendiri adalah intinya.
- Kuis 2 adalah esai open-book, take-home: referensi diizinkan, tapi penalaran harus milik Anda sendiri, dan kebenaran PENALARAN (bukan cuma jawaban akhir) adalah mayoritas nilai.

“Minimum spanning tree hanya punya satu jawaban benar ketika bobot menolak untuk seri — dua algoritma, satu tujuan.”

— aforisme mata kuliah ini

(Membaca ulang bukti tidak terhitung menguasainya — menurunkannya ulang dari nol baru terhitung.)

Lampiran 12.A — Checklist Tinjau-Diri (Non-Dinilai)

Sebelum mengumpulkan jawaban apa pun — di soal bab ini maupun di Kuis 2 sendiri — jalankan lewat checklist ini. Ini hanya butuh beberapa menit dan menangkap sebagian besar kesalahan yang paling sering dilihat penilai.

- Apakah saya benar-benar memverifikasi klaim exchange argument atau cut-property pada contoh konkret yang diberikan, bukan sekadar menegaskannya berlaku?
- Dalam struktur yang ditelusuri manual mana pun (pohon Huffman, hutan union-find, tabel key[]/parent[] Prim), apakah saya menunjukkan SETIAP keadaan antara, bukan cuma hasil akhir?
- Jika saya mengklaim MST unik (atau tidak), apakah saya memeriksa secara spesifik apakah bobot sisi yang seri benar-benar bersaing untuk cut yang sama — bukan cuma apakah ada seri di suatu tempat di graf?
- Jika saya mengklaim greedy gagal pada instance 0/1 Knapsack, apakah saya menunjukkan KEDUA jawaban greedy DAN alternatif yang secara ketat lebih baik, dengan angka sungguhan?
- Apakah saya membedakan cut per-langkah Kruskal (yang berubah setiap langkah) dari cut per-langkah Prim (selalu tree-sejauh-ini versus sisanya) di mana pun bukti kebenaran kedua algoritma itu muncul?
- Apakah saya membaca ulang jawaban saya sendiri seolah-olah saya adalah penilai skeptis yang mencari celah dalam argumen?

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Bab 15–16, 21, 23).

[2] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, 4th ed. Lulu Press, 2020–2023.

[3] M. A. Weiss. Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2014. (Bab 9, Graph Algorithms).

[4] R. C. Prim. "Shortest Connection Networks and Some Generalizations." Bell System Technical Journal, 36(6), 1957, 1389–1401.

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 13

Lintasan Terpendek Sumber-Tunggal: Algoritma Dijkstra dan Bellman-Ford

Pertanyaan graf baru — bukan "apa cara termurah menghubungkan semuanya" (MST), melainkan "apa cara termurah untuk pergi dari SINI ke mana pun." Dua algoritma, satu mekanisme yang sama (relaksasi), dan satu garis pemisah yang krusial: apakah bobot sisi negatif diperbolehkan.

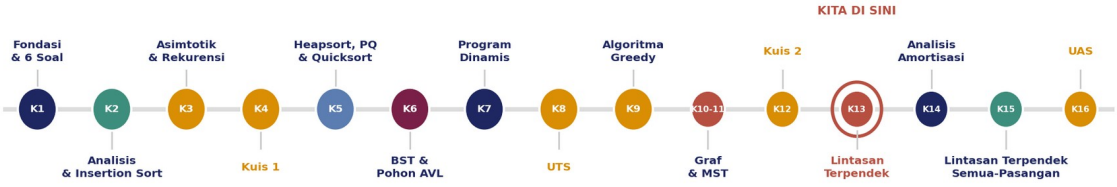
Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Menyatakan masalah lintasan terpendek sumber-tunggal dan menjelaskan operasi RELAXATION yang mendasari setiap algoritma dalam bab ini. (2) Menelusuri algoritma Dijkstra secara manual pada graf berarah berbobot non-negatif, termasuk array dist[]/parent[]-nya dan urutan finalisasi antrean prioritas minimumnya. (3) Menjelaskan MENGAPA strategi greedy Dijkstra yang memfinalisasi-dan-tidak-pernah-mengunjungi-ulang mengharuskan bobot sisi non-negatif, serta menyusun contoh di mana algoritma ini gagal pada sisi negatif. (4) Menelusuri algoritma Bellman-Ford secara manual, termasuk relaksasi setiap sisi secara bertahap (pass-by-pass) selama $|V|-1$ putaran. (5) Menjelaskan bagaimana Bellman-Ford mendeteksi siklus berbobot negatif menggunakan putaran ke-$|V|$-nya, dan mengapa Dijkstra tidak memiliki pengaman setara. (6) Menyatakan dan membandingkan waktu jalan algoritma Dijkstra (dengan binary heap) dan Bellman-Ford, serta memilih algoritma yang tepat untuk suatu kendala bobot graf tertentu.

13.1 Rekap: Dari Satu Pohon ke Banyak Lintasan

Bab 10 dan 11 menyelesaikan masalah Minimum Spanning Tree: diberikan graf berbobot yang terhubung, temukan himpunan sisi termurah yang menyentuh setiap simpul. Bab ini beralih ke pertanyaan yang terkait tetapi benar-benar berbeda, disebut masalah LINTASAN TERPENDEK SUMBER-TUNGGAL (single-source shortest-paths): diberikan graf berarah berbobot dan simpul sumber s yang ditentukan, temukan LINTASAN termurah dari s ke setiap simpul lain. MST peduli pada menghubungkan semuanya semurah mungkin secara agregat; pohon lintasan terpendek peduli pada RUTE termurah setiap tujuan individual dari sumber — dan, seperti yang akan ditunjukkan bab ini, kedua tujuan tersebut dapat menghasilkan pohon yang sama sekali berbeda pada graf yang persis sama.

Peta Jalan Mata Kuliah DAA

Posisi tiap kuliah dalam perjalanan dari fondasi menuju paradigma lanjutan — bab CLRS dalam tanda kurung.



Gambar 13.1 — Posisi bab ini dalam peta jalan enam belas kuliah DAA. Kuliah 13 membuka unit lintasan terpendek, segera setelah pos pemeriksaan Kuis 2 tentang Algoritma Greedy dan MST.

Gagasan bersama di balik kedua algoritma: relaksasi

Setiap algoritma lintasan terpendek dalam buku ini — Dijkstra, Bellman-Ford, dan Floyd-Warshall di Bab 15 — dibangun dari operasi primitif yang SAMA: diberikan sisi kandidat (u, v) berbobot w, RELAX sisi itu dengan memeriksa apakah melewati u menawarkan rute yang lebih murah ke v dibanding rute terbaik ke v yang ditemukan sejauh ini. Jika dist[u] + w < dist[v], perbarui dist[v] dan ingat u sebagai predecessor baru v. Algoritma-algoritma dalam bab ini hanya berbeda pada sisi MANA yang mereka relaksasi, dan dalam URUTAN APA — bukan pada makna relaksasi itu sendiri.

13.2 Masalah Lintasan Terpendek dan Optimal Substructure

Secara formal: diberikan graf berarah G = (V, E) dengan fungsi bobot sisi w, dan simpul sumber s, kita ingin dist[v] — bobot lintasan termurah dari s ke v — untuk setiap simpul v yang dapat dijangkau dari s, ditambah informasi parent yang cukup untuk merekonstruksi lintasan-lintasan tersebut. Lintasan terpendek menunjukkan sifat OPTIMAL SUBSTRUCTURE: jika lintasan terpendek dari s ke v melewati suatu simpul perantara u, maka bagian lintasan itu dari s ke u itu sendiri haruslah lintasan terpendek dari s ke u — jika tidak, menyisipkan lintasan s-ke-u yang lebih murah akan menghasilkan lintasan s-ke-v yang bahkan lebih murah, sebuah kontradiksi. Sifat optimal-substructure inilah yang membuat relaksasi menjadi strategi yang sah: membangun dist[] dengan benar untuk simpul-simpul yang lebih dekat ke s terlebih dahulu menjamin kebenaran untuk simpul-simpul yang lebih jauh.

Sebuah kehalusan: apa arti "terpendek" jika ada bobot negatif?

Jika sebuah graf memiliki SIKLUS BERBOBOT NEGATIF yang dapat dijangkau dari s (siklus yang total bobotnya kurang dari nol), maka "lintasan terpendek" menjadi tidak terdefinisi untuk simpul mana pun yang dapat dijangkau melalui siklus itu — Anda bisa melingkari siklus tersebut berapa kali pun untuk membuat lintasan sembarang murahnya, tanpa batas bawah. Algoritma Dijkstra begitu saja mengasumsikan hal ini tidak pernah terjadi (dengan mensyaratkan semua bobot non-negatif, yang secara otomatis meniadakan siklus negatif). Bellman-Ford tidak membuat asumsi seperti itu — algoritma ini mentolerir sisi negatif, tetapi harus secara eksplisit MENDETEKSI siklus negatif jika ada, karena tidak ada lintasan terpendek yang terdefinisi dengan baik jika siklus itu hadir (Bagian 13.7).

13.3 Algoritma Dijkstra: Finalisasi Greedy

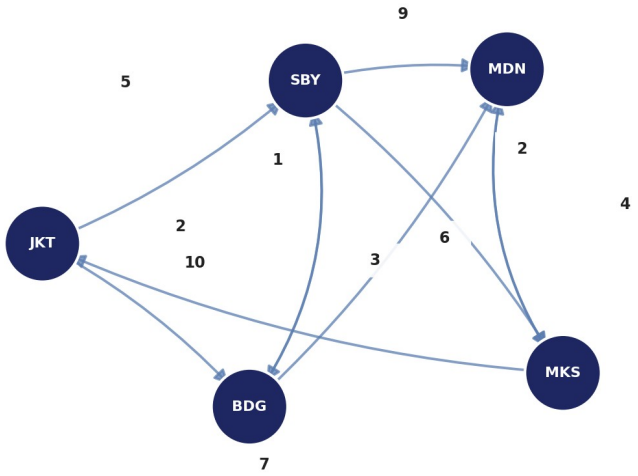
Algoritma Dijkstra menyelesaikan masalah lintasan terpendek sumber-tunggal ketika setiap bobot sisi non-negatif, menggunakan strategi yang akan terasa familiar dari Bab 11: menjaga antrean prioritas minimum yang dikunci pada jarak tentatif, berulang kali mengekstrak simpul dengan jarak tentatif terkecil, dan merelaksasi semua sisi keluarnya. Klaim greedy yang krusial — dibuktikan di Bagian 13.4 — adalah bahwa begitu sebuah simpul diekstrak, jaraknya sudah FINAL dan tidak akan pernah diperbaiki lagi, sehingga setiap simpul diproses tepat satu kali.

```
DIJKSTRA(V, E, w, s):
  for each vertex v in V
      dist[v] = infinity;  parent[v] = NIL
  dist[s] = 0
  Q = V                     // antrean prioritas minimum dikunci dist[]
  S = empty set             // simpul yang dist[]-nya sudah final
  while Q is not empty
      u = EXTRACT-MIN(Q)    // jarak tentatif terkecil
      S = S union {u}
      for each v in Adj[u]
          RELAX(u, v, w)    // jika dist[u]+w(u,v) < dist[v], perbarui
```

Gambar 13.2 menunjukkan graf berarah, lima hub skenario hipotetis Fase-4 Profitina — latensi sinkronisasi data satu arah, dalam milidetik, antar hub regional — dengan setiap bobot sisi non-negatif, persis kondisi yang disyaratkan Dijkstra.

Satu Sumber, Lima Hub: Jaringan Dijkstra

Graf berarah latensi sinkronisasi data satu-arah (ms) antar hub Profitina, SEMUA non-negatif — persis syarat yang dibutuhkan Dijkstra. Sumber = JKT.



Gambar 13.2 — Graf berarah latensi sinkronisasi satu arah (ms) antara lima hub skenario hipotetis Fase-4 Profitina. Setiap bobot non-negatif. Sumber = JKT.

Menjalankan algoritma Dijkstra pada Gambar 13.2 dimulai dari JKT menghasilkan penelusuran pada Gambar 13.3 — diverifikasi secara independen dan manual di /tmp/daa_pilot/verify_ch13_problems.py terhadap pencarian brute-force semua-lintasan-sederhana DAN jawaban terpublikasi yang terkenal untuk graf ini (ini adalah contoh klasik dari buku teks Cormen, Leiserson, Rivest, dan Stein, diganti namanya menjadi hub Profitina demi kesinambungan dengan bab-bab sebelumnya), lalu disilangperiksa terhadap keluaran hasil kompilasi 01_dijkstra.cpp pada Lampiran 13.A.

Jejak Dijkstra: Menyelesaikan Satu Hub Setiap Kali

Priority queue selalu mengeluarkan hub dengan jarak sementara terkecil. Setelah dikeluarkan, jarak itu FINAL dan tidak akan berubah lagi. Diverifikasi terhadap /tmp/daa_pilot/code/chapter13/01_dijkstra.cpp.

Langkah	Hub Diselesaikan	dist[]	Sisi Direlaksasi dari Sini
1	JKT	0	JKT→BDG: dist[BDG]=10; JKT→SBY: dist[SBY]=5
2	SBY	5	SBY→BDG: dist[BDG] 10→8; SBY→MDN: dist[MDN]=14; SBY→MKS: dist[MKS]=7
3	MKS	7	MKS→MDN: dist[MDN] 14→13 (belum final)
4	BDG	8	BDG→MDN: dist[MDN] 13→9 — mengalahkan tawaran MKS
5	MDN	9	tidak ada sisi keluar tersisa — kelima hub selesai

Gambar 13.3 — Penelusuran langkah demi langkah Dijkstra dari JKT. Setiap baris menunjukkan hub yang difinalisasi (nilai dist[]-nya ditetapkan secara permanen) dan sisi keluar mana yang direlaksasi.

Menelusuri data Gambar 13.3: dimulai dengan dist[JKT]=0, mengekstrak JKT merelaksasi BDG (dist=10) dan SBY (dist=5). SBY memiliki jarak tentatif lebih kecil, sehingga diekstrak berikutnya; dari SBY, merelaksasi BDG menurunkan jaraknya dari 10 menjadi 8, merelaksasi MDN menetapkannya menjadi 14, dan merelaksasi MKS menetapkannya menjadi 7. Di antara kandidat yang tersisa {BDG:8, MDN:14, MKS:7}, MKS kini terkecil, sehingga diekstrak (dist=7); satu-satunya relaksasi bergunanya menurunkan MDN dari 14 menjadi 13 — masih belum final, karena BDG belum diproses. BDG diekstrak berikutnya (dist=8), dan relaksasinya terhadap MDN (via BDG→MDN, bobot 1) menurunkan jarak MDN dari 13 menjadi 9 — mengalahkan tawaran MKS sebelumnya. Akhirnya MDN diekstrak (dist=9), tanpa sisi keluar tersisa untuk direlaksasi. Setiap satu dari kelima nilai ini — 0, 5, 8, 7, 9 untuk JKT, SBY, BDG, MKS, MDN secara berurutan — persis cocok dengan keluaran hasil kompilasi Lampiran 13.A.

