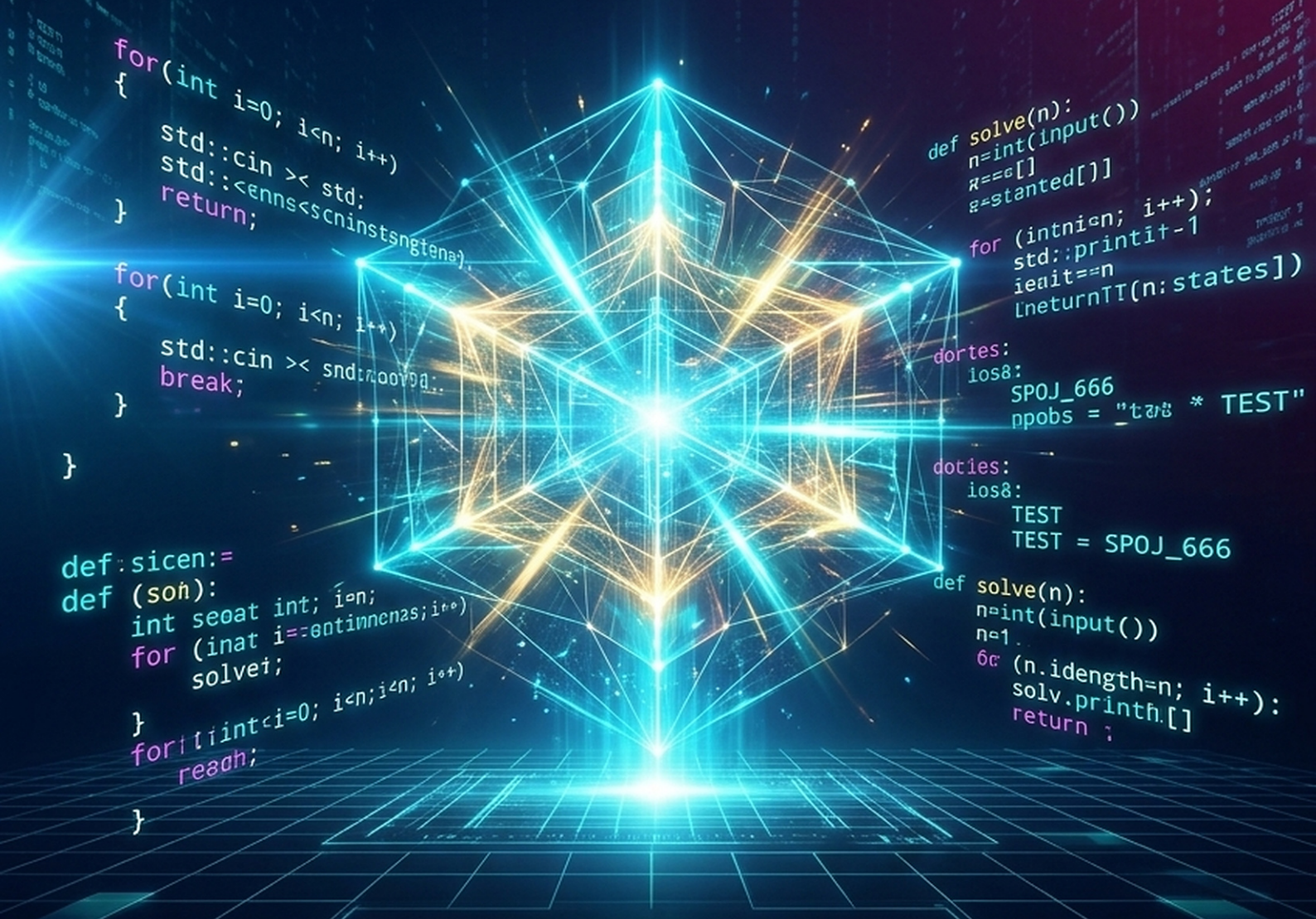
Pemrograman Kompetitif dengan SPOJ

Dari Membaca Soal hingga Accepted, dengan C++ dan Python



Irfan Subakti

Profitina
profitina.com

Pemrograman Kompetitif dengan SPOJ

Dari Membaca Soal hingga Accepted, dengan C++ dan Python

Irfan Subakti

Profitina

Hak cipta © 2026 Irfan Subakti. Hak cipta dilindungi undang-undang.

Diterbitkan oleh Profitina · profitina.com

Edisi pertama, September 2026.

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

Daftar Isi

Kata Pengantar ..... **xx**
Untuk siapa buku ini ditulis xx
Satu gagasan, dua bahasa xx
Isi buku ini xxi
Cara membacanya xxii
Tentang penulis xxii
Ucapan terima kasih xxiii

BAB 1 Selamat Datang di Pemrograman Kompetitif dan SPOJ **1**
1.1 Vonis Merah Tengah Malam 1
1.2 Apa Itu Pemrograman Kompetitif, dan Apa yang Ditawarkan SPOJ 2
1.3 Apa yang Dilakukan Juri terhadap Program Anda 3
1.4 Cara Kerja Buku Ini 5
1.5 Meja Kerja Anda: Apa yang Dipasang dan Mengapa 6
1.6 Memasang di Windows 10 dan 11 7
1.6.1 Kompiler C++ dari MSYS2 7
1.6.2 Python 8
1.6.3 VS Code 8
1.7 Memasang di macOS 8
1.7.1 Kompiler C++ 8
1.7.2 Python dan VS Code 8
1.8 Memasang di Linux 9
1.9 VS Code: Dua Ekstensi dan Satu *Build Task* 9
1.10 Program Pertama Anda: Memeriksa Meja Kerja 10
1.11 Soal SPOJ Pertama Anda: TEST 12
1.12 Membangun Juri Sendiri 14
1.12.1 Uji sebagai pasangan berkas 14
1.12.2 Jurinya 15
1.12.3 Menjuri program yang benar dan yang salah 17
1.13 Memakainya Sungguh-sungguh 19
1.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 20
1.15 Ringkasan Bab 20
1.16 Soal Latihan 21
1.17 Penyelesaian Soal Latihan 21
Daftar Pustaka 22

BAB 2 Membaca Soal Seperti Juri **24**
2.1 Baris yang Tak Dibaca Siapa Pun 24
2.2 Anatomi Pernyataan Soal 25
2.3 Daftar Periksa Membaca 26
2.4 Tiga Bentuk Masukan Juri 28
2.5 ADDREV: Membaca “Abaikan Nol di Depan” 29
2.5.1 Versi yang salah, dan alat yang menemukannya 31
2.6 FCTRL2: Membaca Ukuran Jawaban 35
2.7 ONP: Membaca Aturan yang Tak Lazim 38
2.8 Memakainya Sungguh-sungguh 43
2.9 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 44
2.10 Ringkasan Bab 45
2.11 Soal Latihan 45

2.12 Penyelesaian Soal Latihan 46
Daftar Pustaka 47

BAB 3 Memilih Algoritma Sesuai Batasan 48
3.1 Antrean di Tempat Fotokopi 48
3.2 Dari Batas Waktu ke Anggaran Langkah 49
3.3 Menghitung Langkah Sebuah Algoritma 51
3.4 FCTRL: Ketika Rumus Menggantikan Gelung 53
3.5 LASTDIG: Membagi Dua Pangkat 58
3.6 PRIME1: Saringan Bersegmen 60
3.7 Isyarat dalam Pernyataan Soal 64
3.8 Memakainya Sungguh-sungguh 65
3.9 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 66
3.10 Ringkasan Bab 66
3.11 Soal Latihan 67
3.12 Penyelesaian Soal Latihan 67
Daftar Pustaka 68

BAB 4 Input dan Output Cepat 69
4.1 Satu Perjalanan per Paket 69
4.2 Ke Mana Waktu Pergi 70
4.2.1 Apa yang sebenarnya dikerjakan satu `std::cin >> x` 71
4.3 Mengukur Cara-cara Membaca 71
4.3.1 Biaya memori membaca semuanya 76
4.4 Mengukur Cara-cara Menulis 78
4.5 Aturan-aturannya 79
4.6 INTEST: Masukan Raksasa 80
4.7 INOUTEST: Masukan dan Keluaran Raksasa 82
4.8 Catatan Portabilitas 84
4.9 Memakainya Sungguh-sungguh 85
4.10 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 85
4.11 Ringkasan Bab 86
4.12 Soal Latihan 87
4.13 Penyelesaian Soal Latihan 87
Daftar Pustaka 89

BAB 5 Flag Kompiler, Pragma, dan Portabilitas 90
5.1 Program yang Jalan di Rumah 90
5.2 Gagasannya dengan Kata-Kata Sederhana 91
5.2.1 Dari teks sumber ke program yang berjalan 91
5.2.2 Flag, pragma, portabilitas 91
5.2.3 Tiga jenis perbedaan 92
5.3 Soal yang Kita Bawa Sepanjang Bab: FCTRL 92
5.3.1 Soalnya dengan kata-kata kita sendiri 92
5.3.2 Gagasannya: hitung faktor lima 92
5.3.3 Mengapa program membagi, bukan mengalikan 93
5.3.4 Programnya 93
5.4 Apa yang Dilakukan Juri terhadap Sumber Anda 95
5.4.1 Kompiler di seberang sana 95
5.4.2 Menanyai kompiler tentang dirinya 95
5.4.3 Ukuran tipe bilangan bulat 97
5.5 Flag yang Menemukan Bug 97
5.5.1 Peringatan 97
5.5.2 Sanitizer 98
5.5.3 Pengoptimasi menganggap perilaku tak terdefinisi tidak pernah terjadi 99

5.5.4 Mode debug pustaka standar 100
5.6 Flag yang Membuat Kode Lebih Cepat 100
5.6.1 Tingkat optimasi 100
5.6.2 Apa yang kami ukur 101
5.6.3 Optimasi dari dalam sumber 103
5.7 Tangga Risiko Pragma 104
5.8 Daftar Periksa Portabilitas 105
5.8.1 C++ 105
5.8.2 Python 105
5.8.3 Skrip pemeriksa sebelum mengirim 106
5.9 Mengujinya 107
5.10 Menggunakannya Sungguhan 108
5.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 108
5.12 Ringkasan Bab 109
5.13 Soal Latihan 109
5.14 Penyelesaian Soal Latihan 110
Daftar Pustaka 111

BAB 6 Memori dan Pilihan Struktur Data dalam Batas Byte 112
6.1 Koper dengan Batas Berat 112
6.2 Gagasannya dengan Kata-Kata Sederhana 113
6.2.1 Tempat program menyimpan datanya 113
6.2.2 Rumus penimbangan 113
6.2.3 Memori juga memakan waktu 113
6.3 Harga Sebenarnya Satu Elemen 114
6.3.1 Mengukurnya di C++ 114
6.3.2 Membaca angka-angkanya 116
6.3.3 Mengukurnya di Python 117
6.4 PRIME1: Rentang yang Terlalu Besar untuk Disimpan 119
6.4.1 Soalnya 119
6.4.2 Menimbang solusi yang paling jelas 119
6.4.3 Gagasannya: saringan kecil dan sebuah jendela 120
6.4.4 Contoh yang dikerjakan dengan tangan 121
6.4.5 Program-programnya 121
6.5 LKS: Ketika Tabel Tidak Muat 124
6.5.1 Soalnya 124
6.5.2 Solusi buku teks dan beratnya 124
6.5.3 Satu baris sudah cukup 125
6.5.4 Contoh yang dikerjakan dengan tangan 126
6.5.5 Python tidak dapat lolos LKS 127
6.6 Stack Lebih Kecil daripada Dugaan Anda 127
6.6.1 Graf yang berbentuk garis 127
6.6.2 Di mana rekursi patah 128
6.6.3 Kisah yang sama di Python 129
6.7 Mengujinya 130
6.8 Menggunakannya Sungguhan 130
6.9 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 131
6.10 Ringkasan Bab 131
6.11 Soal Latihan 132
6.12 Penyelesaian Soal Latihan 132
Daftar Pustaka 133

BAB 7 Jebakan Implementasi Klasik Penyebab Wrong Answer 135
7.1 Sepuluh Meter Pagar, Sebelas Tiang 135
7.2 Gagasannya dengan Kata-Kata Sederhana 136

7.3 Luapan: EIGHTS 136
7.3.1 Soalnya 136
7.3.2 Gagasannya, beserta bukti 136
7.3.3 Seberapa besar bilangannya? 137
7.3.4 Program yang sama dengan `int` 138
7.3.5 Mengujinya 139
7.4 Melampaui Semua Tipe Bawaan: FCTRL2 140
7.4.1 Soalnya 140
7.4.2 Seberapa jauh `unsigned long long`? 140
7.4.3 Digit dalam larik, seperti di kertas 141
7.4.4 Gejalanya 142
7.4.5 Mengujinya 143
7.5 Nol yang Harus Hilang: ADDREV 143
7.5.1 Soalnya 143
7.5.2 Tempat jebakannya 143
7.5.3 Program-programnya 143
7.5.4 Mengujinya 145
7.6 Keadaan yang Bocor Antarkasus Uji: CANDY 145
7.6.1 Soalnya 145
7.6.2 Gagasannya, beserta bukti 146
7.6.3 Kebocorannya 146
7.7 Pembagian, Sisa Bagi, dan Bilangan Pecahan 149
7.7.1 Dua bahasa, dua jawaban 149
7.7.2 Bilangan pecahan 151
7.8 Selisih Satu dan Bentuk Masukan 151
7.8.1 Menghitung tiang 151
7.8.2 Masukan tidak selalu seperti kelihatannya 152
7.8.3 Bentuk keluaran 152
7.8.4 Pembanding yang berkata “kurang dari atau sama dengan” 152
7.9 Mengujinya 153
7.10 Menggunakannya Sungguhan 153
7.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 154
7.12 Ringkasan Bab 155
7.13 Soal Latihan 155
7.14 Penyelesaian Soal Latihan 156
Daftar Pustaka 156

BAB 8 Men-debug TLE dan WA: Stress Test dan Pemeriksa Brute-Force 158
8.1 Menghitung Uang Dua Kali 158
8.2 Gagasannya dengan Kata-Kata Sederhana 159
8.3 Pengikat Uji 159
8.3.1 Memeriksa generatornya sendiri 161
8.4 INVCNT: Menghitung Inversi 162
8.4.1 Soalnya 162
8.4.2 Brute force-nya 162
8.4.3 Gagasan cepatnya: hitung sambil menggabung 163
8.5 Memburu Jawaban yang Salah 166
8.5.1 Program dengan dua kekeliruan 166
8.5.2 Menemukan dan memperkecilnya 166
8.5.3 Bug yang tak terlihat oleh uji kecil 167
8.6 Memburu Batas Waktu 167
8.6.1 Ukur, jangan menebak 167
8.6.2 Bagaimana waktu tumbuh? 168
8.6.3 Python pada masukan terbesar 169
8.7 PALIN: Jawaban Salah dan Langkah Kuadratik Tersembunyi 170

8.7.1 Soalnya 170
8.7.2 Gagasannya 170
8.7.3 Program-programnya 170
8.7.4 Jawaban salah: simpanan yang hilang 173
8.7.5 Batas waktu yang tak terlihat oleh uji acak 173
8.8 Perkakas Lain untuk Runtime Error 174
8.8.1 Berhenti di baris yang keliru 174
8.8.2 Asersi 175
8.8.3 Cetak ke aliran galat 175
8.8.4 Membelah berkas berisi banyak kasus uji 175
8.8.5 Bila jawabannya tidak tunggal 176
8.9 Menggunakannya Sungguhan 176
8.10 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 177
8.11 Ringkasan Bab 177
8.12 Soal Latihan 178
8.13 Penyelesaian Soal Latihan 178
Daftar Pustaka 179

BAB 9 Pola Pemrograman Dinamis 180
9.1 Hari Pasar Bu Wati 180
9.2 Gagasannya dengan Kata-Kata Sederhana 181
9.2.1 Pertanyaan yang lebih kecil dengan bentuk yang sama 181
9.2.2 Mengerjakannya dengan tangan 181
9.2.3 Lima pertanyaan untuk setiap DP 181
9.3 Percobaan Pertama: Rekursi Polos 182
9.4 Pola 1: Deretan Pilihan (FARIDA) 184
9.5 Pola 2: Menghitung Banyak Cara (ACODE) 186
9.6 Pola 3: Lintasan pada Grid (BYTESM2) 188
9.7 Pola 4: Memilih Barang dengan Anggaran Terbatas (KNAPSACK) 190
9.7.1 Satu baris, dan arah yang menentukan 190
9.7.2 Programnya 192
9.8 Pola 5: Dua String (EDIST) 194
9.9 Pola 6: Interval (MIXTURES) 197
9.10 Mengujinya: Setiap Pola terhadap Brute Force 199
9.11 Membuatnya Lebih Baik: Kecepatan dalam Kedua Bahasa 202
9.12 Memakainya dalam Kehidupan Nyata 203
9.13 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 204
9.14 Ringkasan Bab 204
9.15 Soal Latihan 205
9.16 Penyelesaian Soal Latihan 205
Daftar Pustaka 206

BAB 10 Algoritma Greedy dan Argumen Pertukaran 208
10.1 Balai Warga Pak Harun 208
10.2 Gagasannya dengan Kata-Kata Sederhana 208
10.3 Percobaan Pertama: Kembalian di Kasir 209
10.4 Memilih Kegiatan (BUSYMAN) 211
10.4.1 Aturan yang mana? 212
10.4.2 Mengapa selesai paling awal itu benar 213
10.4.3 Programnya 214
10.4.4 Bila acara punya nilai 216
10.5 Memasangkan Dua Daftar (FASHION) 216
10.6 Tempat bagi Keinginan (BAISED) 218
10.7 Memindahkan Barang di Sepanjang Garis (GERGOVIA) 220
10.8 Mengujinya: Setiap Greedy terhadap Brute Force 222

10.9 Membuatnya Lebih Baik: Kecepatan dalam Kedua Bahasa ... 223
10.9.1 Bila tumpukan mana pun boleh digabung ... 224
10.10 Memakainya dalam Kehidupan Nyata ... 225
10.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya ... 226
10.12 Ringkasan Bab ... 226
10.13 Soal Latihan ... 226
10.14 Penyelesaian Soal Latihan ... 227
Daftar Pustaka ... 229

BAB 11 Binary Search pada Jawaban dan Dua Penunjuk ... 230
11.1 Pasar Malam Bu Ratna ... 230
11.2 Gagasannya dengan Kata-Kata Sederhana ... 231
11.2.1 Pertanyaan yang berganti jawaban sekali ... 231
11.2.2 Templat yang sama pada larik terurut ... 231
11.2.3 Bila ujung atas tidak diketahui ... 232
11.3 Percobaan Pertama: Coba Setiap Nilai ... 232
11.4 Memaksimalkan yang Terkecil (AGGRCOW) ... 234
11.5 Gergaji Setinggi Mungkin (EKO) ... 236
11.6 Binary Search pada Bilangan Real (PIE) ... 239
11.7 Meminimalkan yang Terbesar, dan Membangun Kembali Jawabannya (BOOKS1) ... 241
11.8 Dua Penunjuk (HOTELS) ... 245
11.8.1 Soal cerminnya: deret terpendek yang sudah cukup ... 248
11.9 Mengujinya: Setiap Pencarian terhadap Brute Force ... 248
11.10 Membuatnya Lebih Baik: Kecepatan dan Bantuan Pustaka ... 249
11.11 Memakainya dalam Kehidupan Nyata ... 250
11.12 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya ... 250
11.13 Ringkasan Bab ... 251
11.14 Soal Latihan ... 251
11.15 Penyelesaian Soal Latihan ... 252
Daftar Pustaka ... 253

BAB 12 Perkakas Algoritma Graf ... 255
12.1 Seorang Sopir, Sebuah Jadwal Pelajaran, dan Papan Catur ... 255
12.2 Gagasannya dengan Kata-Kata Sederhana ... 256
12.2.1 Simpul, sisi, dan istilah di sekitarnya ... 256
12.2.2 Menyimpan graf: adjacency list ... 256
12.2.3 Tujuh pertanyaan, tujuh perkakas ... 257
12.3 Pohonkah Ini? Union-Find (PT07Y) ... 257
12.3.1 Soalnya dengan kata-kata kita sendiri ... 257
12.3.2 Gagasannya ... 257
12.3.3 Cara kerja union-find ... 258
12.3.4 Menjalankannya ... 260
12.4 Breadth-First Search: Langkah Kuda (NAKANJ) ... 260
12.4.1 Soalnya dengan kata-kata kita sendiri ... 260
12.4.2 Gagasannya: jelajahi per cincin ... 261
12.4.3 Menjalankannya ... 263
12.5 Dua Kelompok Tanpa Konflik: Graf Bipartit (BUGLIFE) ... 264
12.5.1 Soalnya dengan kata-kata kita sendiri ... 264
12.5.2 Gagasannya ... 264
12.5.3 Menjalankannya ... 267
12.6 Lintasan Terpanjang di Pohon (PT07Z) ... 268
12.6.1 Soalnya dengan kata-kata kita sendiri ... 268
12.6.2 Gagasannya: dua kali penelusuran ... 268
12.6.3 Menjalankannya ... 270
12.7 Lintasan Terpendek Berbobot: Dijkstra (SHPATH) ... 271

12.7.1 Soalnya dengan kata-kata kita sendiri 271
12.7.2 Gagasannya: tetapkan kota terdekat lebih dulu 271
12.7.3 Heap dengan penghapusan malas 272
12.7.4 Menjalankannya, dan biaya 1.000 kueri 274
12.8 Mengerjakan Sesuatu Secara Berurutan: Pengurutan Topologis (TOPOSORT) 275
12.8.1 Soalnya dengan kata-kata kita sendiri 275
12.8.2 Algoritma Kahn 276
12.8.3 Menjalankannya 278
12.9 Menghubungkan Semuanya dengan Murah: Kruskal (MST) 279
12.9.1 Soalnya dengan kata-kata kita sendiri 279
12.9.2 Algoritma Kruskal dan mengapa ia benar 279
12.9.3 Menjalankannya 281
12.10 Menguji Perkakasnya 281
12.11 Menggunakannya Secara Nyata 282
12.12 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 283
12.13 Ringkasan Bab 284
12.14 Soal Latihan 284
12.15 Penyelesaian Soal Latihan 285
Daftar Pustaka 286

BAB 13 Perkakas Teori Bilangan, Geometri, dan String 288
13.1 Sebidang Kebun, Selembar Struk, dan Sebuah Obrolan 288
13.2 Gagasannya dengan Kata-Kata Sederhana 289
13.2.1 Sisa bagi menjaga bilangan tetap kecil 289
13.2.2 Bilangan bulat eksak untuk geometri 289
13.2.3 Jangan pernah melihat karakter yang sama dua kali 289
13.3 Pangkat Raksasa: Pemangkatan Cepat (LASTDIG) 290
13.3.1 Soalnya dengan kata-kata kita sendiri 290
13.3.2 Gagasannya: kuadratkan dan kalikan 290
13.3.3 Menjalankannya 292
13.4 Menghitung Bilangan Koprima untuk Sejuta Bilangan: Totient Euler (ETF) 292
13.4.1 Soalnya dengan kata-kata kita sendiri 292
13.4.2 Rumus hasil kali 292
13.4.3 Menjalankannya 294
13.5 Primakah Bilangan 19 Digit? Miller–Rabin (PON) 294
13.5.1 Soalnya dengan kata-kata kita sendiri 294
13.5.2 Dari Fermat ke Miller–Rabin 294
13.5.3 Hasil kali 128 bit 295
13.5.4 Menjalankannya 297
13.6 Perkakas Geometri: Orientasi, Selimut Konveks, dan Luas 297
13.6.1 Perkalian silang 297
13.6.2 Selimut konveks 297
13.6.3 Luas poligon 298
13.7 Pasangan Titik Terdekat (CLOPPAIR) 301
13.7.1 Soalnya dengan kata-kata kita sendiri 301
13.7.2 Bagi dan kuasai 301
13.7.3 Menjalankannya 305
13.8 Mencari Jarum dalam Tumpukan Jerami: Knuth–Morris–Pratt (NHAY) 306
13.8.1 Soalnya dengan kata-kata kita sendiri 306
13.8.2 Border dan fungsi prefiks 306
13.8.3 Mencari dengan fungsi prefiks 307
13.8.4 Menjalankannya 309
13.9 Prefiks yang Berulang: Periode Sebuah String (PERIOD) 309
13.9.1 Soalnya dengan kata-kata kita sendiri 309
13.9.2 Gagasannya 309

13.9.3 Menjalankannya 311
13.10 Menghitung Substring Berbeda: Suffix Array (DISUBSTR) 311
13.10.1 Soalnya dengan kata-kata kita sendiri 311
13.10.2 Gagasannya: setiap substring adalah prefiks sebuah sufiks 311
13.10.3 Membangun lariknya 311
13.10.4 Menjalankannya 314
13.11 Menguji Perkakasnya 314
13.12 Menggunakannya Secara Nyata 314
13.13 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 315
13.14 Ringkasan Bab 316
13.15 Soal Latihan 316
13.16 Penyelesaian Soal Latihan 317
Daftar Pustaka 318

BAB 14 Menuju Accepted: Studi Kasus Utuh dan Langkah Berikutnya 319
14.1 Dari Pengiriman Pertama Menuju Kebiasaan 319
14.2 Alur Kerja 319
14.3 Studi Kasus 1: Lintasan Prima (PPATH) 320
14.3.1 Langkah 1: membaca 320
14.3.2 Langkah 2: anggaran 321
14.3.3 Langkah 3: idenya dan mengapa benar 321
14.3.4 Langkah 4: brute force 321
14.3.5 Langkah 5: solusi cepat 322
14.3.6 Langkah 6 dan 7: uji kita sendiri, lalu stress test 323
14.3.7 Langkah 8: mengirim 324
14.4 Studi Kasus 2: Lintasan ABC Terpanjang (ABCPATH) 324
14.4.1 Langkah 1 sampai 3 324
14.4.2 Langkah 4 dan 5 324
14.4.3 Langkah 7 beraksi: menangkap bug yang ditanam 326
14.5 Studi Kasus 3: Satu dan Nol (ONEZERO) 327
14.5.1 Langkah 1 dan 2 327
14.5.2 Langkah 3: penelusuran atas sisa bagi 328
14.5.3 Langkah 5, dan jalan salah yang menggoda 329
14.5.4 Langkah 6 dan 7: pemeriksa alih-alih brute force 331
14.6 Ketika Vonisnya Bukan Accepted 332
14.7 Langkah Berikutnya 332
14.7.1 Juri untuk latihan sehari-hari 332
14.7.2 Kontes pada 2026 333
14.7.3 Buku untuk selalu di meja 334
14.7.4 Rencana dua belas minggu 334
14.8 Menggunakannya Secara Nyata 334
14.9 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya 335
14.10 Ringkasan Bab 335
14.11 Soal Latihan 336
14.12 Penyelesaian Soal Latihan 336
Daftar Pustaka 338

Daftar Gambar

Gambar 1.1 Apa yang dilakukan juri terhadap sebuah kiriman: mengompilasi berkas sumber sekali, menjalankan program pada setiap uji tersembunyi dalam batas waktu dan memori, membandingkan setiap keluaran dengan keluaran harapan, dan berhenti pada kegagalan pertama.... 3

Gambar 2.1 Enam zona pernyataan soal dan apa yang dicatat dari masing-masing. Batasan, yang ditandai kuning tua, menentukan algoritma sekaligus kasus tepi.... 25

Gambar 3.1 Langkah yang diperlukan kelas-kelas kompleksitas yang umum seiring bertambahnya ukuran masukan, dibandingkan dengan sekitar satu detik C++ (pita hijau kebiruan) dan Python (pita kuning tua). Titik potong sebuah kurva dengan sebuah pita adalah masukan terbesar yang dapat ditangani kelas itu dalam satu detik.... 52

Gambar 4.1 Waktu terukur untuk membaca dan menulis lima juta bilangan, terbaik dari tiga kali eksekusi, C++ berwarna hijau kebiruan dan Python kuning tua, metode paling lambat berwarna merah. Waktu mutlaknya bergantung pada mesin; yang penting adalah perbandingannya.... 77

Gambar 5.1 4×108 perbandingan yang sama dengan sembilan pengaturan flag, pada skala logaritmik; batang merah memakai instruksi AVX2.... 102

Gambar 5.2 Tangga risiko pragma: setiap anak tangga ke atas membeli kecepatan di sebagian mesin dan menambah satu cara gagal di mesin lain.... 104

Gambar 6.1 Byte per elemen untuk 106 nilai, diukur dengan allocator penghitung di C++ dan dengan `tracemalloc` di Python; skalanya logaritmik.... 119

Gambar 6.2 Saringan bersegmen: bilangan prima kecil sampai 109 mencoret kelipatannya di dalam jendela $[m, n]$; bilangan di luar jendela tidak pernah disimpan.... 120

Gambar 7.1 `int` 32-bit sebagai lingkaran: satu langkah melewati `INT_MAX` mendarat di `INT_MIN`; rumus EIGHTS untuk k besar membutuhkan 49 bit.... 139

Gambar 7.2 ADDREV pada pasangan 4301 dan 6: jumlah 1040 dibalik menjadi "0401", dan nol di depan harus dibuang sebelum dicetak.... 143

Gambar 8.1 Brute force dan program merge sort pada larik acak, dengan skala logaritmik; garis putus-putus adalah batas waktu INVCNT.... 169

Gambar 9.1 Satu barang berukuran 2 dan bernilai 3 ditambahkan ke baris yang semuanya nol. Bila menurun, setiap sel membaca nilai dari sebelum barang itu, sehingga barang dipakai paling banyak sekali; bila menaik, sel 4 membaca sel 2 yang sudah berisi barang itu, sehingga barang terhitung dua kali, dan sel 6 tiga kali.... 191

Gambar 9.2 Tabel jarak sunting untuk SUNDAY (baris) dan SATURDAY (kolom); setiap sel diperoleh dari tetangga kiri, atas, dan kiri-atasnya dengan Persamaan (9.5), dan sel yang diarsir membentuk satu lintasan termurah dengan jarak 3.... 195

Gambar 10.1 Selesai paling awal lebih dulu pada enam kegiatan: tiga kegiatan yang diarsir diambil menurut urutan bernomor; setiap kegiatan yang dilewati dimulai sebelum orang itu kembali luang.... 214

Gambar 10.2 Meluruskan dua pasangan bersilang tidak pernah menurunkan jumlah hasil kali; di sini jumlahnya naik sebesar $(7-2)(9-3) = 30$.... 217

Gambar 11.1 Jawaban atas pertanyaan "dapatkah empat lapak menjaga jarak d?" membentuk satu blok ya dan satu blok tidak; tanda bernomor adalah empat pertanyaan pertama binary search.... 234

Gambar 11.2 Jendela dua penunjuk pada harga 2, 1, 3, 4, 5 dengan anggaran 12: ujung kanan maju satu langkah pada setiap baris, dan ujung kiri melompat maju hanya ketika jumlahnya melampaui anggaran.... 246

Gambar 12.1 Perkakas graf bab ini: pertanyaan yang diajukan soal, teknik yang menjawabnya, dan soal SPOJ yang dipakai untuk berlatih.... 257

Gambar 12.2 Peta uji empat kota kita dengan jarak Dijkstra yang dihasilkan dari Surabaya; sisi yang disorot membentuk pohon lintasan terpendek.... 272

Gambar 13.1 Tiga belas titik pada **Kode 13.7**: tujuh titik sudut selimut, lima titik di dalam, dan satu titik pada sisi selimut yang dibuang oleh *monotone chain*.... 301

Gambar 13.2 Langkah penggabungan algoritma pasangan terdekat: pasangan terbaik tiap separuh memberi d, dan hanya titik di jalur selebar $2d$ di sekitar garis pembagi yang perlu dibandingkan menyeberanginya. 302
Gambar 14.1 Alur kerja utuh yang dipakai dalam setiap studi kasus bab ini, dengan jalan kembali setelah *stress test* gagal atau vonis selain *Accepted* 320
Gambar 14.2 *Breadth-first search* atas sisa bagi modulo 7: lintasan yang disorot menambahkan 0, 0, 1 pada angka 1 di awal dan mencapai sisa 0, sehingga jawabannya 1001. 328

Daftar Tabel

Tabel 1.1 Vonis-vonis juri, namanya di SPOJ, artinya, dan tempat untuk mulai mencari *bug*. 4
Tabel 1.2 Meja kerja buku ini dan versi yang berlaku saat buku ini ditulis. 6
Tabel 2.1 Daftar periksa membaca: delapan pertanyaan yang dijawab sebelum menulis kode, dan vonis yang cenderung muncul jika masing-masing dilewati. 27
Tabel 2.2 *Shunting-yard algorithm* pada `a+b*c-d` dengan prioritas ONP. 39
Tabel 3.1 Anggaran kompleksitas dalam C++: ukuran masukan terbesar dalam batasan dan kompleksitas paling lambat yang muat dalam sekitar satu detik. Untuk Python, turunlah satu atau dua baris. 52
Tabel 3.2 Isyarat dalam pernyataan soal dan keluarga metode yang biasanya ditunjuknya, dengan bab buku ini yang membahas setiap keluarga. 65
Tabel 4.1 Pembaca `readInt` dari **Kode 4.2** pada masukan `-307`, satu karakter per baris tabel. 74
Tabel 5.1 Apa yang terjadi pada kiriman C++ di SPOJ, dan siapa yang mengendalikan setiap langkah. 91
Tabel 5.2 Ukuran bilangan bulat dalam dua model data 64-bit; baris tebal adalah yang berbeda. 97
Tabel 5.3 Keluaran `ub_check.cpp` dengan berbagai kompiler dan tingkat optimasi. 100
Tabel 5.4 Waktu `flags_bench.cpp` dengan sembilan pengaturan (tercepat dari tiga kali jalan, g++ 13.3, Xeon 2,1 GHz). 102
Tabel 5.5 Fitur yang rusak ketika program C++ pindah ke platform lain, dan apa yang sebaiknya ditulis. 105
Tabel 6.1 Aturan praktis memilih kontainer menurut beratnya (g++ 13 dengan libstdc++, CPython 3.11). 119
Tabel 6.2 Baris tunggal `best` pada `lks.cpp` setelah setiap barang, untuk kapasitas 10. 126
Tabel 6.3 DFS rekursif dan iteratif pada sebuah lintasan, dengan berbagai ukuran stack (g++ 13.3, `-O2`). 129
Tabel 7.1 Tujuh keluarga jebakan implementasi, gejala yang ditinggalkan masing-masing, dan tempat bab ini membahasnya. 136
Tabel 7.2 Pembagian bulat, sisa bagi, dan pembulatan di C++ dan Python. 150
Tabel 7.3 Hasil uji stres untuk program-program bab ini (300 uji acak per putaran). 153
Tabel 8.1 Apa yang biasanya berarti dari setiap vonis, dan perkakas pertama yang sebaiknya diraih. 159
Tabel 8.2 Waktu kedua program INVCNT pada larik acak yang membesar (tercepat dari tiga kali jalan, September 2026). 168
Tabel 8.3 Waktu tiga program PALIN pada masukan sejuta digit. 174
Tabel 9.1 Brute force dan masukan kecil yang dipakai untuk memeriksa keenam program DP. 200
Tabel 9.2 Masukan terbesar dan waktu CPU yang terukur; kolom terakhir adalah penilaian kami tentang seberapa longgar ruang bagi Python. 202
Tabel 9.3 Enam pola DP: state-nya, keputusan terakhir yang memberi rekurensi, dan urutan pengisiannya. 203
Tabel 10.1 Brute force yang dipakai untuk memeriksa keempat program greedy. 223
Tabel 10.2 Masukan terbesar dan waktu CPU terukur untuk keempat program greedy. 223
Tabel 10.3 Pertanyaan yang memutuskan antara algoritma greedy dan sesuatu yang lebih kuat. 225
Tabel 11.1 Brute force yang dipakai untuk memeriksa kelima program bab ini. 249
Tabel 11.2 Masukan terbesar dan waktu CPU terukur untuk kelima program. 249

Tabel 11.3 Bentuk-bentuk khas soal yang diselesaikan dengan binary search pada jawaban dan dengan dua penunjuk. 250
Tabel 12.1 Tiga cara menyimpan graf, dengan biaya dua pertanyaan yang paling sering diajukan algoritma. 256
Tabel 12.2 Waktu pada masukan terbesar yang kita bangkitkan per soal, dibulatkan dari jalankan di atas, dan apakah pengiriman Python realistis. 282
Tabel 13.1 Soal-soal bab ini, batas yang menyingkirkan cara naif, dan teknik yang sesuai. 290
Tabel 13.2 Fungsi prefiks `aabaab`, satu prefiks per baris. 306
Tabel 14.1 Delapan langkah alur kerja, pertanyaan yang dijawab setiap langkah, dan tempat buku ini mengajarkannya. 320
Tabel 14.2 Arti yang biasa dari setiap vonis, apa yang dilakukan berikutnya, dan tempat buku ini membahasnya. 332
Tabel 14.3 Tempat berlatih setelah SPOJ, dan keunggulan masing-masing juri. 333
Tabel 14.4 Rencana latihan dua belas minggu setelah buku ini, dengan satu kebiasaan yang dilatih dengan sengaja di setiap blok. 334

Daftar Kode

Kode 1.1 `.vscode/tasks.json`: *build task* yang mengompilasi berkas di editor dengan opsi buku ini. 9
Kode 1.2 `check_env.cpp`: melaporkan kompiler, standar C++, dan ukuran tipe bilangan bulat. 10
Kode 1.3 `check_env.py`: melaporkan versi Python dan menunjukkan bahwa bilangan bulat Python tidak berukuran tetap. 11
Kode 1.4 `test_life.cpp`: solusi TEST dalam C++. 12
Kode 1.5 `test_life.py`: solusi TEST dalam Python. 13
Kode 1.6 `gen_test_life.py`: menulis empat uji untuk TEST, dengan keluaran harapan yang dihitung oleh kode sesederhana mungkin. 14
Kode 1.7 `minijudge.py`: mengompilasi program C++ (atau menjalankan program Python) pada setiap uji di sebuah folder, dengan batas waktu, dan mencetak vonis seperti SPOJ. 15
Kode 1.8 `test_wa.cpp`: solusi salah yang memeriksa 42 baru setelah mencetaknya. 17
Kode 1.9 `test_re.py`: solusi salah yang menyimpan bilangan dalam list berukuran tetap dan berhenti paksa pada masukan yang panjang. 18
Kode 1.10 `test_ce.cpp`: solusi dengan satu titik koma yang hilang. 18
Kode 2.1 `sum_until_eof.cpp`: bentuk masukan "sampai akhir berkas" dalam C++. 28
Kode 2.2 `sum_until_eof.py`: bentuk masukan yang sama dalam Python, dengan membaca semua token sekaligus. 28
Kode 2.3 `addrev.cpp`: ADDREV dengan pembalikan digit pada bilangan bulat. 30
Kode 2.4 `addrev.py`: ADDREV dalam Python, membalik melalui string. 30
Kode 2.5 `addrev_bug.cpp`: ADDREV yang salah, yang mempertahankan nol di depan pada jumlah yang dibalik. 31
Kode 2.6 `stress.py`: menjalankan generator, *brute force*, dan program pada banyak *seed*, lalu mencetak masukan pertama yang jawabannya berbeda. 32
Kode 2.7 `gen_addrev.py`: masukan ADDREV acak yang kecil dan kaya nol. 34
Kode 2.8 `brute_addrev.py`: ADDREV langsung dari definisinya. 34
Kode 2.9 `fctrl2.cpp`: FCTRL2 dengan perkalian cara sekolah pada vektor digit desimal. 36
Kode 2.10 `fctrl2.py`: FCTRL2 dalam Python, yang bilangan bulatnya tidak berukuran tetap. 36
Kode 2.11 `check_fctrl2.py`: memeriksa program FCTRL2 pada setiap masukan yang mungkin. 37
Kode 2.12 `onp.cpp`: ONP dengan *shunting-yard algorithm*; prioritasnya adalah prioritas dari pernyataan soal. 39
Kode 2.13 `onp.py`: algoritma yang sama dalam Python, dengan list sebagai tumpukan. 41

Kode 2.14 `brute_onp.py`: ONP dengan *recursive descent*, satu tingkat tata bahasa per prioritas. 41
Kode 2.15 `gen_onp.py`: ekspresi ONP acak dengan kurung minimal dan kurung tambahan. 42
Kode 3.1 `ops_budget.cpp`: mengukur langkah gelung sederhana per detik dan waktu mengurutkan 107 bilangan dalam C++. 49
Kode 3.2 `ops_budget.py`: pengukuran yang sama dalam Python. 50
Kode 3.3 `fctrl_naive.cpp`: FCTRL dengan menghitung faktor 5 dari setiap kelipatan 5, benar tetapi jauh terlalu lambat. 53
Kode 3.4 `fctrl.cpp`: FCTRL dengan rumus Legendre, sekitar 13 pembagian per kasus. 54
Kode 3.5 `fctrl.py`: rumus yang sama dalam Python, dengan membaca seluruh masukan sekaligus. 55
Kode 3.6 `brute_fctrl.py`: FCTRL dengan menghitung $n!$ seutuhnya lalu menghitung nol di ujungnya. 56
Kode 3.7 `gen_fctrl.py`: masukan FCTRL kecil dengan nilai batas pangkat 5. 56
Kode 3.8 `make_fctrl_tests.py`: contoh soal dan uji berukuran maksimum untuk FCTRL, dalam format `minijudge.py` 57
Kode 3.9 `lastdig.cpp`: LASTDIG dengan pengkuadratan berulang modulo 10. 58
Kode 3.10 `lastdig.py`: metode yang sama dalam Python; `pow(a, b, 10)` bawaan melakukan hal yang sama. 59
Kode 3.11 `prime1.cpp`: PRIME1 dengan saringan bersegmen; bilangan prima dasar dihitung sekali, setiap rentang disaring terpisah. 61
Kode 3.12 `prime1.py`: saringan bersegmen yang sama dalam Python, mencoret dengan penugasan irisan. 63
Kode 4.1 `gen_intest.py`: menulis masukan berbentuk INTEST dengan n bilangan acak. 72
Kode 4.2 `read_bench.cpp`: empat cara membaca masukan yang sama dalam C++, dari `cin` biasa sampai pembaca buatan sendiri di atas `fread` 72
Kode 4.3 `read_bench.py`: tiga cara membaca masukan yang sama dalam Python. 74
Kode 4.4 `write_bench.cpp`: empat cara menulis keluaran yang sama dalam C++. 78
Kode 4.5 `write_bench.py`: tiga cara menulis keluaran yang sama dalam Python. 78
Kode 4.6 `intest.cpp`: INTEST dengan pembaca bilangan tak negatif berbasis `fread`. 81
Kode 4.7 `intest.py`: INTEST dalam Python, dengan membaca semua token sekaligus. 81
Kode 4.8 `inoutest.cpp`: INOUTEST dengan aliran tanpa sinkronisasi dan keluaran yang dikumpulkan dalam satu string. 83
Kode 4.9 `inoutest.py`: INOUTEST dalam Python, dengan membaca semua token dan menggabungkan semua hasil kali. 83
Kode 5.1 `fctrl.cpp`: banyaknya nol di ujung $N!$, ditulis agar berperilaku sama di setiap platform. 93
Kode 5.2 `fctrl.py`: hitungan yang sama dalam Python, dengan membaca seluruh masukan sekaligus. 94
Kode 5.3 `platform_probe.cpp`: mencetak fakta tentang kompiler dan mesin yang menentukan apakah sebuah program berperilaku sama di tempat lain. 95
Kode 5.4 `platform_probe.py`: versi, implementasi, dan batas Python yang bisa berbeda antara mesin Anda dan juri. 96
Kode 5.5 `fctrl_mulloop.cpp`: rumus yang sama dengan pangkat 5 yang membesar; benar di dalam batasan soal, perilaku tak terdefinisi tepat di luarnya. 98
Kode 5.6 `ub_check.cpp`: syarat loop `i > 0` bergantung pada luapan, sehingga banyaknya langkah bergantung pada tingkat optimasi. 99
Kode 5.7 `flags_bench.cpp`: 4×108 perbandingan yang kecepatannya hanya bergantung pada flag kompiler. 100
Kode 5.8 `flags_bench_pragma.cpp`: uji kecepatan dengan pragma optimize dan pragma target, diletakkan sesudah include. 103
Kode 5.9 `portability_check.py`: mengompilasi setiap program dengan beberapa standar, kompiler, dan mode pemeriksaan lalu melaporkan kegagalannya. 106
Kode 5.10 `brute_fctrl.py`: brute force untuk FCTRL membangun $N!$ dan menghitung nolnya; hanya N kecil yang layak. 107
Kode 6.1 `memsize.cpp`: allocator penghitung mengukur byte heap yang dipakai setiap kontainer untuk 106 nilai. 114