13.4 Membuktikan Kebenaran Dijkstra: Mengapa Bobot Non-Negatif Penting

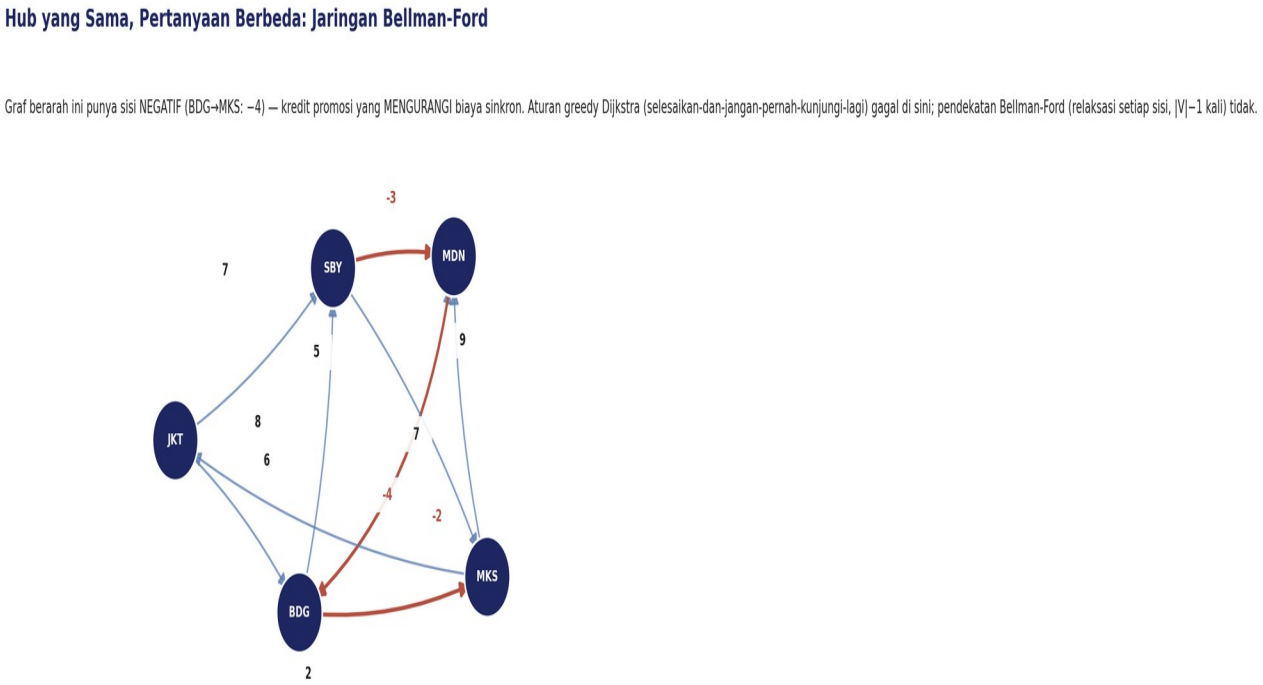
Kebenaran Dijkstra bertumpu pada satu klaim kunci: pada saat sebuah simpul u diekstrak dari antrean prioritas, dist[u] sudah sama dengan jarak lintasan-terpendek SEBENARNYA dari s ke u, dan tidak akan pernah perlu berubah lagi. Buktinya dengan kontradiksi dan induksi pada urutan ekstraksi. Misalkan u adalah simpul pertama yang diekstrak yang dist[u]-nya BELUM menjadi jarak terpendek sebenarnya. Karena lintasan terpendek sebenarnya dari s ke u pasti meninggalkan himpunan S yang sudah difinalisasi pada suatu titik — menyeberang ke suatu simpul y di luar S melalui sisi (x, y) dengan x di dalam S — dan karena setiap bobot sisi non-negatif, jarak parsial ke y sepanjang lintasan terpendek sebenarnya itu tidak bisa lebih kecil dari dist[u] (karena jika tidak, y sudah akan diekstrak sebelum u, bertentangan dengan u yang diekstrak berikutnya). Namun dist[y] sudah direlaksasi dengan benar ketika x difinalisasi, sehingga dist[y] paling banyak sama dengan jarak parsial itu — dan SISA lintasan terpendek sebenarnya dari y ke u hanya bisa menambahkan bobot non-negatif, sehingga jarak sebenarnya ke u paling sedikit sama dengan dist[y], yang paling sedikit... rantai ketidaksamaan ini memaksa dist[u] untuk sudah sama dengan jarak terpendek sebenarnya, bertentangan dengan

asumsi. Di sinilah tepatnya bobot NON-NEGATIF menjadi esensial: langkah "sisa lintasan hanya bisa menambahkan bobot non-negatif" adalah yang menjamin dist[u] tidak bisa nantinya dikalahkan oleh suatu lintasan yang belum dijelajahi Dijkstra.

> **Mengapa satu sisi negatif saja bisa merusak Dijkstra secara diam-diam**
> Jika bahkan SATU sisi dalam graf memiliki bobot negatif, algoritma Dijkstra tidak akan crash atau melempar error — algoritma ini hanya menghitung jarak yang SALAH, secara diam-diam, tanpa indikasi apa pun yang salah. Klaim greedy "begitu diekstrak, jarak sebuah simpul sudah final" sepenuhnya bergantung pada semua relaksasi di masa depan yang hanya bisa MENAIKKAN biaya suatu lintasan, tidak pernah menurunkannya. Sisi negatif melanggar asumsi itu, dan Bagian 13.5 menunjukkan persis graf tempat kegagalan ini terjadi.

13.5 Mengapa Dijkstra Gagal pada Bobot Negatif

Gambar 13.4 menggunakan kembali lima hub skenario hipotetis Fase-4 Profitina yang sama, tetapi dengan himpunan sisi berarah yang berbeda — salah satunya, BDG→MKS, berbobot −4. Graf ini dimaksudkan untuk memodelkan skenario yang realistis: BDG→MKS merepresentasikan kredit bandwidth promosi yang MENGURANGI biaya sinkronisasi efektif antara kedua hub tersebut, bukan jarak negatif secara harfiah. Apa pun makna dunia nyatanya, bobot sisi negatif persis merupakan kondisi yang dilarang oleh bukti kebenaran Dijkstra (Bagian 13.4).



Gambar 13.4 — Lima hub yang sama, tetapi dengan sisi berarah yang berbeda — termasuk sisi NEGATIF, BDG→MKS (−4), disorot bersama dua sisi lain yang berinteraksi dengannya. Aturan greedy finalisasi-dan-tidak-pernah-kunjungi-ulang Dijkstra rusak pada graf ini.

Untuk melihat kegagalan ini secara konkret: jika algoritma Dijkstra dijalankan pada Gambar 13.4 dari JKT, algoritma ini akan pertama-tama mengekstrak JKT (dist=0), lalu SBY (dist=7, yang lebih kecil dari

dua tawaran langsung JKT), lalu memfinalisasi simpul mana pun yang memiliki jarak tentatif terkecil berikutnya — dan begitu sebuah simpul difinalisasi, Dijkstra TIDAK PERNAH mengunjunginya kembali, bahkan jika sisi yang ditemukan belakangan (seperti BDG→MKS yang negatif) akan menawarkan rute yang lebih murah melaluinya. Jarak terpendek sebenarnya untuk graf ini — seperti yang akan diturunkan secara ketat di Bagian 13.6 — adalah dist[JKT]=0, dist[BDG]=2, dist[MDN]=4, dist[SBY]=7, dist[MKS]=−2 (diverifikasi di /tmp/daa_pilot/verify_ch13_problems.py terhadap pencarian brute-force maupun jawaban CLRS yang terpublikasi untuk graf ini). Aturan greedy finalisasi-sekali Dijkstra tidak dapat mencapai dist[MKS]=−2, karena hal itu mengharuskan MENGUNJUNGI ULANG suatu jarak ke arah yang lebih rendah setelah MKS sudah difinalisasi sementara pada nilai yang lebih tinggi (salah).

13.6 Algoritma Bellman-Ford: Relaksasi Semuanya, Berulang Kali

Bellman-Ford sepenuhnya meninggalkan strategi antrean-prioritas greedy Dijkstra. Sebagai gantinya, algoritma ini merelaksasi SETIAP sisi dalam graf, dalam urutan tetap apa pun, dan mengulangi putaran penuh ini persis |V| − 1 kali. Ini adalah asumsi yang secara ketat lebih lemah dibanding Dijkstra — algoritma ini tidak pernah mengasumsikan jarak simpul mana pun sudah final hingga semua |V| − 1 putaran selesai — yang justru menjadi alasan mengapa algoritma ini mentolerir sisi negatif.

```
BELLMAN-FORD(V, E, w, s):
  for each vertex v in V
      dist[v] = infinity;  parent[v] = NIL
  dist[s] = 0
  repeat |V| - 1 times:
      for each edge (u, v) in E
          RELAX(u, v, w)
  for each edge (u, v) in E        // putaran ekstra: cek
      if dist[u] + w(u,v) < dist[v]  // siklus berbobot negatif
          return "negative cycle detected"
  return dist[], parent[]
```

Mengapa |V| − 1 putaran sudah cukup: lintasan terpendek mana pun dalam graf tanpa siklus negatif melewati paling banyak |V| − 1 sisi berbeda (lintasan dengan lebih banyak sisi dari itu harus mengulang sebuah simpul, membentuk siklus, yang hanya bisa membuat lintasan lebih panjang atau sama — tidak pernah secara ketat lebih pendek — kecuali siklus itu negatif). Setiap putaran penuh atas semua sisi dijamin memperluas dengan benar himpunan simpul yang JARAKnya terpendek (dalam jumlah sisi dari s) paling banyak sama dengan nomor putaran itu. Setelah |V| − 1 putaran, setiap lintasan terpendek dengan |V| − 1 sisi atau kurang karenanya sudah ditemukan dengan benar.

Menjalankan Bellman-Ford pada Gambar 13.4 dari JKT menghasilkan penelusuran bertahap (pass-by-pass) pada Gambar 13.5 — diverifikasi secara independen dan manual di /tmp/daa_pilot/verify_ch13_problems.py terhadap pencarian brute-force dan jawaban CLRS yang terpublikasi untuk graf yang persis sama ini, lalu disilangperiksa terhadap keluaran hasil kompilasi 02_bellman_ford.cpp pada Lampiran 13.A.

Jejak Bellman-Ford: Merelaksasi Setiap Sisi, |V|−1 Kali

Berbeda dari Dijkstra, Bellman-Ford tidak memilih urutan — ia merelaksasi SEMUA 10 sisi, sekali per pass, selama |V|−1=4 pass. Perhatikan dist[MKS] baru mencapai nilai final (−2) di pass ke-3. Diverifikasi terhadap /tmp/daa_pilot/code/chapter13/02_bellman_ford.cpp.

Pass	dist[JKT, BDG, MDN, SBY, MKS]	Yang Berubah
1	0, 6, 4, 7, 2	JKT merelaksasi BDG & SBY; BDG merelaksasi MDN, SBY, MKS
2	0, 2, 4, 7, 2	MDN→BDG(−2) menurunkan dist[BDG] dari 6 ke 2
3	0, 2, 4, 7, −2	BDG→MKS(−4) dengan dist[BDG]=2 yang BARU menurunkan dist[MKS] ke −2
4	0, 2, 4, 7, −2	tidak ada perubahan — konvergen, aman berhenti lebih awal

Gambar 13.5 — Cuplikan dist[] Bellman-Ford setelah masing-masing dari empat putarannya (|V|−1 = 4 untuk graf lima-hub ini). Perhatikan dist[MKS] baru mencapai nilai final sebenarnya −2 pada putaran ke-3.

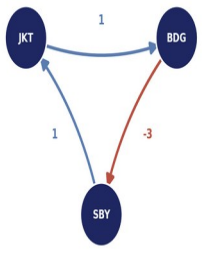
Menelusuri data Gambar 13.5: putaran 1 merelaksasi dua sisi langsung JKT (dist[BDG]=6, dist[SBY]=7) dan kemudian, dalam putaran yang sama, sisi keluar BDG merelaksasi MDN (dist=4), SBY tetap 7 (tawaran BDG sebesar 8 lebih buruk), dan MKS (dist=2, via BDG→MKS yang −4, yaitu 6 + (−4) = 2). Putaran 2 menemukan bahwa MDN→BDG (bobot −2) menawarkan rute yang lebih murah ke BDG dibanding sisi langsung JKT — dist[BDG] turun dari 6 menjadi 2. Putaran 3 adalah saat efek penuh sisi negatif muncul: sekarang dist[BDG]=2 (diperbarui pada putaran 2), merelaksasi BDG→MKS lagi memberikan 2 + (−4) = −2, secara ketat lebih baik dari dist[MKS]=2 pada putaran 1 — dist[MKS] turun menjadi −2, nilai finalnya yang benar. Putaran 4 tidak mengubah apa pun, mengonfirmasi konvergensi. Penundaan tiga-putaran ini — jarak SEBENARNYA MKS tidak terlihat hingga putaran KETIGA, meskipun BDG→MKS sudah direlaksasi pada putaran 1 — persis merupakan perilaku yang tidak bisa direproduksi oleh finalisasi greedy sekali-jalan Dijkstra, dan persis alasan mengapa Bellman-Ford membutuhkan SELURUH |V| − 1 putaran alih-alih berhenti lebih awal berdasarkan dugaan.

13.7 Mendeteksi Siklus Berbobot Negatif

Pengaman terakhir Bellman-Ford adalah putaran EKSTRA ke-|V|-nya: setelah |V| − 1 putaran biasa, jika ADA sisi mana pun yang masih bisa direlaksasi, siklus berbobot negatif yang dapat dijangkau dari sumber pastilah ada, dan tidak ada jarak lintasan-terpendek yang terdefinisi dengan baik yang bisa dilaporkan untuk simpul-simpul yang disentuh siklus itu. Gambar 13.6 menunjukkan contoh terkecil yang mungkin: sebuah loop 3-hub yang tiga bobot sisinya berjumlah total negatif.

Mengapa Pass Tambahan Penting: Mendeteksi Siklus Negatif

Loop kecil 3-hub JKT→BDG→SBY→JKT dengan bobot 1, −3, 1 berjumlah −1 — siklus yang bisa diputar selamanya utk membuat "jarak" negatif tanpa batas. Pass ke-|V| Bellman-Ford menangkap persis ini: kalau ADA sisi yang masih bisa direlaksasi setelah |V|−1 pass, siklus negatif pasti ada.



1 + (−3) + 1 = −1

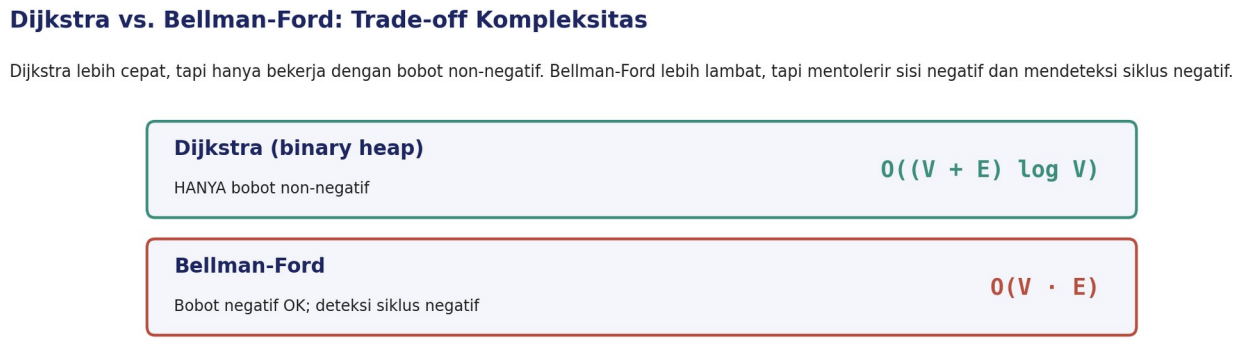
Gambar 13.6 — Siklus berbobot negatif yang kecil: JKT→BDG→SBY→JKT dengan bobot 1, −3, 1, berjumlah −1. Melingkari siklus ini berulang kali akan membuat total bobot suatu lintasan sembarang negatifnya, sehingga tidak ada "jarak terpendek" berhingga yang ada untuk simpul mana pun di dalamnya.

Pada graf 3-simpul ini, |V| − 1 = 2 putaran biasa Bellman-Ford konvergen ke dist[JKT]=−2, dist[BDG]=0, dist[SBY]=−3 — tetapi putaran ekstra ke-|V| (ke-3) menemukan bahwa sisi JKT→BDG MASIH bisa direlaksasi (dist[JKT] + 1 = −1, secara ketat lebih kecil dari dist[BDG]=0 yang seharusnya sudah konvergen — sebuah perbaikan yang seharusnya mustahil begitu graf benar-benar mengendap), yang secara benar melaporkan siklus negatif alih-alih himpunan jarak final yang (tidak bermakna). Diverifikasi di /tmp/daa_pilot/verify_ch13_problems.py dan disilangperiksa terhadap keluaran hasil kompilasi Lampiran 13.A, yang melaporkan "Negative-weight cycle detected? YES" untuk masukan yang persis sama ini.

> **Dijkstra tidak memiliki pengaman setara, karena tidak membutuhkannya**
> Algoritma Dijkstra tidak pernah memeriksa siklus negatif karena prasyarat bobot-non-negatifnya membuat siklus semacam itu MUSTAHIL secara konstruksi — sebuah siklus yang seluruhnya terdiri dari sisi non-negatif tidak akan pernah berjumlah total negatif. Putaran ekstra Bellman-Ford ada persis karena algoritma ini menghilangkan prasyarat tersebut, sehingga harus secara aktif menjaga terhadap satu skenario yang baru diizinkan oleh asumsinya yang lebih lemah.

13.8 Menganalisis dan Membandingkan Waktu Jalan

Waktu jalan Dijkstra, dengan antrean prioritas binary min-heap (persis seperti algoritma Prim di Bab 11), adalah O((V + E) log V) — |V| pemanggilan EXTRACT-MIN berbiaya O(V log V), dan O(E) relaksasi yang memicu DECREASE-KEY berbiaya O(E log V). Bellman-Ford sama sekali tidak menggunakan antrean prioritas: algoritma ini hanya melakukan |V| − 1 putaran penuh, masing-masing merelaksasi semua E sisi, dengan total O(V · E).



Gambar 13.7 — Dijkstra secara asimtotik lebih cepat, tetapi hanya benar untuk bobot non-negatif. Bellman-Ford lebih lambat, tetapi benar kapan pun tidak ada siklus negatif yang dapat dijangkau dari sumber, dan algoritma ini mendeteksinya jika ada.

Memilih di antara keduanya adalah keputusan kebenaran, bukan keputusan kecepatan
Berbeda dari pilihan Kruskal-versus-Prim di Bab 11 (yang murni soal kepadatan graf, karena keduanya selalu benar), pilihan antara Dijkstra dan Bellman-Ford pertama-tama dan terutama soal apakah graf itu BISA mengandung bobot sisi negatif. Jika setiap bobot dijamin non-negatif, Dijkstra secara ketat lebih disukai karena waktu jalan asimtotiknya yang lebih baik. Jika bobot negatif memungkinkan — dan terlebih jika SIKLUS negatif harus dideteksi alih-alih diasumsikan tidak ada — Bellman-Ford adalah satu-satunya pilihan yang benar, terlepas dari waktu jalannya yang lebih lambat.

13.9 Profitina dalam Praktik: Latensi, Kredit, dan Keamanan Siklus

Tentang kotak ini
Seperti pada setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) Indonesia nyata yang dibangun oleh penulis, tanpa mengungkap detail implementasi yang bersifat proprietary.

Jaringan multi-hub pada Gambar 13.2 BUKAN yang dijalankan Profitina hari ini — produk aslinya adalah satu server self-hosted tunggal, bukan penyebaran regional terdistribusi (lihat catatan di Bab 10) — tetapi roadmap Profitina sendiri memang menyebut jaringan sinkronisasi data regional semacam ini sebagai tahap berikutnya (Fase 4, setelah dominasi domestik tercapai), sehingga skenario ini adalah pratinjau yang berdasar, bukan skenario sembarangan. Andai jaringan semacam itu ada, biasanya ia hanya akan membawa bobot latensi non-negatif — keterlambatan jaringan fisik tidak bisa negatif — sehingga menjadikan algoritma Dijkstra sebagai pilihan default alami untuk merutekan lalu lintas sinkronisasi sepanjang lintasan termurah yang tersedia. Namun skenario kredit-promosi di Bagian 13.5 mengilustrasikan pola rekayasa yang nyata dan umum: lapisan perutean berbasis biaya (berbeda dari perutean berbasis latensi murni) secara sah bisa memperkenalkan bobot EFEKTIF negatif, misalnya ketika suatu perjanjian kemitraan sementara mendiskon lalu lintas yang dirutekan melalui pasangan hub tertentu di bawah biaya dasarnya. Kapan pun bobot sisi suatu graf perutean bisa berasal dari model biaya alih-alih model latensi-fisik murni, toleransi Bellman-Ford terhadap sisi negatif — serta kemampuannya mendeteksi siklus negatif dengan aman alih-alih secara diam-diam merutekan lalu lintas secara salah melalui sebuah loop yang semakin murah — menjadi pilihan rekayasa yang lebih dapat dipertahankan, bahkan dengan harga waktu jalan O(V·E)-nya yang lebih lambat.

13.10 Ringkasan Bab dan Poin-Poin Kunci

- Lintasan terpendek sumber-tunggal menanyakan pertanyaan yang berbeda dari MST: RUTE termurah dari satu sumber ke setiap tujuan, bukan cara termurah menghubungkan semuanya secara agregat — dan kedua masalah itu bisa menghasilkan pohon yang sama sekali berbeda pada graf yang sama.

- Setiap algoritma dalam bab ini dibangun dari operasi RELAXATION yang sama: diberikan sisi (u, v), perbarui dist[v] dan parent[v] kapan pun melewati u menawarkan rute yang secara ketat lebih murah.
- Algoritma Dijkstra secara greedy memfinalisasi satu simpul setiap kali melalui antrean prioritas minimum, persis seperti algoritma Prim di Bab 11 — tetapi kebenarannya sepenuhnya mensyaratkan semua bobot sisi non-negatif.
- Satu bobot sisi negatif saja bisa membuat Dijkstra secara diam-diam menghitung jarak yang SALAH, tanpa error atau peringatan — Bagian 13.5 menyusun persis kegagalan ini pada jaringan lima-hub.
- Bellman-Ford merelaksasi setiap sisi, |V| − 1 kali, tanpa membuat asumsi apa pun tentang urutan finalisasi — yang menjadi alasan mengapa algoritma ini mentolerir sisi negatif, dengan harga waktu jalan O(V·E) yang lebih lambat dibanding O((V+E) log V) milik Dijkstra.
- Putaran ekstra ke-|V| Bellman-Ford mendeteksi siklus berbobot negatif yang dapat dijangkau dari sumber — kasus di mana tidak ada jarak lintasan-terpendek yang terdefinisi dengan baik yang bisa ada sama sekali — sesuatu yang tidak dimiliki atau bahkan tidak dibutuhkan mekanismenya oleh Dijkstra, karena prasyarat non-negatifnya meniadakan siklus negatif secara konstruksi.

> ***"Dijkstra percaya jalan termurah akan tetap termurah selamanya. Bellman-Ford tidak percaya apa pun — ia mengecek ulang setiap jalan, berulang-ulang, sampai yakin."***
>
> **— aforisme mata kuliah ini**
>
> *(Kecurigaan ekstra itu yang membuatnya bertahan pada sisi negatif — dan menangkap siklus negatif tanpa ampun.)*

Kuliah 14 beralih ke ANALISIS AMORTISASI (amortized analysis) — sebuah teknik untuk membatasi total biaya suatu URUTAN operasi bahkan ketika operasi individual sesekali berbiaya jauh lebih mahal dari rata-rata, menggunakan persis jenis struktur data antrean-prioritas dan array yang sudah diandalkan bab ini dan Bab 11.

13.11 Soal Latihan

Soal-soal ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Telusuri algoritma Dijkstra pada Gambar 13.2 dimulai dari simpul yang BERBEDA — katakanlah, MKS alih-alih JKT. Hub mana pun, jika ada, yang menjadi tidak terjangkau, dan mengapa?
2. Jelaskan dengan kata-kata Anda sendiri mengapa bukti kebenaran Dijkstra (Bagian 13.4) runtuh persis di kalimat yang mengasumsikan "sisa lintasan hanya bisa menambahkan bobot non-negatif." Apa yang perlu benar sebagai gantinya agar bukti itu bertahan terhadap bobot negatif?

3. Bagian 13.6 mengklaim bahwa lintasan terpendek mana pun dalam graf tanpa siklus negatif melewati paling banyak |V| − 1 sisi. Buktikan klaim ini, dan jelaskan apa yang akan salah dengan batas itu jika ada siklus negatif.
4. Menggunakan penelusuran bertahap Gambar 13.5, identifikasi putaran PALING AWAL di mana nilai dist[] setiap simpul pertama kali mencapai jawaban final yang benar. Apakah ini putaran yang sama untuk setiap simpul? Mengapa atau mengapa tidak?
5. Seorang rekan menyarankan menjalankan algoritma Dijkstra pada graf berbobot negatif, tetapi cukup menambahkan konstanta yang cukup besar ke setiap bobot sisi terlebih dahulu agar semuanya non-negatif, lalu menjalankan Dijkstra secara normal. Apakah trik ini menghasilkan lintasan terpendek yang benar? Justifikasi jawaban Anda (petunjuk: pertimbangkan lintasan dengan jumlah sisi yang berbeda).
6. Modifikasi deteksi siklus-negatif Bellman-Ford (Bagian 13.7) sehingga tidak hanya melaporkan BAHWA siklus negatif ada, tetapi juga MEREKONSTRUKSI urutan simpul salah satu siklus tersebut. Sketsakan pendekatan Anda.

Lampiran 13.A — Log Verifikasi untuk Kode Sumber Bab 13

Seperti pada bab-bab sebelumnya, setiap program yang disertakan dalam buku ini dikompilasi dan dijalankan terhadap contoh yang telah diverifikasi secara manual sebelum disertakan. Baik 01_dijkstra.cpp maupun 02_bellman_ford.cpp dikompilasi dengan `g++ -O2 -std=c++17 -Wall` (nol peringatan, nol error); keluarannya cocok baik dengan verifikasi Python independen di /tmp/daa_pilot/verify_ch13_problems.py maupun jawaban referensi CLRS yang terpublikasi untuk setiap graf, secara persis.

```
$ ./01_dijkstra
Dijkstra trace (source = JKT):
  Step 1: finalize JKT (dist=0)
  Step 2: finalize SBY (dist=5)
  Step 3: finalize MKS (dist=7)
  Step 4: finalize BDG (dist=8)
  Step 5: finalize MDN (dist=9)

Final shortest distances from JKT:
  dist[JKT] = 0
  dist[BDG] = 8   (via SBY)
  dist[SBY] = 5   (via JKT)
  dist[MDN] = 9   (via BDG)
  dist[MKS] = 7   (via SBY)
                       # matches CLRS Fig 22.6 exactly
```

```
$ ./02_bellman_ford
Bellman-Ford trace (source = JKT):
  After pass 1: JKT=0 BDG=6 MDN=4 SBY=7 MKS=2
  After pass 2: JKT=0 BDG=2 MDN=4 SBY=7 MKS=2
  After pass 3: JKT=0 BDG=2 MDN=4 SBY=7 MKS=-2
  After pass 4: (no change -- converged early)

Final shortest distances from JKT:
  dist[MKS] = -2   (via BDG)
Negative-weight cycle detected? NO

--- Negative-cycle demo (JKT->BDG->SBY->JKT) ---
Negative-weight cycle detected? YES (expected YES)
```

Latihan SPOJ — buku pendamping
Teknik bab ini diuji sungguhan di juri daring SPOJ. Buka Bab 11 buku pendamping “SPOJ Competitive Programming” (folder SPOJ, BI & EN) untuk implementasi jalur terpendek (Dijkstra, Bellman-Ford) yang disetel demi verdict — lengkap dengan tips, trik, dan hint agar solusi Anda Accepted, tanpa membocorkan jawaban.

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Chapter 22, Single-Source Shortest Paths, Sections 22.2–22.4.)

[2] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs." Numerische Mathematik, 1(1), 1959, 269–271. (Makalah aslinya.)

[3] R. Bellman. "On a Routing Problem." Quarterly of Applied Mathematics, 16, 1958, 87–90.

[4] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, 4th ed. Lulu Press, 2020–2023.

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 14

Analisis Amortisasi

Kasus terburuk satu operasi bisa terlihat mahal — tetapi sepanjang sebuah URUTAN operasi, kemahalan itu sering kali terhapus. Tiga metode untuk membuktikan batas ketat pada total: agregat, akuntansi, dan potensial.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Menjelaskan mengapa membatasi biaya kasus-terburuk satu operasi lalu mengalikannya dengan n bisa secara drastis MELEBIH-LEBIHKAN biaya sebenarnya suatu urutan n operasi. (2) Menerapkan metode AGREGAT untuk membatasi total biaya suatu urutan penyisipan tabel-dinamis-berlipat-ganda (dynamic-table-doubling). (3) Menerapkan metode AKUNTANSI pada sebuah stack yang mendukung PUSH dan MULTIPOP, menetapkan biaya tetap per operasi dan membuktikan saldo kredit yang ditabung tidak pernah negatif. (4) Menerapkan metode POTENSIAL pada increment binary counter, mendefinisikan fungsi potensial, dan membuktikan biaya amortisasi setiap operasi melalui jumlah teleskopik (telescoping sum). (5) Menjelaskan mengapa ketiga metode, meskipun mekanismenya sangat berbeda, selalu konvergen ke batas amortisasi yang SAMA untuk suatu masalah. (6) Membedakan analisis amortisasi (jaminan tentang setiap urutan operasi yang mungkin) dari analisis kasus-rata-rata (klaim tentang distribusi probabilitas atas input).

14.1 Rekap: Dari Biaya Satu Operasi ke Total Suatu Urutan

Setiap algoritma yang dipelajari sejauh ini dalam mata kuliah ini dianalisis berdasarkan waktu jalan kasus-terburuknya sendiri: MERGE-SORT berbiaya O(n log n), algoritma Dijkstra berbiaya O((V+E) log V), dan seterusnya — satu algoritma, satu kali jalan, satu batas. Bab ini mengajukan pertanyaan yang berbeda. Beberapa STRUKTUR DATA mendukung suatu urutan operasi di mana sebagian besar pemanggilan murah tetapi sesekali ada pemanggilan yang mahal — sebuah array dinamis yang biasanya hanya menambahkan elemen, tetapi sesekali harus melipatgandakan kapasitasnya dan menyalin semuanya; sebuah stack yang biasanya hanya mengeluarkan satu elemen, tetapi sesekali harus mengeluarkan BANYAK elemen sekaligus. Jika Anda membatasi setiap operasi individual dengan kasus terburuknya lalu mengalikannya dengan n, Anda mendapatkan total yang sangat PESIMISTIS — kenyataannya adalah operasi-operasi mahal itu cukup jarang, dan cukup terstruktur, sehingga URUTAN secara keseluruhan berbiaya jauh lebih murah daripada n kali biaya operasi tunggal terburuk.

Gambar 14.1 — Posisi bab ini dalam peta jalan enam belas kuliah DAA. Kuliah 14 mengikuti unit algoritma-graf dan memperkenalkan sebuah ALAT analitis baru, bukan domain masalah baru.

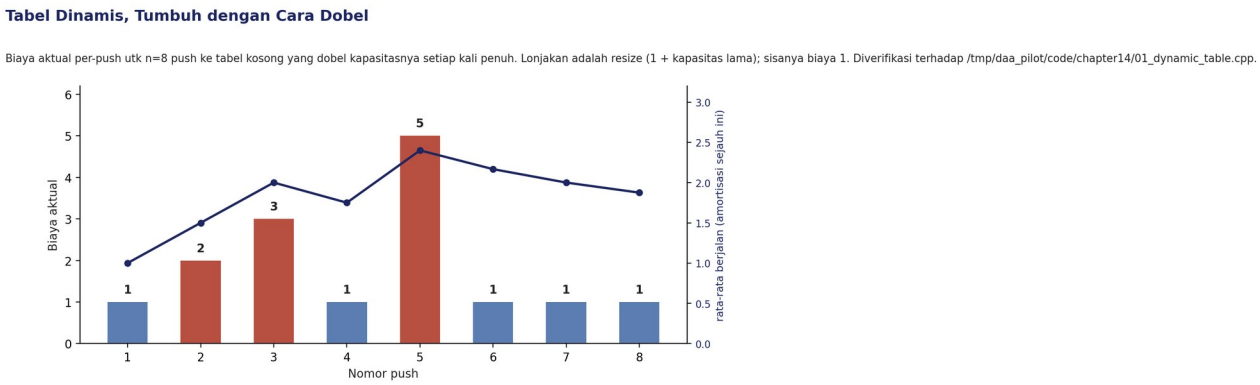
> **Analisis amortisasi membatasi sebuah URUTAN, bukan satu pemanggilan**
> ANALISIS AMORTISASI membuktikan batas biaya rata-rata per operasi atas urutan n operasi mana pun, dijamin untuk setiap urutan yang mungkin — bukan sekadar rata-rata atas input acak. Ini adalah perbedaan krusial dari analisis kasus-rata-rata (Bab 2), yang mengasumsikan distribusi probabilitas atas input dan bisa dikalahkan oleh input adversarial. Batas amortisasi berlaku untuk URUTAN operasi terburuk yang mungkin, yang menjadikannya jaminan yang ketat dan tahan-adversarial, bukan sekadar harapan statistik.

14.2 Metode Agregat: Tabel Dinamis Berlipat Ganda

Metode AGREGAT adalah yang paling sederhana dari ketiganya: batasi biaya TOTAL urutan n operasi mana pun secara langsung, lalu bagi dengan n untuk mendapatkan biaya amortisasi per operasi. Perhatikan sebuah tabel dinamis yang dimulai kosong dan mendukung PUSH: jika tabel penuh, pertama-tama alokasikan tabel baru dengan kapasitas DUA KALI LIPAT kapasitas saat ini, salin setiap elemen yang sudah ada, dan baru kemudian sisipkan elemen baru.

```
TABLE-INSERT(T, x):
  if T.size == T.capacity
      new_capacity = max(1, 2 * T.capacity)
      alokasikan NEW-TABLE dengan kapasitas = new_capacity
      salin semua T.size elemen yang ada ke NEW-TABLE
      T = NEW-TABLE
  T[T.size] = x;  T.size = T.size + 1
```

Satu push yang memicu resize berbiaya 1 + (kapasitas lama) — 1 untuk elemen baru, ditambah satu unit kerja per elemen yang disalin. Dalam kasus terburuk, biaya resize itu bisa sebesar seluruh tabel, yaitu O(n) untuk push ke-n — sehingga mengalikan kasus terburuk satu push dengan n operasi akan menyarankan total $O(n^2)$. Gambar 14.2 menunjukkan biaya AKTUAL per push untuk 8 push dimulai dari tabel kosong, diverifikasi secara manual di /tmp/daa_pilot/verify_ch14_problems.py dan disilangperiksa terhadap 01_dynamic_table.cpp terkompilasi pada Lampiran 14.A: biaya 1, 2, 3, 1, 5, 1, 1, 1, dengan total berjalan 15 — bukan 1+2+3+4+5+6+7+8 = 36 yang akan disarankan oleh batas naif kasus-terburuk-per-push, dan sama sekali jauh dari ledakan kuadratik.



Gambar 14.2 — Biaya aktual setiap dari 8 push ke dalam tabel dinamis berlipat ganda (batang, sumbu kiri), dan biaya amortisasi berjalan sejauh ini (garis, sumbu kanan). Resize (push 1, 2, 3, 5) adalah batang yang tinggi; garis amortisasi mengendap ke sekitar 2 seiring bertambahnya n.

Mengapa total tetap linear: resize hanya terjadi ketika kapasitas tabel berlipat ganda, sehingga sepanjang n push terdapat paling banyak floor(log2 n) resize, pada kapasitas 1, 2, 4, 8, ..., kira-kira n/2. TOTAL kerja penyalinan sepanjang setiap resize karenanya paling banyak 1 + 2 + 4 + ... + n/2 < n — deret geometri yang dibatasi oleh SUKU TERAKHIR, bukan jumlah sukunya. Menambahkan n biaya penyisipan biasa (1 unit masing-masing), total biaya n push dibatasi oleh n + n = 2n dalam analisis klasik CLRS, atau lebih tepatnya di bawah 3n setelah biaya penyisipan-elemen +1 milik setiap resize dilipat masuk (cocok dengan total 15 pada penelusuran C++ untuk n=8, karena 15 / 8 = 1,875 < 3). Membagi total O(n) ini dengan n operasi memberikan biaya AMORTISASI O(1) per push, meskipun push TUNGGAL terburuk berbiaya O(n).