Kode 6.2 `memsize.py`: `tracemalloc` mengukur byte per elemen kontainer Python yang memuat 106 nilai. 117
Kode 6.3 `prime1.cpp`: saringan bersegmen; memorinya bergantung pada n dan $n - m$, tidak pernah pada n. 121
Kode 6.4 `prime1.py`: saringan bersegmen pada jendela `bytearray`, mencoret satu irisan utuh sekaligus. 122
Kode 6.5 `lks_table.cpp`: knapsack dengan tabel penuh; benar, tetapi uji terbesar membutuhkan sekitar 4 GB. 124
Kode 6.6 `lks.cpp`: knapsack dengan satu baris berisi $K + 1$ nilai, dikunjungi dari kapasitas tinggi ke rendah. 125
Kode 6.7 `lks.py`: knapsack satu baris dalam Python; benar, tetapi jauh terlalu lambat untuk uji terbesar. 127
Kode 6.8 `deep_dfs.cpp`: penelusuran depth-first pada lintasan berisi n simpul, secara rekursif dan dengan stack eksplisit. 128
Kode 6.9 `brute_lks.py`: brute force LKS mencoba setiap himpunan bagian barang. 130
Kode 7.1 `eights.cpp`: rumus Persamaan (7.1), dihitung dengan aritmetika 64-bit. 137
Kode 7.2 `eights.py`: rumus yang sama dalam Python, yang bilangan bulatnya tidak pernah meluap. 138
Kode 7.3 `eights_int.cpp`: rumus dengan `int`; kompilasi bersih, contoh benar, jawaban salah untuk k besar. 138
Kode 7.4 `brute_eights.py`: brute force untuk EIGHTS mencari ke atas dan tidak memerlukan rumus. 139
Kode 7.5 `fctrl2_ull.cpp`: faktorial dalam tipe bawaan terbesar; benar sampai 20! dan diam-diam salah mulai 21! 140
Kode 7.6 `fctrl2.cpp`: faktorial sampai 158 digit, disimpan mulai dari digit paling tidak signifikan dan dikalikan seperti di kertas. 141
Kode 7.7 `fctrl2.py`: FCTRL2 dalam Python, dengan `math.factorial`. 142
Kode 7.8 `addrev.cpp`: pembalikan yang selalu membuang nol di depan, dan penjumlahan bersusun pada string digit. 144
Kode 7.9 `addrev.py`: ADDREV dalam Python; `int()` atas string yang dibalik membuang nol di depan. 144
Kode 7.10 `candy.cpp`: langkah-langkah Persamaan (7.5), dengan setiap variabel sebuah blok dibuat baru di dalam loop. 146
Kode 7.11 `candy_leak.cpp`: program yang sama dengan total dideklarasikan di luar loop; setiap blok setelah yang pertama mewarisi permen dari blok-blok sebelumnya. 147
Kode 7.12 `candy.py`: CANDY dalam Python, membaca semua token sekaligus dan berhenti pada penanda -1. 148
Kode 7.13 `arith_demo.cpp`: pembagian bulat, sisa bagi, pembulatan, dan perbandingan bilangan pecahan di C++. 149
Kode 7.14 `arith_demo.py`: pertanyaan yang sama dalam Python; beberapa jawabannya berbeda dari C++. 150
Kode 8.1 `stress.py`: menjalankan generator, brute force, dan kandidat untuk banyak benih dan berhenti pada perbedaan pertama. 159
Kode 8.2 `gen_invcnt.py`: masukan INVCNT acak dengan nilai berbeda dan baris kosong sesuai format masukan aslinya; ukuran larik terbesar dapat dipilih. 161
Kode 8.3 `invcnt_brute.cpp`: brute force memeriksa setiap pasangan; jelas benar dan berwaktu kuadratik. 162
Kode 8.4 `invcnt.cpp`: penghitungan inversi dengan merge sort, memakai penghitung 64-bit. 164
Kode 8.5 `invcnt.py`: merge sort bottom-up yang menghitung inversi, tanpa rekursi. 164
Kode 8.6 `measure.py`: menjalankan perintah pada berkas masukan dan melaporkan waktu serta memori puncaknya. 167
Kode 8.7 `palin.cpp`: palindrom berikutnya dalam waktu linear pada string digit. 170
Kode 8.8 `palin.py`: palindrom berikutnya dalam Python, hanya pada string. 171
Kode 8.9 `brute_palin.py`: brute force PALIN menghitung naik sampai bertemu palindrom. 172

Kode 8.10 `bounds_demo.cpp`: loop jumlah awalan yang membaca satu elemen melewati ujung vector. 174
Kode 9.1 `dp_intro.cpp`: rekursi polos, rekursi dengan memo, dan tabel, ketiganya menghitung Persamaan (9.1). 182
Kode 9.2 `farida.cpp`: FARIDA dengan dua nilai bergulir. 184
Kode 9.3 `farida.py`: penyelesaian yang sama dalam Python, dengan membaca seluruh masukan sekaligus. 185
Kode 9.4 `acode.cpp`: menghitung bacaan dengan Persamaan (9.2). 186
Kode 9.5 `acode.py`: hitungan yang sama dalam Python. 187
Kode 9.6 `bytesm2.cpp`: DP grid dengan dua baris. 188
Kode 9.7 `bytesm2.py`: DP yang sama dalam Python, dengan baris berbantalan sebagai pengganti pemeriksaan tepi. 189
Kode 9.8 `knap_direction.cpp`: kedua arah gelung dibandingkan dengan brute force 191
Kode 9.9 `knapsack.cpp`: 0/1 knapsack dalam satu baris, gelung kapasitas menurun. 192
Kode 9.10 `knapsack.py`: DP yang sama dalam Python, satu baris baru per barang. 193
Kode 9.11 `edist.cpp`: jarak sunting dengan dua baris. 195
Kode 9.12 `edist.py`: DP yang sama dalam Python, dengan gelung dalam yang ramping. 196
Kode 9.13 `mixtures.cpp`: DP interval yang diisi menurut panjang yang menaik. 197
Kode 9.14 `mixtures.py`: DP interval yang sama dalam Python, dengan warna semua interval dihitung lebih dulu. 198
Kode 9.15 `tests/stress.py`: pengendali stress test untuk keenam soal bab ini. 200
Kode 10.1 `coin_change.cpp`: pemberian kembalian secara greedy dibandingkan dengan minimum yang dihitung DP, untuk dua sistem koin. 210
Kode 10.2 `greedy_keys.cpp`: tiga aturan greedy untuk pemilihan kegiatan dibandingkan dengan brute force. 212
Kode 10.3 `busyman.cpp`: BUSYMAN, urutkan menurut waktu selesai dan ambil yang muat. 214
Kode 10.4 `busyman.py`: greedy yang sama dalam Python. 215
Kode 10.5 `fashion.cpp`: urutkan kedua daftar lalu pasangkan menurut urutan. 217
Kode 10.6 `fashion.py`: pemasangan yang sama dalam Python. 217
Kode 10.7 `baised.cpp`: urutkan keinginan lalu jumlahkan jaraknya. 219
Kode 10.8 `baised.py`: greedy yang sama dengan counting sort. 219
Kode 10.9 `gergovia.cpp`: jumlah nilai mutlak jumlah awalan. 221
Kode 10.10 `gergovia.py`: jumlah yang sama dengan `itertools.accumulate` 221
Kode 10.11 `fractional.cpp`: knapsack pecahan menurut nilai per kilogram, berdampingan dengan jawaban 0/1 yang ditemukan brute force. 227
Kode 11.1 `bsearch_demo.cpp`: pemeriksaan untuk sebuah jarak, pemindaian linear atas semua jarak, dan binary search, dengan pemeriksaan yang dihitung. 233
Kode 11.2 `aggrcow.cpp`: binary search pada jarak dengan pemeriksaan greedy. 235
Kode 11.3 `aggrcow.py`: pencarian yang sama dalam Python, dengan keluar lebih awal begitu semua sapi muat. 235
Kode 11.4 `eko.cpp`: binary search pada tinggi gergaji, dengan masukan per blok dan jumlah 64 bit. 237
Kode 11.5 `eko.py`: EKO tanpa binary search, dengan Persamaan (11.4) pada tinggi yang sudah terurut. 239
Kode 11.6 `pie.cpp`: binary search pada bilangan real dengan banyak putaran yang tetap. 240
Kode 11.7 `pie.py`: pencarian yang sama dalam Python. 240
Kode 11.8 `books1.cpp`: binary search untuk batas terkecil, lalu pembagian yang dibangun kembali dari kanan. 242
Kode 11.9 `books1.py`: pencarian dan pembangunan kembali yang sama dalam Python. 244
Kode 11.10 `hotels.cpp`: jendela dua penunjuk. 246
Kode 11.11 `hotels.py`: jendela yang sama dalam Python. 247
Kode 11.12 `lis.cpp`: subbarisan menaik tegas terpanjang dengan `lower_bound` 252
Kode 12.1 `pt07y.cpp`: graf adalah pohon bila memiliki $n - 1$ sisi dan tidak satu pun menutup siklus. 258

Kode 12.2 `pt07y.py`: uji yang sama dalam Python. 259
Kode 12.3 `nakanj.cpp`: satu *breadth-first search* per kueri pada 64 petak papan. 261
Kode 12.4 `nakanj.py`: penelusuran yang sama dalam Python, dengan `collections.deque` sebagai antrean. 262
Kode 12.5 `buglife.cpp`: pewarnaan dua warna dengan BFS pada setiap komponen terhubung, dengan pembaca *buffer* byte untuk masukan sejuta baris. 264
Kode 12.6 `buglife.py`: pewarnaan dua warna yang sama dalam Python. 266
Kode 12.7 `pt07z.cpp`: diameter pohon dari dua kali *breadth-first search*. 268
Kode 12.8 `pt07z.py`: dua penelusuran yang sama dalam Python. 269
Kode 12.9 `shpath.cpp`: Dijkstra dengan *heap* biner dan penghapusan malas, satu penelusuran per kueri, berhenti di tujuan. 272
Kode 12.10 `shpath.py`: algoritma yang sama dengan `heapq`. 274
Kode 12.11 `toposort.cpp`: algoritma Kahn dengan *min-heap* berisi pekerjaan siap, dan pendeteksian siklus dengan menghitung. 276
Kode 12.12 `toposort.py`: algoritma yang sama dengan `heapq`. 277
Kode 12.13 `mst.cpp`: algoritma Kruskal dengan *union-find* dan total 64 bit. 279
Kode 12.14 `mst.py`: algoritma yang sama dalam Python; bilangan bulat Python tidak pernah meluap. 280
Kode 13.1 `lastdig.cpp`: pemangkatan cepat modulo 10. 290
Kode 13.2 `lastdig.py`: gelung yang sama dalam Python; fungsi bawaan `pow(a, b, 10)` melakukan hal yang sama dalam satu panggilan. 291
Kode 13.3 `etf.cpp`: saringan totient; setiap kueri dijawab dari tabel. 293
Kode 13.4 `etf.py`: $\varphi(n)$ lewat pemfaktoran dengan bilangan prima di bawah 1.000. 293
Kode 13.5 `pon.cpp`: Miller–Rabin deterministik dengan dua belas basis prima dan hasil kali 128 bit. 295
Kode 13.6 `pon.py`: uji yang sama dalam Python. 296
Kode 13.7 `geometry.cpp`: orientasi dengan perkalian silang, selimut konveks *monotone chain*, dan luas *shoelace*. 298
Kode 13.8 `geometry.py`: perkakas yang sama dalam Python. 299
Kode 13.9 `cloppair.cpp`: bagi dan kuasai atas jarak kuadrat bulat; setiap panggilan meninggalkan bagian lariknya terurut menurut y. 302
Kode 13.10 `cloppair.py`: algoritma yang sama dalam Python, setiap panggilan mengembalikan *list* baru terurut menurut y. 304
Kode 13.11 `nhay.cpp`: KMP yang mengalirkan *haystack* karakter demi karakter, hanya menyimpan *needle* dan fungsi prefiksnya. 307
Kode 13.12 `nhay.py`: algoritma yang sama dalam Python, yang membaca setiap *haystack* sebagai satu *string*. 308
Kode 13.13 `period.cpp`: fungsi prefiks dan Persamaan (13.9) untuk setiap prefiks. 309
Kode 13.14 `period.py`: hal yang sama dalam Python. 310
Kode 13.15 `disubstr.cpp`: *suffix array* dengan *prefix doubling*, LCP dengan Kasai, dan Persamaan (13.10). 312
Kode 13.16 `disubstr.py`: sufiks terurut dan LCP antartetangga, cukup untuk $n \leq 1000$. 313
Kode 14.1 `ppath.cpp`: saringan di bawah 104 dan satu *breadth-first search* per kasus uji. 322
Kode 14.2 `ppath.py`: penelusuran yang sama dalam Python, dengan *generator* untuk tetangga dan penugasan irisan dalam saringan. 322
Kode 14.3 `abcpath.cpp`: penelusuran bermemo; tabel memo dikosongkan untuk setiap grid. 324
Kode 14.4 `abcpath.py`: rekurens yang sama dievaluasi dari bawah ke atas, dari `Z` turun ke `A`. 325
Kode 14.5 `onezero_wrong.cpp`: BFS atas bilangan itu sendiri, yang meluap diam-diam begitu melewati dua puluh digit. 329
Kode 14.6 `onezero.cpp`: BFS atas sisa bagi modulo n dengan tautan induk, lalu digit-digitnya dibaca kembali dari sisa 0. 330
Kode 14.7 `onezero.py`: penelusuran yang sama dalam Python, dengan *list* dan indeks kepala sebagai antrean. 330

Daftar Pseudokode

Pseudokode 9.1 Memulihkan hari-hari terpilih dari tabel FARIDA dengan berjalan mundur dari hari terakhir. 206
Pseudokode 10.1 Menggabungkan tumpukan dua demi dua dengan biaya total terkecil: selalu gabungkan dua yang terkecil. 224

Daftar Persamaan

(1.1) 5
(1.2) 11
(2.1) 29
(2.2) 35
(3.1) 49
(3.2) 51
(3.3) 51
(3.4) 54
(3.5) 58
(3.6) 61
(3.7) 61
(4.1) 70
(5.1) 92
(5.2) 93
(5.3) 93
(6.1) 113
(6.2) 120
(6.3) 120
(6.4) 121
(6.5) 124
(6.6) 124
(7.1) 136
(7.2) 137
(7.3) 141
(7.4) 141
(7.5) 146
(7.6) 151
(7.7) 151
(7.8) 151
(8.1) 162
(8.2) 163
(8.3) 163
(8.4) 169
(9.1) 181
(9.2) 186
(9.3) 188
(9.4) 190
(9.5) 194
(9.6) 197

(10.1)....213
(10.2)....216
(10.3)....217
(10.4)....218
(10.5)....221
(11.1)....231
(11.2)....231
(11.3)....237
(11.4)....239
(11.5)....240
(11.6)....242
(11.7)....246
(12.1)....271
(12.2)....272
(13.1)....289
(13.2)....290
(13.3)....292
(13.4)....294
(13.5)....294
(13.6)....297
(13.7)....298
(13.8)....306
(13.9)....309
(13.10)....311
(14.1)....321
(14.2)....324
(14.3)....328

Kata Pengantar

Seorang mahasiswi mengirimkan solusi pertamanya ke sebuah *online judge* tengah malam, dan diberi tahu, dengan huruf merah, bahwa solusinya salah. Programnya mencetak jawaban yang benar di laptopnya. Program itu mencetaknya untuk setiap contoh dalam soal. Yang dilihat juri, dan tidak dilihatnya, adalah sebaris kalimat dalam soal yang hanya dibacanya sekilas, masukan yang sepuluh ribu kali lebih besar daripada contohnya, sebuah bilangan yang tidak lagi muat dalam `int`, dan kompiler yang bukan kompiler di mejanya. Pemrograman kompetitif adalah keterampilan menutup celah itu: menulis, dalam waktu terbatas, program yang benar untuk setiap masukan yang diizinkan soal dan cukup cepat untuk masukan yang terbesar, serta mengetahui bahwa program itu memang demikian sebelum juri mengatakannya.

SPOJ, *Sphere Online Judge*, adalah salah satu tempat tertua dan terluas untuk mempelajari keterampilan itu. SPOJ memiliki ribuan soal yang terkumpul selama dua dasawarsa, menerima program dalam puluhan bahasa, dan menjawab setiap pengiriman dalam hitungan detik dengan sebuah vonis: *Accepted*, *Wrong answer*, *Time limit exceeded*, *Runtime error*, atau *Compilation error*. Buku ini memakai soal-soal klasiknya — TEST, PRIME1, ADDREV, FCTRL2, AGGRCOW, SHPATH, dan banyak lagi — sebagai latihan untuk mengajarkan seluruh keterampilan itu, dari membaca soal sampai membuat soal yang sulit diterima.

Untuk siapa buku ini ditulis

Buku ini lengkap dengan sendirinya, dan Anda dianggap seorang pemula. Jika Anda belum pernah ikut lomba, belum pernah mengirim program ke *online judge*, dan belum yakin apa arti "O(n log n)", Andalah pembaca yang dibayangkan saat buku ini ditulis. Setiap istilah diperkenalkan dengan kata-kata sederhana sebelum dipakai, setiap gagasan lebih dulu dikerjakan dengan tangan pada contoh kecil, matematika yang diperlukan dijelaskan di tempat ia diperlukan, dan setiap program dalam buku ini sudah dijalankan sebelum keluarannya dicetak di sini. Bab 1 dibuka dengan sebuah vonis merah tengah malam dan ditutup dengan meja kerja, soal pertama yang diterima, dan sebuah juri kecil buatan Anda sendiri.

Pembaca yang sudah bisa memprogram, dan siswa atau mahasiswa yang bersiap menghadapi olimpiade, lomba nasional, atau ICPC, akan menemukan separuh lain buku ini: setiap algoritma dibuktikan kebenarannya dan dihitung biayanya, anggaran waktu dan memori juri diterjemahkan menjadi batas nyata tentang apa yang boleh dilakukan sebuah program, jebakan yang mengubah gagasan yang benar menjadi jawaban yang salah didaftar satu per satu, serta program yang utuh dan teruji, siap disesuaikan. Prinsip itu membuat buku ini tebal. Buku yang lebih tipis akan menganggap Anda sudah menguasai bagian-bagian yang sulit, dan buku semacam itu tidak berguna bagi orang yang belum menguasainya.

Satu gagasan, dua bahasa

Setiap soal dalam buku ini diselesaikan lebih dulu dalam **C++17**, karena C++ adalah bahasa pemrograman kompetitif: cepat, pustaka standarnya memuat sebagian besar struktur data yang diperlukan dalam lomba, dan setiap juri menerimanya. Jika Python dapat lolos batas waktu, solusi yang sama juga ditulis dalam **Python 3**, dan kedua versi itu membaca masukan yang sama dan mencetak keluaran yang identik. Python ringkas dan mudah dibaca, dan melihat satu gagasan

ditulis dengan dua cara adalah jalan tercepat untuk memahami gagasan itu sendiri, bukan sekadar kebiasaan satu bahasa. Jika Python tidak dapat lolos, bahkan dengan teknik masukan dan keluaran tercepat, buku ini mengatakannya, dan menjelaskan sebabnya.

Tidak ada program dalam buku ini yang dipercaya hanya karena lolos contoh soal. Setiap program **diuji dengan *stress test* terhadap *brute force***: program kedua yang sengaja dibuat sederhana dan lambat, yang kebenarannya jelas, dijalankan berdampingan dengannya pada ribuan masukan acak kecil buatan sebuah *generator*, dan pengujian berhenti pada perbedaan pertama. Program-programnya adalah program ajar kita sendiri, ditulis untuk buku ini; setiap soal diceritakan ulang dengan kata-kata kita sendiri, disertai kode dan alamat SPOJ-nya agar Anda dapat membaca aslinya. Semuanya ditulis dan dijalankan di **Visual Studio Code**, dengan kompiler GNU C++ dan Python 3.14, sama saja di Windows, macOS, maupun Linux; Bab 1 memasangnya langkah demi langkah pada ketiga sistem itu.

Isi buku ini

Bab 1 menjelaskan apa itu pemrograman kompetitif, apa yang ditawarkan SPOJ, dan apa yang sebenarnya dilakukan juri terhadap program yang dikirim. Bab ini memasang meja kerja, menyelesaikan soal SPOJ pertama, TEST, dan membangun juri kecil buatan sendiri yang mengompilasi, menjalankan, dan membandingkan program seperti yang dilakukan SPOJ.

Bab 2 sampai 4 membahas soal sebelum algoritmanya. Bab 2 mengajari Anda membaca soal seperti juri: anatominya, daftar periksa membaca, tiga bentuk masukan juri, dan frasa-frasa yang menentukan segalanya, pada ADDREV, FCTRL2, dan ONP. Bab 3 mengubah batas waktu menjadi anggaran langkah, menghitung langkah sebuah algoritma, dan menunjukkan bagaimana batasan soal menunjuk ke metodenya, dengan FCTRL, LASTDIG, dan *segmented sieve* pada PRIME1. Bab 4 mengukur setiap cara membaca dan menulis data dalam kedua bahasa dan menarik aturan untuk masukan dan keluaran yang sangat besar, pada INTEST dan INOUTEST.

Bab 5 sampai 8 membahas segala hal yang berdiri di antara gagasan yang benar dan program yang diterima. Bab 5 menjelaskan apa yang dilakukan juri terhadap kode sumber Anda: *flag* kompiler yang menemukan *bug*, *flag* dan *pragma* yang mempercepat kode, tangga risiko *pragma*, dan daftar periksa portabilitas. Bab 6 membelanjakan anggaran kedua, memori: berapa sebenarnya biaya satu elemen, rentang yang terlalu besar untuk disimpan pada PRIME1, tabel yang tidak muat pada LKS, dan *stack* yang lebih kecil daripada dugaan Anda. Bab 7 mendaftar jebakan implementasi klasik penyebab *Wrong answer* — *overflow*, bilangan yang melampaui setiap tipe bawaan, nol yang harus dihilangkan, keadaan yang bocor antarkasus uji, pembagian dan *floating point*, serta kesalahan selisih satu. Bab 8 membangun perangkat uji yang diandalkan seluruh sisa buku: *generator*, *brute force*, dan *stress test* yang memburu jawaban salah dan langkah lambat yang tersembunyi, pada INVCNT dan PALIN.

Bab 9 sampai 11 adalah inti algoritmanya. Bab 9 mengajarkan pemrograman dinamis melalui enam pola — deretan pilihan, menghitung banyak cara, lintasan pada kisi, *knapsack*, dua string, dan interval — pada FARIDA, ACODE, BYTESM2, KNAPSACK, EDIST, dan MIXTURES. Bab 10 menunjukkan kapan pilihan *greedy* aman dan bagaimana argumen pertukaran membuktikannya, pada BUSYMAN, FASHION, BAISED, dan GERGOVIA. Bab 11 mencari jawaban alih-alih menghitungnya, dengan *binary search* pada jawaban dan dua penunjuk, pada AGGRCOW, EKO, PIE, BOOKS1, dan HOTELS.

Bab 12 dan 13 adalah kotak perkakas. Bab 12 merakit algoritma graf yang paling sering diperlukan dalam lomba: *union-find*, *breadth-first search*, pemeriksaan graf bipartit, lintasan

terpanjang pada pohon, lintasan terpendek Dijkstra, pengurutan topologis, dan pohon merentang minimum Kruskal. Bab 13 melakukan hal yang sama untuk teori bilangan, geometri, dan string: perpangkatan cepat, fungsi totient Euler, uji keprimaan Miller–Rabin, orientasi, *convex hull* dan luas poligon, pasangan titik terdekat, Knuth–Morris–Pratt, periode sebuah string, dan *suffix array*.

Bab 14 menyatukan seluruh keterampilan itu. Bab ini merangkai bab-bab sebelumnya menjadi satu alur kerja dan menempuhnya dari awal sampai akhir pada tiga studi kasus, PPATH, ABCPATH, dan ONEZERO, menunjukkan apa yang harus dilakukan ketika vonisnya bukan *Accepted*, dan menyarankan langkah berikutnya.

Setiap bab menempuh jalan yang sama: sebuah kisah dan masalah yang dimunculkannya, gagasannya dalam kata-kata sederhana dengan contoh kecil yang dihitung dengan tangan, soal-soal SPOJ sungguhan yang diceritakan ulang dengan kata-kata kita sendiri, gagasan yang dibuktikan kebenarannya dan dihitung biayanya, program acuan dalam C++17 dan Python, menjalankannya, mengujinya terhadap *brute force*, menyempurnakannya, memakainya sungguh-sungguh, kesalahan yang sering terjadi, ringkasan, soal latihan dengan penyelesaian lengkap, dan daftar pustaka.

Setiap program dalam buku ini sudah dijalankan sebelum keluarannya dicetak. Program-program itu dikumpulkan, bab demi bab, dalam folder kode yang menyertai buku ini, dari `Chapter01` sampai `Chapter14`, masing-masing dengan folder `cpp`, folder `python`, folder `tests` yang berisi *generator*, *brute force*, dan skrip *stress test*, serta README yang menjelaskan cara mengompilasi dan menjalankannya di terminal dan di VS Code, pada Windows, macOS, dan Linux.

Cara membacanya

Dengan VS Code terbuka dan akun juri siap. Bab 1 sampai 4 didahulukan bagi semua pembaca, karena bab-bab itulah yang membangun meja kerja, kebiasaan membaca, dan anggaran waktu yang diandalkan setiap bab sesudahnya, dan perangkat *stress test* dari Bab 8 layak dipelajari sebelum bab-bab algoritma; setelah itu bab-bab mendaki dari yang mudah menuju yang sulit, dan setiap bab menyebutkan bab-bab sebelumnya yang menjadi pijakannya. Bacalah setiap soal dan cobalah sendiri sebelum membaca solusinya, walau hanya dua puluh menit: pergulatan itulah yang membuat gagasannya melekat. Kerjakan contoh-contoh kecilnya dengan tangan, ketikkan programnya, jangan hanya menyalinnya, dan setelah sebuah program berjalan, rusaklah dengan sengaja — buat sebuah larik kurang satu elemen, pakai `int` di tempat yang memerlukan `long long` — lalu saksikan *stress test* menangkapnya. Semua soal latihan di akhir tiap bab dilengkapi penyelesaian lengkap — gunakan sesudah usaha Anda sendiri, bukan sebagai penggantinya.

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

Tentang penulis

Irfan Subakti menggambarkan dirinya dalam satu baris — *Father | Educator | Entrepreneur | Bridging Tech & Nature: CS Lecturer × Crypto/Digital Business × AI/ML Specialist × Fruit Farmer | Storyteller* — ayah, pendidik, wirausahawan, penjembatan teknologi dan alam: dosen ilmu komputer, pelaku bisnis kripto dan digital, spesialis AI/ML, petani buah, dan pencerita. Setiap bagian dari baris itu ikut masuk ke dalam buku ini. Ia mengajar di Departemen Teknik Informatika ITS sejak 1998. Ia membimbing dan menguji tugas akhir mahasiswa sarjana,

mengajar MLOps dalam program IBM SkillsBuild, dan membangun produk: ia mendirikan Profitina, platform visibilitas AI, yang para insinyurnya menghadapi anggaran waktu dan memori yang sama dengan yang diajarkan buku ini. Jauh dari papan ketik, ia merawat kebun buah dan durian keluarga — di sanalah ia belajar bahwa perangkat lunak yang baik, seperti kebun yang baik, ditumbuhkan dengan sabar, satu bagian yang terawat demi satu bagian. Jiwa penceritanya adalah alasan setiap bab dalam buku ini dibuka dengan seseorang dan masalahnya, bukan dengan sebuah definisi.

Ucapan terima kasih

Kepada para mahasiswa yang mempelajari versi-versi awal materi ini, dan yang pengiriman-pengirimannya yang ditolak, yang ditanyakan larut malam, menyingkapkan di mana sebuah penjelasan masih tipis, serta kepada rekan-rekan di ITS yang bersama-sama mengajarkan dan menyempurnakan materi algoritma dan pemrograman dari tahun ke tahun. Kepada para pembuat soal dan pengelola SPOJ, yang soal-soalnya telah melatih satu generasi pemrogram, dan kepada komunitas pemrograman kompetitif yang lebih luas, yang pembahasan dan diskusinya menjaga keterampilan ini tetap hidup. Standar buku ini juga berutang pada dokumentasi resmi C++, GCC, Python, dan VS Code, serta pada buku teks klasik dan literatur riset algoritma, yang dikutip di setiap tempat penggunaannya.

Surabaya, September 2026



Irfan Subakti · yifana@gmail.com

Profitina

profitina.com

Bab 1

Selamat Datang di Pemrograman Kompetitif dan SPOJ

Program yang berjalan di laptop Anda belum tentu program yang benar. Sebuah *online judge* (juri daring) mengambil berkas sumber Anda, mengompilasinya dengan kompiler yang tidak Anda pilih, menjalankannya pada masukan yang belum pernah Anda lihat, memberinya jatah detik dan megabita yang pasti, lalu membandingkan setiap karakter yang dicetaknya dengan jawaban yang sudah ditetapkan jauh sebelum Anda mulai mengetik. Bab ini menjelaskan mesin itu dan menyiapkan Anda untuk menghadapinya. Di akhir bab Anda akan mampu menjelaskan apa itu pemrograman kompetitif dan apa yang ditawarkan SPOJ; menguraikan langkah demi langkah apa yang dilakukan juri terhadap sebuah kiriman dan apa arti setiap vonis (AC, WA, TLE, RE, CE) bagi pencarian *bug* Anda; memasang meja kerja C++17 dan Python 3.14 yang lengkap, dengan g++, VS Code, dan ekstensinya, di Windows, macOS, atau Linux; menyelesaikan dan menjalankan soal SPOJ pertama Anda dalam kedua bahasa; serta membangun juri kecil milik Anda sendiri yang memberi program Anda vonis yang sama di komputer Anda, sebelum SPOJ sempat melihatnya.

1.1 Vonis Merah Tengah Malam

Rina kuliah di tahun kedua program studi informatika di sebuah universitas di Surabaya. Tim kampusnya ingin ikut kompetisi pemrograman tingkat nasional, dan kakak tingkat yang melatih tim memberi setiap anggota baru pekerjaan rumah yang sama: buat akun di SPOJ, Sphere Online Judge, lalu selesaikan soal berkode TEST sebelum pertemuan berikutnya. Soal itu tampak nyaris terlalu mudah. Masukannya adalah deretan bilangan kecil, satu per baris; cetak kembali bilangan-bilangan itu, satu per baris, dan berhentilah begitu bertemu bilangan 42.

Rina menulis sepuluh baris C++ dalam beberapa menit. Ia mengetikkan contoh masukan ke programnya, contoh keluaran muncul di layar persis seperti yang tercetak di halaman soal, lalu ia mengirimkannya. Beberapa detik kemudian halaman status menampilkan kirimannya dengan warna merah: *wrong answer*. Ia membaca kodenya lagi. Program itu mencetak bilangan; program itu berhenti pada 42. Ia mengetik contohnya sekali lagi, dan sekali lagi keluarannya benar. Tengah malam ia mengirim pesan kepada pelatihnya: “Jurinya rusak.”

Tentu saja jurinya tidak rusak. Program Rina ikut mencetak 42 sebelum berhenti, dan contoh di halaman soal kebetulan berakhir dengan cara yang menyembunyikan perbedaan itu dari matanya yang lelah. Subbab 1.12 membangun juri kecil yang menemukan kesalahan ini dalam sepersekian detik dan menunjukkan pada baris berapa kesalahan itu terjadi. Namun pelajaran yang lebih penting adalah pelajaran yang ingin ditanamkan sang pelatih sejak hari pertama: juri tidak peduli bahwa program itu *hampir* benar, tidak membaca kode, dan tidak mendengarkan penjelasan. Juri menjalankan program dan membandingkan apa yang dicetaknya. Ketegasan itulah yang membuat juri berguna.

Siapa yang diuntungkan jika belajar memuaskan penguji setegas itu? Jelas Rina dan rekan-rekan setimnya, yang ingin mengikuti International Collegiate Programming Contest (ICPC), kompetisi pemrograman terbesar bagi mahasiswa, yang babak regionalnya berujung pada final dunia [1].

Siswa SMA yang bersiap menghadapi olimpiade informatika. Siapa pun yang melamar pekerjaan sebagai pengembang perangkat lunak, karena wawancara teknis di banyak perusahaan berupa soal pemrograman kompetitif kecil yang diselesaikan di depan pewawancara. Dan, meskipun kurang kentara, setiap pemrogram yang sudah bekerja: kebiasaan yang dipaksakan oleh juri, yakni membaca spesifikasi sampai kata terakhir, menanyakan seberapa besar masukan bisa menjadi, menguji kasus terkecil dan terbesar, serta memeriksa program cepat terhadap program lambat yang jelas benar, adalah kebiasaan yang membedakan perangkat lunak yang biasanya jalan dari perangkat lunak yang memang jalan.

Bayangkan ujian praktik SIM. Penguji tidak peduli seberapa baik Anda menyetir kemarin atau seberapa sungguh-sungguh Anda berniat melirik spion; ia mengamati apa yang Anda lakukan di rute yang ia pilih, yang tidak Anda lihat sebelumnya, dan ia memegang daftar periksa. Juri adalah penguji dengan daftar periksa dan stopwatch, dan rutenya dirahasiakan. Buku ini mengajari Anda menyetir begitu baik sehingga rutenya tidak lagi penting.

1.2 Apa Itu Pemrograman Kompetitif, dan Apa yang Ditawarkan SPOJ

> **DEFINISI.** *Pemrograman kompetitif* (*competitive programming*) adalah kegiatan menyelesaikan soal algoritmik yang terspesifikasi secara tepat dengan menulis program yang diperiksa secara otomatis oleh sebuah mesin, yaitu *juri* (*judge*), dari segi kebenaran keluaran, waktu berjalan, dan pemakaian memori. Sebuah *soal* terdiri atas pernyataan soal, masukan uji yang dirahasiakan, keluaran yang diharapkan untuk masukan itu, batas waktu, dan batas memori. Sebuah *kiriman* (*submission*) adalah satu berkas sumber yang dikirim ke juri; tanggapan juri terhadapnya disebut *vonis* (*verdict*).

Pemrograman kompetitif hadir dalam dua bentuk. Dalam sebuah *kontes*, banyak orang atau tim menyelesaikan kumpulan soal yang sama pada waktu yang sama, biasanya lima sampai dua belas soal dalam dua sampai lima jam, dan papan skor mengurutkan mereka menurut banyaknya soal yang terselesaikan dan waktu yang dibutuhkan. ICPC adalah kontes untuk tim beranggotakan tiga mahasiswa yang berbagi satu komputer [1]; olimpiade informatika adalah kontes perorangan untuk siswa sekolah; dan banyak situs web menyelenggarakan kontes daring setiap minggu. Dalam sebuah *arsip*, soal-soal tersedia selamanya dan Anda menyelesaikannya kapan saja, dengan kecepatan Anda sendiri, sebagai latihan. SPOJ terutama adalah sebuah arsip.

SPOJ, Sphere Online Judge, termasuk juri publik tertua di web. Isinya ribuan soal yang ditulis oleh sukarelawan dari seluruh dunia, dan setiap soal punya kode pendek berupa paling banyak delapan huruf kapital: TEST, PRIME1, ADDREV, FCTRL2 [2]. Kode itu sekaligus alamat soalnya: soal TEST berada di `https://www.spoj.com/problems/TEST/` [3]. SPOJ mengelompokkan soalnya dalam beberapa bagian. Bagian *classical*, yang dipakai buku ini, berisi soal yang jawabannya benar sepenuhnya atau tidak sama sekali, dengan vonis biner. Bagian *challenge* menilai seberapa baik sebuah jawaban, misalnya seberapa pendek programnya atau seberapa dekat sebuah heuristik dengan nilai optimum. Bagian *tutorial* berisi soal-soal yang lebih mudah untuk berlatih. Setiap soal classical juga menunjukkan berapa banyak orang yang telah menyelesaikannya, ukuran kasar yang berguna tentang tingkat kesulitannya: TEST telah diselesaikan oleh sangat banyak pengguna, sementara beberapa soal baru diselesaikan oleh segelintir orang.

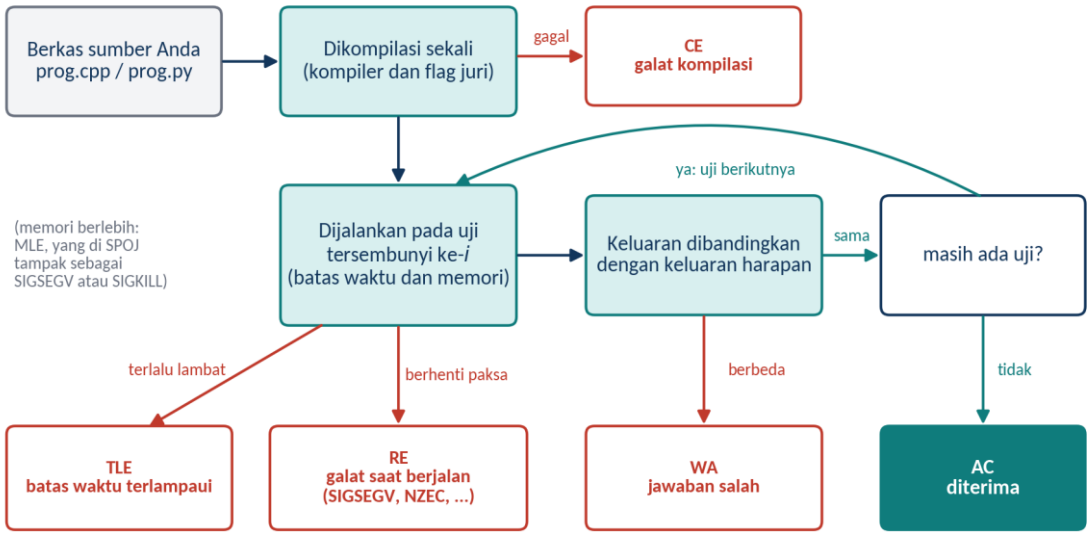
Yang membuat SPOJ sangat baik untuk belajar adalah luasnya pilihan bahasa dan umurnya. SPOJ menerima program dalam puluhan bahasa pemrograman, dan soal-soalnya terkumpul selama dua dasawarsa, sehingga hampir setiap teknik klasik perancangan algoritma muncul dalam suatu

soal, biasanya dalam beberapa varian dengan kesulitan yang meningkat. Umurnya juga membawa harga yang akan terus diperhatikan buku ini: kompiler dan mesin juri bukanlah yang terbaru. Pada saat buku ini ditulis, kompiler C++ yang diterima sebagian besar soal SPOJ adalah GCC 8.3, yang ditawarkan sebagai "C++14 (gcc 8.3)" [4], versi yang dirilis pada 2019, dan Python 3 di SPOJ juga bertahun-tahun lebih tua daripada Python 3.14 yang kita pasang di bab ini. Subbab 1.4 menjelaskan bagaimana program-program dalam buku ini menyiasatinya.

Pemrograman kompetitif tidak sama dengan rekayasa perangkat lunak. Soal kontes itu kecil, spesifikasinya lengkap dan tepat, tidak pernah berubah setelah Anda mulai, dan satu-satunya penggunanya adalah juri. Perangkat lunak sungguhan punya kebutuhan yang kabur dan berubah setiap bulan, banyak pengguna, dan kode yang hidup bertahun-tahun. Meski begitu, keduanya bertemu pada satu titik penting: sebagian besar penyebab perangkat lunak menjadi lambat atau salah adalah hal yang dilatih oleh pemrograman kompetitif, yaitu memilih algoritma dan struktur data yang tepat untuk ukuran masukan, serta menangani kasus tepi yang tidak tersentuh oleh uji sekilas. Pemrogram yang telah menyelesaikan beberapa ratus soal juri membaca sebuah kebutuhan dengan cara berbeda, dan lebih awal bertanya "bagaimana kalau daftarnya kosong?" dan "berapa banyak rekaman yang mungkin ada?".

1.3 Apa yang Dilakukan Juri terhadap Program Anda

Ketika Anda menekan tombol kirim, dimulailah rangkaian langkah yang sama di semua juri di dunia, dan **Gambar 1.1** menggambarkannya.



Gambar 1.1 Apa yang dilakukan juri terhadap sebuah kiriman: mengompilasi berkas sumber sekali, menjalankan program pada setiap uji tersembunyi dalam batas waktu dan memori, membandingkan setiap keluaran dengan keluaran harapan, dan berhenti pada kegagalan pertama.

Langkah 1: kompilasi. Untuk bahasa yang dikompilasi seperti C++, juri lebih dulu mengompilasi berkas sumber dengan kompilernya sendiri dan opsinya sendiri. Jika kompiler menolak berkas itu, vonisnya adalah *compilation error* (CE, galat kompilasi) dan tidak ada yang dijalankan. Untuk Python tidak ada langkah kompilasi terpisah di juri; galat sintaksis muncul ketika program mulai berjalan dan biasanya dilaporkan sebagai galat saat berjalan.

Langkah 2: jalankan pada setiap uji tersembunyi. Penyusun soal telah menyiapkan sekumpulan berkas uji, kadang satu berkas besar berisi banyak kasus uji, kadang selusin berkas

terpisah. Juri menjalankan program Anda sekali untuk setiap berkas uji, menghubungkan berkas itu ke *masukan standar* program (aliran yang dibaca oleh `std::cin` dan `input()`, yang di komputer Anda biasanya berupa papan ketik) dan menangkap semua yang ditulis program ke *keluaran standar* (aliran `std::cout` dan `print()`, biasanya layar). Program Anda tidak pernah membuka berkas menurut namanya; program hanya membaca dan menulis, dan juri telah menaruh hal yang benar di kedua ujungnya. Selama program berjalan, juri mengukur waktu dan memorinya. Jika batas waktu terlampaui sebelum program selesai, juri menghentikannya: *time limit exceeded* (TLE, batas waktu terlampaui). Jika program berakhir secara tidak wajar, misalnya karena membaca di luar larik, membagi dengan nol, atau melempar eksepsi yang tidak ditangkap, vonisnya adalah *runtime error* (RE, galat saat berjalan), dan SPOJ menambahkan nama sinyal atau penyebabnya, misalnya `SIGSEGV` (pelanggaran akses memori), `SIGFPE` (kesalahan aritmetika seperti pembagian dengan nol), `SIGABRT` (program menghentikan dirinya sendiri), atau `NZEC` (*non-zero exit code*, kode keluar bukan nol, khas untuk eksepsi Python yang tidak ditangkap).

Langkah 3: bandingkan. Jika program berakhir secara wajar dalam batas yang ditentukan, juri membandingkan keluarannya dengan keluaran harapan. Kebanyakan soal classical memakai perbandingan yang mengabaikan perbedaan *whitespace*, yaitu spasi dan pindah baris yang berlebih, tetapi tidak mengabaikan hal lain; beberapa soal menerima lebih dari satu jawaban benar dan memakai program pemeriksa khusus. Jika keluarannya berbeda, vonisnya adalah *wrong answer* (WA, jawaban salah).

Langkah 4: berhenti pada kegagalan pertama, atau terima. Juri berpindah ke uji berikutnya hanya jika uji saat ini lolos. Kegagalan pertama menentukan vonis; hanya program yang lolos semua uji yang mendapat *accepted* (AC, diterima). **Tabel 1.1** mendaftar vonis-vonis itu beserta hal pertama yang perlu diperiksa untuk masing-masing.

Tabel 1.1 Vonis-vonis juri, namanya di SPOJ, artinya, dan tempat untuk mulai mencari *bug*.

Vonis	Tampilan SPOJ	Yang terjadi	Periksa dulu
AC	accepted	Semua uji lolos dalam batas	Tidak ada: lanjutkan, tetapi baca Subbab 1.13
WA	wrong answer	Keluaran berbeda pada suatu uji	Format keluaran, kasus tepi, *overflow*
TLE	time limit exceeded	Program tidak selesai tepat waktu	Kompleksitas algoritma (Bab 3), masukan dan keluaran yang lambat (Bab 4), gelung tanpa akhir
RE	runtime error (SIGSEGV, SIGFPE, SIGABRT, NZEC, ...)	Program berhenti paksa	Batas larik, pembagian dengan nol, kedalaman rekursi, membaca melewati akhir masukan
CE	compilation error	Kompiler menolak berkas sumber	Fitur bahasa yang belum dikenal kompiler juri, *header* yang hilang, salah memilih bahasa
MLE	biasanya tampak sebagai runtime error	Program memakai terlalu banyak memori	Ukuran larik dibandingkan batas memori (Bab 6)

> **WAWASAN KUNCI:** Vonis adalah petunjuk, bukan nilai rapor. WA berarti algoritma selesai dan mencetak sesuatu yang salah, sehingga yang dicari adalah gagasan yang keliru atau kasus yang tidak tertangani; TLE sama sekali tidak mengatakan apa pun tentang kebenaran, hanya bahwa program terlalu lambat pada suatu uji; RE berarti program rusak, yang sering kali merupakan *bug* pengindeksan dan bukan kesalahan gagasan. Membaca vonis dengan cara ini mengubah "gagal" menjadi pencarian yang sempit.

Dua sifat proses ini perlu diucapkan terang-terangan, karena pemula terkejut oleh keduanya. Pertama, uji tersembunyi bukanlah sampel acak. Penyusun soal yang cermat menulis uji yang memburu *bug*: masukan terkecil yang sah, masukan terbesar yang sah, masukan yang penuh nilai

kembar, masukan yang jawabannya nol, masukan yang dirancang agar program dengan kompleksitas yang salah berjalan berjam-jam. Contoh di halaman soal hanya ada untuk menunjukkan format. Lolos dari contoh itu membuktikan bahwa Anda membaca format dengan benar, dan nyaris tidak ada yang lain. Kedua, batas waktu berlaku untuk mesin juri, bukan mesin Anda. Program yang butuh 0,9 detik di laptop baru Anda mungkin butuh 2 detik di mesin juri yang lebih tua. Bab 3 menunjukkan cara memperkirakan waktu berjalan sebelum program ditulis, satu-satunya pertahanan yang andal.

Waktu yang diperlukan sebuah program kira-kira sama dengan banyaknya langkah sederhana yang dikerjakannya dibagi banyaknya langkah yang dapat dikerjakan mesin per detik:

$$t \approx \frac{\text{banyak langkah}}{\text{langkah per detik}} \tag{1.1}$$

Komputer modern yang menjalankan C++ terkompilasi mengerjakan antara 10^8 dan 10^9 langkah sederhana per detik, dan Python kira-kira lima puluh sampai seratus kali lebih sedikit. Persamaan (1.1) tampak terlalu sederhana untuk berguna, tetapi persamaan ini memutuskan banyak sekali soal hanya dengan sekali lihat, dan Bab 3 dibangun di atasnya.

1.4 Cara Kerja Buku Ini

Setiap bab buku ini disusun di sekitar soal SPOJ sungguhan. Untuk setiap soal kita mengerjakan tujuh hal yang sama, dalam urutan yang sama, karena urutan inilah yang benar-benar dipakai oleh peserta kompetisi yang berpengalaman.

1. **Menceritakan ulang soal dengan kata-kata kita sendiri.** Kita tidak pernah menyalin pernyataan soal; kita menyampaikan apa yang diminta, dan kita memberikan kode serta alamat SPOJ-nya agar Anda dapat membaca aslinya. Menceritakan ulang soal dengan kata-kata sendiri adalah keterampilan tersendiri, dan Bab 2 dikhususkan untuk itu.
2. **Membaca batasan** dan membiarkannya memberi tahu algoritma mana yang cukup cepat, dengan Persamaan (1.1).
3. **Menemukan gagasannya** dalam bahasa sehari-hari, biasanya dengan mengerjakan contoh kecil dengan tangan.
4. **Menunjukkan bahwa gagasan itu benar**, dengan pembuktian sesingkat yang dimungkinkan oleh gagasannya, dan **menghitung biayanya**.
5. **Menulis program acuan dalam C++17**, bahasa utama buku ini, **dan dalam Python 3**. Keduanya mencetak keluaran yang persis sama. Jika Python terlalu lambat untuk suatu soal bahkan dengan teknik tercepat, kita katakan demikian, beserta alasannya.
6. **Menjalankannya** pada contoh dan pada masukan buatan kita sendiri, dan menampilkan keluaran yang sesungguhnya.
7. **Mengujinya** terhadap sebuah *brute force*: program kedua yang sengaja dibuat sederhana dan lambat, yang kebenarannya jelas. Sebuah *generator* membuat banyak masukan acak yang kecil, dan sebuah *stress test* menjalankan kedua program pada setiap masukan itu dan berhenti pada ketidaksesuaian pertama. Mulai Bab 2, setiap soal dilengkapi ketiga alat ini.

Program-program itu adalah program pengajaran kita sendiri, ditulis untuk buku ini dan diuji dengan masukan kita sendiri. Program itu bukan salinan kiriman orang lain yang telah diterima, dan ditulis agar dapat dibaca: nama yang jelas, satu gagasan per fungsi, dan komentar pada baris yang tidak jelas dengan sendirinya. Program itu tidak dikirimkan ke SPOJ untuk Anda;

mengirimkannya tidak mengajarkan apa pun, sedangkan mengetiknya, menjalankannya, lalu mengubahnya mengajarkan banyak hal.

Semua kode tersimpan di folder `SPOJ/Code_new`, dengan satu folder per bab: `Chapter01/cpp` berisi program C++, `Chapter01/python` program Python, dan `Chapter01/tests` generator, *brute force*, dan berkas uji. Setiap folder bab punya `README.md` yang mendaftar programnya dan mengulang perintah untuk membangun dan menjalankannya di setiap sistem operasi.

Tentang versi. Kita mengompilasi setiap program C++ dengan `g++ -std=c++17 -O2 -Wall -Wextra`, tanpa peringatan, dan kita juga memeriksa bahwa setiap program bersih di bawah *sanitizer* `-fsanitize=address,undefined`, yang membuat program berhenti dengan pesan yang jelas pada akses di luar batas atau operasi tak terdefinisi yang pertama, alih-alih diam-diam berjalan terus. Karena kompiler C++ di SPOJ lebih tua, program-program itu sengaja hanya memakai fitur yang dipahami GCC 8.3, dan setiap program juga dapat dikompilasi dengan `-std=c++14`. Solusi Python ditulis untuk Python 3.14 tetapi tidak memakai apa pun yang lebih baru daripada *f-string*, sehingga juga berjalan pada Python 3 juri yang lebih tua; alat uji lokalnya memerlukan Python 3.11 atau yang lebih baru. Semua eksekusi yang ditampilkan dalam buku ini dilakukan pada mesin Linux dengan GCC 13.3 dan Python 3.14.7.

1.5 Meja Kerja Anda: Apa yang Dipasang dan Mengapa

Kita memerlukan empat hal: kompiler C++, Python, sebuah editor, dan dua ekstensi yang mengajari editor itu tentang C++ dan Python. **Tabel 1.2** mendaftarnya beserta versi yang berlaku pada September 2026.

Tabel 1.2 Meja kerja buku ini dan versi yang berlaku saat buku ini ditulis.

Alat	Apa itu	Versi pada September 2026
GCC (`g++`)	Kompiler C++ GNU	16.2, dirilis 7 Agustus 2026 [5]; Ubuntu 26.04 menyertakan GCC 15 [6]
MSYS2 (Windows)	Kumpulan paket yang membawa GCC ke Windows	paket `mingw-w64-ucrt-x86_64-gcc` 16.2.0 [7]
Command Line Tools (macOS)	Kompiler Clang dari Apple, yang juga menjawab panggilan `g++`	dipasang dengan `xcode-select --install` [8]
Python	Interpreter Python dan pustaka standarnya	3.14.7, dirilis 5 Agustus 2026 [9]
Visual Studio Code	Editor gratis untuk Windows, macOS, dan Linux	1.139, dirilis 23 September 2026 [10]
Ekstensi C/C++	IntelliSense, membangun, dan men-*debug* C++ di VS Code	`ms-vscode.cpptools` [11]
Ekstensi Python	Menjalankan, memeriksa, dan men-*debug* Python di VS Code	`ms-python.python` [12]

Mengapa g++? Karena itulah kompiler juri. SPOJ, seperti hampir semua juri, mengompilasi C++ dengan GCC, dan GCC punya beberapa kebiasaan yang tidak dimiliki kompiler lain, misalnya *header* `<bits/stdc++.h>` yang menyertakan seluruh pustaka standar sekaligus. Program yang dapat dikompilasi dengan GCC di mesin Anda juga dapat dikompilasi dengan GCC di juri, asalkan Anda menghindari fitur yang lebih baru daripada versi juri. Di Windows kita mengambil GCC dari MSYS2, jalur yang juga dianjurkan dokumentasi VS Code [13]; di Linux GCC datang dari pengelola paket sistem; di macOS kompiler sistemnya adalah Clang, yang memadai untuk buku ini, dan Subbab 1.7 menunjukkan cara mendapatkan GCC yang asli juga. GCC mana pun mulai versi 8

dapat mengompilasi semua program dalam buku ini; versi terbaru hanyalah pilihan terbaik untuk dipasang hari ini.

Mengapa C++17 dan bukan yang lebih baru? Bahasa C++ mendapat standar baru setiap tiga tahun: C++17, C++20, C++23. Standar yang lebih baru menawarkan lebih banyak kemudahan, tetapi program yang memakai fitur C++20 akan gagal dengan galat kompilasi di juri yang kompilernya lebih tua. C++17 cukup baru untuk nyaman dipakai dan cukup tua untuk aman, dan dengan menghindari sedikit tambahan C++17 yang belum dimiliki GCC 8, kita tetap aman bahkan di SPOJ. Kompiler terbaru memakai standar yang lebih baru secara bawaan, karena itulah setiap perintah dalam buku ini menyebutkan `-std=c++17` secara eksplisit.

Mengapa Python 3.14? Python 3.14 adalah seri yang berlaku saat ini, dan rilis pemeliharaan ketujuhnya, 3.14.7, terbit pada 5 Agustus 2026 [9]. Python adalah bahasa kedua yang sangat baik untuk pemrograman kompetitif: bilangan bulatnya tidak pernah *overflow*, string dan list-nya nyaman, dan *brute force* yang benar cukup ditulis dalam beberapa baris. Kelemahannya adalah kecepatan, dan buku ini jujur tentang hal itu: untuk sebagian soal, program Python dengan algoritma yang benar tetap terlalu lambat bagi juri, dan kita akan mengukur seberapa lambatnya.

Mengapa VS Code? Karena gratis, berjalan di ketiga sistem, menangani kedua bahasa, dan punya terminal bawaan. Editor apa pun dapat dipakai; petunjuk dalam buku ini memakai VS Code agar satu set nama menu cocok untuk semua orang.

1.6 Memasang di Windows 10 dan 11

1.6.1 Kompiler C++ dari MSYS2

Windows tidak membawa kompiler C++ yang dapat dipakai dari terminal, jadi kita memasang GCC dari MSYS2, kumpulan perkakas yang dipakai pemrogram Linux yang dirawat dengan baik [14].

1. Unduh penginstalnya dari `https://www.msys2.org` dan jalankan. Terima folder bawaan `C:\msys64`; jalur tanpa spasi menghindarkan masalah di kemudian hari.
2. Dari menu Start, buka **MSYS2 UCRT64**. Namanya penting: UCRT64 adalah varian modern, dan kompiler yang dipasang dari terminal MSYS2 lain tidak cocok dengannya. Ketik perintah berikut, tekan Enter untuk menerima semua anggota grup, lalu jawab `Y`:
   ```
   $ pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain
   ```
3. Tambahkan folder `C:\msys64\ucrt64\bin` ke `Path`, daftar folder tempat Windows mencari perintah: tekan tombol Windows, ketik *environment*, buka **Edit the system environment variables**, klik **Environment Variables**, pilih **Path** di bawah *User variables*, klik **Edit**, lalu **New**, tempelkan folder itu, dan konfirmasikan setiap dialog dengan **OK**.
4. Tutup semua terminal, buka **PowerShell** yang baru (jendela yang dibuka sebelum perubahan tidak melihatnya), lalu periksa:
   ```
   PS C:\Users\you> g++ --version
   g++.exe (Rev..., Built by MSYS2 project) 16.2.0
   ```

Baris pertama semestinya menyebut MSYS2 dan versi 13 atau yang lebih baru. Jika PowerShell menjawab bahwa `g++` "is not recognized", entri `Path` belum ada atau salah ketik, atau terminalnya lebih tua daripada perubahan itu.

Alternatifnya, jika Anda lebih suka lingkungan Linux sungguhan di dalam Windows, adalah Windows Subsystem for Linux: `wsl --install` di PowerShell administrator memasang Ubuntu, dan di dalamnya Anda mengikuti Subbab 1.8. Kedua jalur dapat dipakai untuk seluruh buku.

1.6.2 Python

Python di Windows dipasang dengan **Python install manager**, program kecil yang mengunduh dan berpindah-pindah antarversi Python [15]. Pasang dari Microsoft Store (cari "Python Install Manager") atau dari `https://www.python.org/downloads/windows/`, lalu buka PowerShell baru dan ketik:

```
PS C:\Users\you> py install 3.14
PS C:\Users\you> python --version
Python 3.14.7
```

Setelah itu, `python` dan `py` sama-sama menjalankan Python 3.14. Jika `python --version` malah membuka Microsoft Store, tutup semua terminal, buka yang baru, dan coba lagi. Di mana pun buku ini menulis `python3`, pengguna Windows mengetik `python` atau `py`.

1.6.3 VS Code

Pasang VS Code dengan `winget install Microsoft.VisualStudioCode` atau dengan **User Installer** dari `https://code.visualstudio.com`. Centang **Add to PATH**, yang memungkinkan Anda mengetik `code .` di terminal untuk membuka folder saat ini di editor.

1.7 Memasang di macOS

1.7.1 Kompiler C++

Buka **Terminal** (di **Applications › Utilities**) dan ketik:

```
$ xcode-select --install
```

Sebuah dialog menawarkan pemasangan **Command Line Tools**; terima tawarannya. Unduhannya beberapa ratus megabita dan membawa kompiler Clang dari Apple bersama `git`, `make`, dan perkakas klasik lainnya [8]. Anda tidak memerlukan aplikasi Xcode yang lengkap. Sesudahnya:

```
$ g++ --version
Apple clang version 17.x.x (clang-17xx.x.x.x)
```

Angka tepatnya bergantung pada versi macOS Anda. Perhatikan apa yang dikatakan baris pertama: di Mac, perintah `g++` bukanlah GCC melainkan Clang dari Apple dengan nama lain. Untuk hampir semua hal dalam buku ini hal itu tidak berpengaruh, karena Clang memahami C++17 standar sama baiknya. Pengaruhnya ada di satu tempat: Clang tidak punya *header* pribadi GCC `<bits/stdc++.h>`, sehingga program yang menyertakannya dapat dikompilasi di juri tetapi gagal di Mac. Buku ini menyertakan *header* standar satu per satu (`<iostream>`, `<vector>`, `<string>`), yang berfungsi di mana saja. Jika Anda menginginkan GCC yang asli, yang dipakai juri, pasang **Homebrew** dari `https://brew.sh` lalu `brew install gcc`; kompilernya kemudian bernama `g++-16`, dengan nomor versi di namanya, agar tidak bertabrakan dengan `g++` milik Apple [16].

1.7.2 Python dan VS Code

macOS tidak menyertakan Python yang mutakhir. Unduh **macOS installer** untuk Python 3.14.7 dari `https://www.python.org/downloads/`, jalankan, dan di akhir klik dua kali **Install Certificates.command** di folder yang terbuka. Setelah itu `python3 --version` mencetak `Python 3.14.7`. (Dengan Homebrew, `brew install python@3.14` juga dapat dipakai.) Untuk VS Code, unduh versi macOS dari `https://code.visualstudio.com`, seret ke **Applications**, jalankan, buka Command Palette dengan **Cmd+Shift+P**, lalu jalankan **Shell Command: Install 'code' command in PATH**.

1.8 Memasang di Linux

Di Linux semuanya datang dari pengelola paket distribusi. Di Ubuntu, Debian, dan kerabatnya:

```
$ sudo apt update
$ sudo apt install build-essential gdb python3 python3-venv
$ g++ --version
g++ (Ubuntu 15.2.0-...) 15.2.0
$ python3 --version
Python 3.14.x
```

Paket `build-essential` membawa `g++`, `make`, dan *header* pustaka C; `gdb` adalah *debugger*-nya. Ubuntu 26.04 LTS menyertakan GCC 15 sebagai kompiler bawaan [6] dan Python 3.14 sebagai `python3`-nya; Ubuntu 24.04 yang lebih tua menyertakan GCC 13.3 dan Python 3.12, dan keduanya dapat dipakai untuk seluruh buku, jadi tidak perlu diganti. Di Fedora padanannya adalah `sudo dnf install gcc-c++ gdb python3`. Jangan pernah menghapus atau mengganti Python sistem di Linux; sebagian sistem operasi bergantung padanya.

Untuk VS Code, unduh paket `.deb` (Ubuntu, Debian, Mint) atau paket `.rpm` (Fedora, openSUSE) dari `https://code.visualstudio.com` dan pasang dengan `sudo apt install ./code_*.deb` atau `sudo dnf install ./code-*.rpm`.

1.9 VS Code: Dua Ekstensi dan Satu *Build Task*

Jalankan VS Code, klik ikon **Extensions** di bilah kiri (atau tekan **Ctrl+Shift+X**, di Mac **Cmd+Shift+X**), lalu pasang dua ekstensi terbitan Microsoft:

- **C/C++** (pengenal `ms-vscode.cpptools`). Ekstensi ini melengkapi nama dan menandai galat sambil Anda mengetik (IntelliSense), serta menjalankan *debugger* [11]. Jika editor menggarisbawahi kode yang benar dengan warna merah, buka Command Palette (**Ctrl+Shift+P**), jalankan **C/C++: Select IntelliSense Configuration**, lalu pilih `g++` yang telah Anda pasang.
- **Python** (pengenal `ms-python.python`). Ekstensi ini membawa pemeriksa Pylance dan *debugger* Python [12]. Klik versi Python di bilah status di bawah untuk memilih interpreter yang dipakai VS Code; pilih 3.14.

Selalu buka sebuah *folder* (**File › Open Folder**), bukan satu berkas: untuk bab ini, buka `Chapter01`. Terminal terpadu (**Terminal › New Terminal**) lalu dimulai di folder itu.

VS Code tidak tahu dengan sendirinya opsi apa yang dipakai untuk mengompilasi C++, jadi kita memberitahunya sekali, dalam berkas kecil `.vscode/tasks.json` di dalam folder bab. **Kode 1.1** adalah berkas itu, dan berkas yang sama ada di folder setiap bab.

Kode 1.1 `.vscode/tasks.json`: *build task* yang mengompilasi berkas di editor dengan opsi buku ini.

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "C++17: build active file",
      "type": "shell",
      "command": "g++",
      "args": ["-std=c++17", "-O2", "-Wall", "-Wextra",
               "${file}", "-o", "${fileDirname}/${fileBasenameNoExtension}"],
      "group": { "kind": "build", "isDefault": true },
      "problemMatcher": ["$gcc"]
    }
  ]
}
```

Task ini menjalankan `g++` pada berkas yang tampil di editor (`${file}`) dan menulis programnya di sebelah berkas itu, dengan nama yang sama tanpa `.cpp` (di Windows, g++ menambahkan `.exe` dengan sendirinya). Karena *task* ini ditandai sebagai *build task* bawaan, **Ctrl+Shift+B** (Mac: **Cmd+Shift+B**) menjalankannya, dan *problem matcher* `$gcc` mengubah setiap galat kompiler menjadi garis bergelombang merah di baris yang tepat. Di Mac dengan GCC dari Homebrew, ubah `"g++"` menjadi `"g++-16"`.

Program kompetitif membaca masukan standar, jadi ada gunanya belajar menyuapinya dengan sebuah berkas alih-alih mengetik. Di terminal Linux dan macOS, dan di **Command Prompt** klasik Windows, operator `<` menghubungkan berkas ke masukan standar dan `>` mengirim keluaran standar ke sebuah berkas:

```
$ ./test_life < sample.in > my.out
```

PowerShell, terminal bawaan VS Code di Windows, tidak menerima `<`. Di sana Anda menulis `Get-Content sample.in | .\test_life.exe`, atau mengganti terminal menjadi Command Prompt melalui tanda **∨** kecil di sebelah **+** pada panel terminal. Jika Anda mengetik masukan dengan tangan, akhiri dengan **Ctrl+D** di Linux dan macOS, dan dengan **Ctrl+Z** lalu Enter di Windows; tombol itu mengirim *end of file* (akhir berkas), yang memberi tahu program bahwa tidak ada masukan lagi.

1.10 Program Pertama Anda: Memeriksa Meja Kerja

Dua program pertama tidak melakukan apa pun selain melaporkan perkakas yang membangun dan menjalankannya, cara tercepat untuk memastikan bahwa pemasangan berhasil dan untuk melihat beberapa fakta yang penting kelak. **Kode 1.2** adalah versi C++-nya.

Kode 1.2 `check_env.cpp`: melaporkan kompiler, standar C++, dan ukuran tipe bilangan bulat.

```cpp
// check_env.cpp: menampilkan kompiler dan standar C++ yang membangun program ini.
#include <climits>
#include <cstdio>

int main() {
#if defined(__clang__)
    std::printf("compiler: clang %d.%d.%d\n", __clang_major__, __clang_minor__,
                __clang_patchlevel__);
#elif defined(__GNUC__)
    std::printf("compiler: g++ %d.%d.%d\n", __GNUC__, __GNUC_MINOR__,
                __GNUC_PATCHLEVEL__);
#else
    std::printf("compiler: not GCC or Clang\n");
#endif
    std::printf("C++ standard: %ld\n", static_cast<long>(__cplusplus));
    std::printf("bytes: int %zu, long %zu, long long %zu, pointer %zu\n",
                sizeof(int), sizeof(long), sizeof(long long), sizeof(void*));
    std::printf("INT_MAX   = %d\n", INT_MAX);
    std::printf("LLONG_MAX = %lld\n", LLONG_MAX);
    return 0;
}
```

Baris yang diawali `#if` dibaca oleh *preprocessor* sebelum kompiler yang sebenarnya: setiap kompiler mendefinisikan beberapa nama yang mengenalinya, misalnya `__clang__` untuk Clang dan `__GNUC__` untuk GCC, dan hanya cabang yang cocok yang dikompilasi. Nama `__cplusplus` menyimpan standar bahasa sebagai angka, `201703` untuk C++17. Kompilasi dan jalankan di terminal VS Code, dari folder `Chapter01`:

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o cpp/check_env cpp/check_env.cpp
$ ./cpp/check_env
```

```
compiler: g++ 13.3.0
C++ standard: 201703
bytes: int 4, long 8, long long 8, pointer 8
INT_MAX   = 2147483647
LLONG_MAX = 9223372036854775807
```

Bacalah baris ketiga dengan saksama, karena di sana tersembunyi jebakan klasik. Di Linux dan macOS, `long` berukuran 8 bita; di Windows hanya 4, sama dengan `int`, bahkan dengan kompiler 64-bit. Program yang menyimpan bilangan besar dalam `long` berjalan benar di Linux dan *overflow* di Windows. Tipe `long long` berukuran 8 bita di mana saja, dan itulah tipe yang dipakai buku ini setiap kali sebuah nilai dapat melampaui sekitar dua miliar, batas `INT_MAX` yang tercetak di baris keempat. Persamaan (1.2) menyatakan kedua batas itu secara tepat.

$$\text{INT\_MAX} = 2^{31} - 1 = 2\,147\,483\,647, \qquad \text{LLONG\_MAX} = 2^{63} - 1 \approx 9.2 \times 10^{18} \tag{1.2}$$

Versi Python, **Kode 1.3**, melaporkan interpreternya.

Kode 1.3 `check_env.py`: melaporkan versi Python dan menunjukkan bahwa bilangan bulat Python tidak berukuran tetap.

```python
"""check_env.py: menampilkan Python yang menjalankan program ini dan ukuran int."""
import platform
import sys


def main():
    print("python:", platform.python_version(), platform.python_implementation())
    print("system:", platform.system(), platform.machine())
    print("sys.maxsize =", sys.maxsize)
    print("2**100 =", 2**100)
    print("inside a virtual environment:", sys.prefix != sys.base_prefix)


if __name__ == "__main__":
    main()
```

```
$ python3 python/check_env.py
```

```
python: 3.14.7 CPython
system: Linux x86_64
sys.maxsize = 9223372036854775807
2**100 = 1267650600228229401496703205376
inside a virtual environment: False
```

Baris keempat menunjukkan mengapa Python begitu menyenangkan untuk soal dengan bilangan besar: 2^{100} terdiri atas 31 digit, dan Python menghitungnya dengan tepat tanpa diminta. `sys.maxsize` bukanlah bilangan bulat terbesar, karena hal itu tidak ada, melainkan ukuran terbesar sebuah list, yang di mesin 64-bit kebetulan sama dengan `LLONG_MAX`. Baris terakhir menyatakan apakah sebuah *virtual environment* (lingkungan virtual), folder pribadi berisi paket terpasang, sedang aktif; program Bab 1 sampai 4 hanya memakai pustaka standar dan tidak memerlukannya.

1.11 Soal SPOJ Pertama Anda: TEST

Soalnya (SPOJ TEST, "Life, the Universe, and Everything", `https://www.spoj.com/problems/TEST/`) [3]. Masukannya adalah deretan bilangan bulat, satu per baris; masing-masing paling banyak dua digit. Cetak bilangan-bilangan itu satu per baris, dengan urutan yang sama, sampai Anda membaca bilangan 42; jangan cetak 42 maupun apa pun sesudahnya. Soal tidak menyebutkan berapa banyak bilangannya, dan tidak ada jaminan bahwa masukan berakhir tepat setelah 42: bilangan lain boleh menyusul, dan program harus mengabaikannya.

Contoh masukan di halaman soal adalah lima bilangan 1, 2, 88, 42, 99, dan contoh keluarannya 1, 2, 88.

Gagasannya. Baca sebuah bilangan; jika bilangan itu 42, berhenti; jika bukan, cetak dan ulangi. Tidak ada yang perlu dibuktikan. Yang perlu dibuat benar adalah urutan ketiga tindakan, *baca, uji, cetak*, dan perilaku di akhir masukan, tempat bersarangnya *bug* para pemula.

Kode 1.4 `test_life.cpp`: solusi TEST dalam C++.

```cpp
// SPOJ TEST (Life, the Universe, and Everything): cetak ulang bilangan sampai 42.
#include <iostream>

int main() {
    int x;
    // Membaca lebih dulu: gelung berhenti pada 42 atau saat masukan habis.
    while (std::cin >> x && x != 42) {
        std::cout << x << '\n';
    }
    return 0;
}
```

Seluruh program berada di kondisi gelung `while`. Ekspresi `std::cin >> x` membaca satu bilangan bulat ke `x` lalu *bernilai aliran itu sendiri*; aliran yang dipakai sebagai kondisi bernilai benar jika pembacaan terakhir berhasil dan salah jika gagal, misalnya karena masukan telah habis. Operator `&&` mengevaluasi sisi kanannya hanya jika sisi kirinya benar, sehingga `x != 42` hanya diuji pada bilangan yang benar-benar terbaca. Karena itu gelung berhenti tepat dalam dua keadaan, yaitu dua keadaan yang diizinkan soal: bilangan 42 terbaca, atau masukan habis. Menulis `'\n'` alih-alih `std::endl` mengakhiri baris tanpa memaksa keluaran dikirim seketika; Bab 4 mengukur mengapa hal itu penting.

Kode 1.5 `test_life.py`: solusi TEST dalam Python.

```python
"""SPOJ TEST (Life, the Universe, and Everything): cetak bilangan sampai 42."""
import sys


def main():
    for line in sys.stdin:
        line = line.strip()
        if not line:  # lewati baris kosong, bila ada
            continue
        x = int(line)
        if x == 42:
            break
        print(x)


if __name__ == "__main__":
    main()
```

Versi Python membaca masukan baris demi baris dengan `for line in sys.stdin`, yang juga berhenti dengan sendirinya di akhir masukan. Program membuang pindah baris dengan `strip()` dan melewati baris kosong, yang tidak dijanjikan soal tetapi tidak merugikan untuk ditangani. Dua baris terakhir adalah konvensi Python: `main()` hanya berjalan jika berkas itu dijalankan sebagai program, bukan ketika program lain mengimpornya, sehingga alat uji di bab-bab berikutnya dapat memakai ulang fungsi dari solusi kita.

Kedua program memberi hasil yang sama pada contoh. Kompilasi versi C++ dengan **Ctrl+Shift+B** atau dengan perintah di bawah, lalu suapi keduanya dengan contoh:

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o cpp/test_life cpp/test_life.cpp
$ printf '1\n2\n88\n42\n99\n' | ./cpp/test_life
$ printf '1\n2\n88\n42\n99\n' | python3 python/test_life.py
```

```
1
2
88
```

(Setiap program mencetak ketiga baris ini; keluarannya ditampilkan sekali.) Perintah `printf` menuliskan argumennya, dengan `\n` diubah menjadi pindah baris, dan pipa `|` menghubungkannya

ke masukan standar program. Di PowerShell, `"1","2","88","42","99" | .\cpp\test_life.exe` melakukan hal yang sama.

1.12 Membangun Juri Sendiri

Pesan tengah malam Rina berasal dari celah dalam cara kerjanya: ia tidak punya cara cepat untuk mencoba programnya pada lebih dari satu masukan dan membandingkan hasilnya secara mekanis. Membandingkan dengan mata gagal tepat ketika Anda lelah, dan gagalnya diam-diam. Obatnya adalah melakukan di komputer Anda sendiri apa yang dilakukan juri, yang sudah digambarkan lengkap oleh **Gambar 1.1**: kompilasi sekali, jalankan pada setiap uji dengan batas waktu, bandingkan dengan keluaran harapan, berhenti pada kegagalan pertama. Semuanya cukup ditulis dalam kurang dari seratus baris Python.

1.12.1 Uji sebagai pasangan berkas

Sebuah uji terdiri atas dua berkas dengan nama yang sama: `01.in`, masukannya, dan `01.out`, keluaran yang dicetak program yang benar untuk masukan itu. Sebuah folder berisi pasangan-pasangan seperti itu adalah sebuah himpunan uji. Dari mana keluaran harapannya? Untuk soal sesederhana TEST, kita dapat menuliskannya dengan beberapa baris kode yang jelas benar. **Kode 1.6** menulis empat uji: contoh soal, daftar yang diawali 42, daftar yang memuat nol, dan daftar panjang berisi lima ribu bilangan dengan bilangan-bilangan lain sesudah 42.

Kode 1.6 `gen_test_life.py`: menulis empat uji untuk TEST, dengan keluaran harapan yang dihitung oleh kode sesederhana mungkin.

```
"""gen_test_life.py: menulis uji untuk TEST ke tests/TEST (.in dan .out)."""
import random
from pathlib import Path

HERE = Path(__file__).resolve().parent / "TEST"


def expected(numbers):
    """Jawaban lambat yang jelas benar: semua bilangan sebelum 42 yang pertama."""
    return numbers[:numbers.index(42)]


def write(name, numbers):
    HERE.mkdir(exist_ok=True)
    (HERE / f"{name}.in").write_text("".join(f"{x}\n" for x in numbers))
    (HERE / f"{name}.out").write_text("".join(f"{x}\n" for x in expected(numbers)))


def main():
    rng = random.Random(2026)
    write("01", [1, 2, 88, 42, 99])  # contoh dari soal
    write("02", [42, 1, 2])  # 42 di awal: tidak mencetak apa pun
    write("03", [7, 0, 42])  # sebuah nol sebelum akhir
    other = [x for x in range(100) if x != 42]
    long_run = [rng.choice(other) for _ in range(5000)]
    write("04", long_run + [42] + [rng.randrange(100) for _ in range(20)])
    print("wrote", len(list(HERE.glob("*.in"))), "tests to", HERE.name)


if __name__ == "__main__":
    main()
```

Fungsi `expected` adalah spesifikasi soal yang ditulis sebagai kode: semua bilangan sebelum 42 yang pertama. Bilangan acaknya berasal dari `random.Random(2026)`, pembangkit dengan *seed*

tetap, sehingga setiap eksekusi menghasilkan uji yang sama; masukan acak tetapi dapat diulang itulah yang membuat kegagalan dapat direproduksi. Uji `02` ada karena keluaran kosong adalah jawaban sah yang bisa saja salah ditangani program yang ceroboh, dan uji `04` karena program yang benar untuk lima bilangan mungkin gagal untuk lima ribu.

```
$ cd tests
$ python3 gen_test_life.py
```

```
wrote 4 tests to TEST
```

1.12.2 Jurinya

Kode 1.7 adalah juri itu sendiri.

Kode 1.7 `minijudge.py`: mengompilasi program C++ (atau menjalankan program Python) pada setiap uji di sebuah folder, dengan batas waktu, dan mencetak vonis seperti SPOJ.

```
"""minijudge.py: juri lokal kecil yang bekerja seperti juri SPOJ.

pemakaian: python3 minijudge.py TESTDIR SECONDS PROGRAM
PROGRAM adalah berkas .cpp (dikompilasi dulu, seperti juri) atau berkas .py.
Setiap uji adalah pasangan NAMA.in / NAMA.out di TESTDIR; uji dijalankan
menurut urutan nama dan penjurian berhenti pada uji pertama yang gagal.
"""
import os
import subprocess
import sys
import tempfile
import time
from pathlib import Path

CXX = os.environ.get("CXX", "g++")
FLAGS = ["-std=c++17", "-O2", "-Wall", "-Wextra"]


def compile_cpp(source, workdir):
    """Kompilasi source; kembalikan (command, "") atau (None, pesan kompiler)."""
    exe = Path(workdir) / ("prog.exe" if os.name == "nt" else "prog")
    result = subprocess.run([CXX, *FLAGS, "-o", str(exe), source],
                            capture_output=True, text=True)
    if result.returncode != 0:
        return None, result.stderr
    return [str(exe)], ""


def same_output(got, expected):
    """Sama per baris, abaikan spasi di ujung baris dan baris kosong di akhir."""
    a = [line.rstrip() for line in got.rstrip().splitlines()]
    b = [line.rstrip() for line in expected.rstrip().splitlines()]
    return a == b


def first_difference(got, expected):
    a, b = got.splitlines(), expected.splitlines()
    for i in range(max(len(a), len(b))):
        x = a[i].rstrip() if i < len(a) else "<nothing>"
        y = b[i].rstrip() if i < len(b) else "<nothing>"
        if x != y:
            return f"line {i + 1}: got {x!r}, expected {y!r}"
    return "only blank lines differ"
```

Kode 1.7 (Lanjutan) *minijudge.py: mengompilasi program C++ (atau menjalankan program Python) pada setiap uji di sebuah folder, dengan batas waktu, dan mencetak vonis seperti SPOJ.*

```
def run_one(command, infile, seconds):
    """Jalankan sekali dengan masukan standar infile; hasil (status, waktu, out)."""
    with open(infile, "rb") as f:
        start = time.perf_counter()
        try:
            r = subprocess.run(command, stdin=f, capture_output=True,
                               timeout=seconds)
        except subprocess.TimeoutExpired:
            return "TLE", seconds, ""
        used = time.perf_counter() - start
    if r.returncode != 0:
        lines = r.stderr.decode(errors="replace").strip().splitlines()
        return "RE", used, lines[-1] if lines else f"exit code {r.returncode}"
    return "OK", used, r.stdout.decode(errors="replace")


def judge(testdir, seconds, program):
    tests = sorted(Path(testdir).glob("*.in"))
    with tempfile.TemporaryDirectory() as work:
        if program.endswith(".cpp"):
            command, errors = compile_cpp(program, work)
            if command is None:
                print("compilation error")
                lines = errors.splitlines()
                print([x for x in lines if "error" in x][0] if lines else "")
                return "CE"
        else:
            command = [sys.executable, program]
        for test in tests:
            status, used, out = run_one(command, test, seconds)
            if status == "OK":
                expected = test.with_suffix(".out").read_text()
                status = "AC" if same_output(out, expected) else "WA"
            print(f"  test {test.stem}: {status:3} {used:5.2f} s")
            if status == "WA":
                print("    ", first_difference(out, expected))
            if status == "RE":
                print("    ", out)
            if status != "AC":
                return status
    return "AC"
```

Kode 1.7 (Lanjutan) *minijudge.py: mengompilasi program C++ (atau menjalankan program Python) pada setiap uji di sebuah folder, dengan batas waktu, dan mencetak vonis seperti SPOJ.*

```
def main():
    if len(sys.argv) != 4:
        sys.exit(__doc__)
    testdir, seconds, program = sys.argv[1], float(sys.argv[2]), sys.argv[3]
    verdict = judge(testdir, seconds, program)
    names = {"AC": "accepted", "WA": "wrong answer", "TLE": "time limit exceeded",
             "RE": "runtime error", "CE": "compilation error"}
    print(f"verdict: {verdict} ({names[verdict]})")


if __name__ == "__main__":
    main()
```

Telusuri kode ini menurut urutan **Gambar 1.1**. Fungsi `compile_cpp` menjalankan kompiler dengan opsi buku ini, persis seperti juri mengompilasi sekali sebelum uji apa pun; jika kompiler gagal, `judge` mencetak baris pertama pesan kompiler yang memuat kata *error* dan

mengembalikan CE. Untuk berkas Python tidak ada yang perlu dikompilasi, dan perintahnya menjadi interpreter Python yang sedang berjalan (`sys.executable`) diikuti nama berkas.

Fungsi `run_one` menjalankan program sekali dengan berkas `.in` sebagai masukan standarnya dan menangkap keluaran standarnya. Argumen `timeout` pada `subprocess.run` mewujudkan batas waktu: ketika waktunya habis, Python menghentikan program dan melempar `TimeoutExpired`, yang menjadi TLE. Program yang berakhir dengan kode keluar bukan nol telah berhenti paksa, yang menjadi RE, dan baris terakhir dari apa yang ditulisnya ke aliran galat disimpan, karena bagi Python baris itulah yang menyebut nama eksepsinya.

Fungsi `same_output` mewujudkan perbandingan pada langkah 3. Fungsi ini membuang spasi di ujung setiap baris dan baris kosong di akhir keluaran sebelum membandingkan, kurang lebih seperti perlakuan perbandingan bawaan SPOJ terhadap *whitespace*; baris yang berbeda pada karakter lain dianggap berbeda. Jika vonisnya WA, `first_difference` menampilkan baris pertama yang berbeda, satu-satunya informasi paling berguna ketika Anda memburu jawaban salah, dan informasi yang tidak pernah diberikan oleh juri sungguhan.

Akhirnya, `judge` menjalankan uji menurut urutan nama dan kembali pada kegagalan pertama, seperti SPOJ. Waktu yang dicetak untuk setiap uji adalah waktu jam dinding yang diukur juri, termasuk waktu untuk memulai program; untuk Python, memulai interpreter saja sudah makan beberapa perseratus detik.

1.12.3 Menjuri program yang benar dan yang salah

Sekarang kita dapat berperan sebagai juri. Pertama, kedua solusi dari Subbab 1.11, dari folder `Chapter01`:

```
$ python3 python/minijudge.py tests/TEST 1 cpp/test_life.cpp
$ python3 python/minijudge.py tests/TEST 1 python/test_life.py
```

```
  test 01: AC   0.01 s
  test 02: AC   0.00 s
  test 03: AC   0.00 s
  test 04: AC   0.01 s
verdict: AC (accepted)
  test 01: AC   0.02 s
  test 02: AC   0.03 s
  test 03: AC   0.02 s
  test 04: AC   0.04 s
verdict: AC (accepted)
```

Waktu di komputer Anda akan sedikit berbeda dari satu eksekusi ke eksekusi lain; vonisnya tidak pernah berubah. Sekarang program yang dikirim Rina, **Kode 1.8**.

Kode 1.8 `test_wa.cpp`: solusi salah yang memeriksa 42 baru setelah mencetaknya.

```cpp
// Versi TEST yang salah: memeriksa 42 baru setelah mencetak bilangannya.
#include <iostream>

int main() {
    int x;
    while (std::cin >> x) {
        std::cout << x << '\n';
        if (x == 42) break;
    }
    return 0;
}
```

```
$ python3 python/minijudge.py tests/TEST 1 cpp/test_wa.cpp
```

```
  test 01: WA   0.00 s
    line 4: got '42', expected '<nothing>'
verdict: WA (wrong answer)
```

Juri menemukan *bug* itu pada uji yang paling pertama, yaitu contoh soal, dan menyebutkan barisnya: program mencetak baris keempat, `42`, di tempat keluaran yang benar sudah berakhir. Rina melewatkannya karena baris tambahan itu berada tepat di tempat mata berhenti membaca. Urutan *baca, cetak, uji* alih-alih *baca, uji, cetak* termasuk *bug* paling umum dalam gelung berbentuk seperti ini, dan perbandingan mekanis menemukannya seketika.

Program salah berikutnya, **Kode 1.9**, lebih halus: program ini menyimpan bilangan dalam list berukuran tetap 100 sebelum mencetaknya, kebiasaan yang terbawa dari bahasa dengan larik berukuran tetap.

Kode 1.9 `test_re.py`: solusi salah yang menyimpan bilangan dalam list berukuran tetap dan berhenti paksa pada masukan yang panjang.

```python
"""Versi TEST yang salah: menyimpan bilangan dalam list berukuran tetap 100."""
import sys


def main():
    kept = [0] * 100
    count = 0
    for line in sys.stdin:
        x = int(line)
        if x == 42:
            break
        kept[count] = x  # gagal dengan IndexError pada bilangan ke-101
        count += 1
    for i in range(count):
        print(kept[i])


if __name__ == "__main__":
    main()
```

```
$ python3 python/minijudge.py tests/TEST 1 python/test_re.py
```

```
  test 01: AC   0.02 s
  test 02: AC   0.04 s
  test 03: AC   0.03 s
  test 04: RE   0.09 s
    IndexError: list assignment index out of range
verdict: RE (runtime error)
```

Program ini lolos dari tiga uji pendek dan berhenti paksa pada uji yang panjang, dengan eksepsi yang disebut Python di baris terakhirnya. Di SPOJ vonis ini berbunyi *runtime error (NZEC)*, karena eksepsi Python yang tidak ditangkap mengakhiri program dengan kode keluar bukan nol. Perhatikan bahwa pernyataan soal tidak pernah menyebutkan berapa banyak bilangannya; program mengasumsikan sebuah angka, dan uji dengan bilangan yang lebih banyak mematahkan asumsi itu. Karena itulah himpunan uji yang baik selalu memuat uji yang besar.

Yang terakhir, **Kode 1.10**, bahkan tidak dapat dikompilasi.

Kode 1.10 `test_ce.cpp`: solusi dengan satu titik koma yang hilang.