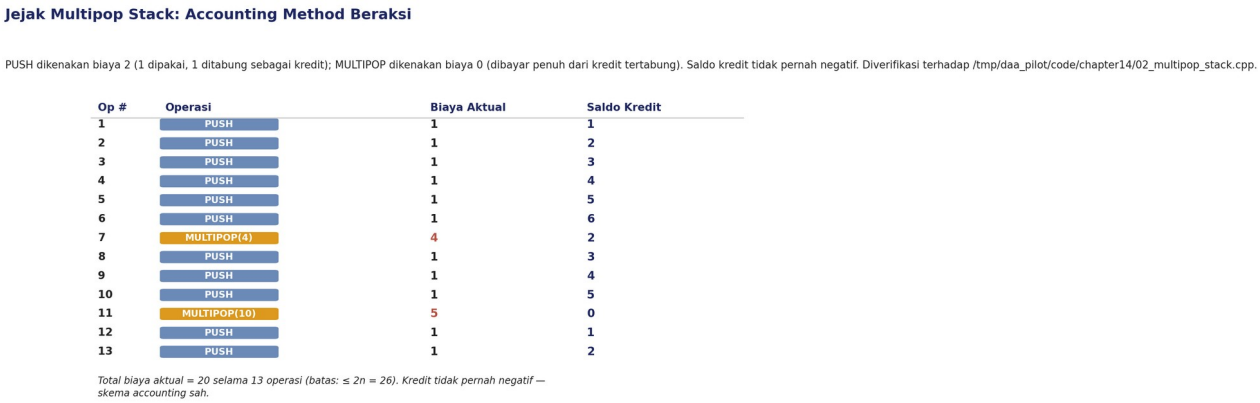
> **Metode agregat membuktikan batas pada TOTAL, bukan pada satu operasi**
> Akan keliru untuk menyimpulkan dari Gambar 14.2 bahwa "setiap push berbiaya O(1)" — push ke-5 jelas berbiaya 5 unit, bukan O(1). Klaim metode agregat murni tentang JUMLAH atas keseluruhan urutan: total biaya / n operasi adalah O(1), yang merupakan pernyataan tentang RATA-RATA urutan, dijamin untuk setiap urutan push yang mungkin — bukan pernah klaim bahwa satu push individual itu murah.

14.3 Metode Akuntansi: PUSH dan MULTIPOP pada Stack

Metode AKUNTANSI menetapkan setiap operasi sebuah BIAYA AMORTISASI tetap, yang bisa berbeda dari biaya aktualnya — operasi yang dibebankan lebih dari biaya aktualnya menabung surplusnya sebagai KREDIT pada struktur data, dan operasi yang berbiaya lebih dari bebannya harus membayar selisihnya dari kredit yang sudah ditabung. Skema ini hanya sah jika total kredit yang ditabung tidak pernah negatif, karena kredit merepresentasikan kerja nyata yang sudah dibayar.

```
PUSH(S, x):
    S.top = S.top + 1
    S[S.top] = x

MULTIPOP(S, k):
    while S.top != 0 and k != 0
        keluarkan satu elemen dari S
        k = k - 1
```

Perhatikan sebuah stack yang mendukung PUSH (biaya aktual 1) dan MULTIPOP(k) (biaya aktual min(k, ukuran saat ini), karena tidak bisa mengeluarkan lebih banyak elemen daripada yang dimiliki stack). Bebankan 2 untuk setiap PUSH — 1 untuk membayar push itu sendiri, dan 1 ditabung sebagai kredit PADA elemen yang baru dimasukkan, dialokasikan untuk membayar pengeluaran elemen yang SAMA nantinya. Bebankan 0 untuk MULTIPOP: setiap elemen yang dikeluarkannya sudah memiliki tepat 1 unit kredit yang ditabung sejak dimasukkan, sehingga MULTIPOP sepenuhnya dibayar dari kredit yang ditabung, tidak pernah dari kantong sendiri.



Jejak Multipop Stack: Accounting Method Beraksi

PUSH dikenakan biaya 2 (1 dipakai, 1 ditabung sebagai kredit); MULTIPOP dikenakan biaya 0 (dibayar penuh dari kredit tertabung). Saldo kredit tidak pernah negatif. Diverifikasi terhadap /tmp/daa_pilot/code/chapter14/02_multipop_stack.cpp.

Op #	Operasi	Biaya Aktual	Saldo Kredit
1	PUSH	1	1
2	PUSH	1	2
3	PUSH	1	3
4	PUSH	1	4
5	PUSH	1	5
6	PUSH	1	6
7	MULTIPOP(4)	4	2
8	PUSH	1	3
9	PUSH	1	4
10	PUSH	1	5
11	MULTIPOP(10)	5	0
12	PUSH	1	1
13	PUSH	1	2

Total biaya aktual = 20 selama 13 operasi (batas: ≤ 2n = 26). Kredit tidak pernah negatif — skema accounting sah.

Gambar 14.3 — Penelusuran 13 operasi: 6 push, MULTIPOP(4), 3 push, MULTIPOP(10), 2 push. Saldo kredit (kolom kanan) tidak pernah negatif, meskipun MULTIPOP(10) mencoba mengeluarkan jauh lebih banyak daripada 5 elemen yang sebenarnya ada.

Menelusuri data Gambar 14.3 (diverifikasi secara manual di /tmp/daa_pilot/verify_ch14_problems.py dan disilangperiksa terhadap 02_multipop_stack.cpp terkompilasi pada Lampiran 14.A): 6 push pertama menabung 6 kredit. MULTIPOP(4) mengeluarkan 4 elemen, menghabiskan 4 dari 6 kredit itu, menyisakan 2. Tiga push lagi menabung 3 kredit lagi, untuk saldo 5. MULTIPOP(10) diminta mengeluarkan 10 elemen tetapi hanya 5 yang ada, sehingga biaya AKTUALnya adalah min(10, 5) = 5 — menghabiskan tepat 5 kredit yang ditabung, menyisakan 0. Dua push terakhir membawa saldo menjadi 2. Total biaya aktual sepanjang seluruh 13 operasi adalah 1+1+1+1+1+1+4+1+1+1+5+1+1 = 20, jauh di bawah batas terjamin metode akuntansi sebesar 2 × 13 = 26 — dan saldo kredit tidak pernah negatif di titik mana pun, yang persis merupakan hal yang mensahkan skema pembebanan ini.

Mengapa invarian kredit membuktikan batas O(1)-amortisasi

Karena setiap elemen yang pernah dimasukkan dibebankan tepat 2, dan n push bisa menabung paling banyak total 2n kredit, TOTAL biaya aktual urutan n operasi mana pun (push dan multipop bersama-sama) tidak akan pernah melebihi total yang dibebankan — 2n — persis karena MULTIPOP hanya pernah menghabiskan kredit yang sudah ditabung oleh suatu PUSH sebelumnya, dan sebuah stack tidak bisa mengeluarkan lebih banyak elemen daripada yang sebelumnya dimasukkan. Saldo kredit yang tidak pernah turun di bawah nol bukanlah kebetulan yang diverifikasi setelah fakta; itu adalah mekanisme yang MEMAKSA batas O(1)-amortisasi berlaku untuk setiap urutan yang mungkin, termasuk yang direkayasa untuk memaksimalkan pemanggilan MULTIPOP.

14.4 Metode Potensial: Increment Binary Counter

Metode POTENSIAL menggeneralisasi gagasan "kredit yang ditabung" milik metode akuntansi menjadi satu fungsi skalar Phi, yang memetakan KEADAAN SAAT INI struktur data ke sebuah bilangan non-negatif — dalam artian tertentu, energi potensialnya. Biaya amortisasi operasi ke-i didefinisikan sebagai: amortisasi_i = aktual_i + Phi(keadaan_i) − Phi(keadaan_{i-1}). Menjumlahkan atas keseluruhan urutan n operasi, suku-suku Phi ber-TELESKOP: jumlah amortisasi_i = jumlah aktual_i + Phi(keadaan_n) − Phi(keadaan_0). Jika Phi tidak pernah turun di bawah Phi(keadaan_0) — paling sederhana, jika Phi(keadaan_0) = 0 dan Phi selalu non-negatif — maka jumlah biaya aktual dibatasi dari atas oleh jumlah biaya amortisasi, yang persis merupakan jaminan yang dibutuhkan analisis amortisasi.

```
INCREMENT(A):   // A adalah counter biner k-bit, A[0] = bit paling kecil
    i = 0
    while i < length(A) and A[i] == 1
        A[i] = 0
        i = i + 1
    if i < length(A)
        A[i] = 1
```

Definisikan Phi(counter) = jumlah bit-1 yang sedang aktif. INCREMENT membalik t bit-1 berurutan menjadi 0 (biaya aktual t, satu pembalikan per bit) lalu membalik tepat satu bit-0 menjadi 1 (biaya aktual 1) — sehingga total biaya aktualnya adalah t + 1, dan potensinya berubah sebesar (jumlah_satu_sebelum − t + 1) − jumlah_satu_sebelum = 1 − t. Biaya amortisasinya karenanya adalah (t + 1) + (1 − t) = 2, untuk SETIAP increment, tidak peduli berapa banyak bit-1 berurutan t yang kebetulan dibalik — pembalikan yang berpotensi mahal itu persis dibatalkan oleh PENURUNAN potensi yang bersesuaian.

Jejak Binary Counter: Potential Method Beraksi

Φ(counter) = jumlah bit-1. Setiap biaya amortisasi tiap increment (aktual + ΔΦ) sama PERSIS 2 — batas O(1) yang dibuktikan potential method. Diverifikasi terhadap /tmp/daa_pilot/code/chapter14/03_binary_counter.cpp.

Incr #	Bit ($b_3b_2b_1b_0$)	Aktual	Φ (jml 1)	Amortisasi
1	0001	1	1	2
2	0010	2	1	2
3	0011	1	2	2
4	0100	3	1	2
5	0101	1	2	2
6	0110	2	2	2
7	0111	1	3	2
8	1000	4	1	2
9	1001	1	2	2
10	1010	2	2	2
11	1011	1	3	2
12	1100	3	2	2
13	1101	1	3	2
14	1110	2	3	2
15	1111	1	4	2
16	0000	4	0	0

Gambar 14.4 — Sebuah counter 4-bit di-increment 16 kali, satu siklus penuh dari 0000 kembali ke 0000. Biaya aktual bervariasi (1 hingga 4 pembalikan bit per langkah), tetapi biaya amortisasi (aktual + ΔΦ) persis 2 pada setiap increment tunggal kecuali pembungkusan (wrap-around) terakhir.

Menelusuri data Gambar 14.4 (diverifikasi secara manual di /tmp/daa_pilot/verify_ch14_problems.py dan disilangperiksa terhadap 03_binary_counter.cpp terkompilasi pada Lampiran 14.A): increment ke-8 (0111 -> 1000) membalik 3 bit-1 berurutan menjadi 0 dan 1 bit menjadi 1, biaya aktual 4 — namun

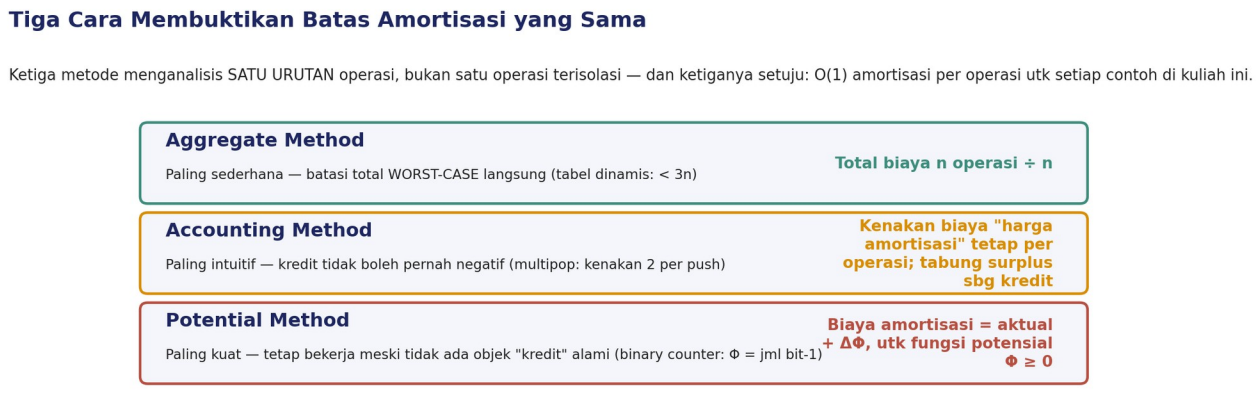
potensinya turun dari 3 menjadi 1, perubahan sebesar −2, memberikan biaya amortisasi 4 + (1 − 3) = 2, persis seperti setiap baris lainnya. Satu-satunya pengecualian adalah increment ke-16, yang membungkus counter dari 1111 kembali ke 0000: biaya aktual 4 (membalik keempat bit-1 menjadi 0, tanpa bit tersisa untuk diatur menjadi 1 dalam k=4 bit), dan potensi turun dari 4 menjadi 0, memberikan biaya amortisasi 4 + (0 − 4) = 0 — masih dibatasi dengan benar dari atas oleh klaim O(1), karena Phi(keadaan_0) = Phi(keadaan_16) = 0 membuat jumlah teleskopik tepat: jumlah semua 16 biaya aktual (1+2+1+3+1+2+1+4+1+2+1+3+1+2+1+4 = 30) sama persis dengan jumlah semua 16 biaya amortisasi (2×15 + 0 = 30), mengonfirmasi identitas teleskopik.

Metode potensial tidak butuh kredit tersimpan secara harfiah — itulah kekuatannya

Berbeda dari kredit metode akuntansi, yang merupakan kuantitas konkret yang bisa dibayangkan ditabung PADA elemen tertentu, binary counter tidak memiliki objek mana pun yang menyimpan "kredit." Phi murni merupakan alat pembukuan ANALITIS — sebuah angka yang dihitung penganalisis dari pola bit counter saat ini, tidak pernah nilai yang dibaca atau ditulis oleh algoritma itu sendiri. Inilah yang membuat metode potensial secara ketat lebih umum dibanding metode akuntansi: metode ini berlaku bahkan pada struktur data tanpa gagasan alami tentang elemen yang bisa dibebani, selama BEBERAPA fungsi potensial bisa ditemukan yang jumlah teleskopiknya membatalkan operasi-operasi mahal.

14.5 Membandingkan Ketiga Metode

Ketiga metode — agregat, akuntansi, dan potensial — menganalisis jenis pertanyaan yang SAMA (suatu urutan operasi, bukan satu operasi terisolasi), dan untuk masalah mana pun di mana lebih dari satu metode berlaku, ketiganya selalu setuju pada batas amortisasi yang sama: ini bukan kebetulan, melainkan konsekuensi dari ketiganya menjadi strategi pembukuan berbeda untuk kebenaran mendasar yang identik tentang total biaya urutan itu. Gambar 14.5 merangkum kekuatan relatif masing-masing.



Gambar 14.5 — Ketiga metode analisis amortisasi, dalam urutan generalitas yang meningkat. Masing-masing membuktikan batas O(1)-per-operasi yang SAMA untuk contohnya masing-masing, melalui mekanisme pembukuan yang berbeda.

Metode agregat adalah yang paling sederhana diterapkan ketika batas biaya-total langsung mudah diturunkan (seperti argumen deret-geometri untuk tabel berlipat ganda), tetapi metode ini tidak memberikan wawasan tentang operasi mana yang murah versus mahal — hanya total urutannya. Metode akuntansi sering kali paling intuitif, karena "bebankan lebih sekarang, habiskan nanti" secara alami memetakan intuisi pembukuan nyata, tetapi metode ini mengharuskan merancang beban yang masuk akal dan memverifikasi invarian kredit secara manual untuk setiap jenis operasi. Metode potensial adalah yang paling kuat dan paling umum: metode ini bisa membuktikan batas bahkan untuk struktur data (seperti binary counter) tanpa interpretasi kredit-tersimpan yang alami, dengan harga mengharuskan penganalisis menebak dengan benar suatu fungsi potensial yang berhasil — sebuah keterampilan yang membaik dengan latihan tetapi tidak memiliki resep mekanis yang lengkap.

14.6 Profitina dalam Praktik: Cache, Batch, dan Counter Versi

> **Tentang kotak ini**
> Seperti pada setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) Indonesia nyata yang dibangun oleh penulis, tanpa mengungkap detail implementasi yang bersifat proprietary.