```cpp
// Versi TEST yang tidak dapat dikompilasi: satu titik koma hilang.
#include <iostream>

int main() {
    int x;
    while (std::cin >> x && x != 42) {
        std::cout << x << '\n'
    }
    return 0;
}
```

```
$ python3 python/minijudge.py tests/TEST 1 cpp/test_ce.cpp
compilation error
cpp/test_ce.cpp:7:31: error: expected ‘;’ before ‘}’ token
verdict: CE (compilation error)
```

Kompiler melaporkan berkas, baris (7), dan kolom (31) tempat ia menyadari masalahnya, yaitu ujung baris 7, tempat titik koma hilang. Tentu Anda akan melihat hal ini di komputer sendiri sebelum mengirim; di juri, CE biasanya punya penyebab lain, misalnya fitur yang lebih baru daripada kompiler juri, atau salah memilih bahasa di formulir pengiriman.

Satu-satunya vonis yang belum kita hasilkan adalah TLE. Bab 3 menghasilkannya secara jujur, dengan program yang benar tetapi algoritmanya terlalu lambat, dan dengan `minijudge.py` yang sama.

1.13 Memakainya Sungguh-sungguh

Semua yang sejauh ini terjadi di komputer Anda. Berikut cara membawanya ke SPOJ, dan cara membangun kebiasaan di sekitarnya.

Mengirim. Buat akun di `https://www.spoj.com` (tautan **sign up** di bagian atas). Buka sebuah soal, misalnya `https://www.spoj.com/problems/TEST/`, lalu klik **Submit**. Pilih bahasa dari daftar, dan pilihlah dengan cermat: untuk C++ pilih entri yang menyebut GCC terbaru yang ditawarkan, yang pada saat buku ini ditulis adalah “C++14 (gcc 8.3)” [4]; untuk Python pilih “Python 3”, jangan pernah Python 2, yang `print` dan pembagiannya bekerja berbeda. Tempelkan kode sumber atau unggah berkasnya, lalu tekan **Send**. Halaman status menampilkan vonis dalam hitungan detik, bersama waktu berjalan dan memori, dan kiriman Anda tetap tersimpan di daftar kiriman Anda, tempat Anda dapat membukanya lagi.

Sebelum setiap pengiriman, jalankan juri Anda sendiri. Untuk soal baru, buat folder `tests` dengan contoh soal sebagai `01.in`/`01.out`, tambahkan kasus terkecil yang terpikirkan, dan satu kasus besar. Bab 2 menunjukkan cara membuat kasus besar dengan generator, dan cara mendapatkan keluaran harapannya dari *brute force* ketika Anda tidak dapat menuliskannya dengan tangan. Dua menit yang dipakai di sini menghemat banyak vonis merah dan, yang lebih penting, melatih Anda meramalkan di mana program akan patah.

Simpan buku catatan. Untuk setiap soal yang Anda selesaikan, catat kodenya, gagasannya dalam satu kalimat, dan setiap vonis yang Anda terima beserta penyebabnya: “ADDREV: WA, mencetak nol di depan.” Setelah lima puluh soal, buku catatan itu menjadi daftar pribadi kesalahan favorit Anda, alat *debugging* paling berharga yang akan Anda miliki. Subbab Kesalahan yang Sering Terjadi di buku ini adalah daftar semacam itu, yang dikumpulkan dari banyak orang.

Untuk klub sekolah atau kampus. `minijudge.py` dapat dipakai untuk soal apa pun yang keluarannya dibandingkan baris demi baris. Seorang pembina dapat memberi klubnya sebuah folder uji untuk suatu soal, dan setiap anggota dapat menjuri programnya secara lokal, tanpa

koneksi internet dan tanpa menunggu antrean, sebelum mengirim ke SPOJ. Mengubah opsi di `FLAGS` menjadi `-std=c++14` membuat juri kecil itu menolak tepat program-program yang akan ditolak kompiler juri.

Irama yang manjur. Pemrograman kompetitif lebih menghargai latihan teratur daripada akhir pekan yang panjang. Menyelesaikan satu atau dua soal sehari, membaca pernyataan soal dengan benar, meramalkan kompleksitas sebelum menulis kode, menguji secara lokal, lalu mengirim, membangun keterampilan buku ini jauh lebih cepat daripada membaca tentangnya. Jika sebuah soal telah melawan Anda selama satu jam, bacalah petunjuk atau penjelasan gagasannya, lalu tutup dan tulis programnya sendiri: solusi yang Anda ketik setelah memahaminya akan melekat; solusi yang Anda salin tidak.

1.14 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

- **Memercayai contoh soal.** Lolos dari contoh menunjukkan bahwa formatnya benar dan hampir tidak ada yang lain. Selalu tambahkan paling tidak masukan terkecil dan masukan terbesar sebelum mengirim.
- **Memeriksa sesudah bertindak.** Dalam gelung yang harus berhenti pada nilai penanda seperti 42, urutannya adalah *baca, uji, bertindak*. `test_wa.cpp` bertindak lebih dulu.
- **Mengasumsikan ukuran masukan yang tetap.** Jika pernyataan soal tidak membatasi banyaknya baris, jangan mengarang batas. Bacalah sampai masukan habis, dan simpan hanya yang Anda perlukan.
- **Mencetak kalimat ajakan.** Juri membandingkan setiap karakter; `Masukkan bilangan:` yang ramah adalah jawaban salah. Cetak persis apa yang diminta format keluaran, tidak lebih.
- **Memakai `long` untuk bilangan besar.** Ukurannya 4 bita di Windows. Pakai `long long`, seperti ditunjukkan Persamaan (1.2) dan `check_env.cpp`.
- **Menyertakan `<bits/stdc++.h>` di Mac.** Clang dari Apple tidak memilikinya. Sertakan *header* standar yang Anda perlukan, atau pasang GCC dengan Homebrew.
- **Salah memilih bahasa di SPOJ.** Kode C++ yang dikirim sebagai C, atau kode Python 3 yang dikirim sebagai Python 2, menghasilkan CE atau RE tanpa alasan yang tampak. Periksa bahasanya sebelum menekan Send.
- **Mengetik `<` di PowerShell.** Ditolak. Pakai Command Prompt, `Get-Content file | program`, atau `minijudge.py` dari bab ini, yang menyuapkan berkasnya sendiri.
- **Membaca vonis sebagai nilai.** TLE tidak mengatakan apa pun tentang kebenaran, dan WA tidak mengatakan apa pun tentang kecepatan. Pakai **Tabel 1.1** untuk memutuskan di mana mencari.

1.15 Ringkasan Bab

- Pemrograman kompetitif adalah menyelesaikan soal yang terspesifikasi secara tepat dengan program yang diperiksa juri dari segi kebenaran keluaran dalam batas waktu dan batas memori. SPOJ adalah arsip permanen yang besar untuk soal-soal seperti itu, masing-masing dengan kode pendek seperti TEST; bagian classical-nya memberi vonis biner.
- Juri mengompilasi berkas sumber sekali, menjalankannya pada setiap uji tersembunyi dengan masukan standar terhubung ke berkas uji, membandingkan keluarannya dengan keluaran harapan, dan berhenti pada kegagalan pertama (**Gambar 1.1**). Vonis AC, WA, TLE, RE, dan CE masing-masing menunjuk jenis *bug* yang berbeda (**Tabel 1.1**).

- Waktu yang diperlukan program kira-kira sama dengan banyaknya langkah dibagi langkah per detik mesin (Persamaan (1.1)), sekitar 10^8 sampai 10^9 untuk C++ dan jauh lebih sedikit untuk Python.
- Meja kerja buku ini adalah g++ (MSYS2 di Windows, Command Line Tools atau Homebrew di macOS, `build-essential` di Linux), Python 3.14, dan VS Code dengan ekstensi C/C++ dan Python (**Tabel 1.2**), dengan *build task* yang mengompilasi memakai `-std=c++17 -O2 -Wall -Wextra` (**Kode 1.1**). Setiap program juga dapat dikompilasi sebagai C++14, sehingga cocok dengan kompiler SPOJ yang lebih tua.
- `long` berukuran 4 bita di Windows; `long long` menampung sampai sekitar 9.2×10^{18} di mana saja (Persamaan (1.2)). Bilangan bulat Python tidak berbatas.
- TEST diselesaikan dengan gelung yang membaca, menguji 42, dan baru kemudian mencetak, dalam C++ (**Kode 1.4**) dan Python (**Kode 1.5**).
- Juri lokal (**Kode 1.7**) dengan sebuah folder uji (**Kode 1.6**) memberi vonis yang sama dengan SPOJ dalam sepersekian detik dan menunjukkan baris pertama yang berbeda. Juri itu menangkap urutan tindakan yang salah (**Kode 1.8**), list berukuran tetap (**Kode 1.9**), dan galat sintaksis (**Kode 1.10**).

1.16 Soal Latihan

1. Untuk setiap situasi berikut, sebutkan vonis yang diberikan SPOJ dan hal pertama yang akan Anda periksa: (a) program mencetak `Case 1: 5` padahal format keluaran meminta `5`; (b) program C++ membaca sejuta bilangan ke dalam `int a[1000]`; (c) program yang benar melakukan gelung atas semua bilangan dari 1 sampai 10^{12}; (d) program yang ditulis dengan fitur C++20 dikirim sebagai "C++14 (gcc 8.3)".
2. Pakai Persamaan (1.1) untuk memperkirakan berapa lama program C++ memerlukan 3×10^9 langkah sederhana, dan apakah program itu muat dalam batas satu detik. Berapa lama program Python memerlukan banyak langkah yang sama, dengan 10^7 langkah per detik?
3. Tulis uji `05` untuk TEST yang masukannya memuat bilangan 42 dua kali, dan jelaskan keluaran yang benar. Program mana saja dari bab ini yang lolos?
4. Ubah `minijudge.py` sehingga, alih-alih berhenti pada kegagalan pertama, ia menjalankan semua uji dan mencetak ringkasan seperti `3 of 4 tests passed`. Mengapa juri sungguhan berhenti pada kegagalan pertama?
5. Jalankan `check_env.cpp` di komputer Anda sendiri dan bandingkan keluarannya dengan yang ada di Subbab 1.10. Pada sistem mana baris ketiga berbeda, dan mengapa hal itu penting bagi program yang menyimpan bilangan 3×10^9?
6. Kawan Rina berkata: "Programmu lolos contoh, jadi pasti jurinya yang salah." Tulislah, dalam tiga atau empat kalimat, jawaban yang disarankan bab ini.

1.17 Penyelesaian Soal Latihan

1. (a) WA: format keluaran tidak diikuti; cetak persis `5`. (b) RE, kemungkinan besar `SIGSEGV`, karena program menulis melewati ujung larik; tentukan ukuran larik dari batasan soal atau pakai `std::vector`. (c) TLE: 10^{12} langkah memerlukan jauh lebih dari seribu detik menurut Persamaan (1.1); algoritmanya yang harus diganti, bukan kode di sekitarnya. (d) CE: kompiler juri tidak mengenal fitur itu; kompilasi secara lokal dengan `-std=c++14` sebelum mengirim.

2. Dengan 10^8 sampai 10^9 langkah per detik, 3×10^9 langkah memerlukan antara 3 dan 30 detik, sehingga program tidak muat dalam batas satu detik bahkan di mesin yang cepat. Dengan 10^7 langkah per detik Python memerlukan $3 \times 10^9 / 10^7 = 300$ detik, lima menit.

3. Masukan `5`, `42`, `7`, `42`, `9`, satu per baris; keluaran yang benar adalah satu baris `5`, karena program harus berhenti pada 42 yang *pertama*. Berkas harapan `05.out` karenanya berisi `5`. Kedua program yang benar, `test_life.cpp` dan `test_life.py`, lolos, karena keduanya berhenti pada 42 pertama dan tidak pernah melihat lebih jauh. `test_re.py` juga lolos, karena masukannya pendek. `test_wa.cpp` gagal, karena mencetak `5` dan `42`. Untuk menambahkan uji itu, tambahkan `write("05", [5, 42, 7, 42, 9])` pada `main` di `gen_test_life.py`; fungsi `expected` sudah memotong pada 42 yang pertama.

4. Di `judge`, hapus baris `if status != "AC": return status`, hitung uji yang bervonis AC dalam variabel `passed`, dan setelah gelung cetak `f"{passed} of {len(tests)} tests passed"` lalu kembalikan `"AC"` hanya jika `passed == len(tests)`, dan jika tidak, kembalikan vonis kegagalan pertama, yang Anda simpan dalam sebuah variabel. Juri sungguhan berhenti pada kegagalan pertama karena dua alasan: menghemat waktu mesin ketika ribuan kiriman menunggu di antrean, dan menghindari memberi tahu peserta uji mana yang lolos, yang memungkinkan mereka meraba-raba isi uji tersembunyi.

5. Baris itu berbunyi `long 8` di Linux dan macOS dan `long 4` di Windows, baik dengan g++ dari MSYS2 maupun dengan kompiler Microsoft. Bilangan 3×10^9 lebih besar daripada `INT_MAX`, $2^{31} - 1 \approx 2.1 \times 10^9$ (Persamaan (1.2)), sehingga `long` 4 bita tidak dapat menampungnya: program benar di Linux dan salah di Windows. `long long` menampungnya di semua sistem.

6. Contoh soal hanya memperlihatkan format, dan juri menjalankan program pada banyak uji tersembunyi yang dirancang untuk menemukan kesalahan: masukan terkecil dan terbesar, nilai-nilai khusus, dan masukan yang panjang. Juri tidak membaca kode, juri membandingkan keluaran, sehingga program yang mencetak satu baris berlebih tetap gagal meskipun "hampir benar". Tanggapan yang tepat terhadap WA adalah menulis lebih banyak uji, membandingkan secara mekanis dengan alat seperti `minijudge.py`, dan mencari masukan yang membuat program berperilaku salah.

Daftar Pustaka

[1] ICPC Foundation, "The ICPC International Collegiate Programming Contest." [Online]. Available: https://icpc.global (accessed Sep. 2026).

[2] Sphere Research Labs, "SPOJ: Sphere Online Judge." [Online]. Available: https://www.spoj.com (accessed Sep. 2026).

[3] SPOJ, "TEST — Life, the Universe, and Everything." [Online]. Available: https://www.spoj.com/problems/TEST/ (accessed Sep. 2026).

[4] hkktr, "polski-spoj: solutions for POLSKI SPOJ problems," README, stating the language used as "C++14 (gcc 8.3)". [Online]. Available: https://github.com/hkktr/polski-spoj (accessed Sep. 2026).

[5] GNU Project, "GCC, the GNU Compiler Collection," release list with GCC 16.2 (Aug. 7, 2026) and GCC 15.3 (Jun. 12, 2026). [Online]. Available: https://gcc.gnu.org/ (accessed Sep. 2026).

[6] L. Rendek, "How to Install GCC and Compile C/C++ Programs on Ubuntu 26.04," LinuxConfig, Apr. 1, 2026. [Online]. Available: https://linuxconfig.org/how-to-install-gcc-and-compile-c-c-programs-on-ubuntu-26-04

[7] MSYS2 Developers, "Package: mingw-w64-ucrt-x86_64-gcc," version 16.2.0-4, built Sep. 20, 2026. [Online]. Available: https://packages.msys2.org/packages/mingw-w64-ucrt-x86_64-gcc

[8] Apple Inc., "Installing the command-line tools," *Xcode Documentation*. [Online]. Available: https://developer.apple.com/documentation/xcode/installing-the-command-line-tools (accessed Sep. 2026).

[9] Python Software Foundation, "Python 3.14.7," Aug. 5, 2026. [Online]. Available: https://www.python.org/downloads/latest/python3.14/

[10] Microsoft, "Visual Studio Code Updates, version 1.139," Sep. 23, 2026. [Online]. Available: https://code.visualstudio.com/updates

[11] Microsoft, "C/C++ for Visual Studio Code," *Visual Studio Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/languages/cpp (accessed Sep. 2026).

[12] Microsoft, "Python in Visual Studio Code," *Visual Studio Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/languages/python (accessed Sep. 2026).

[13] Microsoft, "Using GCC with MinGW," *Visual Studio Code Documentation*. [Online]. Available: https://code.visualstudio.com/docs/cpp/config-mingw (accessed Sep. 2026).

[14] MSYS2 Developers, "MSYS2: Software distribution and building platform for Windows." [Online]. Available: https://www.msys2.org (accessed Sep. 2026).

[15] Python Software Foundation, "Using Python on Windows," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/using/windows.html (accessed Sep. 2026).

[16] Homebrew, "gcc: GNU compiler collection," formula version 16.2.0. [Online]. Available: https://formulae.brew.sh/formula/gcc (accessed Sep. 2026).

Bab 2
Membaca Soal Seperti Juri

Sebagian besar kiriman yang ditolak bukanlah kegagalan keterampilan algoritmik, melainkan kegagalan membaca. Sebuah frasa seperti “paling banyak”, sebuah kalimat seperti “abaikan nol di depan”, sebaris batasan yang mengatakan bahwa masukan dapat memuat seratus kasus, atau daftar prioritas yang berbeda dari yang Anda pelajari di sekolah: masing-masing menentukan apakah gagasan yang benar berubah menjadi program yang diterima. Bab ini mengajarkan cara membaca pernyataan soal secara disiplin sebelum menulis kode apa pun. Di akhir bab Anda akan mampu menemukan enam zona sebuah pernyataan soal dan menyebutkan apa yang perlu diambil dari masing-masing; menjawab daftar periksa membaca yang singkat, yang mencegah sebagian besar jawaban salah sebelum terjadi; mengenali tiga bentuk kedatangan masukan juri dan membaca masing-masing dengan benar dalam C++ dan Python; menceritakan ulang soal dengan kata-kata sendiri, termasuk jebakannya; serta menyelesaikan tiga soal SPOJ sungguhan, ADDREV, FCTRL2, dan ONP, masing-masing diperiksa terhadap *brute force* oleh sebuah *stress test* yang akan Anda pakai ulang di sepanjang buku.

2.1 Baris yang Tak Dibaca Siapa Pun

Setiap orang Indonesia yang pernah memperpanjang SIM atau mengisi formulir pajak mengenal rasanya: formulir dikembalikan di loket karena satu kotak diisi dengan format yang salah, tanggal ditulis hari-bulan padahal yang diminta bulan-hari, nama ditulis dengan huruf kecil padahal diminta huruf kapital. Petugasnya tidak sedang kejam. Petunjuknya tercetak di formulir; petunjuk itu hanya tidak dibaca, karena semua orang merasa sudah tahu cara menulis tanggal.

Tim kampus Rina bertemu petugas yang sama dalam kontes daring pertama mereka. Program mereka untuk sebuah soal bilangan prima benar dan cepat. Program itu ditolak dengan *wrong answer* tiga kali. Pernyataan soalnya, menjelang akhir, dalam kalimat yang tidak dibaca sampai kata terakhir oleh siapa pun di tim, meminta sebuah baris kosong di antara keluaran kasus uji yang berurutan. Tim itu mencetak bilangan prima setiap kasus satu demi satu tanpa baris kosong tersebut. Setelah kontes, sang pelatih menyuruh mereka membaca pernyataan soal keras-keras, kalimat demi kalimat, dan mencatat setiap aturan yang mereka dengar. Mereka menemukan baris kosong itu pada menit kedua.

Pelajarannya bukan “lebih berhati-hatilah”, nasihat yang tidak dapat diikuti siapa pun. Pelajarannya adalah bahwa membaca pernyataan soal merupakan keterampilan yang punya prosedur, seperti membaca peta atau kontrak, dan prosedur itu dapat dipelajari. Juri membaca keluaran Anda terhadap pernyataan soal dengan perhatian sempurna pada setiap karakter. Satu-satunya pertahanan adalah membaca pernyataan soal dengan perhatian yang sama, sekali, secara sistematis, sebelum baris kode pertama.

Siapa yang diuntungkan? Setiap orang yang menulis program berdasarkan spesifikasi, yaitu setiap pemrogram. Pengembang yang mewujudkan antarmuka yang dijabarkan oleh bank, kantor pajak, atau *payment gateway* menghadapi teks jenis ini persis: format masukan, format keluaran, batas ukuran, aturan untuk kasus khusus, dan contoh yang menggambarkan format tanpa

mencakup aturannya. Membaca dokumen seperti itu dengan cara yang diajarkan bab ini adalah bagian besar dari mengerjakan pekerjaan itu dengan baik.

2.2 Anatomi Pernyataan Soal

Hampir setiap pernyataan soal di SPOJ, dan di juri mana pun, dibangun dari enam zona yang sama. Zona-zona itu tidak selalu berjudul, dan urutannya tidak selalu sama, tetapi selalu ada, dan **Gambar 2.1** mendaftarnya beserta apa yang sebaiknya Anda catat dari masing-masing.



Gambar 2.1 Enam zona pernyataan soal dan apa yang dicatat dari masing-masing. Batasan, yang ditandai kuning tua, menentukan algoritma sekaligus kasus tepi.

Kepala: judul, kode, batas. Bagian atas setiap halaman soal SPOJ menampilkan kode soal, batas waktu per berkas uji, dan batas memori, dan sering juga daftar bahasa yang diizinkan. Semua itu bagian dari spesifikasi, sama seperti format masukan. Catat batas waktunya, karena Bab 3 mengubahnya menjadi anggaran langkah.

Cerita dan tugas. Banyak pernyataan soal membungkus tugasnya dengan sebuah cerita: tokoh, sapi, raja dengan harta karun. Cerita membuat soal mudah diingat, dan kadang menyamarkan soal baku sebagai sesuatu yang asing. *Tugasnya* biasanya satu kalimat, sering kalimat terakhir cerita, yang menyebutkan apa yang harus dihitung. Temukan kalimat itu dan garis bawahi; selebihnya dari cerita adalah hiasan, kecuali jika mendefinisikan istilah yang dipakai tugas.

Format masukan. Zona ini adalah kontrak. Zona ini menyebutkan berapa banyak bilangan atau baris yang datang, dalam urutan apa, dan bagaimana masukan berakhir. “Baris pertama memuat banyaknya kasus uji” dan “masukan berakhir di akhir berkas” memerlukan gelung yang berbeda, dan Subbab 2.4 menunjukkan keduanya.

Format keluaran. Kontrak lain, dan pelanggarannya paling menjengkelkan, karena menghasilkan WA pada algoritma yang benar. Carilah ejaan yang tepat, label kasus seperti `Case 1:`, pemisah antarbilangan, baris kosong antarkasus, banyaknya digit di belakang titik desimal, dan apa yang harus dicetak jika tidak ada jawaban.

Batasan. Spesifikasi yang sebenarnya. Nilai-nilai terbesar menentukan algoritma mana yang cukup cepat, seperti ditunjukkan Bab 3 secara rinci. Nilai-nilai terkecil, dan kata-kata di

sekitarnya, memberi tahu kasus tepi mana yang ada: bolehkah n bernilai 0 atau 1? bolehkah bilangannya negatif, kembar, atau nol? apakah masukan sudah terurut? apakah nilainya muat dalam 32 bit? Pernyataan yang berbunyi $1 \leq n$ baru saja memberi tahu Anda bahwa Anda tidak perlu menangani daftar kosong; pernyataan yang berbunyi $0 \leq n$ baru saja memberi tahu bahwa Anda harus menanganinya.

Contoh. Sebuah contoh masukan dan keluarannya. Contoh memperlihatkan format, dan mengerjakannya dengan tangan adalah pemeriksaan terbaik bahwa Anda memahami tugasnya. Contoh hampir tidak pernah mencakup kasus tepi, dan tidak pernah merupakan himpunan uji yang lengkap.

> **WAWASAN KUNCI:** Bacalah batasan dua kali: sekali untuk ukuran, sekali untuk bentuk. Putaran pertama menjawab "seberapa besar masukan bisa menjadi?", yang memilih algoritma. Putaran kedua menjawab "seperti apa nilai-nilainya bisa tampak?", nol, negatif, kembar, sangat besar, yang memilih kasus tepi yang harus diuji. Kebanyakan pemula hanya melakukan putaran pertama.

Untuk melihat kedua putaran itu bekerja, ambil batasan PRIME1, soal yang membuat tim Rina tersandung, yang diselesaikan secara lengkap di Bab 3: paling banyak 10 kasus uji, masing-masing sebuah rentang dari m sampai n dengan $1 \leq m \leq n \leq 10^9$ dan $n - m \leq 10^5$. Putaran ukuran berbunyi: bilangan sampai satu miliar, tetapi setiap rentang memuat paling banyak seratus ribu bilangan, dan paling banyak ada sepuluh rentang, sehingga total paling banyak sejuta bilangan yang harus diperiksa; algoritma yang bekerja sedikit untuk setiap bilangan dalam rentang tidak masalah, algoritma yang menyentuh setiap bilangan sampai n tidak bisa. Putaran bentuk berbunyi: m bisa bernilai 1, yang bukan bilangan prima, sehingga program tidak boleh mencetaknya; m bisa sama dengan n, sehingga sebuah rentang bisa hanya memuat satu bilangan; dan bilangannya mencapai 10^9, yang muat dalam `int`, sedangkan hasil kali seperti $p \cdot p$ untuk p di sekitar 31 623 sudah mendekati batas `int` dan harus dihitung dalam `long long`. Enam pengamatan singkat, dan masing-masing berpadanan dengan *bug* yang pernah dikirim seseorang. Tambahkan aturan keluaran yang terlewat oleh tim Rina, sebuah baris kosong di antara kasus uji, dan seluruh spesifikasi muat di sehelai kartu indeks.

2.3 Daftar Periksa Membaca

Daftar periksa mengubah perhatian menjadi prosedur. **Tabel 2.1** memuat delapan pertanyaan; jawablah masing-masing, secara tertulis jika soalnya baru bagi Anda, sebelum menulis kode. Di sebelah setiap pertanyaan tercantum jenis vonis yang biasanya muncul jika pertanyaan itu dilewati.

Tabel 2.1 Daftar periksa membaca: delapan pertanyaan yang dijawab sebelum menulis kode, dan vonis yang cenderung muncul jika masing-masing dilewati.

#	Pertanyaan	Tempat mencari	Jika dilewati
1	Apa tepatnya yang ditanyakan? Dapatkah saya menyatakannya dalam satu kalimat?	Cerita dan tugas	WA di semua uji
2	Berapa banyak kasus uji, dan bagaimana masukan berakhir?	Format masukan	WA, RE, atau TLE karena menunggu masukan
3	Berapa masukan terbesar, dan kompleksitas apa yang diizinkannya?	Batasan, batas waktu	TLE
4	Berapa masukan terkecil, dan bolehkah nilainya 0, negatif, kembar?	Batasan	WA pada kasus tepi, RE
5	Apakah semua nilai dan hasil antara muat dalam `int`? dalam `long long`?	Batasan	WA karena *overflow*
6	Apa tepatnya yang harus dicetak, dengan pemisah dan baris kosong apa?	Format keluaran	WA pada algoritma yang benar
7	Apakah jawabannya tunggal? Jika tidak, apa yang harus dicetak?	Tugas, format keluaran	WA karena jawaban sah yang berbeda
8	Apakah perhitungan tangan saya atas contoh menghasilkan contoh keluaran?	Contoh	WA karena salah memahami tugas

Cara terbaik menjawab pertanyaan 1 adalah menulis sebuah *penceritaan ulang*: dua sampai lima kalimat, dengan kata-kata Anda sendiri, yang menyatakan apa masukannya, apa keluarannya, dan setiap aturan khusus. Penceritaan ulang berguna justru karena memaksa Anda memutuskan arti setiap kalimat aslinya. Jika Anda tidak dapat menuliskannya, Anda belum memahami soalnya, dan sebanyak apa pun menulis kode tidak akan memperbaikinya. Mulai sekarang, setiap soal SPOJ dalam buku ini diawali penceritaan ulang semacam itu, dengan gaya yang sebaiknya Anda pakai di buku catatan Anda sendiri. Buku-buku latihan baku pemrograman kompetitif menganjurkan disiplin yang sama: membaca dengan cermat, memodelkan, dan menguji [1], [2].

Beberapa jebakan cukup sering muncul di SPOJ sehingga layak diberi nama.

- **"Paling banyak" lawan "tepat".** "Paling banyak 100 kasus uji" mengizinkan satu kasus; "paling banyak 10 digit" mengizinkan satu digit. Gelung Anda tidak boleh mengasumsikan nilai maksimum.
- **Aturan yang disebut sekali.** Pernyataan soal yang di paragraf pertamanya menyebut bahwa posisi dinomori mulai 1 tidak akan menyebutkannya lagi di bagian batasan.
- **Satuan tersembunyi.** Waktu dalam menit, sudut dalam derajat, uang dalam sen: ceritanya biasanya menyebutkan, format masukannya tidak.
- **Nilai seri.** "Cetak yang terkecil", "cetak yang mana saja", atau tidak disebut sama sekali: tiga soal yang berbeda.
- **Rentang dan ujung-ujungnya.** "Dari a sampai b" biasanya mencakup kedua ujung, "di antara" bermakna ganda dalam bahasa sehari-hari, dan "k yang pertama" bisa dihitung dari 0 atau dari 1. Jika pernyataan soal tegas, misalnya dengan kata *inclusive*, patuhilah; jika tidak, contoh soal biasanya yang memutuskan, jadi kerjakan contoh itu dengan tangan.
- **Contoh yang menyembunyikan aturan.** Contoh yang hanya berisi nilai positif, berbeda, dan kecil tidak mengatakan apa pun tentang nol, nilai kembar, atau nilai besar, meskipun batasannya mengizinkan ketiganya.

2.4 Tiga Bentuk Masukan Juri

Masukan juri datang dalam tiga bentuk, dan masing-masing memerlukan gelungnya sendiri.

Banyaknya di depan. Baris pertama memuat bilangan T, lalu menyusul T kasus uji. Inilah bentuk yang paling umum di SPOJ, dan ADDREV, FCTRL2, serta ONP di bab ini semuanya memakainya. Gelungnya membaca T lalu berjalan tepat T kali.

Penanda. Masukan diakhiri nilai khusus yang bukan kasus uji, misalnya 42 pada TEST di Bab 1, atau baris berisi `0 0`. Gelungnya membaca, membandingkan dengan penanda, lalu berhenti.

Sampai akhir berkas. Tanpa banyaknya dan tanpa penanda: kasus uji terus berdatangan sampai masukan habis. Gelung harus mendeteksi sendiri akhir masukan. **Kode 2.1** memperlihatkan idiom C++-nya untuk sebuah soal rekaan yang meminta jumlah setiap pasang bilangan.

Kode 2.1 `sum_until_eof.cpp`: bentuk masukan "sampai akhir berkas" dalam C++.

```cpp
// Masukan "sampai akhir berkas": jumlahkan tiap pasang, berapa pun banyaknya.
#include <iostream>

int main() {
    long long a, b;
    int line = 0;
    while (std::cin >> a >> b) {  // bernilai false saat masukan habis
        line++;
        std::cout << "pair " << line << ": " << a + b << '\n';
    }
    return 0;
}
```

Seperti di Bab 1, `std::cin >> a >> b` membaca dua bilangan dan bernilai aliran itu sendiri, yang bernilai salah begitu sebuah pembacaan gagal; di akhir masukan pembacaan gagal dan gelung berakhir. Tidak perlu tahu sebelumnya berapa pasang yang ada, dan tidak perlu pemeriksaan terpisah seperti `std::cin.eof()`. Pemeriksaan itu malah merupakan jebakan: `eof()` baru bernilai benar *setelah* sebuah pembacaan mencoba melewati akhir, sehingga gelung `while (!std::cin.eof())` berjalan sekali terlalu banyak dan memproses pasangan terakhir dua kali atau sepasang sampah sekali.

Kode 2.2 `sum_until_eof.py`: bentuk masukan yang sama dalam Python, dengan membaca semua token sekaligus.

```python
"""Masukan "sampai akhir berkas": jumlahkan tiap pasang, berapa pun banyaknya."""
import sys


def main():
    numbers = [int(t) for t in sys.stdin.read().split()]
    for line, i in enumerate(range(0, len(numbers) - 1, 2), start=1):
        print(f"pair {line}: {numbers[i] + numbers[i + 1]}")


if __name__ == "__main__":
    main()
```

Versi Python membaca seluruh masukan dengan `sys.stdin.read()`, memecahnya menjadi *token* (rangkaian karakter terpanjang tanpa spasi atau pindah baris) dengan `split()`, dan mengubahnya menjadi bilangan bulat. Gaya ini punya keunggulan besar yang mudah terlewat: ia tidak peduli bagaimana bilangan-bilangan tersebar di baris-baris. Kedua program memberi

jawaban yang sama, baik ketiga pasang ditulis sepasang per baris maupun semuanya dalam satu baris:

```
$ printf '1 2\n3 4\n100 200\n' | ./cpp/sum_until_eof
$ printf '1 2 3 4\n\n100\n200' | python3 python/sum_until_eof.py
```

```
pair 1: 3
pair 2: 7
pair 3: 300
```

(Setiap program mencetak baris-baris ini, untuk masukan mana pun.) Masukan kedua punya baris kosong di tengah dan tidak punya pindah baris di akhir, dua hal yang mematahkan program yang membaca per baris dan yang memang terdapat dalam berkas juri sungguhan. Kecuali struktur baris itu sendiri membawa informasi, bacalah token, bukan baris. Pembacaan berbasis token yang sama juga melindungi Anda dari karakter `\r` yang dibawa berkas buatan Windows di akhir setiap baris: `split()` memperlakukannya sebagai *whitespace*, demikian pula `std::cin >>`.

2.5 ADDREV: Membaca "Abaikan Nol di Depan"

Soalnya (SPOJ ADDREV, "Adding Reversed Numbers", `https://www.spoj.com/problems/ADDREV/`) [3]. Pernyataan soal memperkenalkan *bilangan terbalik*: bilangan yang ditulis dengan digit dalam urutan terbalik, sehingga 1245 menjadi 5421. Jika sebuah bilangan berakhir dengan nol, bentuk terbaliknya kehilangan nol itu, sehingga 1200 dibalik menjadi 21, bukan 0021: bilangan terbalik tidak pernah diawali nol. Masukan diawali banyaknya kasus uji; setiap kasus adalah satu baris berisi dua bilangan bulat positif, yang sudah merupakan bilangan *terbalik*. Ambil kedua bilangan asal mereka, jumlahkan, dan cetak jumlahnya dalam bentuk terbalik, lagi-lagi tanpa nol di depan. Pernyataan soal juga menyebutkan bahwa bilangan asalnya dianggap yang tanpa nol di belakang, karena nol yang hilang saat pembalikan tidak dapat dipulihkan.

Diceritakan ulang dalam satu kalimat: *untuk setiap pasangan* a, b*, cetak* $\mathrm{rev}(\mathrm{rev}(a) + \mathrm{rev}(b))$*, dengan* rev *membalik digit dan membuang nol di depan.*

Membaca jebakannya. Telusuri daftar periksa. Pertanyaan 2: banyaknya di depan. Pertanyaan 4: bilangannya positif, jadi 0 tidak pernah muncul sebagai masukan, tetapi sebuah jumlah bisa berakhir dengan nol, dan pembalikannya lalu diawali nol yang harus dibuang. Pertanyaan 5: bilangan masukan berukuran sedang, tetapi sebuah jumlah bisa satu digit lebih panjang daripada suku-sukunya, dan kita memakai `long long` 64-bit agar tidak ada kasus yang *overflow*. Pertanyaan 6: satu bilangan per baris.

$$\mathrm{rev}(x) = \sum_{i=0}^{k-1} d_i \cdot 10^{k-1-i} \quad \text{dengan } x = \sum_{i=0}^{k-1} d_i \cdot 10^{i} \tag{2.1}$$

Persamaan (2.1) menyatakannya dengan tepat: jika x punya digit $d_{k-1} \ldots d_1 d_0$ dari kiri ke kanan, sehingga d_0 adalah digit satuan, maka $\mathrm{rev}(x)$ punya digit $d_0 d_1 \ldots d_{k-1}$. Jika $d_0 = 0$, digit terdepan $\mathrm{rev}(x)$ adalah 0, yang berarti bilangan itu hanya punya lebih sedikit digit; sebagai bilangan bulat, nol itu lenyap dengan sendirinya.

Contoh yang dikerjakan. Kasus contoh ketiga adalah 305 dan 794.

1. $\mathrm{rev}(305) = 503$, dan $\mathrm{rev}(794) = 497$.
2. $503 + 497 = 1000$.
3. $\mathrm{rev}(1000)$ adalah string digit `0001`, yang sebagai bilangan bernilai 1.

Jadi jawabannya `1`. Kasus ini adalah seluruh soal dalam bentuk mini, dan contoh soal memuatnya, sebuah kebaikan hati penyusun soal: ketiga kasus contohnya adalah 24,1 dengan jawaban 34; 4358,754 dengan jawaban 1998; dan 305,794 dengan jawaban 1.

Gagasannya. Hitung rev pada bilangan bulat, digit demi digit, bukan pada string, seperti yang dilakukan **Kode 2.3**. x mod 10 memberi digit terakhir, dan $\lfloor x/10 \rfloor$ membuangnya; menambahkan digit d ke r adalah $r \leftarrow 10r + d$. Nol di depan hasilnya lenyap dengan sendirinya, karena $10 \cdot 0 + 0 = 0$.

Kode 2.3 `addrev.cpp`: ADDREV dengan pembalikan digit pada bilangan bulat.

```cpp
// SPOJ ADDREV (Adding Reversed Numbers): reverse(reverse(a) + reverse(b)).
#include <iostream>

// Digit x dalam urutan terbalik; nol di depan hasilnya lenyap.
long long reverseDigits(long long x) {
    long long r = 0;
    while (x > 0) {
        r = r * 10 + x % 10;
        x /= 10;
    }
    return r;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int cases;
    std::cin >> cases;
    for (int c = 0; c < cases; c++) {
        long long a, b;
        std::cin >> a >> b;
        std::cout << reverseDigits(reverseDigits(a) + reverseDigits(b)) << '\n';
    }
    return 0;
}
```

Gelung di `reverseDigits` berjalan sekali per digit, sehingga satu kasus memerlukan waktu sebanding dengan banyaknya digit, paling banyak sekitar sepuluh, dan seluruh program linear terhadap ukuran masukan. Dua baris setelah deklarasi `main` membuat masukan dan keluaran lebih cepat; keduanya tidak merugikan di sini, dan Bab 4 menjelaskannya.

Kode 2.4 `addrev.py`: ADDREV dalam Python, membalik melalui string.

```python
"""SPOJ ADDREV (Adding Reversed Numbers): reverse(reverse(a) + reverse(b))."""
import sys


def reverse_digits(x):
    """Digit x dalam urutan terbalik; nol di depan hasilnya lenyap."""
    return int(str(x)[::-1])


def main():
    data = sys.stdin.read().split()
    cases = int(data[0])
    out = []
    for c in range(cases):
        a = int(data[1 + 2 * c])
        b = int(data[2 + 2 * c])
        out.append(str(reverse_digits(reverse_digits(a) + reverse_digits(b))))
    print("\n".join(out))


if __name__ == "__main__":
    main()
```

Versi Python, **Kode 2.4**, membalik melalui string: `str(x)[::-1]` adalah digit-digitnya dalam urutan terbalik, dan `int()` dari string yang diawali nol cukup mengabaikannya, sehingga `int("0001")` bernilai 1. Program mengumpulkan semua jawaban dalam sebuah list dan mencetaknya dengan digabung memakai pindah baris, satu penulisan besar alih-alih banyak penulisan kecil; Bab 4 mengukur mengapa hal itu membantu.

```
$ printf '3\n24 1\n4358 754\n305 794\n' | ./cpp/addrev
$ printf '3\n24 1\n4358 754\n305 794\n' | python3 python/addrev.py
```

```
34
1998
1
```

(Kedua program mencetak baris-baris ini.)

2.5.1 Versi yang salah, dan alat yang menemukannya

Kode 2.5 adalah program yang ditulis banyak orang pada percobaan pertama. Program ini membalik masukan dengan benar, tetapi membalik *jumlahnya* sebagai string dan mencetak string itu apa adanya.

Kode 2.5 `addrev_bug.cpp`: ADDREV yang salah, yang mempertahankan nol di depan pada jumlah yang dibalik.

```cpp
// ADDREV yang salah: membalik jumlah sebagai teks dan lupa nol di depan.
#include <algorithm>
#include <iostream>
#include <string>

long long reverseDigits(long long x) {
    long long r = 0;
    while (x > 0) {
        r = r * 10 + x % 10;
        x /= 10;
    }
    return r;
}

int main() {
    int cases;
    std::cin >> cases;
    for (int c = 0; c < cases; c++) {
        long long a, b;
        std::cin >> a >> b;
        std::string s = std::to_string(reverseDigits(a) + reverseDigits(b));
        std::reverse(s.begin(), s.end());  // 1000 menjadi "0001"
        std::cout << s << '\n';
    }
    return 0;
}
```

Pada contoh soal, program ini mencetak `34`, `1998`, dan `0001`: baris ketiga membongkar kesalahannya, jika Anda melihatnya. Tetapi andaikan contohnya tidak memuat jumlah yang berakhir dengan nol. Apakah Anda akan menemukan *bug* itu? Inilah saatnya memperkenalkan alat yang menemukan *bug* semacam itu secara sistematis, yaitu *stress test*. Alat ini memerlukan tiga program:

- **generator** yang mencetak satu masukan acak, yang ditentukan oleh *seed* yang diberikan di baris perintah, sehingga setiap kegagalan dapat direproduksi;
- ***brute force*** yang menyelesaikan soal dengan cara sesederhana mungkin, tanpa kecerdikan sama sekali, sehingga kebenarannya jelas;
- **program yang diuji**.

Stress test menjalankan generator dengan *seed* 1, 2, 3, dan seterusnya, menyuapkan setiap masukan ke kedua program, dan berhenti pada masukan pertama yang membuat keluaran keduanya berbeda. **Kode 2.6** adalah *stress test* yang kita pakai di sepanjang sisa buku.

Kode 2.6 `stress.py`: menjalankan generator, *brute force*, dan program pada banyak *seed*, lalu mencetak masukan pertama yang jawabannya berbeda.

```
"""stress.py: bandingkan program dengan brute force pada banyak masukan acak.

pemakaian: python3 stress.py GEN.py BRUTE.py PROGRAM [ROUNDS]
GEN.py mencetak satu masukan untuk seed yang diberikan sebagai argumen pertama.
BRUTE.py dan PROGRAM membaca masukan itu; PROGRAM boleh berkas .cpp atau .py.
Masukan pertama yang jawabannya berbeda dicetak, lalu proses berhenti.
"""
import os
import subprocess
import sys
import tempfile
from pathlib import Path

CXX = os.environ.get("CXX", "g++")
FLAGS = ["-std=c++17", "-O2", "-Wall", "-Wextra"]


def command_for(program, workdir):
    if program.endswith(".cpp"):
        exe = str(Path(workdir) / ("prog.exe" if os.name == "nt" else "prog"))
        subprocess.run([CXX, *FLAGS, "-o", exe, program], check=True)
        return [exe]
    return [sys.executable, program]


def run(command, text):
    r = subprocess.run(command, input=text, capture_output=True, text=True,
                       timeout=10)
    return r.stdout if r.returncode == 0 else f"<crash {r.returncode}>"


def normalize(text):
    return [line.rstrip() for line in text.rstrip().splitlines()]


def main():
    if len(sys.argv) < 4:
        sys.exit(__doc__)
    gen, brute, program = sys.argv[1:4]
    rounds = int(sys.argv[4]) if len(sys.argv) > 4 else 500
    with tempfile.TemporaryDirectory() as work:
        fast = command_for(program, work)
        for seed in range(1, rounds + 1):
            test = run([sys.executable, gen, str(seed)], "")
            want = run([sys.executable, brute], test)
            got = run(fast, test)
```

Kode 2.6 (Lanjutan) *stress.py: menjalankan generator, brute force, dan program pada banyak seed, lalu mencetak masukan pertama yang jawabannya berbeda.*

```
            if normalize(got) != normalize(want):
                print(f"round {seed}: DIFFERENT")
                print("input:\n" + test.rstrip())
                print("expected:\n" + want.rstrip())
                print("got:\n" + got.rstrip())
                return
    print(f"{rounds} rounds: all answers equal")


if __name__ == "__main__":
    main()
```

Generator yang baik untuk ADDREV harus menghasilkan masukan yang penting. Bilangan acak dari distribusi seragam jarang berakhir dengan nol, jadi **Kode 2.7** sengaja membuat nol sering muncul: string digit `"0000123456789"` memuat empat nol di antara tiga belas karakter. Digit

pertama tidak pernah nol, karena bilangan masukan positif dan tidak diawali nol; generator harus mematuhi batasan, kalau tidak *stress test* melaporkan "*bug*" pada masukan yang tidak pernah terjadi.

Kode 2.7 `gen_addrev.py`: masukan ADDREV acak yang kecil dan kaya nol.

```
"""gen_addrev.py SEED: masukan ADDREV acak yang kecil dan kaya angka nol."""
import random
import sys


def number(rng):
    digits = rng.randint(1, 6)
    first = str(rng.randint(1, 9))  # bilangan positif tidak diawali nol
    rest = "".join(rng.choice("0000123456789") for _ in range(digits - 1))
    return first + rest


def main():
    rng = random.Random(int(sys.argv[1]))
    cases = rng.randint(1, 5)
    print(cases)
    for _ in range(cases):
        print(number(rng), number(rng))


if __name__ == "__main__":
    main()
```

Brute force-nya, **Kode 2.8**, mengikuti definisi secara harfiah dengan string dan tidak memerlukan penalaran tentang digit atau aritmetika.

Kode 2.8 `brute_addrev.py`: ADDREV langsung dari definisinya.

```
"""brute_addrev.py: ADDREV dengan cara lambat yang jelas, hanya dengan string."""
import sys


def main():
    data = sys.stdin.read().split()
    for c in range(int(data[0])):
        a, b = data[1 + 2 * c], data[2 + 2 * c]
        total = int(a[::-1]) + int(b[::-1])  # int() membuang nol di depan
        print(int(str(total)[::-1]))


if __name__ == "__main__":
    main()
```

Menjalankan *stress test* pada program yang benar dan pada program yang salah, dari folder `tests`:

```
$ python3 stress.py gen_addrev.py brute_addrev.py ../cpp/addrev.cpp
$ python3 stress.py gen_addrev.py brute_addrev.py ../python/addrev.py 200
$ python3 stress.py gen_addrev.py brute_addrev.py ../cpp/addrev_bug.cpp
```

```
500 rounds: all answers equal
200 rounds: all answers equal
round 1: DIFFERENT
input:
2
21049 8739
24 7
expected:
93301
94
got:
093301
94
```

Kedua program yang benar sesuai dengan *brute force* pada 500 dan 200 masukan acak. Program yang salah gagal di putaran pertama: rev(21049) + rev(8739) = 94012 + 9378 = 103390, yang pembalikannya sebagai string adalah `093301`. Masukan kecil dan konkret yang memperlihatkan *bug* lebih berharga daripada satu jam memelototi kode.

> **PERINGATAN:** *Stress test* hanya sebaik generator dan *brute force*-nya. Generator yang tidak pernah menghasilkan masukan sulit, di sini bilangan yang berakhir dengan nol, meloloskan setiap *bug*; *brute force* yang berbagi gagasan dengan program cepat juga berbagi *bug*-nya. Tulislah *brute force* dari definisi, sejauh mungkin berbeda dari program cepat.

2.6 FCTRL2: Membaca Ukuran Jawaban

Soalnya (SPOJ FCTRL2, "Small factorials", `https://www.spoj.com/problems/FCTRL2/`) [4]. Masukan diawali banyaknya kasus uji t, $1 \le t \le 100$; setiap kasus berupa satu bilangan bulat n dengan $1 \le n \le 100$. Untuk setiap kasus, cetak $n!$, hasil kali $1 \cdot 2 \cdot 3 \cdots n$, secara tepat, di barisnya sendiri.

Judulnya menyebut faktorial *kecil*, dan batasannya menyebut $n \le 100$. Pembaca yang tergesa-gesa menyimpulkan bahwa soalnya remeh. Pembaca yang mengikuti daftar periksa berhenti di pertanyaan 5: apakah jawabannya muat dalam `long long`? Banyaknya digit desimal bilangan bulat positif x adalah $\lfloor \log_{10} x \rfloor + 1$, dan logaritma sebuah hasil kali adalah jumlah logaritmanya, yang memberikan Persamaan (2.2).

$$\text{digits}(n!) = \left\lfloor \sum_{k=1}^{n} \log_{10} k \right\rfloor + 1 \tag{2.2}$$

Untuk $n = 100$ jumlahnya sekitar 157.97, jadi 100! terdiri atas 158 digit. `long long` menampung paling banyak 19 digit (Subbab 1.10): faktorial terbesar yang muat adalah $20! = 2\,432\,902\,008\,176\,640\,000$, dan 21! sudah *overflow*. Jadi faktorial yang "kecil" itu bilangan yang sangat besar, dan dalam C++ kita harus mengalikan sendiri, seperti cara kita belajar mengalikan di sekolah.

Gagasannya. Simpan bilangan sebagai list digit desimal, mulai dari yang paling tidak signifikan: 120 disimpan sebagai `0, 2, 1`. Untuk mengalikannya dengan k, telusuri digit dari digit satuan ke atas, kalikan masing-masing dengan k, tambahkan *carry* (simpanan) dari posisi sebelumnya, simpan digit terakhir hasilnya di tempat, dan teruskan sisanya sebagai *carry* baru. Setelah digitnya habis, *carry* yang tersisa menjadi digit-digit baru di atas.

Mengapa cara ini tidak pernah membuat `int` *overflow*? Setiap langkah menghitung $d \cdot k + c$ dengan digit $d \le 9$ dan $k \le 100$. *Carry* yang diteruskan adalah $\lfloor (d \cdot k + c)/10 \rfloor$; jika $c \le 99$, maka

$d \cdot k + c \le 900 + 99 = 999$ dan *carry* berikutnya paling besar 99 lagi. Jadi dengan induksi setiap nilai tetap di bawah 1000, jauh di bawah `INT_MAX`.

Kode 2.9 `fctrl2.cpp`: FCTRL2 dengan perkalian cara sekolah pada vektor digit desimal.

```cpp
// SPOJ FCTRL2 (Small factorials): n! untuk n <= 100, digit demi digit.
#include <iostream>
#include <vector>

// digits[0] adalah digit satuan, digits[1] digit puluhan, dan seterusnya.
std::vector<int> factorial(int n) {
    std::vector<int> digits = {1};
    for (int k = 2; k <= n; k++) {
        int carry = 0;
        for (int& d : digits) {
            int value = d * k + carry;  // paling besar 9 * 100 + carry
            d = value % 10;
            carry = value / 10;
        }
        while (carry > 0) {
            digits.push_back(carry % 10);
            carry /= 10;
        }
    }
    return digits;
}

int main() {
    int cases;
    std::cin >> cases;
    while (cases--) {
        int n;
        std::cin >> n;
        std::vector<int> digits = factorial(n);
        for (auto it = digits.rbegin(); it != digits.rend(); ++it) {
            std::cout << *it;
        }
        std::cout << '\n';
    }
    return 0;
}
```

Menghitung $n!$ memerlukan $n - 1$ perkalian, masing-masing menelusuri paling banyak 158 digit, sehingga satu kasus memerlukan paling banyak sekitar $100 \times 158 \approx 1.6 \times 10^4$ langkah dan seluruh 100 kasus paling banyak 1.6×10^6: bukan apa-apa, menurut ukuran Bab 3. Digitnya dicetak dari yang paling signifikan ke bawah, dengan iterator terbalik.

Mengapa bilangan yang dicetak tidak pernah diawali nol? Vektornya dimulai dengan satu digit 1, dan digit baru hanya ditambahkan selama *carry* positif, berupa digit terakhirnya `carry % 10` diikuti digit-digit yang lebih tinggi dari *carry* yang sama. Digit terakhir yang ditambahkan adalah digit terdepan dari *carry* terakhir, yang bukan nol, karena gelung berhenti begitu *carry* menjadi nol. Jadi digit paling signifikan yang tersimpan tidak pernah 0, dan mencetak dari atas tidak menghasilkan nol di depan, sifat yang justru tidak dimiliki versi ADDREV yang salah.

Kode 2.10 `fctrl2.py`: FCTRL2 dalam Python, yang bilangan bulatnya tidak berukuran tetap.

```python
"""SPOJ FCTRL2 (Small factorials): n! untuk n <= 100; int Python tak berbatas."""
import sys


def factorial(n):
    result = 1
    for k in range(2, n + 1):
        result *= k
    return result


def main():
    data = sys.stdin.read().split()
    cases = int(data[0])
    print("\n".join(str(factorial(int(x))) for x in data[1:1 + cases]))


if __name__ == "__main__":
    main()
```

Dalam Python soal ini memang kecil: bilangan bulat tumbuh sesuai kebutuhan, dan gelung yang jelas adalah seluruh solusinya. (Pustaka standar juga punya `math.factorial`; gelungnya ditulis lengkap untuk menunjukkan bahwa tidak ada hal istimewa yang terjadi.) Inilah salah satu kasus ketika pilihan bahasa mengubah kesulitan soal, dan hal ini patut diingat ketika sebuah soal melibatkan bilangan di atas 64 bit: dalam kontes yang mengizinkan Python, soal bilangan besar bisa menjadi soal lima menit dalam Python dan soal tiga puluh menit dalam C++.

```
$ printf '4\n1\n2\n5\n3\n' | ./cpp/fctrl2
```

```
1
2
120
6
```

Program Python mencetak keempat baris yang sama. Untuk uji yang lebih kuat, masukan acak tidak diperlukan: hanya ada 100 kemungkinan nilai *n*, jadi kita dapat menguji *semuanya*. **Kode 2.11** menyuapkan bilangan 1 sampai 100 ke sebuah program dan membandingkan setiap baris dengan `math.factorial`.

Kode 2.11 `check_fctrl2.py`: memeriksa program FCTRL2 pada setiap masukan yang mungkin.

```python
"""check_fctrl2.py PROGRAM: uji PROGRAM untuk n = 1..100 terhadap math.factorial."""
import math
import subprocess
import sys


def main():
    program = sys.argv[1]
    command = [sys.executable, program] if program.endswith(".py") else [program]
    text = "100\n" + "\n".join(str(n) for n in range(1, 101)) + "\n"
    got = subprocess.run(command, input=text, capture_output=True, text=True).stdout
    lines = got.split()
    bad = [n for n in range(1, 101) if lines[n - 1] != str(math.factorial(n))]
    print("all 100 factorials correct" if not bad else f"wrong for n = {bad}")
    print("digits of 100!:", len(lines[99]))


if __name__ == "__main__":
    main()
```

```
$ python3 check_fctrl2.py ../cpp/fctrl2
$ python3 check_fctrl2.py ../python/fctrl2.py
```

```
all 100 factorials correct
digits of 100!: 158
all 100 factorials correct
digits of 100!: 158
```

> **PERINGATAN:** Bilangan bulat Python tidak berbatas, tetapi sejak Python 3.11 konversi bilangan bulat ke string desimal, seperti yang dilakukan `str()` dan `print()`, menolak bilangan yang lebih dari 4300 digit dan melempar `ValueError`, sebuah perlindungan terhadap serangan *denial-of-service* pada server web [5]. 100! dengan 158 digitnya jauh di bawah batas itu, tetapi 2030! terdiri atas 5835 digit, dan *brute force* yang mencetaknya gagal. Pemanggilan `sys.set_int_max_str_digits(0)` di awal program menghapus batas itu; Bab 3 memerlukannya di salah satu *brute force*-nya.

Jika seluruh ruang masukan kecil, ujilah semuanya; itulah uji terkuat yang ada. Dan uji kasus terbesar, di sini 100! yang 158 digit, mengonfirmasi Persamaan (2.2) secara independen.

2.7 ONP: Membaca Aturan yang Tak Lazim

Soalnya (SPOJ ONP, "Transform the Expression", `https://www.spoj.com/problems/ONP/`) [6]. Masukan diawali banyaknya ekspresi, paling banyak 100; setiap ekspresi ditulis dalam satu baris, panjangnya paling banyak 400 karakter, dan memakai huruf kecil `a` sampai `z` sebagai operan, lima operator biner `+ - * / ^`, dan tanda kurung biasa. Ubah setiap ekspresi menjadi *Reverse Polish Notation* (RPN, notasi Polandia terbalik), yang menuliskan setiap operator sesudah kedua operannya, lalu cetak. Pernyataan soal memberikan prioritas operator dari yang terendah ke yang tertinggi dalam urutan `+`, `-`, `*`, `/`, `^`, dan menjanjikan bahwa setiap ekspresi hanya punya satu bentuk RPN, sehingga rantai seperti `a*b*c`, yang pengelompokannya ambigu, tidak muncul.

Membaca aturan yang bukan aturan biasa. Setiap pembaca tahu bahwa perkalian mengikat lebih kuat daripada penjumlahan, dan setiap pembaca tahu bahwa `+` dan `-` punya prioritas yang *sama*. Pernyataan soal mengatakan lain: di sini `-` mengikat lebih kuat daripada `+`, dan `/` lebih kuat daripada `*`. Pada contoh soal, yang seluruhnya berkurung, perbedaan itu tidak pernah tampak, dan program dengan prioritas sekolah lolos dari contoh tetapi gagal pada uji tersembunyi. Ambil ekspresi `a+b*c-d`. Dengan prioritas ONP, `*` mengikat lebih dulu, lalu `-`, lalu `+`: $a + ((b \cdot c) - d)$, yang RPN-nya `abc*d-+`. Dengan prioritas sekolah hasilnya $(a + b \cdot c) - d$, dengan RPN `abc*+d-`. Keduanya "benar"; hanya satu yang benar *di sini*. Inilah pertanyaan 1 daftar periksa dalam bentuk paling murni: bacalah aturan yang tertulis, bukan aturan yang Anda ketahui.

Apa itu RPN. Dalam RPN, ekspresi `a+b` ditulis `ab+`: operan lebih dulu, baru operatornya. Tanda kurung tidak pernah diperlukan, karena urutan simbol sudah menetapkan urutan operasi. Contoh soal memberikan tiga ekspresi: `(a+(b*c))` menjadi `abc*+`, `((a+b)*(z+x))` menjadi `ab+zx+*`, dan `((a+t)*((b+(a+c))^(c+d)))` menjadi `at+bac++cd+^*`.

Gagasannya: tumpukan operator yang menunggu. Bacalah ekspresi dari kiri ke kanan. Huruf langsung masuk ke keluaran, karena dalam RPN operan mempertahankan urutannya. Operator belum dapat ditulis, karena operan kanannya belum terbaca; ia menunggu di sebuah *tumpukan* (*stack*), tumpukan tempat benda yang terakhir diletakkan menjadi yang pertama diambil. Ketika operator baru tiba, setiap operator yang menunggu dengan prioritas setidaknya setinggi operator baru itu harus diselesaikan lebih dulu, sehingga ia diambil dari tumpukan dan ditulis; lalu operator baru menunggu. Kurung buka menunggu di tumpukan sebagai sekat; kurung tutup mengambil dan menulis operator sampai kurung buka pasangannya. Di akhir, semua operator yang tersisa ditulis. Prosedur ini adalah *shunting-yard algorithm* dari Dijkstra, dinamai menurut langsiran kereta api tempat gerbong menunggu di rel samping [7].

Mengapa cara ini benar? Dua fakta menopang argumennya. Pertama, huruf-huruf keluar dalam urutan aslinya, dan memang harus begitu dalam RPN, karena RPN tidak pernah menukar urutan operan. Kedua, sebuah operator ditulis tepat ketika kedua operannya sudah lengkap di keluaran. Operan kirinya sudah lengkap ketika operator itu tiba. Operan kanannya lengkap begitu algoritma bertemu simbol yang tidak mungkin menjadi bagiannya: operator dengan prioritas lebih rendah atau sama, yang mengakhiri operan kanan setiap operator yang mengikat setidaknya sama kuat; kurung tutup, yang mengakhiri semua isi kurung; atau akhir masukan, yang mengakhiri semuanya. Itulah tepat saat-saat algoritma mengambil dari tumpukan. Di antara dua kurung, operator-operator di tumpukan selalu punya prioritas yang naik tegas dari bawah ke atas, sehingga mengambil dari puncak menulis operator yang paling kuat mengikat, yang operannya paling dalam, lebih dulu, dan itulah urutan yang diwajibkan RPN. Janji pernyataan soal bahwa rantai seperti `a*b*c` tidak muncul berarti pilihan antara "ambil jika prioritasnya sama" dan "jangan ambil" tidak pernah mengubah hasil; kita mengambil, aturan yang lazim untuk operator yang berasosiasi ke kiri.

Penelusuran `a+b*c-d` memperlihatkan tumpukan bekerja; tumpukan ditulis dengan puncaknya di sebelah kanan.

Tabel 2.2 *Shunting-yard algorithm* pada `a+b*c-d` dengan prioritas ONP.

Dibaca	Tindakan	Keluaran sejauh ini	Tumpukan
`a`	tulis	`a`	
`+`	tumpukan kosong: masukkan	`a`	`+`
`b`	tulis	`ab`	`+`
`*`	`+` berprioritas lebih rendah: masukkan	`ab`	`+ *`
`c`	tulis	`abc`	`+ *`
`-`	ambil `*` (prioritas 3 ≥ 2), berhenti di `+` (1 < 2), masukkan	`abc*`	`+ -`
`d`	tulis	`abc*d`	`+ -`
akhir	ambil semuanya	`abc*d-+`	

Kode 2.12 `onp.cpp`: ONP dengan *shunting-yard algorithm*; prioritasnya adalah prioritas dari pernyataan soal.

```cpp
// SPOJ ONP (Transform the Expression): infiks ke Reverse Polish Notation.
#include <iostream>
#include <string>

int priority(char op) {
    switch (op) {
        case '+': return 1;
        case '-': return 2;
        case '*': return 3;
        case '/': return 4;
        case '^': return 5;
    }
    return 0;  // '(' tetap di tumpukan sampai ')' pasangannya
}

std::string toRpn(const std::string& expr) {
    std::string out, ops;  // ops dipakai sebagai tumpukan operator
    for (char ch : expr) {
        if (ch >= 'a' && ch <= 'z') {
            out += ch;
        } else if (ch == '(') {
            ops += ch;
        } else if (ch == ')') {
            while (!ops.empty() && ops.back() != '(') {
                out += ops.back();
                ops.pop_back();
            }
            if (!ops.empty()) ops.pop_back();  // buang '('
        } else {
            while (!ops.empty() && priority(ops.back()) >= priority(ch)) {
                out += ops.back();
                ops.pop_back();
            }
            ops += ch;
        }
    }
    while (!ops.empty()) {
        out += ops.back();
        ops.pop_back();
    }
    return out;
}
```

Kode 2.12 (Lanjutan) *onp.cpp: ONP dengan shunting-yard algorithm; prioritasnya adalah prioritas dari pernyataan soal.*

```cpp
int main() {
    int cases;
    std::cin >> cases;
    while (cases--) {
        std::string expr;
        std::cin >> expr;
        std::cout << toRpn(expr) << '\n';
    }
    return 0;
}
```

Kode 2.12 adalah algoritmanya dalam C++, dan **Kode 2.13** dalam Python. Fungsi `priority` mengodekan daftar dari pernyataan soal, dengan 0 untuk kurung buka, sehingga gelung pengambilan untuk sebuah operator berhenti di kurung. Tumpukannya adalah `std::string` yang hanya dipakai di ujungnya: `back()` melihat puncak, `+=` memasukkan, dan `pop_back()` mengambil. Setiap karakter dimasukkan dan diambil paling banyak sekali, sehingga konversinya linear terhadap panjang ekspresi.

Kode 2.13 `onp.py`: algoritma yang sama dalam Python, dengan list sebagai tumpukan.

```python
"""SPOJ ONP (Transform the Expression): infiks ke Reverse Polish Notation."""
import sys

PRIORITY = {"+": 1, "-": 2, "*": 3, "/": 4, "^": 5, "(": 0}


def to_rpn(expr):
    out, ops = [], []  # ops dipakai sebagai tumpukan operator
    for ch in expr:
        if "a" <= ch <= "z":
            out.append(ch)
        elif ch == "(":
            ops.append(ch)
        elif ch == ")":
            while ops and ops[-1] != "(":
                out.append(ops.pop())
            if ops:
                ops.pop()  # buang "("
        else:
            while ops and PRIORITY[ops[-1]] >= PRIORITY[ch]:
                out.append(ops.pop())
            ops.append(ch)
    while ops:
        out.append(ops.pop())
    return "".join(out)


def main():
    data = sys.stdin.read().split()
    cases = int(data[0])
    print("\n".join(to_rpn(e) for e in data[1:1 + cases]))


if __name__ == "__main__":
    main()
```

```
$ printf '3\n(a+(b*c))\n((a+b)*(z+x))\n((a+t)*((b+(a+c))^(c+d)))\n' | ./cpp/onp
$ printf '1\na+b*c-d\n' | ./cpp/onp
```

```
abc*+
ab+zx+*
at+bac++cd+^*
abc*d-+
```

Program Python mencetak baris-baris yang sama. Baris keempat adalah baris yang tidak diuji oleh contoh soal, dan sesuai dengan penelusuran tangan di **Tabel 2.2**.

Mengujinya. *Brute force* untuk ONP harus *berbeda* dari metode tumpukan. **Kode 2.14** memakai *recursive descent* (penguraian turun rekursif): ekspresi pada tingkat prioritas ℓ adalah deretan ekspresi pada tingkat $\ell + 1$ yang digabung oleh operator tingkat ℓ, dan pada tingkat tertinggi sebuah ekspresi berupa huruf atau ekspresi dalam kurung. Tata bahasanya ditulis langsung, satu tingkat per operator menurut urutan dalam pernyataan soal, sehingga kebenarannya dapat diperiksa dengan membacanya.

Kode 2.14 `brute_onp.py`: ONP dengan *recursive descent*, satu tingkat tata bahasa per prioritas.

```python
"""brute_onp.py: ONP dengan recursive descent, satu aturan per tingkat prioritas."""
import sys

LEVELS = "+-*/^"


def parse(expr, pos, level):
    """Urai operator berprioritas level ke atas; hasil (rpn, posisi berikutnya)."""
    if level == len(LEVELS):
        if expr[pos] == "(":
            rpn, pos = parse(expr, pos + 1, 0)
            return rpn, pos + 1  # lewati ")"
        return expr[pos], pos + 1
    left, pos = parse(expr, pos, level + 1)
    while pos < len(expr) and expr[pos] == LEVELS[level]:
        right, pos = parse(expr, pos + 1, level + 1)
        left = left + right + LEVELS[level]
    return left, pos


def main():
    data = sys.stdin.read().split()
    for expr in data[1:1 + int(data[0])]:
        print(parse(expr, 0, 0)[0])


if __name__ == "__main__":
    main()
```

Generatornya, **Kode 2.15**, membangun pohon ekspresi acak dan mencetaknya dengan kurung tepat di tempat yang diwajibkan prioritas (di sekitar anak yang operatornya tidak mengikat lebih kuat daripada operator induknya), dan kadang dengan kurung tambahan. Generator ini tidak pernah menghasilkan rantai berprioritas sama tanpa kurung, yang dikecualikan pernyataan soal.

Kode 2.15 `gen_onp.py`: ekspresi ONP acak dengan kurung minimal dan kurung tambahan.

```python
"""gen_onp.py SEED: ekspresi ONP acak, dengan kurung di tempat yang diperlukan."""
import random
import sys

OPS = "+-*/^"
PRIORITY = {op: i for i, op in enumerate(OPS, start=1)}


def build(rng, depth):
    """Kembalikan (teks, prioritas operator puncaknya; 9 untuk satu huruf)."""
    if depth == 0 or rng.random() < 0.25:
        return rng.choice("abcdefghijklmnopqrstuvwxyz"), 9
    op = rng.choice(OPS)
    parts = []
    for _ in range(2):
        text, prio = build(rng, depth - 1)
        # kurung anak yang operatornya tidak mengikat lebih kuat, atau secara acak
        if prio <= PRIORITY[op] or (prio < 9 and rng.random() < 0.3):
            text = "(" + text + ")"
        parts.append(text)
    return parts[0] + op + parts[1], PRIORITY[op]


def main():
    rng = random.Random(int(sys.argv[1]))
    cases = rng.randint(1, 4)
    print(cases)
    for _ in range(cases):
        text, _ = build(rng, rng.randint(1, 5))
        print(text if rng.random() < 0.5 else "(" + text + ")")


if __name__ == "__main__":
    main()
```

```
$ python3 gen_onp.py 3
```

```
2
((((g-w)/p^m)+(v+y)^y)/y)*((o^e/e)/(y/u*s/l))/(((v^w)+(s^s))-(c+p)^(c/l))
((m+w)+(i^q))/a
```

```
$ python3 stress.py gen_onp.py brute_onp.py ../cpp/onp.cpp
$ python3 stress.py gen_onp.py brute_onp.py ../python/onp.py 200
```

```
500 rounds: all answers equal
200 rounds: all answers equal
```

Ekspresi yang dihasilkan jauh kurang ramah dibanding contoh soal: kurung dan prioritas bercampur, dan bergantung pada `-` yang mengikat lebih kuat daripada `+`. Kedua program sesuai dengan *brute force* pada setiap ekspresi itu.

2.8 Memakainya Sungguh-sungguh

Tulis penceritaan ulang di buku catatan Anda. Untuk setiap soal, sebelum menulis kode, tulislah penceritaan ulangnya, jawaban atas **Tabel 2.1**, dan perhitungan tangan atas contoh soal. Semua itu memakan tiga menit. Setelah beberapa minggu Anda akan mengerjakan sebagian besar di kepala, tetapi untuk soal yang sulit, atau larut malam, tuliskan lagi.

Simpan perangkat *stress*. Ketiga berkas `stress.py`, sebuah generator, dan sebuah *brute force* dapat dipakai ulang untuk soal apa pun yang jawabannya tunggal. Salin `stress.py` ke folder setiap soal baru, dan tulislah generator serta *brute force*-nya sebelum, atau tepat sesudah, solusi yang sebenarnya. *Brute force* untuk sebuah soal sering hanya beberapa baris: coba semua

kemungkinan, hitung dari definisi, atau pakai pustaka bilangan besar. Ketika solusi yang sebenarnya mendapat WA di juri, Anda sudah siap.

Buat generatornya licik. Masukan kecil menemukan lebih banyak *bug* daripada masukan besar, karena *bug* yang tampak pada masukan mungil mudah dipahami, dan masukan kecil lebih sering mengenai kasus tepi. Condongkan generator ke tepi-tepi yang diizinkan batasan: n terkecil, nilai kembar, nol, nilai terbesar.

Pernyataan soal dalam bahasa asing. Hampir setiap pernyataan soal SPOJ ditulis dalam bahasa Inggris, sebagian dalam bahasa Polandia atau bahasa lain, dan banyak pembaca buku ini membaca bahasa Inggris sebagai bahasa kedua. Hal itu bukan kerugian jika dimanfaatkan dengan baik: pembaca yang tidak dapat membaca sekilas terpaksa membaca setiap kalimat, persis seperti yang diminta daftar periksa. Terjemahkan kalimat tugas, format masukan, format keluaran, dan batasan kata demi kata ke dalam bahasa Anda sendiri, secara tertulis, sebelum mulai; ceritanya boleh dibaca sekilas. Carilah arti setiap kata yang tidak Anda yakini, dan berilah perhatian khusus pada kata-kata kecil yang membawa aturan: *at most* (paling banyak), *at least* (paling sedikit), *exactly* (tepat), *distinct* (berbeda satu sama lain), *non-negative* (tak negatif), *inclusive* (termasuk kedua ujungnya), *respectively* (berturut-turut sesuai urutan), *consecutive* (berurutan), *separated by* (dipisahkan oleh). "Non-negative" mencakup nol; "positive" tidak. "Between 1 and n inclusive" mencakup kedua ujungnya. "The numbers are distinct" berarti Anda boleh mengandalkan tidak adanya nilai kembar. Terjemahan mesin untuk seluruh pernyataan soal berguna sebagai gambaran awal, tetapi sering justru menghilangkan kata-kata kecil ini, jadi periksalah aturannya terhadap teks aslinya.

Berikut kartu penceritaan ulang untuk ADDREV dari Subbab 2.5, jenis kartu yang sebaiknya Anda tulis untuk setiap soal:

```
ADDREV  banyaknya di depan (N kasus)  1 baris per kasus: a b (terbalik, positif)
tugas:  cetak rev(rev(a) + rev(b)), rev membuang nol di depan
tepi:   jumlah berakhir 0 (mis. 305 794 -> 1); satu digit; 1 digit + banyak digit
ukuran: bilangan kecil, jumlah bisa satu digit lebih panjang -> long long
keluaran: satu bilangan per baris, tanpa nol di depan
contoh: 24 1 -> 34; 4358 754 -> 1998; 305 794 -> 1 (diperiksa dengan tangan)
```

Membaca spesifikasi di tempat kerja. Daftar periksa ini berlaku langsung untuk dokumen yang dibaca seorang pengembang: deskripsi antarmuka layanan pembayaran, format berkas yang ditetapkan kantor pajak, pertukaran data yang disepakati dengan sistem rumah sakit. Masing-masing punya format masukan, format keluaran, batas, kasus khusus yang disebut sekali, dan contoh yang hanya mencakup jalur mudah. Kebiasaan menulis penceritaan ulang dan bertanya "bagaimana kalau nol, bagaimana kalau maksimum, bagaimana kalau ada yang kembar?" adalah hal yang tepat mencegah *bug* mahal dalam pekerjaan seperti itu.

2.9 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

- **Membaca sekilas format keluaran.** Baris kosong yang hilang, spasi berlebih, `Case #1:` lawan `Case 1:`: semuanya WA. Bacalah format keluaran kata demi kata, dan bandingkan contoh keluaran Anda karakter demi karakter, atau dengan `minijudge.py`.
- **Mengasumsikan aturan sekolah.** Dalam ONP, `-` mengikat lebih kuat daripada `+`. Jika pernyataan soal mendefinisikan sesuatu, bahkan sesuatu yang Anda "tahu", pakailah definisinya.
- **Gelung dengan `eof()`.** `while (!std::cin.eof())` berjalan sekali terlalu banyak. Letakkan pembacaan di kondisi gelung.

- **Membaca baris padahal token sudah cukup.** Baris kosong, pindah baris akhir yang hilang, dan akhir baris gaya Windows mematahkan pembacaan per baris. Bacalah token kecuali struktur baris penting.
- **Tidak memeriksa ukuran jawaban.** 100! terdiri atas 158 digit. Perkirakan ukuran nilai antara dengan penalaran ala Persamaan (2.2) sebelum memilih tipe.
- **Mempertahankan nol di depan.** Bilangan terbalik, sebuah jumlah, sebuah hasil kali: jika pernyataan soal berkata "tanpa nol di depan", buanglah, dan biarkan generator menghasilkan masukan yang memunculkannya.
- ***Brute force* yang menyalin solusi cepat.** Jika keduanya memakai gagasan yang sama, *stress test* tidak dapat menemukan kesalahan pada gagasan itu. Tulislah *brute force* dari definisi.
- **Kata-kata kecil yang hilang dalam terjemahan.** *Non-negative* bukan *positive*, *at most* bukan *exactly*, dan *distinct* adalah janji yang boleh Anda andalkan. Terjemahkan aturan-aturannya kata demi kata, dan periksa terjemahan mesin terhadap teks aslinya.
- **Generator yang melanggar batasan.** Generator seperti itu melaporkan perbedaan pada masukan yang tidak pernah terjadi, dan Anda membuang waktu untuk hal yang bukan *bug*. Setiap masukan yang dihasilkan harus sah.

2.10 Ringkasan Bab

- Pernyataan soal punya enam zona: kepala dengan batas, cerita dan tugas, format masukan, format keluaran, batasan, dan contoh (**Gambar 2.1**). Batasan menentukan algoritma sekaligus kasus tepi; contoh hanya memperlihatkan format.
- Daftar periksa delapan pertanyaan (**Tabel 2.1**) dan penceritaan ulang yang tertulis menangkap sebagian besar salah baca sebelum kode apa pun ada.
- Masukan juri datang dalam tiga bentuk: banyaknya di depan, penanda, dan sampai akhir berkas. Bentuk terakhir dibaca dengan meletakkan pembacaan di kondisi gelung dalam C++ (**Kode 2.1**) dan dengan `sys.stdin.read().split()` dalam Python (**Kode 2.2**); membaca token alih-alih baris tahan terhadap baris kosong dan akhir baris.
- ADDREV menghitung $\mathrm{rev}(\mathrm{rev}(a) + \mathrm{rev}(b))$ (Persamaan (2.1)); jebakannya adalah nol di depan pada jumlah yang dibalik. *Stress test* (**Kode 2.6**) dengan generator yang kaya nol dan *brute force* dari definisi membongkar versi yang salah di putaran pertamanya.
- FCTRL2 meminta bilangan sampai 158 digit (Persamaan (2.2)), jauh melampaui `long long`; C++ memerlukan perkalian cara sekolah pada digit (**Kode 2.9**), Python tidak memerlukan apa pun yang istimewa (**Kode 2.10**). Dengan hanya 100 masukan yang mungkin, menguji semuanya adalah uji terkuat.
- ONP mendefinisikan prioritasnya sendiri, dengan `-` di atas `+`; *shunting-yard algorithm* (**Tabel 2.2**) mengubah ke RPN dalam waktu linear, dan *brute force recursive descent* (**Kode 2.14**) mengonfirmasinya pada ekspresi yang dihasilkan generator.

2.11 Soal Latihan

1. Tulislah penceritaan ulang dan jawablah kedelapan pertanyaan **Tabel 2.1** untuk SPOJ TEST dari Bab 1.
2. Hitung dengan tangan, memakai Persamaan (2.1), jawaban ADDREV untuk pasangan (1,1), (90,10), dan (5,5). Pada pasangan mana `addrev_bug.cpp` salah?
3. Pakai Persamaan (2.2) untuk menentukan banyaknya digit 25! tanpa menghitungnya, lalu periksa hasilnya dengan program-program di Subbab 2.6.

4. Ubah `a-b+c^d/e*f` menjadi RPN dengan tangan memakai prioritas ONP, dengan penelusuran seperti **Tabel 2.2**, lalu periksa jawaban Anda dengan `onp.py`.
5. Sebuah soal berbunyi: "Masukan memuat beberapa kasus uji. Setiap kasus uji berupa satu baris berisi dua bilangan bulat n dan k. Masukan diakhiri baris berisi `0 0`, yang tidak boleh diproses." Tulislah gelung masukannya dalam C++ dan dalam Python.
6. Ubah `gen_addrev.py` sehingga menghasilkan digit secara seragam dari `0123456789`, lalu jalankan lagi *stress test* pada `addrev_bug.cpp`. Pada putaran ke berapa program itu sekarang gagal? Apa yang disampaikan hal itu tentang generator?

2.12 Penyelesaian Soal Latihan

1. *Penceritaan ulang:* masukannya adalah deretan bilangan bulat paling banyak dua digit, satu per baris; cetak bilangan-bilangan itu satu per baris sampai 42 yang pertama, tanpa mencetak 42 maupun apa pun sesudahnya. (1) Cetak ulang bilangan sebelum 42 yang pertama. (2) Tanpa banyaknya; masukan berakhir pada penanda 42 atau di akhir berkas, dan bilangan lain boleh menyusul 42. (3) Banyaknya baris tidak dibatasi, sehingga program harus linear dan tidak boleh menyimpan bilangan dalam larik berukuran tetap. (4) Bilangan pertama bisa 42, sehingga keluarannya bisa kosong; bilangannya bisa 0. (5) Paling banyak dua digit: `int` lebih dari cukup. (6) Satu bilangan per baris. (7) Jawabannya tunggal. (8) Contoh 1, 2, 88, 42, 99 menghasilkan 1, 2, 88.

2. (1,1): $\text{rev}(1) + \text{rev}(1) = 2$, dibalik tetap 2. (90,10): $\text{rev}(90) = 9$ dan $\text{rev}(10) = 1$, jadi jumlahnya 10, dan $\text{rev}(10) = 1$. (5,5): jumlahnya 10 lagi, jawabannya 1. `addrev_bug.cpp` mencetak `2`, `01`, dan `01`: salah pada dua pasangan terakhir, yang jumlahnya berakhir dengan nol. (Tepatnya, 90 dan 10 tidak dapat muncul sebagai masukan, karena bilangan terbalik tidak pernah berakhir dengan nol; kasus kedua ditampilkan hanya untuk aritmetikanya.)

3. $\sum_{k=1}^{25} \log_{10} k = \log_{10} 25! \approx 25.19$, jadi 25! terdiri atas $\lfloor 25.19 \rfloor + 1 = 26$ digit. Program-program itu mencetak `15511210043330985984000000`, yang terdiri atas 26 digit.

4. Penelusurannya: `a` keluar; `-` masuk; `b` keluar; `+` tiba: `-` berprioritas 2 ≥ 1, ambil, keluaran `ab-`, masukkan `+`; `c` keluar; `^` masuk; `d` keluar; `/` tiba: ambil `^` (5 ≥ 4), berhenti di `+`, masukkan `/`; `e` keluar; `*` tiba: ambil `/` (4 ≥ 3), berhenti di `+`, masukkan `*`; `f` keluar; akhir: ambil `*`, lalu `+`. Hasilnya `ab-cd^e/f*+`, dan `onp.py` mencetak hal yang sama.

5. Dalam C++, pembacaan dan uji penanda diletakkan di kondisi gelung:

```
long long n, k;
while (std::cin >> n >> k && !(n == 0 && k == 0)) {
    // proses satu kasus uji
}
```

Dalam Python, dengan semua token dibaca sekaligus:

```
data = sys.stdin.read().split()
i = 0
while i + 1 < len(data):
    n, k = int(data[i]), int(data[i + 1])
    i += 2
    if n == 0 and k == 0:
        break
    # proses satu kasus uji
```

Kedua gelung juga berhenti di akhir berkas, berjaga-jaga jika sebuah berkas tidak memuat `0 0` penutupnya.

6. Dengan digit seragam, sebuah bilangan berakhir dengan nol dengan peluang 1/10 dan sebuah jumlah hanya sesekali berakhir dengan nol, sehingga *bug* memerlukan lebih banyak putaran untuk tampak: dengan digit seragam, *stress test* baru melaporkan perbedaan pertama di putaran 5, bukan putaran 1. Pada soal ini kedua generator menemukan *bug* dengan cepat, tetapi pada soal yang kasus khususnya lebih jarang, generator seragam bisa berjalan ribuan putaran tanpa pernah mengenainya.

Daftar Pustaka

[1] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Singapore: Lulu, 2020.

[2] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

[3] SPOJ, "ADDREV — Adding Reversed Numbers." [Online]. Available: https://www.spoj.com/problems/ADDREV/ (accessed Sep. 2026).

[4] SPOJ, "FCTRL2 — Small factorials." [Online]. Available: https://www.spoj.com/problems/FCTRL2/ (accessed Sep. 2026).

[5] Python Software Foundation, "Integer string conversion length limitation," *Python 3.14 Documentation: Built-in Types*. [Online]. Available: https://docs.python.org/3/library/stdtypes.html#int-max-str-digits (accessed Sep. 2026).

[6] SPOJ, "ONP — Transform the Expression." [Online]. Available: https://www.spoj.com/problems/ONP/ (accessed Sep. 2026).

[7] E. W. Dijkstra, "Algol 60 translation: An Algol 60 translator for the X1 and making a translator for Algol 60," Mathematisch Centrum, Amsterdam, Report MR 35/61, 1961.

Bab 3

Memilih Algoritma Sesuai Batasan

Sebuah program bisa sepenuhnya benar tetapi tetap ditolak, karena terlalu lambat. Juri memberinya satu atau dua detik, dan metode benar yang memerlukan satu jam, bagi juri, adalah metode yang salah. Kabar baiknya, biasanya Anda dapat mengetahui lebih dulu, sebelum menulis satu baris pun, apakah sebuah metode cukup cepat: batasan soal dan sedikit aritmetika sudah cukup. Bab ini mengajarkan aritmetika itu. Di akhir bab Anda akan mampu mengukur berapa langkah sederhana yang dibuat komputer Anda per detik dalam C++ dan dalam Python; mengubah batas waktu dan masukan terbesar menjadi anggaran langkah; memperkirakan langkah yang diperlukan sebuah algoritma dengan menghitung gelungnya, termasuk gelung atas kasus uji; membaca dari sebuah tabel kelas kompleksitas mana yang cocok untuk ukuran masukan mana; serta menyelesaikan tiga soal SPOJ sungguhan yang solusi gamblangnya terlalu lambat, FCTRL, LASTDIG, dan PRIME1, dengan rumus pencacahan, pengkuadratan berulang, dan saringan bersegmen, masing-masing diperiksa terhadap *brute force* dan, untuk FCTRL, disaksikan ketika versi lambatnya menabrak batas waktu juri buatan kita sendiri.

3.1 Antrean di Tempat Fotokopi

Di dekat setiap kampus di Indonesia ada tempat fotokopi, dan setiap awal semester ada antrean di depannya. Seorang mahasiswa meminta satu salinan formulir dua halaman: petugas meletakkan setiap halaman di kaca dan menekan tombol, dan dalam semenit selesai. Mahasiswa berikutnya meminta empat puluh salinan diktat setebal dua ratus halaman. Meletakkan halaman di kaca satu per satu akan menghabiskan seluruh sore, jadi petugas memasukkan seluruh tumpukan ke pengumpan otomatis, mengatur penghitung ke empat puluh, lalu melayani pelanggan berikutnya. Tidak ada yang mengajari petugas itu teori kompleksitas. Ia hanya tahu bahwa metode terbaik untuk dua halaman tidak mungkin dipakai untuk delapan ribu halaman, dan ia memilih metodenya *setelah* mendengar ukuran pekerjaannya.

Pemrograman kompetitif meminta pertimbangan yang sama, dan batasan soal adalah ukuran pekerjaannya. Tim Rina mempelajari hal ini pada soal FCTRL, yang menanyakan berapa banyak nol di ujung $n!$. Program pertama mereka jelas benar: program itu menelusuri semua bilangan dari 1 sampai n dan menghitung faktor lima. Program itu memberi jawaban benar pada contoh soal dalam sekejap. Di juri program itu mendapat *time limit exceeded*, karena n bisa satu miliar dan banyaknya bisa seratus ribu. Subbab 3.4 memperlihatkan rumus dua baris yang menggantikan gelung itu, dan menjalankan kedua program melalui juri kita dari Bab 1, dengan selisih yang bukan beberapa persen melainkan lebih dari sejuta kali.

Siapa yang diuntungkan dengan belajar memperkirakan waktu berjalan? Setiap orang yang menulis program yang bertemu data sungguhan. Laporan yang lancar pada basis data uji berisi seratus pelanggan dan membeku pada basis data sungguhan berisi dua juta; pencarian yang baik-baik saja saat demo dan terlalu lambat saat dipakai; skrip yang memproses penjualan satu hari dalam satu detik dan penjualan satu tahun dalam enam jam. Aritmetika bab ini meramalkan semua itu sebelum terjadi, yang jauh lebih murah daripada menemukannya belakangan.

3.2 Dari Batas Waktu ke Anggaran Langkah

Bab 1 memberikan perkiraan dasarnya di Subbab 1.3: waktu berjalan sama dengan banyaknya langkah sederhana dibagi langkah per detik. Jika dibalik, perkiraan itu memberikan *anggaran*: banyaknya langkah yang muat dalam batas waktu.

$$\text{anggaran} = \text{batas waktu} \times \text{langkah per detik} \tag{3.1}$$

Batas waktu tercetak di halaman soal. Langkah per detik harus diukur, karena bergantung pada mesin, bahasa, dan apa yang dianggap satu "langkah". **Kode 3.1** mengukurnya dalam C++: program ini menjalankan gelung sebanyak 2×10^8 putaran, masing-masing mengerjakan beberapa pergeseran bit, *exclusive-or*, dan satu penjumlahan, jenis pekerjaan murah yang dikerjakan kebanyakan gelung terdalam, dan mengukur waktu pengurutan sepuluh juta bilangan.

Kode 3.1 `ops_budget.cpp`: mengukur langkah gelung sederhana per detik dan waktu mengurutkan 10^7 bilangan dalam C++.

```cpp
// ops_budget.cpp: berapa langkah sederhana yang dibuat komputer ini per detik?
#include <algorithm>
#include <chrono>
#include <cstdio>
#include <vector>

double secondsSince(std::chrono::steady_clock::time_point start) {
    std::chrono::duration<double> d = std::chrono::steady_clock::now() - start;
    return d.count();
}

int main() {
    const long long steps = 200000000;  // 2 * 10^8
    auto start = std::chrono::steady_clock::now();
    unsigned long long x = 88172645463325252ULL, check = 0;
    for (long long i = 0; i < steps; i++) {
        x ^= x << 13;  // beberapa operasi murah yang tidak dapat dilewati kompiler
        x ^= x >> 7;
        x ^= x << 17;
        check += x & 1023;
    }
    double t = secondsSince(start);
    std::printf("simple loop: %lld steps in %.2f s = %.1e steps/s (check %llu)\n",
                steps, t, steps / t, check % 1000);

    const int n = 10000000;  // urutkan 10^7 bilangan: sekitar n log2 n = 2.3 * 10^8
    std::vector<unsigned> a(n);
    for (int i = 0; i < n; i++) {
        x ^= x << 13;
        x ^= x >> 7;
        x ^= x << 17;
        a[i] = static_cast<unsigned>(x);
    }
    start = std::chrono::steady_clock::now();
    std::sort(a.begin(), a.end());
    std::printf("sort of %d numbers: %.2f s\n", n, secondsSince(start));
    return 0;
}
```

Nilai `check` dicetak agar kompiler tidak dapat memutuskan bahwa gelung itu tidak berdampak apa pun lalu menghapusnya, hal yang diperbolehkan bagi kompiler yang mengoptimasi. Kelas `std::chrono::steady_clock` adalah jam yang tidak pernah melompat, jam yang tepat untuk mengukur durasi. **Kode 3.2** melakukan hal yang sama dalam Python, dengan putaran sepuluh kali

lebih sedikit, karena Python jauh lebih lambat per langkah, dan satu juta bilangan untuk diurutkan.

Kode 3.2 `ops_budget.py`: pengukuran yang sama dalam Python.

```python
"""ops_budget.py: berapa langkah sederhana yang dibuat Python per detik?"""
import random
import time


def main():
    steps = 10_000_000  # 10^7: per langkah, Python jauh lebih lambat daripada C++
    start = time.perf_counter()
    x, check = 88172645463325252, 0
    mask = (1 << 64) - 1
    for _ in range(steps):
        x ^= (x << 13) & mask
        x ^= x >> 7
        x ^= (x << 17) & mask
        check += x & 1023
    t = time.perf_counter() - start
    print(f"simple loop: {steps} steps in {t:.2f} s = {steps / t:.1e} steps/s"
          f" (check {check % 1000})")

    n = 1_000_000
    a = [random.getrandbits(32) for _ in range(n)]
    start = time.perf_counter()
    a.sort()
    print(f"sort of {n} numbers: {time.perf_counter() - start:.2f} s")


if __name__ == "__main__":
    main()
```

Pada mesin yang dipakai untuk buku ini, sebuah server Linux bersama yang saat itu juga sibuk dengan pekerjaan lain:

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o cpp/ops_budget cpp/ops_budget.cpp
$ ./cpp/ops_budget
$ python3 python/ops_budget.py
```

```
simple loop: 200000000 steps in 0.37 s = 5.3e+08 steps/s (check 719)
sort of 10000000 numbers: 0.83 s
simple loop: 10000000 steps in 2.33 s = 4.3e+06 steps/s (check 739)
sort of 1000000 numbers: 0.30 s
```

Dua baris pertama berasal dari C++, dua baris terakhir dari Python. Angka Anda akan berbeda, mungkin dua kali lipat ke arah mana pun, tetapi polanya tidak. C++ terkompilasi mencapai sekitar 5×10^8 putaran gelung per detik, masing-masing empat atau lima operasi mesin. Python mencapai sekitar 4×10^6 putaran yang sama, lebih dari seratus kali lebih sedikit: setiap operasi Python melewati interpreter, yang memeriksa tipe nilai, mengalokasikan objek bilangan bulat baru, dan memeriksa apakah perlu beralih ke bilangan besar. Sebaliknya, pengurutan cepat di kedua bahasa, karena `sort` milik Python ditulis dalam C: sejuta bilangan dalam 0,3 detik. Pelajaran bagi pemrogram Python adalah membiarkan fungsi bawaan, yang ditulis dalam C, mengerjakan gelung-gelung berat sedapat mungkin.

Untuk perencanaan, bulatkan hasil pengukuran ke pangkat sepuluh. Mesin juri mungkin lebih tua daripada mesin Anda, dan satu langkah dalam program sungguhan mungkin lebih mahal daripada putaran gelung murah kita, sehingga aturan kerjanya adalah:

> **WAWASAN KUNCI:** Rencanakan dengan sekitar 10^8 langkah sederhana per detik dalam C++ dan sekitar 10^6 sampai 10^7 dalam Python. Metode yang banyak langkahnya, pada masukan

terbesar, berada di bawah anggaran Persamaan (3.1) dengan faktor sepuluh aman; yang berada dalam faktor tiga berisiko; yang melampauinya tidak akan lolos, betapapun rapinya ditulis.

3.3 Menghitung Langkah Sebuah Algoritma

Untuk membandingkan sebuah metode dengan anggaran, kita memerlukan banyak langkahnya pada masukan terbesar, dan untuk itu kita tidak memerlukan hitungan yang persis, cukup orde besarnya. Alat bakunya adalah notasi *big-O* [1]. Buku-buku panduan untuk peserta pemrograman kompetitif memakai persis perkiraan semacam ini sebagai langkah pertama pada setiap soal [2], [3].

DEFINISI. Sebuah algoritma berjalan dalam waktu $O(f(n))$ jika ada konstanta c dan n_0 sedemikian sehingga pada setiap masukan berukuran $n \geq n_0$ algoritma itu mengerjakan paling banyak $c \cdot f(n)$ langkah. Secara informal: untuk masukan besar, banyak langkahnya tumbuh paling cepat seperti $f(n)$, dengan mengabaikan faktor konstanta.

Dalam praktik, tiga aturan memberikan orde hampir setiap program dalam buku ini.