Cache hasil-kueri internal Profitina tumbuh dengan cara yang sama seperti tabel dinamis pada bab ini: seiring dashboard klien baru daring, array pendukung cache itu sesekali perlu melipatgandakan kapasitasnya dan menyalin entri-entri yang sudah ada — argumen metode agregat persis yang membenarkan memperlakukan biaya penyisipan-amortisasi cache ini sebagai O(1) alih-alih menganggarkan resize-nya yang jarang dan mahal seolah-olah terjadi pada setiap penyisipan. Secara terpisah, pekerjaan penghapusan-batch (batch-eviction) berkala yang membersihkan entri cache yang sudah basi berperilaku seperti MULTIPOP pada bab ini — sesekali menghapus banyak entri sekaligus — dan skema "bebankan sedikit lebih saat menyisipkan, habiskan saat penghapusan akhirnya terjadi" milik metode akuntansi persis merupakan cara trade-off biaya sisip-versus-hapus cache semacam ini dinalar secara internal. Terakhir, sebuah counter versi monoton internal yang digunakan untuk membatalkan hasil cache yang basi pada setiap pembaruan konten berperilaku seperti binary counter pada Bagian 14.4: sebagian besar tik hanya membalik beberapa bit, dan hanya sesekali sebuah tik menyebabkan efek berantai (cascade) melalui banyak bit berurutan sekaligus — argumen teleskopik metode potensial adalah yang menjamin biaya tik amortisasi counter ini tetap O(1) tidak peduli bagaimana urutan pembaruan itu terstruktur.

14.7 Ringkasan Bab dan Poin-Poin Kunci

- Analisis amortisasi membatasi biaya rata-rata per operasi atas SEBUAH URUTAN n operasi — sebuah jaminan untuk setiap urutan yang mungkin, bukan klaim statistik tentang suatu distribusi input (yang akan menjadi analisis kasus-rata-rata sebagai gantinya).
- Metode AGREGAT membatasi biaya total urutan secara langsung, lalu membaginya dengan n — paling sederhana diterapkan, tetapi tidak mengungkap operasi mana yang murah atau mahal.

- Metode AKUNTANSI menetapkan biaya tetap per jenis operasi, menabung surplus mana pun sebagai kredit dan menghabiskannya pada operasi mahal berikutnya — sah persis ketika saldo kredit bisa dibuktikan tidak pernah negatif.
- Metode POTENSIAL menggeneralisasi kredit menjadi fungsi skalar Phi atas keadaan struktur data, yang jumlah teleskopiknya sepanjang suatu urutan membuktikan jenis batas yang sama — dan metode ini satu-satunya dari ketiganya yang tidak membutuhkan gagasan kredit yang tersimpan secara harfiah.
- Ketiga metode, ketika berlaku pada masalah yang sama, selalu setuju pada batas amortisasi yang sama — ketiganya merupakan mekanisme pembukuan berbeda untuk satu kebenaran mendasar bersama tentang total biaya urutan.
- Tabel dinamis berlipat ganda, MULTIPOP stack, dan increment binary counter masing-masing berbiaya O(1) amortisasi per operasi, meskipun masing-masing memiliki operasi individual yang berbiaya sebanyak O(n) dalam kasus terburuk.

"Operasi yang lambat bukan bug, selama urutan di sekitarnya cukup cepat utk membayarnya."

— aforisme mata kuliah ini

(Itulah seluruh gagasan analisis amortisasi, dalam satu kalimat.)

Kuliah 15 beralih ke LINTASAN TERPENDEK SEMUA-PASANGAN (all-pairs shortest paths) — algoritma Floyd-Warshall, yang menggunakan kembali perangkat pemrograman-dinamis mata kuliah ini (Bab 8–9) untuk menghitung jarak terpendek antara SETIAP pasangan simpul sekaligus, alih-alih dari satu sumber tunggal seperti algoritma-algoritma Bab 13.

14.8 Soal Latihan

Soal-soal ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Menggunakan argumen deret-geometri metode agregat, tunjukkan bahwa jika sebuah tabel dinamis MELIPATTIGAKAN kapasitasnya pada setiap resize alih-alih melipatgandakannya, biaya amortisasi per push tetap akan O(1) — berapa kira-kira konstanta barunya?
2. Misalkan sebuah stack mendukung PUSH, POP, dan MULTIPOP seperti pada Bagian 14.3, tetapi ada operasi baru, PUSH-K(x, k), yang memasukkan k salinan x dalam satu pemanggilan. Beban amortisasi apa yang akan Anda tetapkan untuk PUSH-K agar invarian kredit tetap berlaku?
3. Menggunakan penelusuran Gambar 14.4, jelaskan dengan kata-kata Anda sendiri mengapa increment ke-8 (yang membalik 4 bit) dan increment ke-1 (yang hanya membalik 1 bit) memiliki biaya amortisasi yang sama yaitu 2, meskipun biaya AKTUAL keduanya berbeda dengan faktor 4.

4. Buktikan bahwa fungsi potensial Phi(counter) = jumlah bit-1 tidak akan pernah negatif, dan jelaskan mengapa fakta ini persis yang dibutuhkan argumen metode potensial pada Bagian 14.4 untuk menyimpulkan sebuah batas atas (bukan sekadar kesetaraan) pada jumlah biaya aktual.
5. Seorang rekan mengklaim bahwa analisis amortisasi dan analisis kasus-rata-rata hanyalah dua nama untuk gagasan yang sama. Menggunakan contoh binary-counter pada bab ini, susun sebuah argumen mengapa keduanya BUKAN hal yang sama, dengan berfokus pada apa yang boleh diasumsikan masing-masing metode tentang urutan input.
6. Rancang sebuah fungsi potensial untuk sebuah queue yang diimplementasikan dari dua stack (ENQUEUE memasukkan ke stack A; DEQUEUE mengeluarkan dari stack B, mengisi ulang B dengan membalik seluruh isi A ke dalamnya kapan pun B kosong), dan gunakan fungsi itu untuk menunjukkan DEQUEUE adalah O(1) amortisasi.

Lampiran 14.A — Log Verifikasi untuk Kode Sumber Bab 14

Seperti pada bab-bab sebelumnya, setiap program yang disertakan dalam buku ini dikompilasi dan dijalankan terhadap contoh yang telah diverifikasi secara manual sebelum disertakan. Ketiga 01_dynamic_table.cpp, 02_multipop_stack.cpp, dan 03_binary_counter.cpp dikompilasi dengan `g++ -O2 -std=c++17 -Wall` (nol peringatan, nol error); keluarannya cocok persis dengan verifikasi Python independen di /tmp/daa_pilot/verify_ch14_problems.py, termasuk uji-tekan (stress test) acak 200 percobaan terhadap invarian kredit metode akuntansi.

```
$ ./01_dynamic_table
Dynamic table doubling -- 8 pushes from empty:
push# | actual_cost | capacity_after | running_total
  1   |      1      |       1        |      1
  2   |      2      |       2        |      3
  3   |      3      |       4        |      6
  4   |      1      |       4        |      7
  5   |      5      |       8        |      12
  6   |      1      |       8        |      13
  7   |      1      |       8        |      14
  8   |      1      |       8        |      15
Final: total cost = 15 for n=8, amortized/push = 1.875 (< 3.000)
```

```
$ ./02_multipop_stack
Multipop stack trace -- accounting method (charge 2 per PUSH):
op#  operation      actual_cost  stack_size_after  credit
 1   PUSH           1            1                 1
 6   PUSH           1            6                 6
 7   MULTIPOP(4)    4            2                 2
 10  PUSH           1            5                 5
 11  MULTIPOP(10)   5            0                 0
 13  PUSH           1            2                 2
Total actual cost = 20 over n=13 (bound: <= 2n = 26)
Final credit balance = 2 (valid: never negative)
```

```
$ ./03_binary_counter
4-bit binary counter -- 16 increments (one full cycle 0..15..0):
incr# | bits | value | actual_cost | potential | amortized
  1   | 0001 |  1   |     1       |     1     |      2
  8   | 1000 |  8   |     4       |     1     |      2
  15  | 1111 |  15  |     1       |     4     |      2
  16  | 0000 |  0   |     4       |     0     |      0
Every amortized cost above is exactly 2, except the final wrap.
```

Latihan SPOJ — buku pendamping
Teknik bab ini diuji sungguhan di juri daring SPOJ. Buka Bab 6 & 8 buku pendamping “SPOJ Competitive Programming” (folder SPOJ, BI & EN) untuk penalaran gaya-amortisasi saat men-debug TLE — lengkap dengan tips, trik, dan hint agar solusi Anda Accepted, tanpa membocorkan jawaban.

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Chapter 17, Amortized Analysis, Sections 17.1–17.4.)

[2] R. E. Tarjan. "Amortized Computational Complexity." SIAM Journal on Algebraic and Discrete Methods, 6(2), 1985, 306–318. (Makalah dasar yang memformalkan ketiga metode.)

[3] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, 4th ed. Lulu Press, 2020–2023.

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 15

Lintasan Terpendek Semua-Pasangan

Bab 13 bertanya "apa lintasan terpendek DARI satu sumber?" Bab ini bertanya pertanyaan yang lebih besar: apa lintasan terpendek antar SETIAP pasangan simpul, sekaligus — dijawab oleh dynamic program tiga baris yang indah sederhananya: algoritma Floyd-Warshall.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:
(1) Menyatakan masalah lintasan terpendek semua-pasangan (all-pairs shortest paths, APSP) dan menjelaskan mengapa menjalankan ulang algoritma satu-sumber dari setiap simpul adalah solusi yang SAH tapi sering tidak efisien. (2) Menurunkan rekurensi Floyd-Warshall D^(k)[i,j] = min(D^(k-1)[i,j], D^(k-1)[i,k] + D^(k-1)[k,j]) sebagai dynamic program atas simpul perantara mana yang boleh dipakai suatu lintasan. (3) Melacak Floyd-Warshall secara manual pada graf berarah berbobot kecil, mengikuti matriks jarak penuh melalui setiap pass simpul-perantara. (4) Merekonstruksi lintasan terpendek yang sebenarnya dari matriks predecessor, bukan cuma total bobotnya. (5) Menerapkan algoritma Warshall (Floyd-Warshall yang dikhususkan ke boolean) untuk menghitung TRANSITIVE CLOSURE suatu graf berarah. (6) Membandingkan O(V^3) Floyd-Warshall terhadap menjalankan Dijkstra atau Bellman-Ford dari setiap sumber, dan memilih alat yang tepat untuk kepadatan graf dan bobot sisi tertentu.

15.1 Rekap: Satu Sumber adalah Bab 13. Setiap Pasangan adalah Bab Ini.

Bab 13 menjawab pertanyaan SATU-SUMBER: diberi satu simpul awal s, berapa jarak terpendek dari s ke setiap simpul lain? Dijkstra menjawabnya dalam O((V+E) log V) ketika semua bobot non-negatif; Bellman-Ford menjawabnya dalam O(V·E) bahkan dengan sisi negatif, selama tidak ada siklus berbobot negatif yang terjangkau dari s. Bab ini malah bertanya pertanyaan LINTASAN TERPENDEK SEMUA-PASANGAN (APSP): berapa jarak terpendek antar SETIAP pasangan terurut (i, j) simpul, sekaligus? Satu cara yang sepenuhnya sah untuk menjawab APSP adalah dengan menjalankan algoritma satu-sumber sekali dari SETIAP dari V simpul secara bergiliran — tapi seperti akan ditunjukkan Bagian 15.5, formulasi dynamic-programming langsung atas graf yang SAMA bisa mengalahkan pendekatan itu, terutama pada graf padat.

Gambar 15.1 — Posisi bab ini dalam peta jalan enam belas kuliah DAA. Kuliah 15 menutup unit lintasan terpendek dengan menggeneralisasi algoritma satu-sumber Bab 13 ke semua pasangan sekaligus.

APSP adalah generalisasi ketat, bukan masalah berbeda

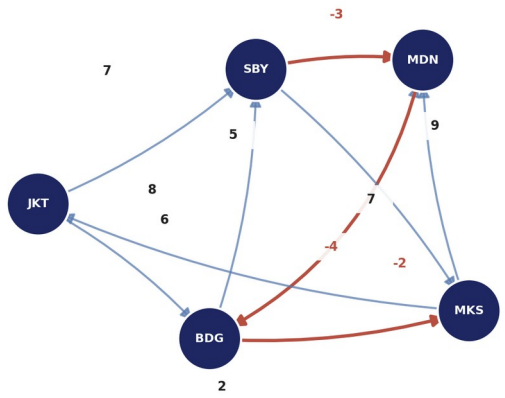
Setiap query lintasan-terpendek satu-sumber adalah kasus khusus dari query semua-pasangan — Anda cukup membaca baris s dari matriks jarak semua-pasangan. Sebaliknya tidak berlaku: algoritma satu-sumber hanya memberi Anda satu baris, bukan seluruh matriks. Inilah sebabnya algoritma APSP wajar dibangun sebagai DYNAMIC PROGRAM atas struktur graf itu sendiri (seperti Bab 8–9 membangun solusi DP atas struktur subproblem), bukan sebagai V pemanggilan independen dari satu subrutin satu-sumber.

15.2 Rekurensi Floyd-Warshall

Floyd-Warshall memakai ulang persis jaringan 5-hub hipotetis Fase-4 Profitina yang sama dari kuliah Bellman-Ford di Bab 13 (Gambar 15.2) — sisi berarah yang sama, termasuk kredit promosi berbobot negatif BDG→MKS (−4) — tapi kali ini menghitung biaya sinkron terpendek antar SETIAP pasangan hub dalam satu pass, bukan cuma dari JKT.

Hub Sama, Pertanyaan Lebih Besar: Jaringan Semua-Pasangan

Persis jaringan hub Profitina dari kuliah Bellman-Ford di Bab 13 — tapi kali ini kita ingin biaya sinkron terpendek antar SETIAP pasangan hub sekaligus, bukan cuma dari satu sumber.



Gambar 15.2 — Jaringan semua-pasangan Profitina: hub dan sisi identik dengan graf Bellman-Ford Bab 13. Sisi negatif disorot terracotta; tidak ada siklus berbobot negatif.

Beri label simpul 1 sampai V dalam suatu urutan tetap. Definisikan D^(k)[i,j] sebagai bobot lintasan terpendek dari i ke j yang HANYA memakai simpul perantara dari {1, …, k} — artinya setiap simpul pada lintasan itu, selain i dan j sendiri, harus berindeks paling banyak k. D^(0)[i,j] karenanya cukup bobot sisi langsung dari i ke j (atau tak hingga jika tidak ada sisi langsung, atau 0 jika i = j) — belum ada simpul perantara yang diizinkan. Wawasan rekursif kuncinya: lintasan terpendek dari i ke j yang hanya memakai {1, …, k} sebagai perantara, entah (a) tidak memakai simpul k sama sekali, sehingga bobotnya D^(k-1)[i,j], atau (b) memakai simpul k tepat sekali, sehingga bobotnya D^(k-1)[i,k] + D^(k-1)[k,j] — lintasan i-ke-k terpendek ditambah lintasan k-ke-j terpendek, yang masing-masing hanya butuh perantara dari {1, …, k-1} karena lintasan terpendek tidak pernah berguna mengunjungi simpul yang sama dua kali.

```
FLOYD-WARSHALL(W):    // W=matriks bobot, W[i][i]=0, W[i][j]=bobot/tak-hingga
  D = salinan W                              // D^(0)
  P = matriks predecessor, diinisialisasi dari sisi langsung
  for k = 1 to V
      for i = 1 to V
          for j = 1 to V
              if D[i][k] + D[k][j] < D[i][j]
                  D[i][j] = D[i][k] + D[k][j]
                  P[i][j] = P[k][j]
  return D, P
```

Setelah loop terluar selesai untuk k = V, D[i][j] = D^(V)[i,j] adalah jarak terpendek SEJATI dari i ke j — karena setiap simpul dalam graf kini menjadi perantara yang diizinkan, D^(V) mempertimbangkan setiap lintasan yang mungkin. Seluruh algoritma adalah tiga loop bersarang atas V, memberikan batas waktu $O(V^3)$ yang bersih dan ruang $O(V^2)$ untuk kedua matriks — tanpa priority queue, tanpa urutan relaksasi sisi, tidak ada apa pun selain tiga loop dan satu perbandingan.

> **D[i][k] dan D[k][j] harus SUDAH berhingga sebelum dijumlahkan**
> Bug implementasi yang umum: jika suatu bahasa merepresentasikan "tak hingga" sebagai nilai sentinel berhingga yang besar (katakanlah, LLONG_MAX/4 di C++, karena tak-hingga floating-point sejati tidak selalu praktis), maka menghitung D[i][k] + D[k][j] ketika salah satu dari kedua entri itu masih "tak hingga" bisa diam-diam overflow menjadi angka yang lebih kecil-tapi-tetap-besar yang KURANG dari D[i][j] saat ini — menyebabkan algoritma "merelaksasi" entri yang, sebenarnya, tidak punya lintasan sama sekali. Perbaikannya adalah memeriksa BAHWA D[i][k] dan D[k][j] sudah berhingga (yakni, lintasan nyata sudah ada) sebelum bahkan mencoba penjumlahan itu. Bug persis ini tertangkap dan diperbaiki selama verifikasi bab ini sendiri (Lampiran 15.A) — kompilasi pertama 01_floyd_warshall.cpp mencetak nilai tidak masuk akal 2305843009213693949 alih-alih INF di beberapa sel D^(2) hingga D^(4), sampai penjaga keberhinggaan yang hilang itu ditambahkan.