1. **Gelung mengalikan.** Gelung yang berjalan n kali mengelilingi badan berbiaya g memerlukan $n \cdot g$. Dua gelung bersarang atas n nilai memerlukan $n \cdot n = n^2$.
2. **Membagi dua menghasilkan logaritma.** Gelung yang membagi dua rentangnya setiap putaran, seperti pencarian biner, atau menggandakan sebuah nilai sampai melampaui n, berjalan sekitar $\log_2 n$ kali. $\log_2 10^9 \approx 30$: satu miliar menjadi satu setelah tiga puluh kali dibagi dua.
3. **Kasus uji dijumlahkan.** Jika ada T kasus uji dan masing-masing memerlukan $f(n)$, seluruh masukan memerlukan $T \cdot f(n)$. Aturan inilah yang dilupakan pemula: metode yang cepat untuk satu kasus bisa tak berdaya untuk seratus ribu kasus.

$$\text{total langkah} \approx T \cdot f(n_{\max}) \tag{3.2}$$

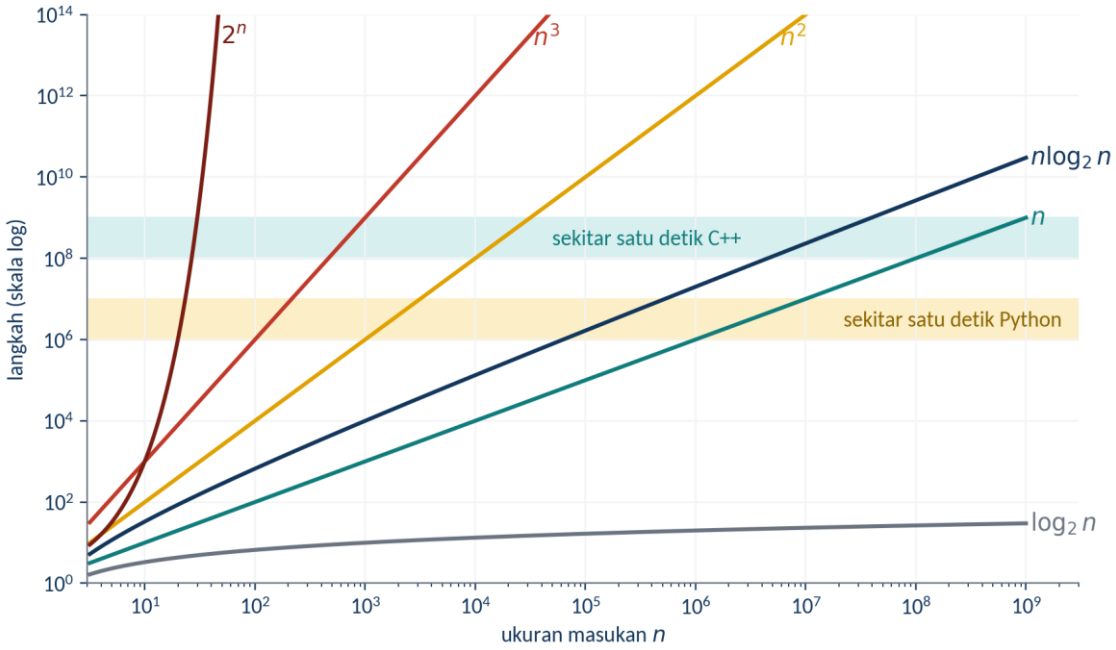
Persamaan (3.2), dibandingkan dengan Persamaan (3.1), adalah seluruh metodenya. Sebagian soal justru menyatakan batas untuk *jumlah* ukuran semua kasus uji; totalnya lalu dibatasi oleh f dari jumlah itu, yang biasanya jauh lebih kecil daripada $T \cdot f(n_{\max})$.

Sebuah hitungan yang dikerjakan memperlihatkan aturan-aturan itu bekerja. Andaikan sebuah soal memberikan $T \leq 10$ larik berisi $n \leq 10^4$ bilangan dan menanyakan, untuk setiap larik, berapa banyak pasangan posisi $i < j$ yang nilainya sama. Program yang gamblang punya dua gelung bersarang: i berjalan dari 0 sampai $n-1$ dan, untuk setiap i, j berjalan dari $i+1$ sampai $n-1$. Gelung dalam berjalan $n-1$ kali untuk $i=0$, $n-2$ kali untuk $i=1$, dan seterusnya sampai 0, sehingga banyaknya perbandingan adalah

$$(n-1) + (n-2) + \cdots + 1 + 0 = \frac{n(n-1)}{2} \tag{3.3}$$

yang untuk $n = 10^4$ bernilai sekitar 5×10^7. Dikali $T = 10$ menjadi 5×10^8 menurut Persamaan (3.2): lima kali anggaran perencanaan 10^8, sehingga gelung ganda itu paling-paling berisiko. Mengurutkan setiap larik lebih dulu lalu menghitung deretan nilai yang sama memerlukan $O(n \log n)$ per larik, total sekitar $10 \cdot 10^4 \cdot 14 \approx 1.4 \times 10^6$ langkah, beberapa ratus kali lebih sedikit. Persamaan (3.3) patut diingat: setiap gelung "semua pasangan" memerlukan sekitar $n^2/2$, separuh dari kuadratnya, yang tetap kuadratik, karena konstanta $1/2$ tidak mengubah laju pertumbuhan.

Gambar 3.1 memperlihatkan bagaimana kelas-kelas kompleksitas yang umum tumbuh, pada sumbu logaritmik yang membuat setiap pangkat n menjadi garis lurus. Kedua pita berbayang adalah sekitar satu detik C++ dan sekitar satu detik Python.



Gambar 3.1 Langkah yang diperlukan kelas-kelas kompleksitas yang umum seiring bertambahnya ukuran masukan, dibandingkan dengan sekitar satu detik C++ (pita hijau kebiruan) dan Python (pita kuning tua). Titik potong sebuah kurva dengan sebuah pita adalah masukan terbesar yang dapat ditangani kelas itu dalam satu detik.

Membaca gambar itu dari kiri ke kanan menghasilkan tabel yang disimpan di kepala para peserta berpengalaman, **Tabel 3.1**. Temukan baris n terbesar dalam batasan; kompleksitas di sebelahnya adalah yang paling lambat yang kemungkinan besar lolos dalam C++.

Tabel 3.1 Anggaran kompleksitas dalam C++: ukuran masukan terbesar dalam batasan dan kompleksitas paling lambat yang muat dalam sekitar satu detik. Untuk Python, turunlah satu atau dua baris.

n Terbesar	Kompleksitas paling lambat yang muat dalam sekitar satu detik	Metode khas
sampai 10–11	$O(n!)$	mencoba semua urutan
sampai 20–25	$O(2^n)$, $O(2^n \cdot n)$	mencoba semua himpunan bagian, pemrograman dinamis *bitmask*
sampai sekitar 500	$O(n^3)$	gelung rangkap tiga, Floyd–Warshall, pemrograman dinamis interval
sampai sekitar 5.000	$O(n^2)$	gelung ganda, pemrograman dinamis sederhana
sampai sekitar 10^6	$O(n\log n)$	pengurutan, pencarian biner, *heap*, *segment tree*
sampai sekitar 10^8	$O(n)$	satu lintasan, jumlah prefiks, dua penunjuk, saringan
di atas 10^9	$O(\sqrt{n})$, $O(\log n)$, $O(1)$	rumus, pengkuadratan berulang, teori bilangan

Dua peringatan tentang tabel ini. Pertama, tabel ini menggambarkan C++; untuk Python, baris $O(n)$ berakhir di sekitar 10^7 alih-alih 10^8, dan baris $O(n^2)$ di sekitar 1.000 sampai 2.000. Kedua, di dekat batas antarbaris, hitunglah alih-alih menebak: $n = 10^5$ dengan metode $O(n^2)$ berarti 10^{10} langkah, seratus kali anggaran, sedangkan $n = 3000$ dengan metode yang sama berarti

9×10^6, dengan nyaman di dalamnya. Lompatan antarbaris yang bertetangga itu besar, dan jarang ada jalan tengah, sehingga tabel ini adalah saringan pertama yang cepat dan andal.

Tabel ini juga bekerja terbalik, dan inilah salah satu kiat paling berguna dalam pemrograman kompetitif: batasan adalah petunjuk. Penyusun soal yang mengizinkan $n \leq 20$ sedang mengisyaratkan metode eksponensial atas himpunan bagian; yang mengizinkan $n \leq 10^5$ sedang menyingkirkan segala yang kuadratik; yang mengizinkan $n \leq 10^{18}$ sedang berkata bahwa jawabannya datang dari rumus atau dari $O(\log n)$ langkah. Jika Anda sama sekali tidak tahu cara menyelesaikan sebuah soal, lihatlah batasannya dan tanyakan baris mana yang cocok.

3.4 FCTRL: Ketika Rumus Menggantikan Gelung

Soalnya (SPOJ FCTRL, "Factorial", `https://www.spoj.com/problems/FCTRL/`) [4]. Masukan diawali banyaknya kasus uji T, sekitar 10^5; setiap kasus berupa satu bilangan bulat n dengan $1 \leq n \leq 10^9$. Untuk setiap kasus, cetak banyaknya nol di ujung representasi desimal $n!$. Cerita di sekelilingnya tentang jaringan telepon seluler, dan murni hiasan: tugasnya satu kalimat, banyaknya nol di ujung $n!$.

Dari mana datangnya nol. Sebuah nol di ujung adalah faktor 10 dari bilangan itu, dan $10 = 2 \cdot 5$. Dalam $n! = 1 \cdot 2 \cdots n$ selalu ada lebih banyak faktor 2 daripada faktor 5 (setiap bilangan kedua genap, hanya setiap bilangan kelima kelipatan 5), sehingga banyaknya nol di ujung sama dengan banyaknya faktor 5 dalam $n!$. Untuk $n = 10$: faktor 5 berasal dari 5 dan 10, sehingga $10! = 3\,628\,800$ punya dua nol di ujung.

Metode gamblang dan biayanya. Telusuri kelipatan 5 sampai n dan hitung berapa kali masing-masing habis dibagi 5. Itu sekitar $n/5$ putaran per kasus. Dengan Persamaan (3.2): $T \cdot n/5 = 10^5 \cdot 2 \times 10^8 = 2 \times 10^{13}$ langkah, yaitu 2×10^5 kali anggaran. **Tabel 3.1** menyatakan hal yang sama sekilas pandang: $n = 10^9$ memerlukan $O(\log n)$ atau rumus. Program gamblangnya, **Kode 3.3**, benar, dan kita menyimpannya sebagai *brute force* untuk uji kita sendiri, tetapi program itu tidak mungkin lolos.

Kode 3.3 `fctrl_naive.cpp`: FCTRL dengan menghitung faktor 5 dari setiap kelipatan 5, benar tetapi jauh terlalu lambat.

```cpp
// SPOJ FCTRL (Factorial), cara lambat: hitung faktor 5 di setiap kelipatan 5.
#include <iostream>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int cases;
    std::cin >> cases;
    while (cases--) {
        long long n;
        std::cin >> n;
        long long zeros = 0;
        for (long long k = 5; k <= n; k += 5) {  // sekitar n / 5 putaran per kasus
            long long m = k;
            while (m % 5 == 0) {
                zeros++;
                m /= 5;
            }
        }
        std::cout << zeros << '\n';
    }
    return 0;
}
```

Rumusnya. Hitung faktor 5 dengan cara lain: bukan bilangan demi bilangan, melainkan lapis demi lapis. Setiap kelipatan 5 sampai n menyumbang paling sedikit satu faktor 5, dan banyaknya $\lfloor n/5 \rfloor$. Setiap kelipatan 25 menyumbang faktor 5 kedua, dan banyaknya $\lfloor n/25 \rfloor$. Setiap kelipatan 125 menyumbang faktor ketiga, dan seterusnya. Bilangan yang habis dibagi tepat 5^j terhitung sekali di setiap lapis dari j lapis pertama, yang persis sama dengan banyaknya faktor 5 bilangan itu. Jadi banyaknya adalah Persamaan (3.4), yang dikenal sebagai rumus Legendre [5].

$$Z(n) = \left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{25} \right\rfloor + \left\lfloor \frac{n}{125} \right\rfloor + \cdots = \sum_{j \geq 1} \left\lfloor \frac{n}{5^j} \right\rfloor \tag{3.4}$$

Dalam praktik jumlah itu berhingga, karena $\lfloor n/5^j \rfloor = 0$ begitu $5^j > n$; untuk $n = 10^9$ hal itu terjadi pada $j = 13$, karena $5^{13} \approx 1.2 \times 10^9$. Jadi setiap kasus memerlukan sekitar 13 pembagian, dan semua 10^5 kasus sekitar 1.3×10^6: sepersejuta metode gamblang. Contoh yang dikerjakan, untuk nilai contoh $n = 100$: $\lfloor 100/5 \rfloor = 20$, $\lfloor 100/25 \rfloor = 4$, $\lfloor 100/125 \rfloor = 0$, jadi $Z(100) = 24$.

Kode 3.4 mewujudkannya dalam C++, dan **Kode 3.5** dalam Python.

Kode 3.4 `fctrl.cpp`: FCTRL dengan rumus Legendre, sekitar 13 pembagian per kasus.

```cpp
// SPOJ FCTRL (Factorial): nol di ujung n! = n/5 + n/25 + n/125 + ...
#include <iostream>

long long trailingZeros(long long n) {
    long long zeros = 0;
    for (long long power = 5; power <= n; power *= 5) {
        zeros += n / power;  // kelipatan 5, lalu kelipatan 25, lalu 125, ...
    }
    return zeros;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int cases;
    std::cin >> cases;
    while (cases--) {
        long long n;
        std::cin >> n;
        std::cout << trailingZeros(n) << '\n';
    }
    return 0;
}
```

Variabel `power` menelusuri 5,25,125, ...; tipenya `long long` karena pangkat terakhir yang dihitung gelung, yang pertama melampaui n, dapat melampaui jangkauan `int` ketika n mendekati 10^9: $5^{13} = 1\,220\,703\,125 < 2^{31}$, tetapi batas yang ceroboh seperti $n \le 2 \times 10^9$ akan mendorongnya melewati batas. Memakai tipe 64-bit untuk nilai yang dikalikan di dalam gelung adalah kebiasaan murah yang mencegah satu kelas *bug*.

Kode 3.5 `fctrl.py`: rumus yang sama dalam Python, dengan membaca seluruh masukan sekaligus.

```python
"""SPOJ FCTRL (Factorial): nol di ujung n! = n//5 + n//25 + n//125 + ..."""
import sys


def trailing_zeros(n):
    zeros, power = 0, 5
    while power <= n:
        zeros += n // power  # kelipatan 5, lalu kelipatan 25, lalu 125, ...
        power *= 5
    return zeros


def main():
    data = sys.stdin.buffer.read().split()
    cases = int(data[0])
    sys.stdout.write("\n".join(str(trailing_zeros(int(x)))
                               for x in data[1:1 + cases]) + "\n")


if __name__ == "__main__":
    main()
```

Program Python membaca semua token sekaligus dengan `sys.stdin.buffer.read().split()` dan menulis semua jawaban dengan satu pemanggilan, yang penting untuk 10^5 baris, seperti ditunjukkan Bab 4. Ketiga program mencetak contoh keluaran:

```
$ printf '6\n3\n60\n100\n1024\n23456\n8735373\n' | ./cpp/fctrl
```

```
0
14
24
253
5861
2183837
```

Mengujinya. *Brute force* untuk FCTRL adalah definisinya sendiri: hitung $n!$ secara tepat dan hitung nolnya, **Kode 3.6**. Cara ini hanya praktis untuk n kecil, sehingga generatornya, **Kode 3.7**, menghasilkan nilai sampai 3000, dengan nilai tepi 1, 4, 5, 24, 25, 124, dan 125, tempat pangkat 5 yang baru mulai terhitung, sering muncul.

Kode 3.6 `brute_fctrl.py`: FCTRL dengan menghitung $n!$ seutuhnya lalu menghitung nol di ujungnya.

```python
"""brute_fctrl.py: FCTRL dengan menghitung n! seutuhnya lalu menghitung nolnya."""
import math
import sys


def main():
    sys.set_int_max_str_digits(0)  # izinkan str() untuk bilangan > 4300 digit
    data = sys.stdin.read().split()
    for x in data[1:1 + int(data[0])]:
        digits = str(math.factorial(int(x)))
        print(len(digits) - len(digits.rstrip("0")))


if __name__ == "__main__":
    main()
```

Kode 3.7 `gen_fctrl.py`: masukan FCTRL kecil dengan nilai batas pangkat 5.

```python
"""gen_fctrl.py SEED: masukan FCTRL kecil (n <= 3000 agar brute force sanggup)."""
import random
import sys


def main():
    rng = random.Random(int(sys.argv[1]))
    cases = rng.randint(1, 8)
    print(cases)
    for _ in range(cases):
        print(rng.choice([1, 4, 5, 24, 25, 124, 125, rng.randint(1, 3000)]))


if __name__ == "__main__":
    main()
```

Brute force inilah program yang menabrak batas Python pada konversi bilangan bulat ke string yang disebut di Bab 2: 2030! punya lebih dari 4300 digit, dan tanpa `sys.set_int_max_str_digits(0)` pemanggilan `str()`-nya gagal. *Stress test* menemukannya pada putaran pertama, satu alasan lagi untuk menjalankan *stress test* pada *brute force* itu sendiri sebelum memercayainya.

```
$ cd tests
$ python3 stress.py gen_fctrl.py brute_fctrl.py ../cpp/fctrl.cpp 300
$ python3 stress.py gen_fctrl.py brute_fctrl.py ../python/fctrl.py 200
$ python3 stress.py gen_fctrl.py brute_fctrl.py ../cpp/fctrl_naive.cpp 200
```

```
300 rounds: all answers equal
200 rounds: all answers equal
200 rounds: all answers equal
```

Ketiganya sesuai dengan *brute force*: program gamblang itu *benar*. Sekarang soal waktu. **Kode 3.8** menulis dua uji untuk juri kita: contoh soal, dan masukan sah terbesar, 10^5 nilai n antara 9×10^8 dan 10^9, dengan jawaban harapan yang dihitung oleh implementasi ketiga Persamaan (3.4) yang independen.

Kode 3.8 `make_fctrl_tests.py`: contoh soal dan uji berukuran maksimum untuk FCTRL, dalam format `minijudge.py`.

```
"""make_fctrl_tests.py: menulis FCTRL/01 (contoh) dan FCTRL/02 (yang terbesar)."""
import random
from pathlib import Path

HERE = Path(__file__).resolve().parent / "FCTRL"


def zeros(n):
    count = 0
    while n:
        n //= 5
        count += n
    return count


def write(name, numbers):
    HERE.mkdir(exist_ok=True)
    text = f"{len(numbers)}\n" + "".join(f"{n}\n" for n in numbers)
    (HERE / f"{name}.in").write_text(text)
    (HERE / f"{name}.out").write_text("".join(f"{zeros(n)}\n" for n in numbers))


def main():
    rng = random.Random(5)
    write("01", [3, 60, 100, 1024, 23456, 8735373])
    write("02", [rng.randint(900_000_000, 1_000_000_000) for _ in range(100_000)])
    print("wrote FCTRL/01 and FCTRL/02")


if __name__ == "__main__":
    main()
```

Dengan batas waktu satu detik, `minijudge.py` dari Bab 1 memberikan:

```
$ python3 make_fctrl_tests.py
$ python3 ../../Chapter01/python/minijudge.py FCTRL 1 ../cpp/fctrl_naive.cpp
$ python3 ../../Chapter01/python/minijudge.py FCTRL 1 ../cpp/fctrl.cpp
$ python3 ../../Chapter01/python/minijudge.py FCTRL 1 ../python/fctrl.py
```

```
wrote FCTRL/01 and FCTRL/02
  test 01: AC   0.01 s
  test 02: TLE  1.00 s
verdict: TLE (time limit exceeded)
  test 01: AC   0.00 s
  test 02: AC   0.03 s
verdict: AC (accepted)
  test 01: AC   0.03 s
  test 02: AC   0.22 s
verdict: AC (accepted)
```

Inilah vonis yang dijanjikan Bab 1: program yang benar, dihentikan pada uji besar setelah satu detik, persis seperti yang dilakukan juri. Menurut perkiraan di atas program itu akan memerlukan berjam-jam. Rumusnya memerlukan 0,03 detik dalam C++ dan 0,22 detik dalam Python, yang sebagian besar habis untuk membaca masukan dan menyalakan interpreter. Versi Python lolos dengan margin lebar: jika algoritmanya benar, Python sering kali cukup cepat.

3.5 LASTDIG: Membagi Dua Pangkat

Soalnya (SPOJ LASTDIG, "The last digit", `https://www.spoj.com/problems/LASTDIG/`) [6]. Masukan diawali banyaknya kasus uji, paling banyak 30; setiap kasus memberikan dua bilangan bulat a dan b dengan $0 \le a \le 20$ dan $0 \le b \le 2\,147\,483\,000$, yang tidak keduanya nol. Cetak digit desimal terakhir dari a^b.

Membaca batasannya. Basisnya mungil dan pangkatnya raksasa. a^b sendiri terdiri atas sampai sekitar $b\log_{10}20 \approx 2.8 \times 10^9$ digit, sehingga menghitungnya mustahil bahkan dalam Python. Digit terakhir sebuah hasil kali hanya bergantung pada digit terakhir faktor-faktornya, sehingga kita dapat mengalikan modulo 10 sepanjang jalan: $a^b \bmod 10$. Mengalikan b kali memerlukan sampai 2.1×10^9 langkah per kasus, 6.4×10^{10} untuk 30 kasus, jauh melampaui anggaran. **Tabel 3.1**, baris "di atas 10^9", menunjuk ke $O(\log b)$.

Pengkuadratan berulang. Tuliskan pangkatnya dalam biner. Untuk $b = 10$, biner `1010`, $a^{10} = a^8 \cdot a^2$. Pangkat-pangkat $a^1, a^2, a^4, a^8, \ldots$ masing-masing adalah kuadrat pangkat sebelumnya, sehingga masing-masing memerlukan satu perkalian, dan kita mengalikan ke dalam hasil tepat pangkat-pangkat yang bitnya pada b bernilai 1:

$$a^b = \prod_{i\,:\,\text{bit } i \text{ dari } b \text{ bernilai } 1} a^{2^i} \qquad \text{dengan} \quad a^{2^{i+1}} = (a^{2^i})^2 \tag{3.5}$$

Persamaan (3.5) berlaku karena $b = \sum_i \beta_i\, 2^i$ dengan bit $\beta_i \in \{0,1\}$, dan $a^{x+y} = a^x a^y$. Karena b punya sekitar $\log_2 b$ bit, paling banyak 31 di sini, metode ini memerlukan paling banyak 31 pengkuadratan dan 31 perkalian, semuanya modulo 10. Setiap nilai antara di bawah 10, sehingga hasil kalinya di bawah 100 dan tidak ada yang dapat *overflow* [1].

Contoh yang dikerjakan, $3^{10} \bmod 10$ dari contoh soal: $b = 10 = 1010_2$. Mulai dengan hasil 1 dan basis 3.

1. Bit 0 dari 10 bernilai 0: hasil tetap 1; basis menjadi $3^2 = 9$.
2. Bit 1 bernilai 1: hasil $1 \cdot 9 = 9$; basis menjadi $9^2 = 81 \equiv 1$.
3. Bit 2 bernilai 0: hasil tetap 9; basis menjadi $1^2 = 1$.
4. Bit 3 bernilai 1: hasil $9 \cdot 1 = 9$; basis 1. Tidak ada bit tersisa.

Digit terakhir $3^{10} = 59\,049$ adalah 9, sesuai kata metode itu.

Kode 3.9 dan **Kode 3.10** mewujudkan metode ini.

Kode 3.9 `lastdig.cpp`: LASTDIG dengan pengkuadratan berulang modulo 10.

```cpp
// SPOJ LASTDIG (The last digit): digit terakhir a^b dengan pengkuadratan berulang.
#include <iostream>

// (base ^ exp) % mod dalam sekitar log2(exp) putaran.
long long powerMod(long long base, long long exp, long long mod) {
    long long result = 1 % mod;
    base %= mod;
    while (exp > 0) {
        if (exp & 1) result = result * base % mod;  // bit exp yang ini bernilai 1
        base = base * base % mod;
        exp >>= 1;
    }
    return result;
}

int main() {
    int cases;
    std::cin >> cases;
    while (cases--) {
        long long a, b;
        std::cin >> a >> b;
        std::cout << powerMod(a, b, 10) << '\n';
    }
    return 0;
}
```

Di `powerMod`, `exp & 1` adalah bit terendah pangkat, dan `exp >>= 1` menggesernya keluar, sehingga gelung mengunjungi bit dari yang terendah ke atas, persis seperti pada contoh yang dikerjakan. Nilai awal `1 % mod` alih-alih `1` membuat fungsi ini benar bahkan untuk `mod = 1`, ketika setiap hasil bernilai 0; biayanya nol dan satu kasus tepi hilang. Kasus $b = 0$ tidak memerlukan kode khusus: gelung tidak berjalan dan hasilnya 1, digit terakhir dari $a^0 = 1$.

Kode 3.10 `lastdig.py`: metode yang sama dalam Python; `pow(a, b, 10)` bawaan melakukan hal yang sama.

```python
"""SPOJ LASTDIG (The last digit): digit terakhir a^b, pengkuadratan berulang."""
import sys


def power_mod(base, exp, mod):
    """(base ** exp) % mod dalam ~log2(exp) putaran, seperti pow(base, exp, mod)."""
    result = 1 % mod
    base %= mod
    while exp > 0:
        if exp & 1:  # bit exp yang ini bernilai 1
            result = result * base % mod
        base = base * base % mod
        exp >>= 1
    return result


def main():
    data = sys.stdin.read().split()
    cases = int(data[0])
    for c in range(cases):
        a, b = int(data[1 + 2 * c]), int(data[2 + 2 * c])
        print(power_mod(a, b, 10))


if __name__ == "__main__":
    main()
```

`pow` bawaan Python dengan tiga argumen menghitung $a^b \bmod m$ dengan metode yang sama, dan dalam kode sungguhan Anda sebaiknya memakainya; fungsi di sini ditulis lengkap untuk

memperlihatkan metodenya, dan *brute force*-nya memakai fungsi bawaan, sehingga *stress test* membandingkan dua implementasi yang independen.

```
$ printf '2\n3 10\n6 2\n' | ./cpp/lastdig
```

```
9
6
```

```
$ python3 stress.py gen_lastdig.py brute_lastdig.py ../cpp/lastdig.cpp
$ python3 stress.py gen_lastdig.py brute_lastdig.py ../python/lastdig.py 200
```

```
500 rounds: all answers equal
200 rounds: all answers equal
```

Generator uji ini, `gen_lastdig.py` di folder `tests`, menghasilkan basis dari 0 sampai 20 dan pangkat yang kecil (0 sampai 5, tempat kasus tepi berada, termasuk $b = 0$ dan 0^b) atau sembarang sampai 2 147 483 000, dan tidak pernah menghasilkan pasangan terlarang $a = b = 0$. Pengkuadratan berulang adalah salah satu gagasan yang paling sering dipakai ulang dalam buku ini. Gelung yang sama, dengan bilangan diganti matriks, menghitung bilangan Fibonacci ke-n modulo m untuk n sampai 10^{18} dalam sekitar 60 perkalian matriks; dengan modulus prima p, gelung itu menghitung invers modular $a^{p-2} \bmod p$; dan gelung itu berlaku untuk operasi apa pun yang asosiatif, karena hanya sifat asosiatif yang dipakai ketika a^{x+y} dipecah menjadi $a^x a^y$. Bab 13 kembali membahasnya. Ada juga solusi yang sama sekali berbeda: digit terakhir $a^1, a^2, a^3, \ldots$ berulang dengan periode yang membagi 4, sehingga hanya b mod 4 yang penting; Soal Latihan 4 menjelajahinya.

3.6 PRIME1: Saringan Bersegmen

Soalnya (SPOJ PRIME1, "Prime Generator", `https://www.spoj.com/problems/PRIME1/`) [7]. Masukan diawali banyaknya kasus uji $t \le 10$; setiap kasus memberikan dua bilangan m dan n dengan $1 \le m \le n \le 10^9$ dan $n - m \le 10^5$. Untuk setiap kasus, cetak setiap bilangan prima p dengan $m \le p \le n$, satu per baris, dan pisahkan keluaran kasus-kasus uji yang berurutan dengan sebuah baris kosong. Inilah soal yang tiga kali salah dikerjakan tim Rina, di Bab 2, karena baris kosong itu; Subbab 2.2 bab tersebut sudah membaca batasannya.

Membaca batasannya. Bilangannya besar, sampai 10^9, tetapi setiap rentang pendek, paling banyak $10^5 + 1$ bilangan, dan paling banyak ada 10 rentang: total paling banyak sekitar 10^6 kandidat. Putaran bentuk menambahkan: $m = 1$ mungkin, dan 1 bukan prima; sebuah rentang bisa hanya memuat satu bilangan; dan $p \cdot p$ untuk p sampai sekitar 31 623 mendekati 10^9, yang muat dalam `int`, tetapi penjumlahan seperti $m + p$ atau hasil kali yang dihitung agak ceroboh dapat melampauinya, sehingga program menghitung posisi dalam `long long`.

Metode gamblang dan biayanya. *Pembagian percobaan* (*trial division*) menguji setiap kandidat x dengan membaginya dengan $2, 3, \ldots, \lfloor\sqrt{x}\rfloor$, yang cukup menurut Persamaan (3.6) di bawah. Bilangan prima di sekitar 10^9 memerlukan sekitar 31 623 pembagian; rentang 10^5 bilangan di sekitar 10^9 memuat sekitar 4.800 bilangan prima, sehingga sepuluh rentang memerlukan sekitar 1.5×10^9 pembagian untuk bilangan primanya saja, lebih dari sepuluh kali anggaran, dan pembagian termasuk operasi mesin yang paling lambat. *Saringan Eratosthenes* biasa sampai n akan memerlukan larik berisi 10^9 entri, satu gigabita, dan satu lintasan atasnya untuk setiap kasus: terlalu lambat sekaligus terlalu besar.

Gagasannya. Dua pengamatan bergabung menjadi solusinya.

Pertama, bilangan komposit x selalu punya faktor prima $p \le \sqrt{x}$. Jika $x = a \cdot b$ dengan $1 < a \le b$, maka $a^2 \le a \cdot b = x$, sehingga $a \le \sqrt{x}$, dan setiap faktor prima a juga paling besar $\sqrt{x}$:

$$x \text{ komposit} \Longrightarrow x \text{ punya faktor prima } p \le \sqrt{x} \le \sqrt{n} \tag{3.6}$$

Untuk $n \le 10^9$, $\sqrt{n} < 31\,623$, sehingga bilangan prima sampai 31 623, yang banyaknya 3.401, cukup untuk mengenali setiap bilangan komposit di rentang mana pun.

Kedua, alih-alih menguji setiap bilangan terhadap bilangan-bilangan prima, biarkan setiap bilangan prima *mencoret* kelipatannya di dalam rentang, seperti dilakukan saringan Eratosthenes. Untuk bilangan prima p dan rentang $[m, n]$, kelipatan pertama p yang paling sedikit m adalah $\lceil m/p \rceil \cdot p$; mulai dari situ, setiap bilangan ke-p adalah kelipatan. Kita juga boleh mulai tidak lebih rendah dari p^2, karena kelipatan yang lebih kecil $k \cdot p$ dengan $k < p$ punya faktor prima lebih kecil yang sudah mencoretnya, dan mulai dari p^2 juga melindungi p sendiri dari tercoret jika p berada di dalam rentang. Jadi setiap bilangan prima mencoret posisi-posisi

$$\max\left(p^2,\ \left\lceil \frac{m}{p} \right\rceil p\right),\ \ldots + p,\ \ldots + 2p,\ \ldots \le n \tag{3.7}$$

dan bilangan yang tidak pernah tercoret, kecuali 1, adalah bilangan prima di rentang itu. Inilah *saringan bersegmen* (*segmented sieve*): saringan biasa sekali sampai $\sqrt{n_{\max}}$, lalu saringan kecil atas setiap rentang.

Biayanya. Bilangan prima p mencoret sekitar $(n - m)/p$ posisi dalam sebuah rentang, sehingga sebuah rentang memerlukan sekitar $(n - m)\sum_{p \le \sqrt{n}} 1/p$ langkah. Jumlah kebalikan bilangan prima sampai x tumbuh seperti $\ln\ln x$, yang untuk $x = 31\,623$ bernilai sekitar 2,6, sehingga sebuah rentang memerlukan beberapa kali 10^5 langkah dan sepuluh rentang beberapa juta, ditambah 3.401 perhitungan titik awal per rentang. Jauh di dalam anggaran, dalam kedua bahasa.

Memorinya. Batas waktu bukan satu-satunya anggaran; batas memori adalah anggaran lain, dan Bab 6 membahasnya secara rinci. Saringan bersegmen memerlukan satu bita per bilangan dalam rentang yang sedang diproses, paling banyak $10^5 + 1$ bita, sekitar 100 KB, ditambah 3.401 bilangan prima dasar, sekitar 14 KB sebagai `int`. Saringan biasa sampai 10^9 dengan satu bita per bilangan akan memerlukan satu gigabita; bahkan jika dipadatkan menjadi satu bit per bilangan, seperti yang dilakukan `std::vector<bool>`, saringan itu memerlukan 125 MB dan satu lintasan atas satu miliar posisi. Jadi saringan bersegmen unggul pada kedua anggaran sekaligus, hal yang lazim: algoritma yang menyentuh lebih sedikit data biasanya lebih cepat sekaligus lebih hemat. Menuliskan banyaknya bita di sebelah banyaknya langkah, sebagai baris kedua perkiraan, hanya memakan sepuluh detik dan mencegah vonis yang dilaporkan sebagai galat saat berjalan padahal sebenarnya batas memori.

Contoh yang dikerjakan pada rentang [10,30]: $\sqrt{30} < 6$, sehingga bilangan prima 2, 3, dan 5 sudah cukup. Prima 2 mulai di $\max(4,10) = 10$ dan mencoret 10, 12, ..., 30. Prima 3 mulai di $\max(9, \lceil 10/3 \rceil \cdot 3) = 12$ dan mencoret 12, 15, 18, 21, 24, 27, 30. Prima 5 mulai di $\max(25,10) = 25$ dan mencoret 25 dan 30. Bilangan yang tersisa adalah 11, 13, 17, 19, 23, 29, persis bilangan prima antara 10 dan 30.

Kode 3.11 adalah program C++-nya dan **Kode 3.12** program Python-nya.

Kode 3.11 `prime1.cpp`: PRIME1 dengan saringan bersegmen; bilangan prima dasar dihitung sekali, setiap rentang disaring terpisah.

```cpp
// SPOJ PRIME1 (Prime Generator): saringan Eratosthenes bersegmen.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>

// Semua bilangan prima sampai limit, dengan saringan biasa.
std::vector<int> smallPrimes(int limit) {
    std::vector<bool> composite(limit + 1, false);
    std::vector<int> primes;
    for (int p = 2; p <= limit; p++) {
        if (composite[p]) continue;
        primes.push_back(p);
        for (long long q = 1LL * p * p; q <= limit; q += p) composite[q] = true;
    }
    return primes;
}

// Tambahkan setiap prima di [m, n] ke out, satu per baris.
void primesBetween(long long m, long long n, const std::vector<int>& base,
                   std::string& out) {
    std::vector<char> composite(n - m + 1, 0);  // composite[i] menyangkut m + i
    for (int p : base) {
        long long pp = 1LL * p * p;
        if (pp > n) break;
        long long start = std::max(pp, (m + p - 1) / p * p);  // kelipatan pertama
        for (long long j = start; j <= n; j += p) composite[j - m] = 1;
    }
    for (long long x = std::max(m, 2LL); x <= n; x++) {
        if (!composite[x - m]) out += std::to_string(x) + '\n';
    }
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    const std::vector<int> base = smallPrimes(31623);  // 31623^2 > 10^9
    int cases;
    std::cin >> cases;
    std::string out;
    for (int c = 0; c < cases; c++) {
        long long m, n;
        std::cin >> m >> n;
        if (c > 0) out += '\n';  // baris kosong di antara kasus uji
        primesBetween(m, n, base, out);
    }
```

Kode 3.11 (Lanjutan) *prime1.cpp: PRIME1 dengan saringan bersegmen; bilangan prima dasar dihitung sekali, setiap rentang disaring terpisah.*

```cpp
    std::cout << out;
    return 0;
}
```

`smallPrimes` adalah saringan biasa sampai 31 623, dijalankan sekali di awal. `primesBetween` mengalokasikan satu bita per bilangan dalam rentang, `composite[i]` mewakili bilangan $m + i$, dan menerapkan Persamaan (3.7) untuk setiap bilangan prima dasar; gelung atas bilangan prima dasar berhenti begitu $p^2 > n$, menurut Persamaan (3.6). Ekspresi `(m + p - 1) / p * p` adalah bentuk bilangan bulat dari $\lceil m/p \rceil \cdot p$. Keluaran semua kasus dikumpulkan dalam satu string dan ditulis di akhir, dengan baris kosong sebelum setiap kasus kecuali yang pertama, persis aturan format keluarannya: *di antara* kasus, bukan sesudah setiap kasus.

Kode 3.12 `prime1.py`: saringan bersegmen yang sama dalam Python, mencoret dengan penugasan irisan.

```
"""SPOJ PRIME1 (Prime Generator): saringan Eratosthenes bersegmen."""
import sys


def small_primes(limit):
    """Semua bilangan prima sampai limit, dengan saringan biasa."""
    composite = bytearray(limit + 1)
    primes = []
    for p in range(2, limit + 1):
        if not composite[p]:
            primes.append(p)
            composite[p * p::p] = b"\x01" * len(range(p * p, limit + 1, p))
    return primes


def primes_between(m, n, base):
    """Setiap prima di [m, n]; composite[i] menyangkut m + i."""
    size = n - m + 1
    composite = bytearray(size)
    for p in base:
        if p * p > n:
            break
        start = max(p * p, (m + p - 1) // p * p) - m  # kelipatan pertama (indeks)
        composite[start::p] = b"\x01" * len(range(start, size, p))
    return [m + i for i in range(size) if not composite[i] and m + i >= 2]


def main():
    data = sys.stdin.read().split()
    base = small_primes(31623)  # 31623 ** 2 > 10 ** 9
    cases = int(data[0])
    blocks = []
    for c in range(cases):
        m, n = int(data[1 + 2 * c]), int(data[2 + 2 * c])
        blocks.append("".join(f"{x}\n" for x in primes_between(m, n, base)))
    sys.stdout.write("\n".join(blocks))  # baris kosong di antara kasus uji


if __name__ == "__main__":
    main()
```

Versi Python harus menghindari gelung tingkat Python atas posisi yang dicoret, yang akan memerlukan beberapa juta langkah terinterpretasi. Program ini memakai *penugasan irisan* (*slice assignment*): `composite[start::p] = b"\x01" * count` mengisi setiap bita ke-p mulai dari `start` dalam satu operasi yang berjalan di dalam kode C interpreter. Sisi kanan harus punya bita sebanyak posisi dalam irisan, yaitu `len(range(start, size, p))`, bilangan yang dihitung Python tanpa membangun rentangnya. *List comprehension* di akhir adalah satu-satunya gelung tingkat Python yang tersisa, sekali per bilangan dalam rentang.

```
$ printf '2\n1 10\n3 5\n' | ./cpp/prime1
```

```
2
3
5
7

3
5
```

Inilah contoh soal: bilangan prima sampai 10, sebuah baris kosong, dan bilangan prima dari 3 sampai 5. Perhatikan bahwa 1 tidak dicetak, dan tidak ada baris kosong sesudah kasus terakhir. Program Python mencetak baris-baris yang sama.

Menguji dan mengukur waktu. *Brute force*-nya, `brute_prime1.py` di folder `tests`, adalah pembagian percobaan, memadai untuk rentang pendek. Generatornya membuat sampai empat rentang, di dekat 1, tempat kasus tepi 1, 2, dan 3 berada, atau tepat di bawah 10^9, tempat bilangan besar berada, masing-masing paling lebar beberapa ratus bilangan agar pembagian percobaan tetap cepat.

```
$ python3 stress.py gen_prime1.py brute_prime1.py ../cpp/prime1.cpp 300
$ python3 stress.py gen_prime1.py brute_prime1.py ../python/prime1.py 200
```

```
300 rounds: all answers equal
200 rounds: all answers equal
```

Untuk waktunya, `make_prime1_worst.py` menulis masukan terbesar yang diizinkan batasan: sepuluh rentang dari 999 900 000 sampai 10^9. Kita mengukur waktu kedua program dengan perintah `time` dari *shell*, yang padanannya di Windows PowerShell adalah `Measure-Command { ... }`:

```
$ python3 make_prime1_worst.py
$ time ../cpp/prime1 < prime1_worst.in > out_cpp.txt
$ time python3 ../python/prime1.py < prime1_worst.in > out_py.txt
```

```
wrote prime1_worst.in
real    0m0.020s
real    0m0.188s
```

Program C++ memerlukan 0,02 detik dan program Python 0,19 detik untuk kasus terburuk, dan keluaran keduanya identik: 48.320 bilangan prima, 4.832 di setiap rentang, dari 999 900 017 sampai 999 999 937. Saringan bersegmen mengubah soal yang tidak dapat diselesaikan tepat waktu oleh pembagian percobaan menjadi soal yang sudah selesai sebelum interpreter benar-benar menyala.

3.7 Isyarat dalam Pernyataan Soal

Ukuran masukan adalah isyarat terkuat, tetapi bukan satu-satunya. Bentuk masukan dan pertanyaan yang diajukan juga menunjuk ke keluarga-keluarga metode, dan **Tabel 3.2** mengumpulkan isyarat-isyarat yang dikembangkan di sisa buku ini. Perlakukan tabel itu sebagai daftar tebakan pertama, yang diperiksa terhadap anggaran, bukan sebagai aturan.

Tabel 3.2 Isyarat dalam pernyataan soal dan keluarga metode yang biasanya ditunjuknya, dengan bab buku ini yang membahas setiap keluarga.

Isyarat dalam pernyataan soal	Keluarga yang dicoba lebih dulu	Bab
"banyaknya cara", "biaya minimum", pilihan yang saling tumpang-tindih	pemrograman dinamis	9
pilihan lokal yang tidak pernah perlu dibatalkan; "jadwal", "pilih sebanyak mungkin"	*greedy*, dengan argumen pertukaran	10
"x terbesar sehingga …", uji ya/tidak yang monoton terhadap x	pencarian biner pada jawaban	11
kota dan jalan, ketergantungan, grid dengan dinding	graf: BFS, DFS, lintasan terpendek	12
bilangan prima, pembagi, "modulo $10^9 + 7$", pangkat raksasa	teori bilangan	13
titik, ruas garis, luas	geometri	13
teks dan pola, palindrom	algoritma string	13
banyak kueri pada larik yang berubah	struktur data: jumlah prefiks, *segment tree*	6, 11

Di bab ini kita bertemu baris teori bilangan dua kali: FCTRL memerlukan rumus pencacahan faktor prima, dan LASTDIG pangkat raksasa modulo bilangan kecil. PRIME1 menggabungkan isyarat teori bilangan (bilangan prima) dengan isyarat ukuran (rentang pendek dari bilangan-bilangan besar), dan gabungan itu menunjuk ke saringan bersegmen.

3.8 Memakainya Sungguh-sungguh

Perkirakan sebelum menulis kode. Untuk setiap soal, tuliskan di buku catatan Anda, di sebelah penceritaan ulang dari Bab 2, masukan terbesar, banyaknya kasus uji, batas waktu, anggaran menurut Persamaan (3.1), dan banyak langkah metode yang Anda rencanakan menurut Persamaan (3.2). Jika metodenya tidak muat, jangan mulai menulisnya; kembalilah ke **Tabel 3.1** dan tanyakan apa yang diisyaratkan batasannya. Lima menit aritmetika menghemat satu jam mengoptimasi metode yang memang tidak pernah bisa lolos.

Berhitunglah dalam pangkat sepuluh. Perkiraan hanya memerlukan satu angka penting, dan tiga fakta membuatnya cepat: $2^{10} \approx 10^3$, sehingga $2^{20} \approx 10^6$ dan $2^{30} \approx 10^9$; karena itu $\log_2 10^3 \approx 10$, $\log_2 10^6 \approx 20$, dan $\log_2 10^9 \approx 30$. Selain itu $\sqrt{10^{2k}} = 10^k$, sehingga $\sqrt{10^9}$ terletak antara 10^4 dan 10^5, sekitar 3×10^4. Dengan fakta-fakta ini, $n\log_2 n$ untuk $n = 10^6$ sekitar 2×10^7 dan untuk $n = 10^5$ sekitar 1.7×10^6, terhitung di kepala dalam beberapa detik. Tuliskan perkiraan sebagai hasil kali pangkat-pangkat sepuluh, jumlahkan eksponennya, lalu bandingkan dengan 8, eksponen anggaran: eksponen 10 berarti seratus kali terlalu lambat, berapa pun konstantanya.

Selalu tulis uji maksimum. Generator untuk masukan sah terbesar, seperti `make_fctrl_tests.py` atau `make_prime1_worst.py`, hanya beberapa baris dan menjawab pertanyaan "apakah cukup cepat?" dengan pengukuran alih-alih harapan. Ukurlah di mesin Anda, dan sisakan faktor keamanan dua sampai lima untuk juri.

Simpan program yang lambat. Program yang benar tetapi lambat tidak sia-sia: program itu adalah *brute force* untuk *stress test* Anda. Gelung gamblang FCTRL, pembagian percobaan untuk PRIME1, dan `pow` Python untuk LASTDIG semuanya berperan seperti itu di bab ini.

Di luar juri. Aritmetika yang sama berlaku untuk setiap program yang bertemu data sungguhan. Sebelum menjalankan gelung bersarang atas dua tabel pelanggan, kalikan ukurannya; sebelum memanggil sebuah layanan web sekali per rekaman, kalikan banyaknya rekaman dengan waktu tanggapnya. Tugas pelaporan di sebuah koperasi yang membandingkan setiap anggota dengan

setiap anggota lain bersifat kuadratik: baik-baik saja untuk 300 anggota, tak berdaya untuk 300.000. Perbaikannya sama seperti di bab ini: urutkan, beri indeks, atau temukan rumus.