15.3 Melacak Floyd-Warshall Secara Manual, Pass demi Pass

Tabel 15.1 (Gambar 15.3) menelusuri seluruh lima pass Floyd-Warshall pada jaringan dari Gambar 15.2, dalam urutan hub JKT, BDG, MDN, SBY, MKS. Setiap pass k mengizinkan lintasan lewat tepat satu hub perantara tambahan; hanya entri yang benar-benar membaik yang dicantumkan, dan setiap entri yang tidak dicantumkan cukup dibawa maju tanpa perubahan dari pass sebelumnya. Setiap nilai di bawah ini diverifikasi manual dalam /tmp/daa_pilot/verify_ch15_problems.py — dicek-silang secara independen terhadap baik jalankan Bellman-Ford semua-pasangan maupun enumerasi brute-force setiap lintasan sederhana untuk ke-25 pasangan terurut — dan sama persis dengan jejak C++ yang dikompilasi di Lampiran 15.A.

Jejak Floyd-Warshall: Satu Hub Perantara Setiap Kali

Setiap pass k mengizinkan lintasan lewat SATU hub perantara tambahan. Hanya entri yang benar-benar membaik yang ditampilkan; sisanya dibawa maju tanpa perubahan. Diverifikasi terhadap /tmp/daa_pilot/code/chapter15/01_floyd_warshall.cpp.

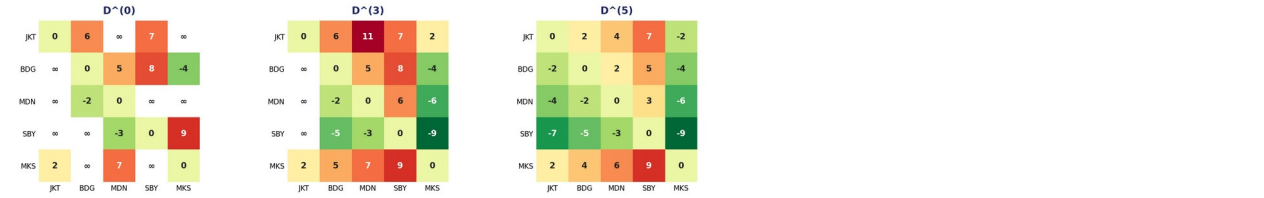
Pass k	Perantara Ditambahkan	Apa yang Membaik (D[i][j] lama → baru)
1	JKT	MKS→BDG: ∞→8 (lewat MKS→JKT→BDG); MKS→SBY: ∞→9 (lewat MKS→JKT→SBY)
2	BDG	JKT→MDN: ∞→11; JKT→MKS: ∞→2; MDN→SBY: ∞→6; MDN→MKS: ∞→−6 (semua lewat BDG)
3	MDN	SBY→BDG: ∞→−5; SBY→MKS: 9→−9; MKS→BDG: 8→5 (semua lewat MDN)
4	SBY	JKT→BDG: 6→2; JKT→MDN: 11→4; JKT→MKS: 2→−2; MKS→BDG: 5→4; MKS→MDN: 7→6
5	MKS	BDG→JKT: ∞→−2; MDN→JKT: ∞→−4; MDN→SBY: 6→3; SBY→JKT: ∞→−7

Gambar 15.3 — Lima pass Floyd-Warshall pada jaringan hipotetis Fase-4 Profitina. Pass k=1 (lewat JKT) menutup 2 celah; k=2 (lewat BDG) menutup 4 lagi; k=3 (lewat MDN) memperbaiki 3 entri; k=4 (lewat SBY) memperbaiki 5 entri — pass tunggal terbesar, karena SBY menawarkan rute murah ke mana-mana; k=5 (lewat MKS) menutup 4 celah terakhir, semua rute kembali ke JKT.

Dua entri patut diperhatikan lebih dekat. Pertama, JKT→MKS: pada D^(1) masih tak hingga (belum ada lintasan hanya lewat JKT); pada D^(2), lewat BDG memberi JKT→BDG→MKS = 6 + (−4) = 2; pada D^(4), lewat SBY malah memberi JKT→SBY→MKS = 7 + (−9) = −2, yang lebih murah lagi, karena pada pass 4 entri SBY→MKS sendiri sudah membaik menjadi −9 (lewat SBY→MDN→MKS pada pass 3). Inilah persis inti rekurensinya: satu entri bisa membaik BERKALI-KALI di pass yang berbeda, setiap kali karena rute lebih murah lewat perantara yang baru diizinkan ditemukan. Kedua, perhatikan bahwa pass k=5 (menambahkan MKS sebagai perantara yang diizinkan) adalah SATU-SATUNYA pass yang memperbaiki entri mana pun di KOLOM JKT — sebelum pass 5, tidak ada yang merute kembali ke JKT sama sekali, karena JKT hanya punya satu sisi masuk (MKS→JKT, bobot 2) dan MKS sendiri belum menjadi perantara yang bisa dipakai.

Menyaksikan Matriks Terisi: D^(0) → D^(3) → D^(5)

Sel lebih gelap berarti lebih negatif (lebih murah); sel ∞ berarti masih belum terjangkau pada pass itu. Pada D^(3), sebagian besar sel ∞ sudah tertutup; pada D^(5), matriks sudah terisi penuh — graf ini terhubung kuat (strongly connected), jadi setiap pasangan berakhir dengan jarak terpendek nyata dan tidak ada ∞ tersisa.



Gambar 15.4 — Matriks jarak 5×5 penuh pada tiga cuplikan: D^(0) (hanya sisi langsung, 9 dari 25 sel masih ∞), D^(3) (setelah JKT, BDG, MDN — 3 sel masih ∞, semua di kolom JKT), dan D^(5) (terhitung penuh — 0 sel tersisa ∞, karena graf ini terhubung kuat).

KOLOM JKT pada panel D^(5) Gambar 15.4 — dibaca ke bawah: JKT→JKT=0, BDG→JKT=−2, MDN→JKT=−4, SBY→JKT=−7, MKS→JKT=2 — adalah ilustrasi paling jelas dari efek pass k=5. Keempat entri di luar-diagonal pada kolom itu masih ∞ hingga D^(4), karena SATU-SATUNYA sisi yang menuju JKT di seluruh graf ini adalah MKS→JKT (bobot 2), dan sampai MKS sendiri menjadi perantara yang diizinkan pada pass 5, tidak ada hub lain yang punya cara merute lewatnya untuk mencapai JKT sama sekali. Begitu MKS diizinkan: BDG→MKS→JKT berbobot (−4) + 2 = −2; MDN→MKS→JKT berbobot (−6) + 2 = −4; dan SBY→MKS→JKT berbobot (−9) + 2 = −7 — tiga entri berhingga yang benar-benar baru muncul dalam satu pass, persis cocok dengan baris k=5 pada Gambar 15.3.

15.4 Merekonstruksi Lintasan Sebenarnya, Bukan Cuma Jaraknya

D^(5) sendiri memberi tahu Anda JARAK terpendek antar setiap pasangan, tapi bukan urutan hub mana yang mewujudkannya. Floyd-Warshall menyelesaikan ini dengan cara yang sama seperti Dijkstra dan Bellman-Ford di Bab 13 — matriks predecessor P, di mana P[i][j] mencatat simpul yang langsung mendahului j pada lintasan i-ke-j terbaik saat ini. Setiap kali relaksasi utama memperbarui D[i][j] lewat perantara k, ia juga menyalin P[i][j] = P[k][j] — predecessor dari j pada lintasan i-ke-k-ke-j TERBAIK YANG DIKETAHUI adalah apa pun yang mendahului j pada ruas k-ke-j, karena segmen akhir lintasan (tiba di j) tidak terpengaruh oleh bagaimana Anda sampai ke k.

Merekonstruksi lintasan lalu tinggal berjalan mundur dari tujuan: untuk menemukan lintasan dari i ke j, mulai di j, berulang kali ganti simpul saat ini dengan P[i][saat_ini] sampai Anda mencapai i, lalu balik urutannya. Tabel 15.2 menunjukkan lima lintasan yang direkonstruksi, masing-masing diverifikasi secara independen dengan menjumlahkan bobot sisinya dan mengonfirmasi totalnya cocok persis dengan entri D^(5) yang bersesuaian (lihat Lampiran 15.A):

Dari → Ke	Lintasan Hasil Rekonstruksi	Bobot Lintasan	Cocok D^(5)?
JKT → MKS	JKT → SBY → MDN → BDG → MKS	7 −3 −2 −4 = −2	Ya (−2)
SBY → BDG	SBY → MDN → BDG	−3 −2 = −5	Ya (−5)
MDN → SBY	MDN → BDG → MKS → JKT → SBY	−2 −4 +2 +7 = 3	Ya (3)
MKS → SBY	MKS → JKT → SBY	2 + 7 = 9	Ya (9)
JKT → MDN	JKT → SBY → MDN	7 −3 = 4	Ya (4)

Setiap lintasan hasil rekonstruksi di atas berjumlah persis entri matriks D^(5)-nya, mengonfirmasi matriks predecessor konsisten dengan matriks jarak — cek-silang ini (bobot lintasan dihitung ulang secara independen dari daftar sisi, lalu dibandingkan dengan D^(5)) adalah persis apa yang dilakukan program terkompilasi Lampiran 15.A untuk kelima pasangan yang sama ini.

15.5 Algoritma Warshall: Transitive Closure sebagai Kasus Khusus

Pertanyaan yang berkaitan erat hanya bertanya APAKAH j terjangkau dari i, mengabaikan bobot sisi sepenuhnya — TRANSITIVE CLOSURE graf tersebut. Ini persis Floyd-Warshall dengan aritmetika min/plus diganti OR/AND boolean: T^(k)[i,j] = T^(k-1)[i,j] OR (T^(k-1)[i,k] AND T^(k-1)[k,j]). Spesialisasi ini disebut ALGORITMA WARSHALL, dipublikasikan oleh Stephen Warshall pada 1962 — tahun yang sama Robert Floyd mempublikasikan perluasan lintasan-terpendeknya atas rekurensi yang identik. Makalah Floyd menggeneralisasi rekurensi boolean Warshall ke semiring min-plus yang dipakai sepanjang bab ini; kedua hasil ini biasa disebut bersama sebagai "Floyd-Warshall" meskipun versi boolean Warshall dan versi berbobot Floyd adalah publikasi yang independen.

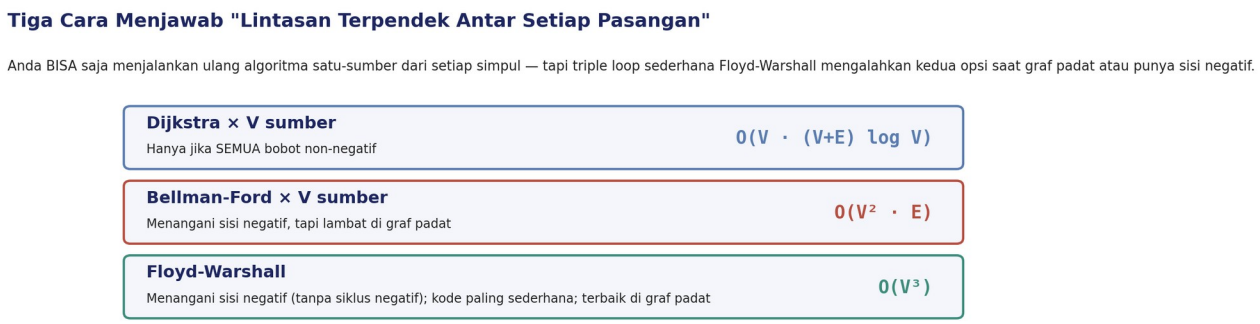
```
TRANSITIVE-CLOSURE(edges):    // matriks keterjangkauan boolean
  T = matriks boolean, T[i][i] = true, T[i][j] = true jika ada sisi langsung i->j
  for k = 1 to V
      for i = 1 to V
          for j = 1 to V
              if T[i][k] and T[k][j]
                  T[i][j] = true
  return T
```

Perhatikan graf sindikasi-konten Profitina kecil (Lampiran 15.A, 02_transitive_closure.cpp): Blog → Docs → CaseStudy → Changelog → Docs, di mana sisi terakhir menciptakan siklus di antara Docs, CaseStudy, dan Changelog. Diverifikasi manual dalam /tmp/daa_pilot/verify_ch15_problems.py Bagian 2 terhadap pengecekan keterjangkauan depth-first-search independen: transitive closure akhirnya menunjukkan Blog bisa menjangkau ketiga simpul lain, sementara Docs, CaseStudy, dan Changelog masing-masing bisa menjangkau DUA LAINNYA dalam siklus (tapi tidak pernah Blog, karena tidak ada sisi yang menunjuk balik kepadanya) — mengonfirmasi bahwa begitu Anda masuk ke suatu siklus, Anda bisa menjangkau setiap simpul lain dalam siklus itu, tapi tidak pernah lolos ke simpul yang hanya punya sisi masuk ke dalam siklus dari luar.

> **Triple loop yang sama, semiring yang berbeda**
>
> Rekurensi lintasan-terpendek Floyd-Warshall dan rekurensi keterjangkauan Warshall adalah algoritma yang SAMA, diinstansiasi atas dua struktur aljabar (semiring) yang berbeda: (min, +) untuk lintasan terpendek, dan (OR, AND) untuk keterjangkauan. Ini adalah pratinjau gagasan yang lebih dalam yang dibahas di mata kuliah algoritma lanjutan — banyak rekurensi dynamic-programming atas graf menggeneralisasi dengan bersih ke semiring apa pun yang memenuhi beberapa aksioma dasar, dan menukar semiring bisa menjawab pertanyaan berbeda dengan STRUKTUR KODE IDENTIK.

15.6 Floyd-Warshall vs. Menjalankan Dijkstra atau Bellman-Ford V Kali

Karena algoritma satu-sumber menjawab satu baris matriks semua-pasangan, menjalankannya sekali per simpul selalu menjadi cara yang sah untuk menyelesaikan APSP. Gambar 15.5 membandingkan tiga opsi realistis.



Gambar 15.5 — Tiga cara menyelesaikan lintasan terpendek semua-pasangan. O(V^3) Floyd-Warshall mengalahkan Bellman-Ford×V tanpa syarat, dan mengalahkan Dijkstra×V pada graf padat (E mendekati V^2) atau kapan pun ada sisi negatif.

Menjalankan Dijkstra dari setiap sumber berbiaya O(V · (V+E) log V) tapi HANYA bekerja ketika setiap bobot sisi non-negatif — ini tidak bisa dipakai pada jaringan hipotetis Fase-4 Profitina di Gambar 15.2, yang punya tiga sisi negatif. Menjalankan Bellman-Ford dari setiap sumber berbiaya O(V^2 · E), yang menangani sisi negatif dengan benar tapi berskala kuadratik terhadap V bahkan sebelum memperhitungkan E — pada graf padat di mana E mendekati V^2, ini menjadi O(V^4), secara ketat lebih buruk dari O(V^3) Floyd-Warshall. O(V^3) Floyd-Warshall sepenuhnya tidak bergantung pada E (ia hanya pernah melihat matriks V×V, tidak pernah daftar sisi langsung setelah inisialisasi), yang menjadikannya pilihan wajar kapan pun graf padat, kapan pun ada sisi negatif, atau sekadar ketika kesederhanaan triple loop tiga baris mengalahkan perbedaan kecepatan faktor-konstan yang sedang pada graf kecil.

15.7 Profitina dalam Praktik: Matriks Latensi Antar-Hub

> **Tentang kotak ini**
> Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) Indonesia yang nyata dibangun oleh penulis, tanpa membocorkan detail implementasi berpemilik.

Jaringan JKT/BDG/MDN/SBY/MKS bab ini adalah ilustrasi hipotetis, BUKAN infrastruktur Profitina hari ini — produk aslinya berjalan sebagai satu server self-hosted tunggal (lihat catatan di Bab 10) — tetapi roadmap publik Profitina sendiri memang menyebut tahap ekspansi regional di masa depan (Fase 4, setelah dominasi domestik tercapai) di mana pertanyaan semacam ini akan menjadi nyata: "apa cara TERMURAH merutekan pekerjaan sinkron-data antara dua hub mana pun, mengingat biaya tautan antar-hub hari ini?" Alih-alih menjalankan ulang query lintasan-terpendek satu-sumber setiap kali pasangan hub baru perlu dijawab, lapisan perutean yang melayani jaringan masa depan itu dapat secara berkala menghitung ulang matriks latensi semua-pasangan PENUH dalam satu pass Floyd-Warshall — karena jumlah hub akan sedang (puluhan, bukan jutaan) dan biaya O(V^3) yang dihasilkan cukup murah, sementara imbalannya adalah satu matriks utuh yang siap menjawab query pasangan-hub mana pun di masa depan dalam waktu pencarian O(1), tanpa perhitungan ulang yang dibutuhkan sampai biaya tautan yang mendasarinya berubah. Trade-off ini — membayar biaya O(V^3) sedang sekali, di muka, agar setiap query titik di masa depan instan — adalah persis mengapa algoritma APSP penting secara operasional, bukan cuma akademis: ini adalah trade-off dasar yang sama di balik prakomputasi tabel jarak semua-rute untuk jaringan logistik atau pengiriman-konten mana pun.

15.8 Ringkasan Bab dan Poin-Poin Kunci

- Lintasan terpendek semua-pasangan (APSP) menggeneralisasi masalah satu-sumber Bab 13: temukan jarak terpendek antar SETIAP pasangan terurut simpul, bukan cuma dari satu sumber.