3.9 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

- **Melupakan banyaknya kasus uji.** Metode yang memerlukan 10^4 langkah per kasus baik-baik saja untuk satu kasus dan terlalu lambat untuk 10^5 kasus. Kalikan, dengan Persamaan (3.2).
- **Memercayai waktu di laptop Anda.** Mesin juri mungkin dua atau tiga kali lebih lambat. Sisakan faktor keamanan.
- **Mengoptimasi hal yang salah.** Gelung yang lebih cepat tidak menyelamatkan algoritma yang sejuta kali melampaui anggaran. Ganti algoritmanya lebih dulu; setel belakangan, jika perlu.
- ***Overflow* pada nilai antara.** `p * p`, `power *= 5`, `result * base`: hitunglah dalam `long long`, atau reduksi modulo sebelum mengalikan.
- **Mencetak 1 sebagai prima, atau melupakan 2.** Ujilah rentang yang dimulai dari 1, 2, dan 3.
- **Baris kosong sesudah setiap kasus.** PRIME1 meminta satu baris kosong *di antara* kasus. Cetaklah sebelum setiap kasus kecuali yang pertama.
- **Gelung Python padahal ada gelung C.** Mencoret dengan gelung `for` Python itu lambat; penugasan irisan, `sum`, `sort`, `pow`, dan `str.join` berjalan dalam C.
- ***Brute force* yang gagal sendiri.** *Brute force* FCTRL menabrak batas 4300 digit Python. Jalankan *stress test* dan bacalah kegagalan pertamanya dengan saksama sebelum menyalahkan program cepat.

3.10 Ringkasan Bab

- Anggaran sebuah soal adalah batas waktunya dikali langkah per detik (Persamaan (3.1)): terukur di sini sekitar 5×10^8 putaran gelung sederhana per detik dalam C++ dan 4×10^6 dalam Python (**Kode 3.1**, **Kode 3.2**); rencanakan dengan 10^8 dan 10^6 sampai 10^7.
- Banyak langkah sebuah algoritma mengikuti tiga aturan: gelung mengalikan, membagi dua menghasilkan logaritma, kasus uji dijumlahkan, dan semua pasangan memerlukan $n(n-1)/2$ (Persamaan (3.3)); hasilnya dibandingkan dengan anggaran pada masukan terbesar (Persamaan (3.2)).
- **Gambar 3.1** dan **Tabel 3.1** mengubah n terbesar menjadi kompleksitas paling lambat yang muat; jika dibaca terbalik, batasan mengisyaratkan metode yang dimaksud.
- FCTRL menghitung nol di ujung $n!$ dengan rumus Legendre (Persamaan (3.4)) dalam sekitar 13 pembagian per kasus; gelung gamblangnya benar tetapi dihentikan juri kita di batas waktu.
- LASTDIG menghitung $a^b \bmod 10$ dengan pengkuadratan berulang (Persamaan (3.5)) dalam paling banyak sekitar 62 perkalian.
- PRIME1 memakai saringan bersegmen: bilangan prima dasar sampai $\sqrt{n}$ (Persamaan (3.6)), masing-masing mencoret kelipatannya di dalam rentang mulai dari titik awal Persamaan (3.7); kasus terburuknya memerlukan 0,02 detik dalam C++ dan 0,19 detik dalam Python.
- Isyarat dalam pernyataan soal menunjuk ke keluarga metode (**Tabel 3.2**); isyarat itu tebakan pertama, yang diperiksa terhadap anggaran.

3.11 Soal Latihan

1. Sebuah soal punya $T \leq 10$ kasus uji, masing-masing larik berisi $n \leq 2 \times 10^5$ bilangan, dan batas waktu 1 detik. Untuk setiap $O(n^2)$, $O(n\log n)$, dan $O(n)$, hitung total langkahnya dengan Persamaan (3.2) dan putuskan apakah muat.
2. Hitung $Z(1000)$ dengan Persamaan (3.4) secara manual, lalu periksa hasilnya dengan `fctrl.cpp`.
3. Berapa banyak perkalian yang dikerjakan `powerMod` untuk $b = 2\,147\,483\,000$? Hitung banyaknya bit b dan berapa di antaranya bernilai satu.
4. Tunjukkan bahwa digit terakhir a^b untuk $b \geq 1$ hanya bergantung pada digit terakhir a dan pada b mod 4, lalu pakai fakta ini untuk menghitung digit terakhir 7^{2023} dengan tangan.
5. Jalankan saringan bersegmen dengan tangan pada rentang [90,110], dengan mendaftar bilangan prima dasar mana mencoret bilangan mana, lalu sebutkan bilangan primanya.
6. Ubah `prime1.cpp` sehingga untuk setiap rentang hanya mencetak banyaknya bilangan prima di dalamnya. Berapa jawabannya untuk rentang $[1, 10^5]$? (Nilai yang terkenal adalah 9.592.)

3.12 Penyelesaian Soal Latihan

1. Dengan $T = 10$ dan $n = 2 \times 10^5$: $O(n^2)$ memberikan $10 \cdot 4 \times 10^{10} = 4 \times 10^{11}$ langkah, empat ribu kali anggaran 10^8: tidak. $O(n\log n)$ memberikan $10 \cdot 2 \times 10^5 \cdot 18 \approx 3.6 \times 10^7$: ya, dengan faktor keamanan sekitar tiga. $O(n)$ memberikan 2×10^6: dengan mudah.

2. $\lfloor 1000/5 \rfloor = 200$, $\lfloor 1000/25 \rfloor = 40$, $\lfloor 1000/125 \rfloor = 8$, $\lfloor 1000/625 \rfloor = 1$, dan $\lfloor 1000/3125 \rfloor = 0$. Jadi $Z(1000) = 200 + 40 + 8 + 1 = 249$. Program mencetak 249 untuk masukan `1` diikuti `1000`.

3. $2^{30} = 1\,073\,741\,824 \leq b < 2^{31}$, sehingga b punya 31 bit dan gelung berjalan 31 kali, dengan satu pengkuadratan setiap kali: 31 pengkuadratan. Banyaknya perkalian ke dalam hasil sama dengan banyaknya bit 1 pada b. Tuliskan $b = (2^{31} - 1) - 647$. Bilangan $2^{31} - 1$ adalah 31 angka satu dalam biner, dan mengurangkan $647 = 1010000111_2$ darinya hanya mengubah kelima bit 1 dari 647 menjadi nol, karena tidak ada peminjaman jika setiap bitnya 1. Jadi b punya $31 - 5 = 26$ bit bernilai 1, dan ada 26 perkalian ke dalam hasil: total 57 perkalian, dan paling banyak 62 untuk pangkat mana pun dalam jangkauan.

4. Digit terakhir sebuah hasil kali hanya bergantung pada digit terakhir faktor-faktornya, sehingga mengganti a dengan a mod 10 tidak mengubah jawaban. Untuk setiap digit terakhir d, barisan $d^1, d^2, d^3, \ldots$ mod 10 berulang dengan periode 1, 2, atau 4: untuk 7 barisannya 7,9,3,1,7,9,3,1, ...; untuk 2 barisannya 2,4,8,6, ...; untuk 0, 1, 5, 6 konstan; untuk 4 dan 9 berperiode 2; untuk 3 dan 8 berperiode 4. Dalam setiap kasus periodenya membagi 4, sehingga untuk $b \geq 1$ jawabannya adalah unsur ke-r dari siklus itu, dengan $r = ((b - 1) \bmod 4) + 1$. Untuk 7^{2023}: $2022 \bmod 4 = 2$, sehingga $r = 3$ dan jawabannya unsur ketiga dari 7,9,3,1: digit 3. (Kasus $b = 0$ istimewa: jawabannya 1.)

5. $\sqrt{110} < 11$, sehingga bilangan prima dasarnya 2, 3, 5, dan 7. Prima 2 mulai di 90 dan mencoret bilangan genap 90 sampai 110. Prima 3 mulai di $\lceil 90/3 \rceil \cdot 3 = 90$: 90, 93, 96, 99, 102, 105, 108. Prima 5 mulai di 90: 90, 95, 100, 105, 110. Prima 7 mulai di $\max(49, \lceil 90/7 \rceil \cdot 7) = 91$: 91, 98, 105. Bilangan yang tidak pernah tercoret adalah 97, 101, 103, 107, dan 109, bilangan prima antara 90 dan 110.

6. Di `primesBetween`, ganti penambahan setiap bilangan prima dengan sebuah pencacah, kembalikan hasil cacahannya, dan cetak per kasus. Untuk $[1, 10^5]$ program mencetak 9592, sesuai dengan nilai terkenal $\pi(10^5)$, banyaknya bilangan prima sampai 10^5.

Daftar Pustaka

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[2] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Singapore: Lulu, 2020.

[3] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

[4] SPOJ, "FCTRL — Factorial." [Online]. Available: https://www.spoj.com/problems/FCTRL/ (accessed Sep. 2026).

[5] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1994, Sec. 4.4, "Factorial factors."

[6] SPOJ, "LASTDIG — The last digit." [Online]. Available: https://www.spoj.com/problems/LASTDIG/ (accessed Sep. 2026).

[7] SPOJ, "PRIME1 — Prime Generator." [Online]. Available: https://www.spoj.com/problems/PRIME1/ (accessed Sep. 2026).

Bab 4

Input dan Output Cepat

Program dengan algoritma yang benar masih bisa melampaui batas waktu jika detik-detiknya habis untuk membaca masukan dan menulis keluaran. Dengan sepuluh juta bilangan yang harus dibaca, selisih antara cara membaca yang paling lambat dan yang paling cepat mencapai sepuluh kali lipat dalam C++ dan lebih besar lagi dalam Python, dan di juri selisih itu adalah selisih antara TLE dan AC. Bab ini menjelaskan ke mana masukan dan keluaran menghabiskan waktunya dan mengukur setiap metode yang umum pada lima juta bilangan yang sama. Di akhir bab Anda akan mampu menjelaskan apa itu penyangga (*buffer*), panggilan sistem (*system call*), sinkronisasi dengan I/O standar C, dan aliran yang terikat (*tied stream*), serta mengapa masing-masing memakan waktu; memilih cara membaca dan menulis yang tepat dalam C++ dan Python untuk sejumlah data tertentu; menulis pembaca kecil berbasis `fread` ketika tidak ada cara lain yang cukup cepat; menghindari dua jebakan klasik, mencampur `cin` dengan `scanf` setelah mematikan sinkronisasi dan mengosongkan penyangga dengan `endl` di setiap baris; serta menyelesaikan dua uji I/O dari SPOJ, INTEST dan INOUTEST, dalam kedua bahasa.

4.1 Satu Perjalanan per Paket

Bayangkan seorang kurir yang harus mengantarkan dua ratus paket dari sebuah gudang ke satu gedung perkantoran di seberang kota. Kurir yang membawa satu paket, menyeberangi kota, mengantarkannya, lalu kembali untuk paket berikutnya menghabiskan seminggu penuh di jalan, meskipun setiap pengantaran hanya semenit. Kurir yang memuat kedua ratus paket ke dalam sebuah mobil boks cukup melakukan satu perjalanan. Pekerjaan di gedung itu sama; yang berbeda adalah banyaknya perjalanan, dan di perjalanan itulah waktu habis.

Masukan dan keluaran bekerja dengan cara yang sama. Membaca sebuah bilangan dari masukan atau menulis sebuah bilangan ke keluaran itu murah; *mengambil* masukan dari sistem operasi atau *menyerahkan* keluaran kepadanya itu mahal, karena setiap permintaan seperti itu, sebuah *panggilan sistem*, meninggalkan program, memasuki *kernel* sistem operasi, lalu kembali. I/O cepat adalah soal melakukan sedikit perjalanan dengan mobil yang penuh: mengambil dan menyerahkan blok-blok besar sekaligus, dan mengerjakan pekerjaan murah per bilangan di dalam program.

Tim Rina mempelajari hal ini dari INTEST, soal SPOJ yang satu-satunya tujuannya menguji kecepatan membaca. Program mereka hanya sebuah gelung dengan satu pembagian, algoritma yang sesederhana dan secepat mungkin. Program itu membaca bilangan dengan `std::cin` biasa, dan mendapat *time limit exceeded*. Dua baris di awal `main`, yang dijelaskan bab ini, membuatnya lolos dengan waktu yang berlebih.

Siapa yang memerlukan ini di luar juri? Siapa pun yang programnya mengolah berkas teks besar: catatan setiap transaksi sebuah koperasi selama setahun, ekspor CSV dari sistem kasir, keluaran sensor yang menulis satu baris setiap detik. Program yang sama bisa memerlukan satu menit atau sepuluh menit untuk berkas seperti itu, hanya bergantung pada cara ia membaca dan menulis, dan aturan-aturan bab ini berlaku tanpa perubahan.

4.2 Ke Mana Waktu Pergi

Di antara `std::cin >> x` Anda dan bita-bita di berkas masukan terdapat tiga lapisan, dan masing-masing dapat memakan waktu.

Penyangga. Program tidak meminta satu karakter sekaligus kepada sistem operasi. Pustaka menyimpan sebuah *penyangga* (*buffer*), sebuah blok memori berukuran beberapa kilobita; ketika penyangga kosong, pustaka meminta satu blok utuh kepada sistem operasi dengan satu panggilan sistem, lalu melayani karakter dari memori sampai blok itu habis. Keluaran bekerja sebaliknya: karakter dikumpulkan dalam penyangga dan diserahkan kepada sistem operasi dalam satu panggilan sistem ketika penyangga penuh, ketika program berakhir, atau ketika program sengaja *mengosongkannya* (*flush*). Dengan penyangga, sejuta bilangan memerlukan beberapa ratus panggilan sistem alih-alih jutaan. Karena itu waktu sebuah program yang berat di I/O kira-kira

$$t_{\text{I/O}} \approx c_{\text{bita}} \cdot \text{bita} + c_{\text{panggilan}} \cdot \text{panggilan sistem} \tag{4.1}$$

dengan $c_{\text{panggilan}}$, biaya satu panggilan sistem, ratusan atau ribuan kali lebih besar daripada c_{bita}, biaya menangani satu bita di memori. Persamaan (4.1) menjelaskan hampir setiap pengukuran di bab ini: metode yang lambat melakukan banyak panggilan sistem, atau banyak bekerja per bita, atau keduanya.

Sinkronisasi dengan I/O standar C. C++ mewarisi dari C keluarga fungsi I/O yang kedua, `scanf` dan `printf`, dengan penyangganya sendiri. Agar sebuah program boleh mencampur kedua keluarga dengan bebas, membaca satu bilangan dengan `scanf` dan bilangan berikutnya dengan `std::cin`, standar C++ secara bawaan mewajibkan aliran C++ *tersinkronisasi* dengan aliran C [1]. Dalam praktik, pustaka GCC mewujudkannya dengan tidak memberi `std::cin` dan `std::cout` penyangga sendiri dan meneruskan setiap karakter melalui pustaka C, yang memakan satu pemanggilan fungsi dan satu penguncian per karakter. Pemanggilan `std::ios::sync_with_stdio(false)` melepaskan hak mencampur kedua keluarga dan, sebagai gantinya, membiarkan aliran C++ memakai penyangganya sendiri yang cepat.

Aliran yang terikat. Secara bawaan `std::cin` *terikat* (*tied*) pada `std::cout`: sebelum setiap operasi masukan, `std::cout` dikosongkan [2]. Alasannya adalah interaksi: program yang mencetak `Your name?` lalu membaca harus memastikan pertanyaannya tampil di layar sebelum menunggu. Di juri tidak ada yang menonton, dan dengan ikatan itu setiap pembacaan bilangan mengosongkan keluaran, mungkin dengan satu panggilan sistem setiap kali. `std::cin.tie(nullptr)` memutus ikatannya.

Kurir dari Subbab 4.1 menjelaskan ketiga biaya itu. Sinkronisasi adalah aturan bahwa setiap paket harus lebih dulu didaftarkan di kantor pos pusat, bahkan jika semua paket menuju gedung yang sama. Ikatan adalah aturan bahwa kurir harus kembali ke gudang dan melapor setelah setiap pengantaran, berjaga-jaga jika ada yang menunggu kabar di sana. Penyangga adalah mobil boksnya. Mematikan sinkronisasi dan memutus ikatan menghapus kedua aturan itu bagi kurir yang bekerja sendirian, dan mobil boks lalu dapat menjalankan tugasnya. Tidak ada aturan yang bodoh: masing-masing melindungi keadaan yang memang terjadi, program yang mencampur I/O C dan C++, atau seseorang yang menunggu di depan papan ketik sebuah ajakan untuk mengetik. Juri hanya bukan keadaan seperti itu, dan program yang ditulis untuk juri boleh melepaskan perlindungan-perlindungan itu.

Mengosongkan dengan sengaja. `std::endl` menulis pindah baris *dan mengosongkan* penyangga keluaran [3]. Mencetak sejuta baris dengan `std::endl` berarti sejuta panggilan sistem, sejuta perjalanan sang kurir. `'\n'` hanya menulis pindah baris.

Python punya lapisan yang sama dengan nama lain. `input()` membaca satu baris melalui `sys.stdin`, sebuah aliran teks, dan mengerjakan pekerjaan tambahan setiap kali dipanggil: memeriksa apakah masukan berasal dari terminal interaktif, membuang pindah baris, dan menjalankan mesin `input()`. `sys.stdin.readline` melewati sebagian besar pekerjaan itu. `sys.stdin.buffer` adalah aliran bita di bawah aliran teks, dan `sys.stdin.buffer.read()` mengambil seluruh masukan dalam satu pemanggilan sebagai bita, tanpa mengubahnya menjadi teks [4]. Pada keluaran, `print()` mengubah argumennya, menambahkan pemisah dan pindah baris, lalu menulis, pekerjaan yang lumayan banyak per baris; mengumpulkan semua baris menjadi satu string dengan `str.join` lalu menulisnya sekali mengerjakan pekerjaan itu di dalam C.

Hitungan kasar memperlihatkan besarnya pengaruh itu. Masukan terbesar INTEST berukuran sekitar 110 MB. Dibaca melalui penyangga 64 KiB, itu berarti $110 \times 10^6 / 65\,536 \approx 1700$ panggilan sistem, yang dengan beberapa mikrodetik masing-masing memakan jauh di bawah seperseratus detik: dengan penyangga, suku kedua Persamaan (4.1) lenyap dan yang tersisa hanya pekerjaan per bita. Sekarang tulislah sepuluh juta baris, masing-masing dikosongkan dengan `std::endl`: sepuluh juta panggilan sistem, dan bahkan dengan satu mikrodetik masing-masing, itu sepuluh detik yang hanya berisi perjalanan ke sistem operasi. Program yang sama, algoritma yang sama, keluaran yang sama, dan selisih seribu kali dalam banyaknya perjalanan.

4.2.1 Apa yang sebenarnya dikerjakan satu `std::cin >> x`

Ada gunanya melihat betapa banyak yang terjadi di balik `std::cin >> x` yang tampak polos untuk bilangan bulat `x`. Mula-mula aliran membuat objek penjaga kecil, *sentry*, yang mengosongkan aliran yang terikat padanya jika ada dan melewati *whitespace* di depan. Lalu aliran menyerahkan konversi yang sebenarnya kepada sebuah *facet* dari *locale* aliran itu, objek yang mengetahui konvensi sebuah bahasa dan wilayah, misalnya karakter mana yang menjadi titik desimal dan apakah digit dikelompokkan dengan koma. *Facet* itu membaca karakter satu per satu melalui penyangga aliran, mengumpulkannya, mencocokkannya dengan format bilangan, mengubahnya, memeriksa *overflow*, dan menyalakan bendera galat aliran jika ada yang salah. Akhirnya nilainya disimpan di `x`. Setiap langkah itu masuk akal dan umum, dan setiap langkah memakan sedikit biaya; jika sinkronisasi menyala, setiap karakter juga melewati pustaka C. Untuk lima juta bilangan sepuluh digit, itu lima puluh juta karakter, masing-masing membayar semuanya.

Pembaca buatan kita sendiri di Subbab 4.3 hanya mengerjakan yang paling minimal: mengambil bita berikutnya dari sebuah larik, membandingkannya dengan `'0'` dan `'9'`, lalu mengalikan dengan sepuluh. Pembaca itu tidak tahu apa-apa tentang *locale*, *overflow*, atau bendera galat, yang dapat diterima hanya karena masukan juri dijamin berbentuk benar. Pertukaran itu, keumuman demi kecepatan, adalah seluruh rahasia pembaca buatan sendiri.

4.3 Mengukur Cara-cara Membaca

Klaim tentang kecepatan harus diukur. Kita memakai masukan berbentuk INTEST (Subbab 4.6): baris pertama berisi n dan k, lalu n bilangan bulat positif sampai 10^9, satu per baris. Tugasnya

menghitung berapa banyak bilangan yang habis dibagi k, yang hampir tidak memerlukan kerja, sehingga waktunya hampir seluruhnya untuk membaca. **Kode 4.1** menulis masukan seperti itu.

Kode 4.1 `gen_intest.py`: menulis masukan berbentuk INTEST dengan n bilangan acak.

```
"""gen_intest.py N [SEED]: masukan berbentuk INTEST dengan N bilangan ke stdout."""
import random
import sys


def main():
    n = int(sys.argv[1])
    rng = random.Random(int(sys.argv[2]) if len(sys.argv) > 2 else 1)
    k = rng.randint(1, 50)
    lines = [f"{n} {k}"]
    lines += [str(rng.randint(1, 10**9)) for _ in range(n)]
    sys.stdout.write("\n".join(lines) + "\n")


if __name__ == "__main__":
    main()
```

Kode 4.2 membaca masukan itu dengan empat cara, yang dipilih melalui angka di baris perintah.

Kode 4.2 `read_bench.cpp`: empat cara membaca masukan yang sama dalam C++, dari `cin` biasa sampai pembaca buatan sendiri di atas `fread`.

```
// read_bench.cpp METHOD: membaca masukan berbentuk INTEST dengan satu dari empat
// cara. 1 = cin apa adanya, 2 = cin tanpa tie dan tanpa sinkronisasi, 3 = scanf,
// 4 = pembaca buatan sendiri di atas fread. Mencetak banyaknya t_i habis dibagi k.
#include <cstdio>
#include <cstdlib>
#include <iostream>

static char buf[1 << 16];
static size_t bufLen = 0, bufPos = 0;

int readByte() {
    if (bufPos == bufLen) {
        bufLen = std::fread(buf, 1, sizeof buf, stdin);  // isi ulang 64 KiB
        bufPos = 0;
        if (bufLen == 0) return EOF;
    }
    return buf[bufPos++];
}

long long readInt() {  // lewati spasi, baca '-' bila ada, lalu digit
    int c = readByte();
    while (c == ' ' || c == '\n' || c == '\r' || c == '\t') c = readByte();
    bool negative = (c == '-');
    if (negative) c = readByte();
    long long x = 0;
    while (c >= '0' && c <= '9') {
        x = x * 10 + (c - '0');
        c = readByte();
    }
    return negative ? -x : x;
}

int main(int argc, char* argv[]) {
    int method = argc > 1 ? std::atoi(argv[1]) : 2;
    long long n = 0, k = 1, t = 0, count = 0;
    if (method == 1 || method == 2) {
        if (method == 2) {
            std::ios::sync_with_stdio(false);
            std::cin.tie(nullptr);
        }
        std::cin >> n >> k;
        for (long long i = 0; i < n; i++) {
            std::cin >> t;
            if (t % k == 0) count++;
        }
    } else if (method == 3) {
```

Kode 4.2 (Lanjutan) *read_bench.cpp: empat cara membaca masukan yang sama dalam C++, dari cin biasa sampai pembaca buatan sendiri di atas fread.*

```
        if (std::scanf("%lld %lld", &n, &k) != 2) return 1;
        for (long long i = 0; i < n; i++) {
            if (std::scanf("%lld", &t) != 1) return 1;
            if (t % k == 0) count++;
        }
    } else {
        n = readInt();
        k = readInt();
        for (long long i = 0; i < n; i++) {
            if (readInt() % k == 0) count++;
        }
    }
    std::printf("%lld\n", count);
    return 0;
}
```

Metode 1 dan 2 adalah kode yang sama kecuali dua baris yang mematikan sinkronisasi dan memutus ikatan. Metode 3 memakai `scanf`, yang mengurai string format `"%lld"` setiap kali dipanggil; nilai kembaliannya, banyaknya butir yang terbaca, diperiksa, yang sekaligus membungkam peringatan kompiler. Metode 4 adalah pembaca buatan kita sendiri. `readByte` melayani karakter dari larik 64 KiB `buf` dan mengisinya ulang dengan satu pemanggilan `fread` ketika kosong, persis penyangga dari Subbab 4.2 yang ditulis dengan tangan; `readInt` melewati *whitespace*, membaca tanda minus bila ada, dan membangun bilangan digit demi digit dengan $x \leftarrow 10x + d$, langkah yang sama seperti pada ADDREV di Bab 2. Pembaca itu tidak mengerjakan apa pun selain itu: tanpa *locale*, tanpa string format, tanpa penanganan galat di luar yang diperlukan masukan juri yang berbentuk benar.

Untuk melihat `readInt` bekerja, telusuri fungsi itu pada karakter-karakter `-307` diikuti pindah baris, yang tersimpan di `buf`:

Tabel 4.1 Pembaca `readInt` dari **Kode 4.2** pada masukan `-307`, satu karakter per baris tabel.

Karakter yang dibaca	Tindakan	`negative`	`x`
spasi	dilewati sebagai *whitespace*		
`-`	menyalakan tanda, membaca karakter berikutnya	true	0
`3`	$x \leftarrow 10 \cdot 0 + 3$	true	3
`0`	$x \leftarrow 10 \cdot 3 + 0$	true	30
`7`	$x \leftarrow 10 \cdot 30 + 7$	true	307
pindah baris	bukan digit: gelung berakhir	true	307

Fungsi itu mengembalikan -307. Pindah baris yang mengakhiri bilangan sudah terpakai, yang tidak merugikan, karena pemanggilan berikutnya toh melewati *whitespace*. Tiga perbandingan dan satu kali kali-tambah per digit: hanya itu.

Kode 4.3 melakukan hal yang sama dalam Python dengan tiga metode.

Kode 4.3 `read_bench.py`: tiga cara membaca masukan yang sama dalam Python.

```
"""read_bench.py METHOD: membaca masukan berbentuk INTEST dengan tiga cara.

1 = input() per baris, 2 = sys.stdin.readline per baris,
3 = sys.stdin.buffer.read().split() sekali. Mencetak banyaknya t_i habis dibagi k.
"""
import sys


def main():
    method = int(sys.argv[1]) if len(sys.argv) > 1 else 3
    count = 0
    if method == 1:
        n, k = map(int, input().split())
        for _ in range(n):
            if int(input()) % k == 0:
                count += 1
    elif method == 2:
        readline = sys.stdin.readline  # cari nama ini sekali, bukan n kali
        n, k = map(int, readline().split())
        for _ in range(n):
            if int(readline()) % k == 0:
                count += 1
    else:
        data = sys.stdin.buffer.read().split()  # semua token, sebagai bytes
        n, k = int(data[0]), int(data[1])
        for token in data[2:2 + n]:
            if int(token) % k == 0:
                count += 1
    print(count)


if __name__ == "__main__":
    main()
```

Metode 2 menyimpan `sys.stdin.readline` ke sebuah nama lokal sekali saja; mencari `sys`, lalu `stdin`, lalu `readline` lima juta kali saja sudah terukur dalam Python. Metode 3 membaca semuanya sekaligus lalu memecahnya menjadi token; `int()` menerima string bita yang dihasilkan `split()`, sehingga tidak perlu mengubahnya menjadi teks.

Skrip `bench_io.py` di folder `tests` bab ini membangkitkan lima juta bilangan, separuh maksimum INTEST, menjalankan setiap metode tiga kali dengan masukan diambil dari berkas dan keluaran dikirim ke berkas, seperti di juri, lalu menyimpan waktu terbaik dari ketiganya. Skrip itu juga memeriksa bahwa semua pembaca mencetak hitungan yang sama. Pada mesin yang dipakai untuk buku ini, sebuah server bersama yang saat itu juga menjalankan pekerjaan lain, sehingga semua angka agak lebih lambat dan lebih bergejolak daripada di laptop yang sedang menganggur:

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o ../cpp/read_bench ../cpp/read_bench.cpp
$ g++ -std=c++17 -O2 -Wall -Wextra -o ../cpp/write_bench ../cpp/write_bench.cpp
$ python3 bench_io.py
```

```
5,000,000 numbers; best of 3 runs
  read   C++ cin, default                 4.28 s
  read   C++ cin, unsynced + untied       0.59 s
  read   C++ scanf                        0.80 s
  read   C++ fread reader                 0.31 s
  read   Python input()                   6.11 s
  read   Python readline                  1.06 s
  read   Python buffer.read().split()     1.26 s
  write  C++ cout << endl                 4.11 s
  write  C++ cout << '\n'                 0.47 s
  write  C++ printf                       0.56 s
  write  C++ fwrite buffer                0.15 s
  write  Python print()                   1.98 s
  write  Python write per line            1.29 s
  write  Python one join                  1.38 s
all readers agree: True
```

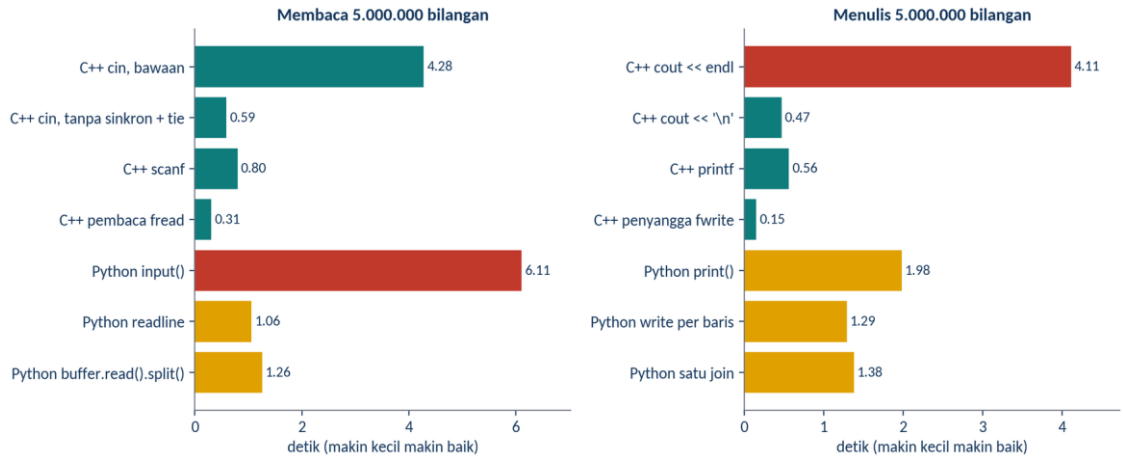
4.3.1 Biaya memori membaca semuanya

Kecepatan bukan satu-satunya biaya. Membaca seluruh masukan sekaligus menyimpan semuanya di memori, dan dalam Python setiap token menjadi objek `bytes` tersendiri dengan kepalanya sendiri: `sys.getsizeof(b"123456789")` bernilai 42 bita untuk sembilan karakter, ditambah delapan bita untuk penunjuknya di dalam list. Kita mengukur memori puncak pembaca-pembaca Python pada masukan lima juta bilangan, sebuah berkas berukuran 49 MB:

```
237925
method 2 max RSS MB 12
237925
method 3 max RSS MB 320
```

Membaca baris demi baris dengan `readline` memerlukan 12 MB, karena setiap baris dibuang setelah dipakai; membaca semuanya lalu memecahnya memerlukan 320 MB, lebih dari enam kali ukuran berkasnya. Pada ukuran penuh INTEST angka itu menjadi sekitar 650 MB. Banyak juri mengizinkan sebesar itu, tetapi sebagian soal punya batas 64 atau 256 MB, dan di sana `readline` adalah pilihan yang aman. Dalam C++, pembaca `fread` kita hanya memerlukan penyangga 64 KiB-nya, berapa pun ukuran masukannya. Aturan umumnya: membaca semuanya sekaligus itu nyaman dan cepat, tetapi periksalah batas memori jika masukannya lebih besar dari sekitar 10 MB.

Gambar 4.1 menggambarkan angka-angka yang sama. Metode paling lambat di setiap panel berwarna merah.



Gambar 4.1 Waktu terukur untuk membaca dan menulis lima juta bilangan, terbaik dari tiga kali eksekusi, C++ berwarna hijau kebiruan dan Python kuning tua, metode paling lambat berwarna merah. Waktu mutlaknya bergantung pada mesin; yang penting adalah perbandingannya.

Bacalah panel membaca dengan Persamaan (4.1) dalam pikiran.

- `cin` **bawaan: 4,28 s.** Setiap karakter melewati jalur tersinkronisasi, satu pemanggilan fungsi per karakter. Untuk 10^7 bilangan penuh INTEST itu berarti lebih dari 8 detik di mesin ini: TLE di juri mana pun.
- `cin` **tanpa sinkronisasi dan tanpa ikatan: 0,59 s,** tujuh kali lebih cepat, dari dua baris kode. Inilah metode yang dipakai secara bawaan.
- `scanf`**: 0,80 s.** Berpenyangga sejak awal, tetapi menafsirkan string format setiap kali dipanggil.
- **Pembaca** `fread` **kita: 0,31 s,** dua kali lebih cepat lagi, karena mengerjakan paling sedikit pekerjaan per bita.
- **Python** `input()`**: 6,11 s.** Lebih lambat daripada metode C++ yang paling lambat, dan jauh terlalu lambat untuk 10^7 baris.
- **Python** `readline`**: 1,06 s,** dan `buffer.read().split()`**: 1,26 s.** Keduanya sekitar enam kali lebih cepat daripada `input()`. Pada masukan ini, dengan satu bilangan per baris, membaca per baris kebetulan sedikit lebih cepat daripada memecah semuanya sekaligus, karena `split()` membangun list berisi lima juta string bita di memori sebelum gelung dimulai; pada masukan dengan banyak bilangan per baris, metode membaca semuanya lebih nyaman dan biasanya lebih cepat.

Mengapa `scanf` di sini lebih lambat daripada `cin` tanpa sinkronisasi, padahal keduanya berpenyangga? Keduanya melakukan sedikit panggilan sistem, sehingga suku kedua Persamaan (4.1) kecil bagi keduanya; perbedaannya ada pada suku pertama, pekerjaan per bita. `scanf` harus membaca dan menafsirkan string formatnya setiap kali dipanggil, dan melewati mesin umum pustaka C untuk setiap bilangan. `cin` tanpa sinkronisasi sudah mengetahui tipe sasarannya saat kompilasi dan melewati penguraian format. Pembaca `fread` kita mengerjakan lebih sedikit lagi per bita, dan menang. Pada kompiler atau pustaka lain urutan `scanf` dan `cin` bisa terbalik; yang dapat dibawa ke mana-mana adalah metodenya, bukan peringkatnya: ukurlah, lalu jelaskan hasil pengukuran dengan kedua suku Persamaan (4.1).

> **PERINGATAN:** Setelah `std::ios::sync_with_stdio(false)`, jangan pernah mencampur `std::cin`/`std::cout` dengan `scanf`/`printf` (atau `getchar`, `puts`) dalam program yang sama. Kedua keluarga kini punya penyangga terpisah, dan bilangan yang sudah ditarik satu keluarga ke penyangganya tidak terlihat oleh keluarga yang lain: masukan terlewati atau terbaca dua

> kali, keluaran muncul tidak berurutan, dan tidak ada kompiler yang memperingatkan Anda. Pilih satu keluarga per program.

4.4 Mengukur Cara-cara Menulis

Kode 4.4 menulis bilangan 1 sampai *N*, satu per baris, dengan empat cara.

Kode 4.4 `write_bench.cpp`: empat cara menulis keluaran yang sama dalam C++.

```cpp
// write_bench.cpp METHOD N: mencetak bilangan 1..N, satu per baris, dengan satu
// dari empat cara. 1 = cout dengan endl, 2 = cout dengan '\n' (tanpa sinkron),
// 3 = printf, 4 = penyangga buatan sendiri yang ditulis dengan fwrite.
#include <cstdio>
#include <cstdlib>
#include <iostream>

static char out[1 << 16];
static size_t outLen = 0;

void flushOut() {
    std::fwrite(out, 1, outLen, stdout);
    outLen = 0;
}

void writeInt(long long x) {  // di sini x >= 0; digit dibuat dari kanan ke kiri
    char digits[24];
    int len = 0;
    do {
        digits[len++] = static_cast<char>('0' + x % 10);
        x /= 10;
    } while (x > 0);
    if (outLen + len + 1 > sizeof out) flushOut();
    while (len > 0) out[outLen++] = digits[--len];
    out[outLen++] = '\n';
}

int main(int argc, char* argv[]) {
    int method = argc > 1 ? std::atoi(argv[1]) : 2;
    long long n = argc > 2 ? std::atoll(argv[2]) : 10;
    if (method == 1) {
        for (long long i = 1; i <= n; i++) std::cout << i << std::endl;
    } else if (method == 2) {
        std::ios::sync_with_stdio(false);
        for (long long i = 1; i <= n; i++) std::cout << i << '\n';
    } else if (method == 3) {
        for (long long i = 1; i <= n; i++) std::printf("%lld\n", i);
    } else {
        for (long long i = 1; i <= n; i++) writeInt(i);
        flushOut();
    }
    return 0;
}
```

`writeInt` adalah pasangan keluaran dari `readInt`: fungsi ini menghasilkan digit x dari kanan ke kiri ke dalam larik kecil dengan `x % 10` dan `x / 10`, menyalinnya dalam urutan yang benar ke penyangga keluaran 64 KiB, dan menyerahkan penyangga itu kepada sistem operasi dengan satu `fwrite` ketika hampir penuh; `flushOut` di akhir menulis sisanya. Fungsi ini hanya menangani bilangan tak negatif, yang sudah cukup bagi program ini; Soal Latihan 4 menambahkan tanda minus.

Kode 4.5 `write_bench.py`: tiga cara menulis keluaran yang sama dalam Python.

```
"""write_bench.py METHOD N: mencetak 1..N, satu per baris, dengan tiga cara.

1 = print() per baris, 2 = sys.stdout.write per baris, 3 = satu join dan satu write.
"""
import sys


def main():
    method = int(sys.argv[1]) if len(sys.argv) > 1 else 3
    n = int(sys.argv[2]) if len(sys.argv) > 2 else 10
    if method == 1:
        for i in range(1, n + 1):
            print(i)
    elif method == 2:
        write = sys.stdout.write
        for i in range(1, n + 1):
            write(f"{i}\n")
    else:
        sys.stdout.write("\n".join(map(str, range(1, n + 1))) + "\n")


if __name__ == "__main__":
    main()
```

Panel menulis di **Gambar 4.1** menceritakan kisah `std::endl`: 4,11 s dibanding 0,47 s dengan `'\n'`, hampir sembilan kali lebih lambat, karena selisih satu karakter di kode sumber. Dengan keluaran menuju berkas, setiap pengosongan adalah satu panggilan sistem, lima juta kali. Penyangga buatan sendiri paling cepat dengan 0,15 s. Dalam Python, `print()` memerlukan 1,98 s, dan menulis setiap baris dengan `sys.stdout.write` 1,29 s. Menggabungkan semua baris menjadi satu string lalu menulisnya sekali memerlukan 1,38 s di sini, sedikit lebih lama daripada menulis per baris, karena string gabungan berukuran sekitar 39 megabita harus dibangun di memori lebih dulu; di mesin yang tidak sesibuk ini keduanya biasanya berdekatan, dan penggabungan selalu jelas mengalahkan `print()`. Program-program Bab 3 dan solusi di bawah mengumpulkan keluarannya lalu menulisnya sekali.

4.5 Aturan-aturannya

Hasil pengukuran itu dapat diringkas menjadi daftar pendek yang mencakup hampir setiap soal.

C++.

1. Awali `main` dengan `std::ios::sync_with_stdio(false);` dan `std::cin.tie(nullptr);` setiap kali masukan atau keluaran berisi lebih dari sekitar 10^5 bilangan. Biayanya nol jika masukannya kecil, sehingga banyak peserta menuliskannya di setiap program.
2. Akhiri baris dengan `'\n'`, jangan pernah dengan `std::endl`.
3. Setelah aturan 1, pakai hanya `std::cin` dan `std::cout`. Jika Anda lebih suka `scanf` dan `printf`, lewati aturan 1 dan pakai hanya keduanya.
4. Untuk lebih dari sekitar 10^7 bilangan, atau batas waktu yang ketat bagi algoritmanya, pakai pembaca `fread` seperti dalam **Kode 4.2**, dan kumpulkan keluaran dalam penyangga atau `std::string` yang ditulis sekali.

Python.

1. Jangan pernah memakai `input()` untuk lebih dari sekitar 10^5 baris.
2. Bacalah dengan `data = sys.stdin.buffer.read().split()` lalu telusuri `data` dengan sebuah indeks, atau dengan `readline = sys.stdin.readline` jika struktur baris penting.

3. Kumpulkan jawaban dalam sebuah list lalu tulis `"\n".join(answers)` sekali, atau pakai `sys.stdout.write`; hindari `print()` dalam gelung untuk keluaran besar.
4. `print(*answers, sep="\n")` adalah alternatif satu baris yang nyaman untuk penggabungan; bentuk ini mengubah setiap jawaban di dalam satu pemanggilan, jauh lebih baik daripada `print()` dalam gelung, meskipun membangun string gabungan sendiri memberi Anda kendali lebih atas pindah baris terakhir.
5. Biarkan pekerjaan per bilangan dikerjakan C: `map(int, ...)`, `sum(...)`, penugasan irisan, `sort`.

> **WAWASAN KUNCI:** I/O cepat bukan pengganti algoritma yang benar. Anggaran Bab 3 didahulukan: metode $O(n^2)$ tidak lolos karena membaca dengan cepat. I/O cepat penting ketika algoritmanya sudah benar dan masukan atau keluarannya besar, dan saat itu pengaruhnya bisa sepuluh kali lipat. Peringatan terkenal bahwa optimasi prematur adalah akar banyak kejahatan [5] juga berlaku untuk pembaca buatan sendiri: pakailah dua baris aturan 1 selalu, dan pembaca `fread` hanya jika pengukuran memintanya.

Mencampur token dan baris. Sebagian masukan mencampur bilangan dan baris teks utuh, misalnya sebuah banyaknya diikuti nama-nama yang mengandung spasi. Dalam C++, `std::cin >> n` membaca bilangan dan meninggalkan pindah baris sesudahnya di penyangga; `std::getline(std::cin, line)` berikutnya lalu membaca *sisa baris itu*, yang kosong, alih-alih baris berikutnya. Obatnya adalah membuang sisa baris setelah membaca bilangan, dengan `std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n')` dari *header* `<limits>`, atau cukup membaca setiap baris dengan `std::getline` lalu mengubah bilangannya sendiri dengan `std::stoi`. Dalam Python masalah ini tidak muncul dengan `readline`, yang selalu mengembalikan baris utuh; dengan metode membaca semuanya, pakai `sys.stdin.buffer.read().split(b"\n")` untuk memecah menjadi baris, bukan token, jika spasi di dalam baris penting.

Soal interaktif adalah pengecualiannya. Sebagian kecil soal bersifat *interaktif*: program mengajukan pertanyaan, program juri menjawab, dan pertanyaan berikutnya bergantung pada jawaban itu. Di sini pertanyaan yang masih tertahan di penyangga dan belum dikosongkan tidak pernah sampai ke juri, dan kedua program saling menunggu selamanya, yang dilaporkan juri sebagai TLE. Dalam soal seperti itu, kosongkan penyangga setelah setiap pertanyaan: `std::cout << question << std::endl;` dalam C++ (satu-satunya pemakaian `endl` yang sah), atau `print(question, flush=True)` dalam Python.

4.6 INTEST: Masukan Raksasa

Soalnya (SPOJ INTEST, “Enormous Input Test”, `https://www.spoj.com/problems/INTEST/`) [6]. Baris pertama memuat dua bilangan bulat positif n dan k, keduanya paling besar 10^7. Lalu menyusul n baris, masing-masing berisi bilangan bulat positif $t_i \le 10^9$. Cetak berapa banyak t_i yang habis dibagi k. Pernyataan soal terang-terangan menyebut bahwa soal ini menguji kecepatan masukan, dan bahwa program semestinya mampu memproses paling sedikit 2,5 MB masukan per detik saat berjalan.

Membaca batasannya. n sampai 10^7 baris berisi sampai 10 digit: sampai sekitar 110 MB masukan. Algoritmanya satu lintasan dengan satu pembagian per bilangan, $O(n)$, 10^7 langkah: bukan apa-apa. Semuanya ada pada pembacaan. Menurut pengukuran di Subbab 4.3, membaca 10^7 bilangan memerlukan lebih dari 8 detik dengan `cin` bawaan di mesin kita, sekitar 1,2 detik dengan `cin` tanpa sinkronisasi, dan sekitar 0,6 detik dengan pembaca `fread`. Solusi C++-nya memakai pembaca `fread`.

Kode 4.6 `intest.cpp`: INTEST dengan pembaca bilangan tak negatif berbasis `fread`.

```cpp
// SPOJ INTEST (Enormous Input Test): hitung t_i yang habis dibagi k, baca cepat.
#include <cstdio>

static char buf[1 << 16];
static size_t bufLen = 0, bufPos = 0;

int readByte() {
    if (bufPos == bufLen) {
        bufLen = std::fread(buf, 1, sizeof buf, stdin);
        bufPos = 0;
        if (bufLen == 0) return EOF;
    }
    return buf[bufPos++];
}

long long readUnsigned() {  // INTEST tidak memuat bilangan negatif
    int c = readByte();
    while (c != EOF && (c < '0' || c > '9')) c = readByte();
    long long x = 0;
    while (c >= '0' && c <= '9') {
        x = x * 10 + (c - '0');
        c = readByte();
    }
    return x;
}

int main() {
    long long n = readUnsigned(), k = readUnsigned();
    long long count = 0;
    for (long long i = 0; i < n; i++) {
        if (readUnsigned() % k == 0) count++;
    }
    std::printf("%lld\n", count);
    return 0;
}
```

Pembaca ini sedikit lebih sederhana daripada yang ada di **Kode 4.2**: INTEST tidak memuat bilangan negatif, sehingga `readUnsigned` melewati setiap karakter yang bukan digit, yang sekaligus membuatnya kebal terhadap karakter `\r` dari akhir baris gaya Windows dan terhadap pemisah lain apa pun. Pembaca ini berhenti di akhir berkas (`EOF`), sehingga pindah baris akhir yang hilang tidak merugikan.

Kode 4.7 `intest.py`: INTEST dalam Python, dengan membaca semua token sekaligus.