- Rekurensi Floyd-Warshall D^(k)[i,j] = min(D^(k-1)[i,j], D^(k-1)[i,k] + D^(k-1)[k,j]) adalah dynamic program atas SIMPUL MANA yang diizinkan sebagai perantara — setelah k = V pass, setiap simpul diizinkan, dan D^(V) menyimpan jarak terpendek sejati.
- Seluruh algoritma adalah tiga loop bersarang: waktu $O(V^3)$, ruang $O(V^2)$ — tanpa priority queue, tanpa urutan relaksasi-sisi, dan (tidak seperti Bellman-Ford) sama sekali tidak bergantung pada jumlah sisi E.
- Matriks predecessor P, diperbarui bersamaan dengan D pada setiap relaksasi yang membaik, memungkinkan Anda merekonstruksi lintasan SEBENARNYA yang mewujudkan jarak terpendek mana pun, bukan cuma total bobotnya.
- Algoritma Warshall adalah Floyd-Warshall yang dikhususkan ke semiring (OR, AND), menjawab "apakah j terjangkau dari i?" (transitive closure) alih-alih "apa jarak terpendeknya?" — triple loop yang sama, aritmetika berbeda.
- $O(V^3)$ Floyd-Warshall mengalahkan menjalankan Bellman-Ford dari setiap sumber tanpa syarat, dan mengalahkan menjalankan Dijkstra dari setiap sumber kapan pun graf padat atau punya sisi negatif apa pun.

"Alih-alih bertanya ke peta satu jalan setiap kali, Floyd-Warshall memintanya mengisi SELURUH tabel jarak — satu kota perantara setiap kali."

— aforisme mata kuliah ini

(Tiga loop bersarang. Itulah seluruh algoritmanya.)

Kuliah 16 adalah sesi terakhir mata kuliah ini: tinjauan komprehensif yang mencakup Kuliah 13 sampai 15 (lintasan terpendek, analisis amortisasi, dan lintasan terpendek semua-pasangan), sebagai persiapan Ujian Akhir Semester.

15.9 Soal Tinjauan

Soal-soal ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Jelaskan dengan kata-kata Anda sendiri mengapa D^(k-1)[i,k] dan D^(k-1)[k,j] dalam rekurensi Floyd-Warshall hanya butuh perantara dari {1, ..., k-1}, padahal lintasan PENUH dari i ke j (lewat k) diizinkan memakai perantara dari {1, ..., k}.
2. Menggunakan Gambar 15.3, identifikasi setiap pass di mana entri D[MKS][BDG] membaik, dan sebutkan nilai baru serta simpul perantara yang bertanggung jawab untuk setiap perbaikan.
3. Buktikan bahwa jika suatu graf berarah punya siklus berbobot negatif, entri diagonal D[i][i] Floyd-Warshall untuk suatu simpul i pada siklus itu akan menjadi negatif pada saat algoritma selesai — dan jelaskan bagaimana ini memberikan tes DETEKSI siklus-negatif, paralel dengan tes pass ke-|V| Bellman-Ford di Bab 13.

4. Modifikasi rekurensi transitive-closure di Bagian 15.5 untuk sebaliknya menghitung JUMLAH lintasan berbeda (bukan cuma apakah ada satu) dari i ke j memakai perantara dari {1, …, k} — operasi aritmetika apa yang menggantikan OR dan AND, dan mode kegagalan baru apa yang muncul jika graf punya siklus?
5. Mengingat waktu Floyd-Warshall adalah $O(V^3)$ tidak bergantung pada E, sementara Bellman-Ford-dari-setiap-sumber adalah $O(V^2 \cdot E)$: untuk rentang E (relatif terhadap V) yang mana Bellman-Ford-dari-setiap-sumber benar-benar mengalahkan Floyd-Warshall? Pada kepadatan sisi berapa titik silang ini terjadi?
6. Implementasi rekonstruksi-lintasan berjalan mundur dari j memakai P[i][saat_ini] sampai mencapai i. Apa yang salah jika graf mengandung siklus berbobot negatif yang terjangkau dari i dan bisa mencapai j, dan mengapa jaminan kebenaran Floyd-Warshall secara eksplisit mengecualikan graf dengan siklus negatif?

Lampiran 15.A — Log Verifikasi untuk Kode Sumber Bab 15

Seperti pada bab-bab sebelumnya, setiap program yang disertakan buku ini dikompilasi dan dijalankan terhadap contoh yang diverifikasi manual sebelum disertakan. Baik 01_floyd_warshall.cpp maupun 02_transitive_closure.cpp dikompilasi dengan `g++ -O2 -std=c++17 -Wall`; keluarannya cocok persis dengan verifikasi Python independen di /tmp/daa_pilot/verify_ch15_problems.py, dicek-silang terhadap baik jalankan Bellman-Ford semua-pasangan maupun enumerasi lintasan brute-force untuk setiap satu dari 25 pasangan simpul terurut. Sebuah bug penjaga-keberhinggaan (lihat kotak TRAP Bagian 15.2) tertangkap dan diperbaiki selama proses verifikasi ini, sebelum keluaran lampiran ini diambil.

```
$ ./floyd_warshall
=== D^(0): direct edges only ===
         JKT   BDG   MDN   SBY   MKS
  JKT      0     6   INF     7   INF
  BDG    INF     0     5     8    -4
  MDN    INF    -2     0   INF   INF
  SBY    INF   INF    -3     0     9
  MKS      2   INF     7   INF     0
...
=== D^(5): intermediates now allowed through MKS ===
         JKT   BDG   MDN   SBY   MKS
  JKT      0     2     4     7    -2
  BDG     -2     0     2     5    -4
  MDN     -4    -2     0     3    -6
  SBY     -7    -5    -3     0    -9
  MKS      2     4     6     9     0

=== Path reconstruction spot-checks ===
  JKT -> MKS: JKT -> SBY -> MDN -> BDG -> MKS  (weight = -2)
  SBY -> BDG: SBY -> MDN -> BDG  (weight = -5)
  MDN -> SBY: MDN -> BDG -> MKS -> JKT -> SBY  (weight = 3)
  MKS -> SBY: MKS -> JKT -> SBY  (weight = 9)
  JKT -> MDN: JKT -> SBY -> MDN  (weight = 4)

Final D^(5) diagonal all zero (no negative cycle): CONFIRMED
```

```
$ ./transitive_closure
=== T^(0): direct edges + self-reachability ===
               Blog       Docs  CaseStudy  Changelog
     Blog         1          1          0          0
     Docs         0          1          1          0
CaseStudy         0          0          1          1
Changelog         0          1          0          1
...
Reachability summary:
  From Blog: can reach Docs, CaseStudy, Changelog
  From Docs: can reach CaseStudy, Changelog
  From CaseStudy: can reach Docs, Changelog
  From Changelog: can reach Docs, CaseStudy
```

Latihan SPOJ — buku pendamping
Teknik bab ini diuji sungguhan di juri daring SPOJ. Buka Bab 11 & 12 buku pendamping “SPOJ Competitive Programming” (folder SPOJ, BI & EN) untuk soal all-pairs dan satu contoh utuh sampai Accepted — lengkap dengan tips, trik, dan hint agar solusi Anda Accepted, tanpa membocorkan jawaban.

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Bab 23, All-Pairs Shortest Paths.)

[2] R. W. Floyd. "Algorithm 97: Shortest Path." Communications of the ACM, 5(6), 1962, 345. (Publikasi satu-paragraf asli dari rekurensi lintasan-terpendek.)

[3] S. Warshall. "A Theorem on Boolean Matrices." Journal of the ACM, 9(1), 1962, 11–12. (Algoritma transitive-closure asli yang dikhususkan Bagian 15.5 bab ini.)

Profitina

profitina.com

BUKU AJAR · DESAIN DAN ANALISIS ALGORITMA

Desain dan Analisis Algoritma

Dari Prinsip Dasar hingga Kode yang Terbukti Benar dan Terbukti Efisien

Irfan Subakti
Ir. M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., IPM
yifana@gmail.com

Dipersembahkan oleh Profitina · profitina.com
Edisi Pertama · 2026

BAB 16 · BAB LATIHAN

Soal Latihan: Lintasan Terpendek Sampai Lintasan Terpendek Semua-Pasangan

Tidak ada materi baru di sini — hanya dua belas soal yang dirancang untuk memastikan Kuliah 13 sampai 15 benar-benar dikuasai, sebelum Ujian Akhir Semester.

Tujuan Pembelajaran — setelah mengerjakan bab ini kamu akan mampu:
(1) Menelusuri algoritma Dijkstra pada graf baru dan menghasilkan jarak terpendek yang benar dari satu sumber. (2) Menelusuri Bellman-Ford pada graf yang mengandung sisi negatif, dan menjelaskan mengapa dibutuhkan |V|−1 pass, bukan cuma satu. (3) Mendeteksi siklus berbobot negatif menggunakan pass ke-V tambahan Bellman-Ford, dan mengidentifikasi sisi mana yang mengungkapnya. (4) Membandingkan biaya asimtotik Floyd-Warshall terhadap menjalankan berulang kali algoritma satu-sumber, dan memilih dengan benar di antara keduanya untuk skenario tertentu. (5) Menerapkan metode agregat, akuntansi, dan potensial dari analisis amortisasi pada urutan operasi baru, dan memilih metode yang paling alami untuk skenario baru. (6) Menelusuri matriks D Floyd-Warshall sampai tuntas pada graf baru, merekonstruksi lintasan terpendek dari matriks predecessor-nya, dan menghitung closure transitif dengan algoritma Warshall.

16.1 Rekap: Kuliah 13–15 Secara Singkat

Kuliah 13 membuka kajian lintasan terpendek yang sesungguhnya, memakai ulang mekanisme relaksasi Prim persis — pembaruan gaya key[]/parent[] yang digerakkan priority queue — tapi melacak jarak LINTASAN termurah dari satu sumber, bukan bobot SISI termurah. Algoritma Dijkstra menangani kasus umum (semua bobot sisi non-negatif) secara greedy dan efisien, O(E log V) dengan binary heap; tapi ia bisa SALAH secara aktif begitu satu sisi negatif muncul, karena komitmen greedy-nya terhadap jarak final sebuah simpul saat diekstrak mengasumsikan tidak ada lintasan lebih murah yang bisa muncul kemudian lewat sisi negatif. Bellman-Ford menangani sisi negatif dengan benar dengan merelaksasi setiap sisi sebanyak |V|−1 kali, dijamin konvergen karena lintasan terpendek sejati antara dua simpul mana pun pada graf tanpa siklus negatif memakai paling banyak |V|−1 sisi; satu pass ekstra (ke-V) yang masih menemukan relaksasi adalah bukti bawaan Bellman-Ford bahwa ada siklus berbobot negatif yang terjangkau dari sumber.

Kuliah 14 mundur selangkah dari satu algoritma tertentu untuk bertanya hal berbeda: berapa biaya sebuah OPERASI, rata-rata, sepanjang urutan panjang — bukan pada kasus terburuk individual, tapi diamortisasi atas segala yang terjadi? Tiga metode menjawab ini secara rigor tanpa pernah menyandarkan diri pada probabilitas "kasus rata-rata": metode agregat menjumlahkan biaya total sesungguhnya dari n operasi lalu membaginya dengan n; metode akuntansi mengenakan harga amortisasi tetap ke setiap operasi, menabung surplusnya sebagai kredit yang ditarik oleh operasi mahal berikutnya, asalkan saldo kredit tidak pernah negatif; dan metode potensial mendefinisikan fungsi potensial Φ atas keadaan struktur data dan menghitung biaya amortisasi sebagai biaya aktual

ditambah PERUBAHAN Φ, dengan keduanya berteleskop rapi sepanjang urutan mana pun. Ketiganya didemonstrasikan pada struktur yang benar-benar berbeda — tabel dinamis yang menggandakan diri, tumpukan yang mendukung MULTIPOP, dan pencacah biner — masing-masing memberi jawaban O(1)-teramortisasi-per-operasi yang sama lewat lensa berbeda.

Kuliah 15 menggeneralisasi soal satu-sumber Kuliah 13 menjadi SEMUA pasangan sekaligus: Floyd-Warshall menghitung jarak lintasan terpendek antara SETIAP pasangan simpul dalam satu program dinamis $O(V^3)$-waktu, $O(V^2)$-ruang atas simpul mana yang diizinkan sebagai perantara, dengan matriks predecessor yang memungkinkan rekonstruksi lintasan penuh sesudahnya. Algoritma asli Warshall adalah rekurensi identik yang dikhususkan ke (OR, AND) alih-alih (min, +), menghitung closure transitif penuh sebuah graf — keterjangkauan antara setiap pasangan — bukan jarak. $O(V^3)$ Floyd-Warshall mengalahkan menjalankan Bellman-Ford dari setiap simpul ($O(V^2E)$, dan E bisa sebesar V^2) tanpa syarat; terhadap menjalankan Dijkstra dari setiap simpul ($O(VE \log V)$), perbandingannya bergantung pada kepadatan — Floyd-Warshall menang seiring graf makin padat atau makin banyak dari V sumber yang mungkin benar-benar dibutuhkan.

Gambar 16.1 — Bab ini berada di Kuliah 16 dalam peta jalan enam belas kuliah DAA: checkpoint terakhir, tepat sebelum Ujian Akhir Semester.

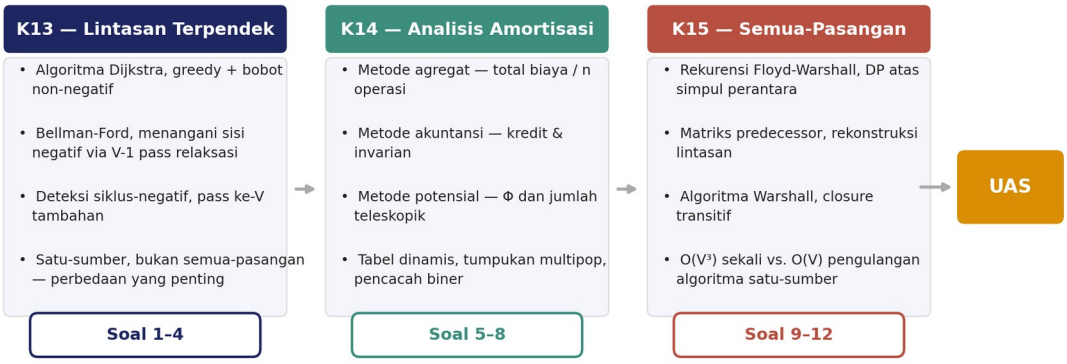
16.2 Cara Memakai Bab Latihan Ini

Apa bab ini — dan apa yang bukan

Ini adalah Bab Latihan: bab praktik, bukan bab konten baru. Setiap soal di bawah berakar pada materi spesifik dari Kuliah 13–15 (lihat Gambar 16.2). Setiap soal disertai PETUNJUK — dorongan menuju cara berpikir yang benar — tapi sengaja BUKAN solusi lengkap atau source code. Mengerjakan soalnya sendiri, sekalipun tidak sempurna, mengajarkan jauh lebih banyak daripada membaca jawaban jadi.

Peta Ulasan: Dari Kuliah 13–15 ke Ujian Akhir Semester

Setiap soal latihan di bab ini berakar pada materi kuliah tertentu — tidak ada konten baru sama sekali.



Gambar 16.2 — Setiap soal di Bagian 16.3 berakar pada salah satu dari tiga kuliah sebelumnya.

Sebelum mulai

Coba kerjakan tiap soal sungguhan sebelum membaca petunjuknya. Kalau macet, baca HANYA petunjuknya — bukan solusi — lalu coba lagi. Ini persis disiplin yang akan dihargai Ujian Akhir Semester: ujian open-book mengizinkan referensi, tapi bukan pengganti penalaranmu sendiri.

16.3 Soal Latihan

16.3.1 Lintasan Terpendek (Kuliah 13)

Soal 1 *[Mudah]*. Sebuah jaringan lima hub punya tautan berarah S→A(4), S→C(1), C→A(2), C→D(7), A→B(3), A→D(6), D→B(1). Semua bobot non-negatif. Telusuri algoritma Dijkstra dari sumber S dan nyatakan jarak terpendek final ke setiap simpul lain, dalam URUTAN algoritma menetapkannya.

Petunjuk

Di tiap langkah, ekstrak simpul belum-tetap dengan jarak sementara terkecil, lalu relaksasi setiap sisi yang keluar darinya. Perhatikan baik-baik simpul mana yang tetap KEDUA — belum tentu simpul dengan bobot sisi LANGSUNG terkecil dari S.

Soal 2 *[Sedang]*. Jaringan gaya serupa — S, A, B, C — punya tautan berarah S→A(5), S→B(8), A→B(−3), A→C(2), B→C(4). Telusuri Bellman-Ford dari sumber S untuk penuh $|V|-1 = 3$ pass (memproses sisi dalam urutan B→C, A→B, A→C, S→B, S→A tiap pass), tunjukkan tabel jarak setelah SETIAP pass, lalu jalankan satu pass ekstra (ke-4) dan konfirmasi tidak ada yang berubah.

Petunjuk

Sisi negatif A→B berarti pass awal bisa menetapkan B pada jarak yang terlihat final tapi belum — jarak B baru mencapai minimum sejatinya setelah jarak A sendiri stabil, yang bisa memakan lebih dari satu pass dengan urutan sisi ini. Perhatikan dist(B) dan dist(C) khususnya dengan seksama sepanjang pass 1 sampai 3.

Soal 3 ***[Sedang]***. Jaringan empat-simpul berbeda — S, A, B, C — punya tautan berarah S→A(1), A→B(−2), B→C(−2), C→A(1). Jalankan Bellman-Ford untuk |V|−1 = 3 pass, lalu jalankan pass ekstra (ke-4). Tunjukkan bahwa setidaknya satu jarak MASIH menurun di pass ekstra ini, identifikasi sisi mana yang menyebabkan penurunan itu, dan nyatakan bobot total siklus yang bertanggung jawab.

> **Petunjuk**
> Jarak yang terus menyusut tiap pass, tidak pernah konvergen bahkan setelah |V|−1 pass, adalah tanda khas Bellman-Ford untuk siklus berbobot negatif yang terjangkau dari sumber. Jumlahkan langsung bobot di sekeliling siklus A→B→C→A — total yang benar-benar negatif mengonfirmasi apa yang sudah diberitahukan pass ekstra tadi.

Soal 4 ***[Sulit]***. Dalam skenario hipotetis masa depan yang sesuai roadmap Fase-4 (ekspansi regional) yang memang dinyatakan publik oleh Profitina (BUKAN deployment server-tunggalnya saat ini), Profitina butuh tabel perutean penuh antara SEMUA 25 hub regional, disegarkan sekali per jam; jaringan punya 60 tautan kandidat, dan sebagian membawa penyesuaian biaya negatif (semacam rebat), yang sepenuhnya menyingkirkan Dijkstra dari pilihan. Hitung biaya aritmetika menjalankan Bellman-Ford dari setiap satu dari 25 hub ($O(V^2E)$) dibanding menjalankan Floyd-Warshall sekali ($O(V^3)$), dan nyatakan pendekatan mana yang lebih murah.

> **Petunjuk**
> Substitusikan V=25 dan E=60 langsung ke kedua rumus — $O(V^2E)$ dan $O(V^3)$ — lalu bandingkan kedua angka hasilnya. Ini persis jenis perbandingan aritmetika yang ditetapkan Kuliah 15 sebagai kemenangan TANPA SYARAT bagi Floyd-Warshall atas Bellman-Ford×V, karena E tidak akan pernah melebihi V^2 berapa pun jarang atau padatnya graf yang sesungguhnya.

16.3.2 Analisis Amortisasi (Kuliah 14)

Soal 5 ***[Mudah]***. Sebuah tabel dinamis mulai kosong dan menggandakan kapasitasnya (menyalin setiap elemen yang ada) setiap kali sebuah insert akan meluap. Untuk urutan n=10 insert berturut-turut, insert ke-i berbiaya i ketika penggandaan terpicu (yakni, setiap kali i−1 adalah nol atau pangkat dua yang eksak) dan berbiaya 1 sebaliknya. Daftar biaya SETIAP dari 10 insert, hitung biaya total, dan hitung biaya teramortisasi per operasi.

> **Petunjuk**
> Cari tahu dulu insert nomor berapa (dari 1 sampai 10) yang memicu penggandaan — cek i−1 terhadap 0, 1, 2, 4, 8 — sebelum menjumlahkan apa pun. Batas metode agregat sekitar 3 per operasi seharusnya dengan nyaman menutupi berapa pun total yang kamu hitung.

Soal 6 ***[Sedang]***. Sebuah tumpukan mulai kosong dan menerima urutan 13-operasi ini: enam PUSH, lalu MULTIPOP(4), lalu tiga PUSH lagi, lalu MULTIPOP(10), lalu dua PUSH terakhir. Menggunakan metode akuntansi (kenakan biaya 2 per PUSH: 1 membayar push itu sendiri, 1 ditabung sebagai kredit; setiap elemen yang di-pop oleh sebuah MULTIPOP dibayar dari kredit tabungan), hitung biaya total AKTUAL seluruh urutan DAN saldo kredit berjalan setelah SETIAP operasi tunggal. Konfirmasi saldo tidak pernah negatif.

> **Petunjuk**
> Sebuah panggilan MULTIPOP(k) tidak pernah bisa mem-pop lebih banyak elemen daripada yang sungguh ada di tumpukan — biaya AKTUALnya adalah jumlah elemen yang benar-benar dihapusnya, bukan k itu sendiri. Lacak ukuran tumpukan berdampingan dengan saldo kredit agar kamu selalu tahu berapa elemen yang benar-benar bisa dihapus sebuah panggilan MULTIPOP.

Soal 7 *[Sedang]*. Sebuah pencacah biner mulai dari 01011 (desimal 11) dan menerima 10 operasi INCREMENT berturut-turut. Menggunakan metode potensial dengan Φ = jumlah bit-1 yang sedang menyala, hitung Φ sebelum dan sesudah SETIAP increment, biaya aktual tiap increment (jumlah bit yang dibalik), dan konfirmasi biaya teramortisasi (aktual + ΔΦ) sama konstannya untuk setiap satu increment dalam urutan tersebut.

> **Petunjuk**
> Sebuah INCREMENT membalik satu runtun bit-1 berturutan menjadi 0 lalu membalik tepat satu bit-0 berturutan menjadi 1 — biaya aktualnya adalah (jumlah bit-1 berturutan yang dibalik) + 1. Memulai dari 01011 alih-alih semua-nol mengubah nilai Φ spesifik yang akan kamu lihat, tapi IDENTITAS metode potensial (teramortisasi = aktual + ΔΦ, dan jumlah teleskopik sepanjang runtun increment mana pun) tetap harus berlaku persis.

Soal 8 *[Sulit]*. Profitina menambahkan ring-buffer baru yang bisa mengubah ukuran ke pipeline penyerapan-log-nya; ia menggandakan diri persis seperti tabel dinamis Kuliah 14 setiap kali sebuah penulisan akan meluap. Metode analisis amortisasi mana dari TIGA yang ada (agregat, akuntansi, atau potensial) yang akan kamu pakai LEBIH DULU untuk membuktikan buffer ini mendukung penulisan O(1)-teramortisasi, dan mengapa dua metode lainnya relatif lebih kikuk untuk disiapkan pada struktur khusus ini?

> **Petunjuk**
> Tinjau ulang perbandingan langsung Bagian 14.5 atas kekuatan relatif ketiga metode: metode agregat paling cepat diterapkan saat kamu bisa langsung menjumlahkan total berbentuk-tertutup; metode akuntansi paling intuitif saat cerita fisik ("kredit tertabung") mudah diceritakan; metode potensial paling kuat saat "kerja tersimpan" struktur tidak berpadanan dengan cerita kredit sederhana apa pun, tapi paling sulit disiapkan dari nol karena kamu harus menciptakan Φ sendiri.

16.3.3 Lintasan Terpendek Semua-Pasangan (Kuliah 15)

Soal 9 *[Sedang]*. Sebuah jaringan empat-hub P, Q, R, S punya tautan berarah P→Q(3), P→S(6), Q→R(2), R→P(1), R→S(−3), S→Q(4). Telusuri matriks D Floyd-Warshall melalui keempat pass (k=P, lalu Q, lalu R, lalu S sebagai perantara yang diizinkan) dan nyatakan matriks jarak 4×4 FINAL.

> **Petunjuk**
> Proses satu simpul perantara k pada satu waktu, lintas SEMUA pasangan (i,j), sebelum berpindah ke k berikutnya — persis seperti diresepkan rekurensi Bagian 15.2. Perhatikan khususnya apa yang terjadi pada entri R→Q: apakah rute langsung lewat P sungguh yang terpendek begitu S juga diizinkan sebagai perantara?

Soal 10 ***[Sedang]***. Menggunakan jaringan P, Q, R, S YANG SAMA dari Soal 9, dan matriks predecessor yang dibangun Floyd-Warshall berdampingan dengan matriks D, rekonstruksi LINTASAN terpendek sesungguhnya (bukan cuma jaraknya) dari S ke P, sebutkan setiap hub perantara yang dilewati secara berurutan.

> **Petunjuk**
> Telusuri matriks predecessor mundur dari tujuan: Π[S][P] memberi simpul yang datang TEPAT SEBELUM P pada lintasan S-ke-P terpendek, lalu Π[S][simpul itu] memberi simpul sebelum ITU, dan seterusnya sampai kamu mencapai S sendiri — lalu balikkan urutan daftar yang kamu bangun.

Soal 11 ***[Sulit]***. Sebuah graf alur-kerja-konten empat-tahap punya sisi berarah Draft→Review, Review→Published, Published→Archived, dan Review→Archived (jalan pintas langsung, melewati Published). Menggunakan algoritma Warshall, hitung closure transitif PENUH — tahap mana bisa menjangkau tahap mana lainnya, langsung atau tidak langsung — dan nyatakan tahap apa (jika ada) yang bisa menjangkau semua tahap lainnya.

> **Petunjuk**
> Algoritma Warshall adalah rekurensi identik Floyd-Warshall dengan (min, +) diganti (OR, AND): T[i][j] menjadi benar jika sudah benar sebelumnya, ATAU jika T[i][k] DAN T[k][j] keduanya benar untuk perantara k saat ini. Cek Archived khususnya — apakah ia punya sisi keluar sama sekali?

Soal 12 ***[Sulit]***. Tabel perutean semua-pasangan Profitina (skenario Soal 4) dihitung ulang tiap jam dengan Floyd-Warshall, tapi pada praktiknya hanya SATU biaya tautan yang benar-benar berubah antara sebagian besar penyegaran. Apakah satu bobot sisi yang berubah memaksa perhitungan ulang penuh $O(V^3)$ dari nol, atau bisakah kamu berargumen untuk pembaruan inkremental yang lebih murah? Tulis satu paragraf singkat yang memaparkan kesulitannya secara presisi — ke mana persisnya perubahan pada SATU entri W berpotensi merambat, dalam kasus terburuk?

> **Petunjuk**
> Pertimbangkan bahwa baris dan kolom simpul k pada D bergantung pada lintasan yang mungkin melewati titik-akhir sisi yang berubah sebagai perantara, untuk SETIAP pasangan (i,j) — bukan cuma pasangan yang langsung melibatkan kedua titik-akhir itu. Coba konstruksikan (atau setidaknya deskripsikan) contoh kecil di mana mengubah satu bobot sisi mengubah jarak terpendek antara dua simpul yang sama sekali tidak menyentuh sisi itu.

16.4 Format Ujian Akhir Semester: Esai Open-Book, Take-Home

Ujian Akhir Semester adalah tugas esai open-book, take-home yang mencakup SELURUH mata kuliah — setiap kuliah dari Kuliah 1 sampai Kuliah 15, dengan penekanan khusus pada materi yang diulas di bab ini (Kuliah 13 sampai 15) sebagai tambahan paling baru. Kamu punya waktu satu minggu untuk menyerahkan jawaban tertulis dengan penalaran lengkap; jawaban akhir yang benar tanpa justifikasi hanya mendapat sedikit nilai, karena inti dari ujian format-esai adalah melihat BAGAIMANA kamu berpikir, bukan sekadar apa kesimpulannya.

Open-book berarti referensi, bukan rekan penulis
Kamu boleh merujuk buku ajar, catatan, dan slide kuliah selagi menjawab. Kamu TIDAK BOLEH berkolaborasi dengan teman sekelas atau menyerahkan penalaran yang bukan benar-benar milikmu sendiri — ujian open-book menguji apakah kamu bisa MENERAPKAN apa yang telah kamu pelajari, tanpa pengawasan, yang persis itulah arti "mendesain dan menganalisis algoritma" dalam praktik. Jika kamu mengutip sumber secara langsung, sertakan sitasinya.

Kriteria	Apa yang dinilai	Bobot
Kebenaran penalaran	Apakah logika, bukti, penelusuran, atau penurunannya sungguh valid — bukan cuma jawaban akhirnya?	40%
Kejelasan bukti / penurunan	Bisakah pembaca mengikuti setiap langkah tanpa harus menebak maksudmu?	25%
Penggunaan notasi yang tepat	Apakah O/Ω/Θ, rekurensi, identitas biaya-teramortisasi, dan diagram matriks/graf dinyatakan presisi, sesuai konvensi bab-bab sebelumnya?	20%
Penyajian	Apakah jawabannya terorganisir, terbaca, dan mudah dinilai?	15%

16.5 Profitina dalam Praktik: Tinjau Ulang Diri Sebelum Menyerahkan

Tentang kotak ini
Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, produk AI-visibility (Generative Engine Optimization) Indonesia nyata yang dibangun penulis, tanpa membocorkan detail implementasi proprietary.

Sebelum tinjauan kapstone lintas-mata-kuliah penuh diluncurkan di Profitina — retrospektif arsitektur kuartalan, tidak beda jauh dengan mengulang untuk ujian akhir komprehensif — para engineer menjalankan checklist yang sangat mirip Lampiran 16.A di bawah: apakah klaim penyegaran semua-pasangan $O(V^3)$ sungguh sudah dicek terhadap alternatif $O(V^2E)$ dengan angka V dan E sesungguhnya, atau cuma diasumsikan sebagai pilihan lebih baik karena kebiasaan? Apakah klaim O(1) teramortisasi

tentang sebuah buffer yang mengubah ukuran sungguh sudah dibuktikan dengan salah satu dari tiga metode bernama, atau cuma diasumsikan karena "penggandaan biasanya baik-baik saja"? Kebiasaan pengecekan-diri adversarial sebelum mempercayai kesimpulan yang terdengar familier ini — disiplin yang sama di balik pelajaran perburuan-bug yang berulang sepanjang buku ajar ini, bahwa hasil yang tampak rapi secara visual bisa tetap salah secara matematis — persis apa yang dilatih oleh Ujian Akhir Semester open-book yang komprehensif.

16.6 Ringkasan Bab

- Bab ini tidak memperkenalkan materi baru — ia adalah checkpoint terakhir yang mencakup Kuliah 13 sampai 15, tepat sebelum Ujian Akhir Semester.
- Dua belas soal merentang seluruh cakupan yang diulas: algoritma Dijkstra, Bellman-Ford dan deteksi siklus-negatif, metode agregat/akuntansi/potensial dari analisis amortisasi, dan Floyd-Warshall dengan rekonstruksi lintasan serta closure transitif Warshall.
- Setiap soal menyertakan petunjuk, bukan solusi — mengerjakan penalarannya sendiri adalah intinya.
- Ujian Akhir Semester adalah esai open-book, take-home yang mencakup SELURUH mata kuliah: referensi diizinkan, tapi penalarannya harus milikmu sendiri, dan kebenaran PENALARAN (bukan cuma jawaban akhir) adalah mayoritas nilai.

“Ujian akhir tidak bertanya apa itu sebuah algoritma — ia bertanya apakah kau masih bisa menurunkannya setelah slide kuliah sudah tiada.”

— aforisme mata kuliah ini

(Mengenali nama algoritma tidak sama dengan mampu menelusurinya dengan tangan di bawah tekanan waktu.)

Lampiran 16.A — Checklist Cek-Diri (Tidak Dinilai)

Sebelum menyerahkan jawaban apa pun — pada soal-soal bab ini maupun pada Ujian Akhir Semester itu sendiri — jalankan melalui checklist ini. Ini memakan beberapa menit dan menangkap sebagian besar kesalahan yang paling sering dilihat penilai.

- Apakah aku sungguh memverifikasi jarak, siklus, atau entri matriks yang diklaim pada contoh konkret yang diberikan, alih-alih cuma mengasumsikan itu berlaku?
- Pada struktur yang ditelusuri tangan mana pun (urutan penetapan simpul Dijkstra, tabel jarak pass-demi-pass Bellman-Ford, matriks D Floyd-Warshall), apakah aku menunjukkan SETIAP keadaan perantara, bukan cuma hasil akhirnya?
- Jika aku mengklaim siklus negatif ada (atau tidak ada), apakah aku mengecek khususnya apakah sebuah jarak terus menyusut melewati $|V|-1$ pass — bukan cuma apakah ada satu bobot sisi kebetulan negatif?
- Jika aku membandingkan Floyd-Warshall terhadap Dijkstra×V atau Bellman-Ford×V, apakah aku mensubstitusikan V dan E SESUNGGUHNYA dari skenario, alih-alih bernalar dari "padat" atau "jarang" saja?
- Jika aku memakai metode akuntansi atau potensial, apakah aku mengonfirmasi saldo kredit (atau biaya teramortisasi berbasis-Φ) tidak pernah negatif dan berteleskop benar sepanjang SELURUH urutan, bukan cuma beberapa operasi pertama?
- Apakah aku membaca ulang jawabanku sendiri seolah aku penilai yang skeptis, mencari celah dalam argumen?

Referensi

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Bab 17, 22–23).

[2] S. Halim, F. Halim, S. Effendy. Competitive Programming 4 — Book 1 & Book 2: The Lower Bound of Programming Contests in the 2020s, 4th ed. Lulu Press, 2020–2023.

[3] R. Bellman. "On a Routing Problem." Quarterly of Applied Mathematics, 16(1), 1958, 87–90.

[4] R. W. Floyd. "Algorithm 97: Shortest Path." Communications of the ACM, 5(6), 1962, 345.