```python
"""SPOJ INTEST (Enormous Input Test): hitung t_i yang habis dibagi k, baca cepat."""
import sys


def main():
    data = sys.stdin.buffer.read().split()
    n, k = int(data[0]), int(data[1])
    print(sum(1 for token in data[2:2 + n] if int(token) % k == 0))


if __name__ == "__main__":
    main()
```

Versi Python membaca semua token sekaligus dan menghitung dengan ekspresi generator di dalam `sum`. Pada contoh soal, tujuh bilangan dengan $k = 3$ yang empat di antaranya habis dibagi:

```
$ printf '7 3\n1\n51\n966369\n7\n9\n999996\n11\n' | ./cpp/intest
4
```

Program Python juga mencetak `4`. Pada masukan lima juta bilangan dari Subbab 4.3, yang jawabannya juga dihitung oleh `brute_intest.py` yang sederhana (membaca dengan `input()`):

```
$ cd tests
$ time ../cpp/intest < big_intest.in
$ time python3 ../python/intest.py < big_intest.in
$ python3 brute_intest.py < big_intest.in
```

```
237925
real    0m0.324s
237925
real    0m1.496s
237925
```

Ketiganya sepakat tentang hitungannya. Program C++ memerlukan 0,32 detik untuk lima juta bilangan, jadi sekitar 0,65 detik untuk maksimum INTEST sepuluh juta, jauh di dalam anggaran. Program Python memerlukan 1,5 detik, jadi sekitar 3 detik untuk maksimumnya, yang bisa muat atau tidak, bergantung pada batas waktu juri dan kecepatan Python-nya yang lebih tua; inilah salah satu soal yang membuat Python berada di tepi jurang, dan C++ dengan I/O cepat adalah pilihan yang aman.

Letakkan angka 2,5 MB per detik dari pernyataan soal di samping pengukuran-pengukuran ini. Berkas lima juta bilangan berukuran 49 MB; program C++ membacanya dalam 0,32 detik, sekitar 150 MB per detik, enam puluh kali persyaratan yang disebutkan, dan program Python dalam 1,5 detik, sekitar 33 MB per detik. `cin` bawaan di Subbab 4.3 hanya mencapai sekitar 11 MB per detik pada berkas yang sama. Persyaratan itu terdengar sederhana, tetapi ditulis untuk mesin juri bertahun-tahun yang lalu, dan batas waktu soalnya disetel sedemikian rupa sehingga hanya metode yang cermat yang lolos. Mengubah waktu menjadi megabita per detik, seperti di sini, adalah cara cepat membandingkan hasil pengukuran Anda dengan pernyataan seperti itu.

Apa yang diukur jam juri. Waktu yang kita ukur adalah waktu *jam dinding* (*wall-clock time*): waktu dari mulai sampai selesai seperti yang ditunjukkan jam di dinding, termasuk waktu program menunggu cakram atau sistem operasi. Banyak juri justru mengukur *waktu CPU*: hanya waktu yang benar-benar dipakai prosesor untuk menjalankan program. Untuk program yang berhitung, keduanya hampir sama; untuk program yang berat di I/O keduanya bisa berbeda, dan di mesin juri yang sibuk waktu jam dinding bisa jauh lebih besar daripada waktu CPU. Inilah satu alasan lagi untuk menyisakan faktor keamanan, dan hal ini menjelaskan mengapa kiriman yang sama dapat menunjukkan waktu yang sedikit berbeda pada hari yang berbeda.

4.7 INOUTEST: Masukan dan Keluaran Raksasa

Soalnya (SPOJ INOUTEST, "Enormous Input and Output Test", `https://www.spoj.com/problems/INOUTEST/`) [7]. Baris pertama memuat N, berorde sejuta. Setiap dari N baris berikutnya memuat dua bilangan bulat, yang boleh negatif. Untuk setiap pasangan, cetak hasil kalinya di barisnya sendiri. Kini kedua arah besar: sejuta baris masuk, sejuta baris keluar.

Membaca batasannya. Nilai-nilainya cukup kecil sehingga hasil kalinya mudah muat dalam 64 bit; kita tetap menghitung dalam `long long`, karena hasil kali dua bilangan adalah tempat klasik tipe 32-bit mengalami *overflow*. Bilangan negatif diizinkan, sehingga pembaca untuk soal ini harus menangani tanda minus, demikian pula penulisnya. Solusi C++-nya memakai aliran tanpa sinkronisasi untuk masukan, yang cukup cepat untuk dua juta bilangan, dan membangun seluruh keluaran dalam satu `std::string`, yang ditulis dengan satu `<<`.

Ukuran keluaran layak diperkirakan seperti ukuran masukan. Dalam uji buatan kita, faktor-faktornya mencapai 65 536 dalam nilai mutlak, seperti pada baris ketiga uji kecil di bawah, sehingga sejuta hasil kali terdiri atas sampai sepuluh digit dan satu tanda, sehingga keluarannya sampai sekitar 12 MB, dan dengan satu `std::endl` per baris akan memakan sejuta panggilan sistem. Mengumpulkannya dalam satu string memakan 12 MB memori, jauh di dalam batas mana pun, dan satu penulisan besar. Perkiraan yang sama untuk soal dengan sepuluh juta baris keluaran akan menghasilkan 120 MB, dan saat itu menulis per blok melalui penyangga, seperti yang dilakukan `writeInt`, lebih baik daripada menampung semuanya sekaligus.

Kode 4.8 `inoutest.cpp`: INOUTEST dengan aliran tanpa sinkronisasi dan keluaran yang dikumpulkan dalam satu string.

```cpp
// SPOJ INOUTEST (Enormous Input and Output Test): cetak a * b untuk tiap pasang.
#include <iostream>
#include <string>

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n;
    std::cin >> n;
    std::string out;
    out.reserve(static_cast<size_t>(n) * 12);  // ruang untuk n baris pendek
    for (int i = 0; i < n; i++) {
        long long a, b;
        std::cin >> a >> b;
        out += std::to_string(a * b);
        out += '\n';
    }
    std::cout << out;  // satu penulisan besar, bukan n yang kecil
    return 0;
}
```

Pemanggilan `out.reserve(...)` mengalokasikan ruang untuk sekitar dua belas karakter per baris di muka, agar string itu tidak perlu tumbuh dan disalin berulang-ulang ketika baris-baris ditambahkan; ini optimasi kenyamanan, bukan keharusan.

Kode 4.9 `inoutest.py`: INOUTEST dalam Python, dengan membaca semua token dan menggabungkan semua hasil kali.

```python
"""SPOJ INOUTEST (Enormous Input and Output Test): cetak a * b untuk tiap pasang."""
import sys


def main():
    data = sys.stdin.buffer.read().split()
    n = int(data[0])
    products = (int(data[1 + 2 * i]) * int(data[2 + 2 * i]) for i in range(n))
    sys.stdout.write("\n".join(map(str, products)) + "\n")


if __name__ == "__main__":
    main()
```

Ekspresi generator `products` menghitung hasil kali satu per satu ketika `join` memakainya, sehingga program tidak pernah menyimpan list kedua berisi sejuta bilangan. Uji kecil dengan bilangan negatif dan hasil kali besar:

```
$ printf '3\n2 3\n-4 5\n65536 -65536\n' | ./cpp/inoutest
```

```
6
-20
-4294967296
```

Hasil kali ketiga, -2^{32}, tidak muat dalam `int` 32-bit: hasil kali `int` akan mencetak `0`, *overflow* diam-diam yang memberi WA hanya pada uji besar. Program Python mencetak ketiga baris yang sama. Pada sejuta pasangan acak buatan `gen_inoutest.py`, kedua program dan `brute_inoutest.py` yang sederhana (dengan `input()` dan `print()`) menulis berkas yang identik:

```
$ python3 gen_inoutest.py 1000000 3 > big_inoutest.in
$ time ../cpp/inoutest < big_inoutest.in > o1.txt
$ time python3 ../python/inoutest.py < big_inoutest.in > o2.txt
$ python3 brute_inoutest.py < big_inoutest.in > o3.txt
$ cmp o1.txt o3.txt && cmp o2.txt o3.txt && echo "all three outputs identical"
```

```
real    0m0.359s
real    0m0.788s
all three outputs identical
```

Sejuta baris masuk dan keluar dalam 0,36 detik dalam C++ dan 0,79 detik dalam Python. Di sini Python lolos dengan nyaman: masukannya sepersepuluh INTEST, dan metode membaca semuanya serta penggabungan menjaga pekerjaan per baris tetap di dalam C.

4.8 Catatan Portabilitas

Beberapa detail berbeda antara sistem dan antara juri, dan detail itu menggigit pada saat yang paling buruk.

- **`getchar_unlocked`.** Banyak templat pembacaan cepat di internet memakai `getchar_unlocked`, fungsi POSIX yang membaca karakter tanpa penguncian. Fungsi itu tidak ada di pustaka C Windows, sehingga templat seperti itu tidak dapat dikompilasi dengan MSYS2 di Windows. Pembaca `fread` di bab ini adalah C standar dan berjalan di mana saja, dengan kecepatan yang sama.
- **Akhir baris.** Berkas yang ditulis di Windows mengakhiri setiap baris dengan `\r\n`. Pembaca berbasis token (`std::cin >>`, `split()`, dan pembaca kita, yang melewati karakter bukan digit) mengabaikan `\r`. Program yang membaca baris utuh dengan `std::getline` lalu membandingkannya dengan sebuah string melihat `\r` itu dan gagal; buanglah karakter itu.
- **Pipa PowerShell.** `Get-Content file | program` di PowerShell membaca berkas sebagai teks, baris demi baris, dan mengodekannya ulang sebelum meneruskannya, yang lambat untuk berkas besar dan dapat mengubah bita-bitanya. Untuk pengukuran waktu dan untuk masukan besar, pakai `program < file` di Command Prompt, atau `bench_io.py` dan `minijudge.py` kita, yang menyerahkan berkas langsung ke program.
- **Mengukur waktu dengan *shell*.** `time` di Linux dan macOS, `Measure-Command { cmd /c "program < file" }` di PowerShell. Selalu ukur waktu dengan keluaran menuju berkas, bukan terminal: terminal menggambar setiap karakter di layar dan jauh lebih lambat daripada metode mana pun di bab ini.
- **Spasi di ujung baris dan pindah baris terakhir.** Kebanyakan juri, termasuk pemeriksa bawaan SPOJ, mengabaikan spasi di ujung baris dan pindah baris yang hilang di akhir keluaran, tetapi sebagian kecil pemeriksa khusus tidak. Kode yang mencetak `" ".join(values)` atau pindah baris setelah setiap baris menghasilkan keluaran yang bersih dan tidak pernah salah; kode yang mencetak spasi setelah setiap bilangan menggantungkan diri pada toleransi pemeriksa.

- **Mode tanpa penyangga di Python.** Jika variabel lingkungan `PYTHONUNBUFFERED` disetel, atau Python dijalankan dengan `-u`, keluaran tidak berpenyangga dan setiap `print()` menjadi panggilan sistem. Sebagian lingkungan pengembangan menyetelnya; eksekusi *benchmark* pertama kita terjadi di lingkungan seperti itu dan mengukur `print()` lebih dari delapan kali biaya sebenarnya. `bench_io.py` membuang variabel itu bagi program-program yang dijalankannya.

4.9 Memakainya Sungguh-sungguh

Dua templat. Simpanlah templat C++ yang diawali dua baris aturan 1 dan templat Python yang diawali `data = sys.stdin.buffer.read().split()` dan diakhiri satu `sys.stdout.write("\n".join(out) + "\n")`. Memulai setiap program dari templat itu membuat I/O cepat menjadi kebiasaan yang tidak perlu dipikirkan, dan menghapus seluruh kelas TLE yang dibahas bab ini. Simpan pembaca `fread` dari **Kode 4.6** di tempat yang sama, untuk soal-soal yang memerlukannya. Buku-buku latihan baku memberikan templat yang sangat mirip [8].

Daftar periksa sebelum mengirim program yang berat di I/O.

1. Hitunglah masukannya: banyaknya bilangan dikali banyaknya digit memberi banyaknya bita; di atas sekitar 1 MB, I/O cepat berpengaruh, di atas sekitar 10 MB, I/O cepat menentukan.
2. C++: dua baris aturan 1 ada, dan tidak ada `scanf` atau `printf` yang bersembunyi di mana pun dalam program, bahkan di baris untuk *debugging*.
3. Tidak ada `std::endl` di gelung mana pun; dalam Python, tidak ada `print()` di gelung atas keluaran.
4. Keluaran dikumpulkan dan ditulis sekaligus, atau paling tidak melalui aliran berpenyangga.
5. Untuk masukan yang sangat besar dalam Python, memori untuk membaca semuanya sekaligus, sekitar enam kali ukuran berkas, muat dalam batas memori.
6. Uji maksimum, yang dibuat oleh generator, berjalan di mesin Anda dalam waktu jauh di bawah separuh batas waktu, diukur dengan keluaran dialihkan ke berkas.

Enam pertanyaan, satu menit, dan seluruh keluarga vonis yang berakar pada I/O lenyap sebelum juri melihat programnya.

Ukurlah mesin Anda sendiri. Jalankan `bench_io.py` sekali di komputer Anda. Angka mutlaknya memberi tahu seberapa cepat mesin Anda dibanding mesin kita; perbandingannya akan serupa, dan melihat `std::endl` memakan waktu sembilan kali lipat di layar Anda sendiri lebih meyakinkan daripada buku mana pun.

Berkas besar di tempat kerja. Catatan transaksi tahunan sebuah koperasi, pembacaan sebuah sensor, ekspor dari mesin kasir sebuah toko: program yang mengolahnya adalah program I/O, dan aturan di Subbab 4.5 berlaku apa adanya. Bacalah dalam blok besar, pecah sekali, simpan pekerjaan per rekaman di bagian bahasa yang cepat, kumpulkan keluarannya, lalu tulis sekali. Skrip yang meringkas berkas CSV sejuta baris semestinya memerlukan hitungan detik, bukan menit; jika memerlukan menit, periksa cara ia membaca dan menulis sebelum memeriksa hal lain.

4.10 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

- **`std::endl` di dalam gelung.** Mengosongkan setiap baris; pakai `'\n'`. Sembilan kali lebih lambat dalam pengukuran kita.
- **Lupa `sync_with_stdio(false)`.** Membaca tujuh kali lebih lambat dengan GCC. Masukkan ke templat Anda.

- **Mencampur `cin` dan `scanf` setelah mematikan sinkronisasi.** Masukan hilang atau terduplikasi. Satu keluarga per program.
- **Mematikan sinkronisasi di soal interaktif dan tidak pernah mengosongkan penyangga.** Juri tidak pernah melihat pertanyaannya; kosongkan setelah setiap pertanyaan.
- **`input()` untuk sejuta baris di Python.** Enam kali lebih lambat daripada `readline` di sini. Bacalah semuanya sekaligus, atau pakai `readline`.
- **`print()` di dalam gelung untuk keluaran besar.** Kumpulkan lalu tulis sekali.
- **Hasil kali `int`.** $65536 \cdot 65536$ membuat `int` *overflow*. Pakai `long long` untuk hasil kali.
- **`getline` setelah `cin >>`.** Pindah baris yang tersisa membuat `getline` mengembalikan baris kosong. Buang sisa baris lebih dulu.
- **Membaca masukan raksasa sekaligus menjadi objek Python.** 320 MB untuk berkas 49 MB dalam pengukuran kita. Periksa batas memori, atau bacalah baris demi baris.
- **Pembaca tanpa tanda minus.** INOUTEST memuat bilangan negatif; pembaca yang mengabaikan `-` mengubah -4 menjadi 4 dan memberi WA.
- **Mengukur waktu dengan keluaran ke terminal.** Terminal itulah hambatannya. Alihkan keluaran ke berkas saat mengukur.

4.11 Ringkasan Bab

- Waktu I/O kira-kira sama dengan banyaknya bita dikali biaya kecil ditambah banyaknya panggilan sistem dikali biaya besar (Persamaan (4.1)); penyangga mengurangi banyaknya panggilan sistem, dan metode yang cepat mengerjakan sedikit pekerjaan per bita.
- Secara bawaan, `std::cin` dan `std::cout` GCC tersinkronisasi dengan I/O C dan `std::cin` terikat pada `std::cout`; `std::ios::sync_with_stdio(false)` dan `std::cin.tie(nullptr)` menghapus kedua biaya itu, setelah itu kedua keluarga I/O tidak boleh dicampur.
- Terukur pada lima juta bilangan (**Gambar 4.1**): membaca memerlukan 4,28 s dengan `cin` bawaan, 0,59 s tanpa sinkronisasi, 0,80 s dengan `scanf`, dan 0,31 s dengan pembaca `fread` (**Kode 4.2**); Python memerlukan 6,11 s dengan `input()` dan sekitar 1,1 sampai 1,3 s dengan `readline` atau membaca semuanya sekaligus (**Kode 4.3**).
- Menulis memerlukan 4,11 s dengan `std::endl` dan 0,47 s dengan `'\n'` (**Kode 4.4**); `print()` Python memerlukan 1,98 s dibanding sekitar 1,3 s untuk menulis string secara langsung (**Kode 4.5**).
- Membaca semuanya sekaligus memakan memori: 320 MB untuk masukan 49 MB dalam Python, dibanding 12 MB baris demi baris; periksa batas memori.
- INTEST diselesaikan dengan pembaca `fread` dalam C++ (**Kode 4.6**) dan dengan satu kali baca dan pecah dalam Python (**Kode 4.7**), dengan 0,32 s dan 1,5 s untuk lima juta bilangan; Python berada di tepi jurang pada ukuran penuh.
- INOUTEST mengumpulkan keluaran dalam satu string (**Kode 4.8**, **Kode 4.9**) dan menghitung hasil kali dalam 64 bit; sejuta baris memerlukan 0,36 s dalam C++ dan 0,79 s dalam Python.
- Soal interaktif harus mengosongkan penyangga setelah setiap pertanyaan; `getchar_unlocked` tidak portabel ke Windows; pengukuran waktu harus mengalihkan keluaran ke berkas.

4.12 Soal Latihan

1. Sebuah soal memuat 3×10^6 bilangan pada masukan dan batas waktu 1 detik. Dengan memakai pengukuran di Subbab 4.3, metode membaca C++ mana yang aman, mana yang berisiko, dan mana yang akan gagal? Jawab pertanyaan yang sama untuk Python.
2. Jelaskan, dengan Persamaan (4.1), mengapa menulis lima juta baris dengan `std::endl` lambat jika keluarannya menuju berkas, tetapi hampir tidak lebih lambat daripada `'\n'` jika program hanya menulis sepuluh baris.
3. Sebuah program memanggil `std::ios::sync_with_stdio(false)`, membaca n dengan `std::cin`, lalu membaca n bilangan dengan `scanf`. Jelaskan apa yang dapat salah, dan berikan dua cara memperbaiki program itu.
4. Perluas `writeInt` di **Kode 4.4** sehingga juga menulis bilangan negatif dengan benar. Hati-hati dengan `long long` yang paling negatif, -2^{63}, yang nilai mutlaknya tidak muat dalam `long long`.
5. Tulis ulang `inoutest.py` sehingga mencetak dengan `print()` di dalam gelung, jalankan pada sejuta pasangan dari Subbab 4.7, dan bandingkan waktunya dengan waktu **Kode 4.9**.
6. Dalam beberapa kalimat, jelaskan mengapa soal interaktif memerlukan pengosongan penyangga setelah setiap pertanyaan, dan vonis apa yang Anda dapat jika melupakannya.
7. Dengan memakai pengukuran memori di Subbab 4.3.1, perkirakan memori puncak `intest.py` pada masukan terbesar INTEST, lalu tulis ulang dengan `readline` sehingga hampir tidak memerlukan memori. Seberapa lebih lambat yang Anda perkirakan?
8. Telusuri `readInt` dari **Kode 4.2**, seperti di **Tabel 4.1**, pada masukan `\n\n42` (dua pindah baris, digit 4 dan 2, sebuah spasi). Apa yang dikembalikannya, dan karakter mana yang akan diserahkan `readByte` berikutnya?

4.13 Penyelesaian Soal Latihan

1. Menskalakan pengukuran lima juta dengan 3/5: `cin` bawaan sekitar 2,6 s, gagal; `cin` tanpa sinkronisasi sekitar 0,35 s, aman; `scanf` sekitar 0,5 s, aman tetapi dengan margin lebih kecil; pembaca `fread` sekitar 0,2 s, aman. Dalam Python, `input()` sekitar 3,7 s, gagal; `readline` atau membaca semuanya sekitar 0,65 sampai 0,75 s, berisiko di juri yang lebih lambat; pekerjaan pada setiap bilangan lalu masih harus ditambahkan.

2. Dengan `std::endl`, banyaknya panggilan sistem sama dengan banyaknya baris. Untuk lima juta baris, suku $c_{\text{panggilan}} \cdot$ panggilan sistem pada Persamaan (4.1) adalah lima juta kali biaya satu panggilan sistem, yang mengalahkan segala yang lain. Dengan sepuluh baris, itu sepuluh panggilan sistem, beberapa mikrodetik; selisihnya dengan `'\n'` tidak terlihat. `std::endl` tidak lambat dengan sendirinya; ia lambat jika dikalikan jutaan.

3. Setelah sinkronisasi dimatikan, `std::cin` punya penyangganya sendiri. Ketika membaca n, ia mengambil satu blok utuh masukan ke penyangga itu, yang memuat bukan hanya n tetapi juga banyak bilangan pertama. `scanf` membaca dari penyangga pustaka C, yang belum melihat bilangan-bilangan itu, sehingga ia mulai membaca setelah blok yang diambil `std::cin`: bilangan-bilangan pertama terlewati, dan program membaca nilai yang salah atau kehabisan masukan. Perbaikan 1: baca semuanya dengan `std::cin`. Perbaikan 2: hapus baris `sync_with_stdio(false)` dan baca semuanya dengan `scanf` (atau pertahankan sinkronisasi dan campurlah, dengan menerima `cin` yang lambat).

4. Tangani tanda lebih dulu, lalu bekerjalah dengan besarnya sebagai `unsigned long long`, yang dapat menampung 2^{63}:

```
void writeInt(long long x) {
    char digits[24];
    int len = 0;
    unsigned long long u = x < 0 ? 0ULL - static_cast<unsigned long long>(x)
                                 : static_cast<unsigned long long>(x);
    do {
        digits[len++] = static_cast<char>('0' + u % 10);
        u /= 10;
    } while (u > 0);
    if (x < 0) digits[len++] = '-';  // ditulis pertama, karena digit terbalik
    if (outLen + len + 1 > sizeof out) flushOut();
    while (len > 0) out[outLen++] = digits[--len];
    out[outLen++] = '\n';
}
```

Ekspresi `0ULL - static_cast<unsigned long long>(x)` menghitung besarnya dengan aritmetika tak bertanda, yang berputar modulo 2^{64} dan karena itu terdefinisi dengan baik bahkan untuk $x = -2^{63}$, sedangkan `-x` akan *overflow*.

5. Ganti dua baris terakhir `main` dengan `for i in range(n): print(int(data[1 + 2 * i]) * int(data[2 + 2 * i]))`. Di mesin buku ini, dengan penyangga Python yang normal, dua kali eksekusi masing-masing menghasilkan 0,74 s dan 0,98 s untuk versi `print()` dibanding 0,60 s dan 0,78 s untuk **Kode 4.9**: sekitar seperempat lebih lambat, dan keluarannya identik. Di sini selisihnya sedang karena membaca dan mengalikan menghabiskan banyak waktu; makin banyak baris yang dicetak program, makin mahal `print()`.

6. Dalam soal interaktif, program juri membaca pertanyaan sebelum dapat menjawab, dan program kita membaca jawaban sebelum dapat mengajukan pertanyaan berikutnya. Jika pertanyaan itu tertahan di penyangga keluaran kita, juri tidak pernah menerimanya dan menunggu; program kita menunggu jawabannya; keduanya tidak bergerak, sampai batas waktu habis. Vonisnya TLE (sebagian juri menyebutnya "idleness limit exceeded"), meskipun logika program bisa saja sempurna. Mengosongkan penyangga setelah setiap pertanyaan menyerahkannya kepada sistem operasi seketika.

7. Berkas lima juta bilangan berukuran 49 MB memerlukan sekitar 320 MB; masukan terbesar INTEST punya bilangan dua kali lebih banyak, jadi sekitar 650 MB. Dengan `readline`, program menyimpan satu baris saja setiap saat:

```
import sys

def main():
    readline = sys.stdin.readline
    n, k = map(int, readline().split())
    print(sum(1 for _ in range(n) if int(readline()) % k == 0))

main()
```

Memorinya tetap sekitar selusin megabita berapa pun n. Menurut Subbab 4.3 kecepatannya kurang lebih sama dengan membaca semuanya sekaligus, 1,06 s dibanding 1,26 s untuk lima juta bilangan di mesin kita, jadi pada bentuk masukan ini tidak ada yang hilang.

8. Kedua pindah baris dilewati sebagai *whitespace*; `4` memberi $x = 4$, `2` memberi $x = 42$; spasinya bukan digit dan mengakhiri gelung. Fungsi mengembalikan 42, dan spasinya sudah terpakai, sehingga karakter berikutnya yang diserahkan `readByte` adalah apa pun yang menyusul spasi itu, atau `EOF` jika tidak ada apa-apa.

Daftar Pustaka

[1] cppreference.com, "std::ios_base::sync_with_stdio." [Online]. Available: https://en.cppreference.com/w/cpp/io/ios_base/sync_with_stdio (accessed Sep. 2026).

[2] cppreference.com, "std::basic_ios::tie." [Online]. Available: https://en.cppreference.com/w/cpp/io/basic_ios/tie (accessed Sep. 2026).

[3] cppreference.com, "std::endl." [Online]. Available: https://en.cppreference.com/w/cpp/io/manip/endl (accessed Sep. 2026).

[4] Python Software Foundation, "sys — System-specific parameters and functions: sys.stdin, sys.stdout," *Python 3.14 Documentation*. [Online]. Available: https://docs.python.org/3/library/sys.html#sys.stdin (accessed Sep. 2026).

[5] D. E. Knuth, "Structured programming with go to statements," *ACM Computing Surveys*, vol. 6, no. 4, pp. 261–301, 1974.

[6] SPOJ, "INTEST — Enormous Input Test." [Online]. Available: https://www.spoj.com/problems/INTEST/ (accessed Sep. 2026).

[7] SPOJ, "INOUTEST — Enormous Input and Output Test." [Online]. Available: https://www.spoj.com/problems/INOUTEST/ (accessed Sep. 2026).

[8] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Singapore: Lulu, 2020.

Bab 5

Flag Kompiler, Pragma, dan Portabilitas

Program tidak dinilai di komputer Anda. Program dinilai di mesin yang tidak pernah Anda lihat, dibangun oleh kompiler yang tidak Anda pilih, dengan sakelar yang tidak Anda atur. Bab ini membahas celah itu. Di akhir bab Anda akan mampu menjelaskan dengan tepat apa yang terjadi antara saat Anda menekan *Submit* dan saat program Anda mulai membaca masukannya; memilih flag kompiler yang menemukan bug di rumah dan flag yang membuat kode cepat; membedakan pragma yang aman dari pragma yang untung-untungan, dan mengukur perbedaannya alih-alih mempercayai sebuah blog; mengenali segelintir fitur bahasa yang berperilaku lain di Windows, macOS, Linux, dan di juri; menulis program untuk soal SPOJ FCTRL yang benar di mana pun; serta menjalankan skrip kecil yang mengompilasi kode Anda dengan beberapa standar, kompiler, dan mode pemeriksaan sebelum Anda mengirimkannya.

5.1 Program yang Jalan di Rumah

Sinta duduk di kelas terakhir sebuah SMA di Malang dan sedang bersiap untuk Olimpiade Sains tingkat provinsi bidang informatika. Setiap malam selepas makan, ia menyelesaikan dua atau tiga soal SPOJ di laptop keluarga, sebuah mesin Windows yang juga dipakai adiknya untuk tugas sekolah. Temannya, Bima, yang berlatih bersamanya setiap akhir pekan, memakai MacBook. Mereka saling berbagi solusi lewat grup obrolan dan membandingkan kecepatannya.

Suatu Sabtu, keduanya menyelesaikan FCTRL, soal klasik SPOJ tentang banyaknya angka nol di ujung sebuah faktorial. Program Sinta diterima pada percobaan pertama. Ia mengirimkannya kepada Bima untuk dibandingkan, dan program itu bahkan tidak bisa dikompilasi di Mac milik Bima: baris pertamanya, `#include <bits/stdc++.h>`, menyebut berkas yang tidak dimiliki kompilernya. Bima memperbaiki baris include, dan program mencetak jawaban yang benar. Lalu ia menambahkan satu baris yang ditemukannya di sebuah blog pemrograman kompetitif, `#pragma GCC target("avx512f")`, yang katanya membuat loop lebih cepat, dan mengirimkan program itu ke SPOJ. Vonisnya *Runtime error (SIGILL)*. Sumber yang sama, benar di dua laptop, mogok di juri.

Minggu berikutnya, Sinta menulis program yang memakai `long` untuk nilai yang bisa mencapai sepuluh miliar. Di juri program itu diterima; di laptopnya program itu mencetak bilangan negatif. Ia menghabiskan satu malam mencari kesalahan logika yang sebenarnya tidak ada: di Windows, `long` berukuran 32 bit, sedangkan di juri yang memakai Linux berukuran 64 bit.

Tak satu pun dari tiga kegagalan ini berkaitan dengan algoritma. Algoritmanya selalu benar. Yang berbeda adalah *lingkungannya*: kompiler, versinya, sakelar yang diberikan kepadanya, sistem operasi, dan prosesor. Siapa pun yang menulis program untuk dijalankan di tempat lain akan bertemu celah ini, bukan hanya peserta lomba. Celah ini sama dengan yang dialami seorang juru masak ketika resep kue yang selalu berhasil di ovennya sendiri justru gosong di oven tetangga, karena oven tetangga dua puluh derajat lebih panas dan resepnya tidak pernah menyebutkan hal itu. Obatnya pun sama: ketahui bagian mana dari hasil Anda yang bergantung pada dapur Anda sendiri, tuliskan, dan cobalah di lebih dari satu dapur sebelum Anda mengandalkan resep itu.

Bab ini memberikan pengetahuan itu untuk C++ dan Python di SPOJ. Teorinya sedikit, eksperimennya banyak, karena setiap klaim tentang kompiler dapat diperiksa dalam beberapa detik, dan memang sebaiknya diperiksa. Semua eksperimen dijalankan pada September 2026 di sebuah mesin virtual Linux dengan dua inti prosesor Intel Xeon 2,1 GHz, dengan g++ 13.3, clang 18.1, dan CPython 3.11; program-programnya ditulis agar berperilaku sama dengan versi-versi yang lebih baru yang dipasang di Bab 1.

5.2 Gagasannya dengan Kata-Kata Sederhana

5.2.1 Dari teks sumber ke program yang berjalan

Berkas sumber C++ adalah teks. Prosesor tidak dapat menjalankan teks; ia menjalankan *kode mesin*, deretan instruksi bernomor seperti "jumlahkan dua register ini" atau "lompat mundur delapan byte jika perbandingan terakhir menyatakan lebih kecil". **Kompiler** menerjemahkan teks menjadi kode mesin satu kali, sebelum program berjalan. Python bekerja dengan cara lain: teks diterjemahkan ke bentuk antara setiap kali program dimulai, lalu **interpreter** menjalankannya langkah demi langkah. Di juri, kedua langkah itu terjadi di mesin milik juri.

Tabel 5.1 mendaftar apa yang terjadi setelah Anda mengirimkan program C++ ke SPOJ, dan langkah mana yang Anda kendalikan.

Tabel 5.1 Apa yang terjadi pada kiriman C++ di SPOJ, dan siapa yang mengendalikan setiap langkah.

Langkah	Apa yang terjadi	Siapa yang menentukan
1. Pilihan bahasa	Anda memilih entri seperti "C++14 (gcc 8.3)" dari sebuah menu	Anda
2. Kompilasi	Juri menjalankan kompilernya pada teks Anda dengan flag miliknya	Juri
3. Eksekusi	Kode mesin berjalan di prosesor juri, sekali per berkas uji	Perangkat keras juri
4. Batas	Waktu dan memori diukur; program dihentikan bila melampauinya	Pembuat soal
5. Perbandingan	Keluaran Anda dibandingkan dengan keluaran yang diharapkan	Pemeriksa milik juri

Hanya langkah pertama dan teks itu sendiri yang menjadi milik Anda. Semua yang dibahas bab ini adalah cara menulis teks yang selamat melewati langkah 2 dan 3 di mesin yang bukan milik Anda.

5.2.2 Flag, pragma, portabilitas

> **DEFINISI.** *Flag kompiler* adalah sakelar yang diberikan kepada kompiler di baris perintahnya, seperti `-O2` (optimasi) atau `-std=c++17` (gunakan standar C++ tahun 2017). *Pragma* adalah baris di dalam berkas sumber, diawali `#pragma`, yang meminta kompiler mengubah perilakunya mulai titik itu. Sebuah program *portabel* bila ia terkompilasi dan memberikan keluaran yang sama dengan setiap kompiler, sistem operasi, dan prosesor yang menjadi sasarannya. *Perilaku tak terdefinisi* (*undefined behavior*) adalah sebutan standar C++ untuk operasi, seperti luapan (*overflow*) bilangan bulat bertanda atau pembacaan melewati ujung larik, yang sama sekali tidak dijanjikan hasilnya oleh bahasa; kompiler boleh menganggap operasi itu tidak pernah terjadi.

Beda flag dan pragma terletak pada siapa yang menulisnya. Anda tidak dapat mengubah flag yang dipakai juri, tetapi Anda dapat menulis pragma, karena pragma adalah bagian dari teks Anda. Itulah sebabnya pragma populer dalam pemrograman kompetitif, dan itu pula sebabnya pragma berbahaya: pragma yang berkata kepada kompiler "prosesor ini punya instruksi X dan Y" adalah janji tentang mesin yang belum pernah Anda lihat.

5.2.3 Tiga jenis perbedaan

Ketika sebuah program berperilaku berbeda di rumah dan di juri, penyebabnya hampir selalu salah satu dari tiga hal ini.

1. **Bahasanya berbeda.** Kompiler Anda memahami C++20, kompiler juri C++14; pustaka standar Anda memiliki header yang tidak dimiliki juri; kompiler Anda menerima perluasan yang ditolak kompiler lain.
2. **Ukurannya berbeda.** `long` berukuran 32 bit di Windows dan 64 bit di Linux dan macOS; `char` bertanda di satu prosesor dan tak bertanda di prosesor lain; stack bawaan berukuran 1 MB di Windows dan 8 MB di kebanyakan sistem Linux.
3. **Program mengandung perilaku tak terdefinisi.** Luapan bilangan bertanda, variabel yang tidak diinisialisasi, atau indeks di luar jangkauan dapat memberikan jawaban benar dengan satu kompiler dan satu set flag, tetapi jawaban salah, atau tidak ada jawaban sama sekali, dengan yang lain. Jenis inilah yang paling licik, karena program *tampak* benar di mesin tempat Anda mengujinya.

Sisa bab ini membahas ketiga jenis itu satu per satu, pada satu soal nyata.

5.3 Soal yang Kita Bawa Sepanjang Bab: FCTRL

5.3.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ FCTRL, "Factorial" [1], mengajukan pertanyaan tentang bilangan $N! = 1 \times 2 \times \cdots \times N$. Masukan diawali sebuah bilangan T, banyaknya nilai uji, sekitar 100.000, lalu T bilangan bulat positif N, masing-masing paling besar 1.000.000.000. Untuk setiap N, program harus mencetak $Z(N)$, banyaknya angka nol di ujung bentuk desimal $N!$. Misalnya, $5! = 120$ berakhir dengan satu nol dan $10! = 3628800$ dengan dua. Batas waktu di halaman soal adalah 6 detik untuk seluruh berkas, dan batas memorinya 1536 MB, nilai bawaan di SPOJ.

Menghitung $N!$ itu sendiri mustahil: 1000! saja sudah 2568 digit, sedangkan N bisa mencapai satu miliar. Kita butuh cara menghitung nol tanpa membangun bilangannya.

5.3.2 Gagasannya: hitung faktor lima

Angka nol di ujung sebuah bilangan adalah faktor 10, dan $10 = 2 \times 5$. Jadi setiap nol di ujung $N!$ memerlukan satu faktor 2 dan satu faktor 5 dari hasil kali $1 \times 2 \times \cdots \times N$. Faktor 2 jauh lebih banyak daripada faktor 5 (setiap bilangan kedua genap, hanya setiap bilangan kelima kelipatan 5), sehingga banyaknya nol sama dengan banyaknya faktor 5 di dalam $N!$.

Berapa banyak faktor 5 itu? Setiap kelipatan 5 sampai N menyumbang paling sedikit satu; banyaknya $\lfloor N/5 \rfloor$, dengan $\lfloor x \rfloor$ berarti x dibulatkan ke bawah menjadi bilangan bulat. Setiap kelipatan 25 menyumbang faktor 5 *kedua*, dan banyaknya $\lfloor N/25 \rfloor$. Setiap kelipatan 125 menyumbang faktor ketiga, dan seterusnya. Menjumlahkan lapisan-lapisan itu menghasilkan rumus Legendre untuk bilangan prima 5 [2]:

$$Z(N) = \sum_{i \geq 1} \left\lfloor \frac{N}{5^i} \right\rfloor = \left\lfloor \frac{N}{5} \right\rfloor + \left\lfloor \frac{N}{25} \right\rfloor + \left\lfloor \frac{N}{125} \right\rfloor + \cdots \tag{5.1}$$

Jumlah itu tampak tak berujung, padahal tidak: begitu $5^i > N$, semua suku berikutnya bernilai nol.

Contoh yang dikerjakan dengan tangan, $N = 125$:

- kelipatan 5 sampai 125: $\lfloor 125/5 \rfloor = 25$;
- kelipatan 25: $\lfloor 125/25 \rfloor = 5$;
- kelipatan 125: $\lfloor 125/125 \rfloor = 1$;
- kelipatan 625: $\lfloor 125/625 \rfloor = 0$, dan semua suku sesudahnya juga 0.

Jadi $Z(125) = 25 + 5 + 1 = 31$: bilangan 125! berakhir dengan tepat 31 nol.

5.3.3 Mengapa program membagi, bukan mengalikan

Ada dua cara memprogram Persamaan (5.1). Cara yang paling jelas menyimpan pangkat $p = 5,25,125,\ldots$ dan menambahkan $\lfloor N/p \rfloor$ selama $p \le N$. Cara lainnya tidak pernah membentuk pangkat itu sama sekali. Cara ini bersandar pada satu fakta kecil tentang pembulatan ke bawah: membagi dengan 5 lalu membulatkan, kemudian membagi dengan 5 lagi dan membulatkan, sama dengan membagi dengan 25 dan membulatkan sekali,

$$\left\lfloor \frac{\lfloor N/5^i \rfloor}{5} \right\rfloor = \left\lfloor \frac{N}{5^{i+1}} \right\rfloor \tag{5.2}$$

(tuliskan $N = 5^{i+1}q + r$ dengan $0 \le r < 5^{i+1}$; kedua ruas sama dengan q). Jadi loop "bagi N dengan 5, tambahkan hasilnya, ulangi sampai N bernilai 0" menghasilkan tepat suku-suku Persamaan (5.1), satu demi satu, dan setiap bilangan yang disentuhnya paling besar N. Tidak ada yang bisa meluap, apa pun tipe N. Di Subbab 5.5 kita akan melihat bahwa loop yang lain, walau tampak tak berbahaya, terjun ke jurang di dekat puncak jangkauan `int`.

Berapa banyak kerjanya? Loop berjalan sekali untuk setiap suku yang tidak nol, yaitu $\lfloor \log_5 N \rfloor + 1$ kali; untuk $N = 10^9$ itu 13 pembagian. Dengan $T = 10^5$ nilai, seluruh berkas memerlukan sekitar

$$T \cdot (\lfloor \log_5 N \rfloor + 1) \le 10^5 \times 13 = 1.3 \times 10^6 \tag{5.3}$$

pembagian, yang praktis bukan apa-apa. Jawabannya pun kecil: setiap suku paling besar seperlima suku sebelumnya, sehingga $Z(N) < N/5 + N/25 + \cdots = N/4$, paling besar 250 juta, dan muat dalam `int` 32-bit. Kita tetap memakai `long long`, agar program tetap benar bila masukannya ternyata lebih besar daripada yang dijanjikan.

5.3.4 Programnya

Kode 5.1 adalah program C++ dan **Kode 5.2** program Python. Keduanya tidak memakai apa pun yang bergantung pada platform: hanya header standar, tanpa pragma, tipe yang berukuran 64 bit di mana pun, dan format `printf` `%lld` yang cocok dengannya.

Kode 5.1 `fctrl.cpp`: banyaknya nol di ujung $N!$, ditulis agar berperilaku sama di setiap platform.

```cpp
// FCTRL (SPOJ): banyak nol di belakang N! untuk banyak N.
// Portabel: hanya header standar, tanpa pragma, tanpa luapan untuk N < 2^31.
#include <cstdio>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        long long n;
        if (std::scanf("%lld", &n) != 1) break;
        long long zeros = 0;
        while (n > 0) {          // floor(N/5) + floor(N/25) + floor(N/125) + ...
            n /= 5;
            zeros += n;
        }
        std::printf("%lld\n", zeros);
    }
    return 0;
}
```

Kode 5.2 `fctrl.py`: hitungan yang sama dalam Python, dengan membaca seluruh masukan sekaligus.

```python
# FCTRL (SPOJ): banyak nol di belakang N! untuk banyak N.
# Ditulis untuk Python 3.8 ke atas: tanpa pernyataan match, tanpa sintaks baru.
import sys


def zeros(n):
    count = 0
    while n:
        n //= 5
        count += n
    return count


def main():
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    out = [str(zeros(int(x))) for x in data[1:1 + t]]
    sys.stdout.write("\n".join(out) + "\n")


if __name__ == "__main__":
    main()
```

Dua rincian program Python layak diberi catatan. Program membaca seluruh masukan dengan `sys.stdin.buffer.read()` lalu memecahnya menjadi token, cara sederhana tercepat untuk membaca banyak bilangan di Python (Bab 4 mengukur alternatif-alternatifnya), dan mengumpulkan semua jawaban dalam sebuah list lalu menuliskannya dengan satu panggilan. Program juga menghindari sintaks yang lebih baru daripada Python 3.8, seperti pernyataan `match`, karena Python di juri bisa lebih tua daripada milik Anda; Subbab 5.8 kembali membahas hal ini.

Kita memberi kedua program berkas uji buatan kita sendiri, dengan nilai-nilai yang dipilih agar mengenai setiap suku Persamaan (5.1) dan puncak jangkauannya:

```
6
7
25
125
1000
999999999
1000000000
```

Dikompilasi dengan `g++ -std=c++17 -O2 -Wall -Wextra -o fctrl fctrl.cpp` dan dijalankan sebagai `./fctrl < sample_fctrl.txt` (di Windows `.\fctrl.exe`), program C++ mencetak baris-baris berikut; `python3 fctrl.py < sample_fctrl.txt` mencetak baris yang persis sama.

```
1
6
31
249
249999989
249999998
```

Baris ketiga adalah hitungan tangan kita untuk $N = 125$. Baris kedua adalah $Z(25) = 5 + 1 = 6$: 25 adalah bilangan pertama yang menyumbang dua faktor lima. Pada berkas berisi 100.000 nilai acak sampai 10^9, masukan terbesar yang diizinkan soal, program C++ memerlukan 0,013 detik dan program Python 0,07 detik, keduanya jauh di dalam batas.

5.4 Apa yang Dilakukan Juri terhadap Sumber Anda

5.4.1 Kompiler di seberang sana

Saat Anda mengirim, menu bahasa SPOJ meminta Anda memilih di antara entri seperti "C++14 (gcc 8.3)" dan "C++ (gcc 8.3)"; arsip solusi yang mencatat entri yang dipilih menunjukkan gcc 8.3 sebagai kompiler di belakangnya [3]. GCC 8 dirilis pada 2018. Kompiler Anda sendiri hampir pasti lebih baru: GCC 16.1 terbit pada 30 April 2026, dan salah satu perubahan utamanya adalah bahwa perintah `g++` polos kini mengompilasi *C++20* secara bawaan, padahal versi-versi sebelumnya memakai C++17 [4]. Dua pelajaran mengikutinya.

Pertama, **selalu sebutkan standar yang Anda maksud**. Kompilasi di rumah dengan `-std=c++17` (atau apa pun yang disebut menu juri), jangan pernah dengan bawaan, karena bawaan adalah sasaran yang terus bergeser. Fitur yang terkompilasi tanpa keluhan di bawah bawaan C++20 milik GCC 16, seperti `std::ranges::sort` atau operator `<=>`, menjadi galat kompilasi di juri yang mengompilasi C++14 atau C++17.

Kedua, **lebih tua bukan berarti lebih buruk, hanya berbeda**. Setiap program di bab ini dan tiga bab berikutnya terkompilasi di bawah C++14 maupun C++17; pemeriksaan di Subbab 5.8 membuktikannya untuk program-program bab ini. Jika Anda memerlukan fitur yang lebih baru, periksa menu juri lebih dahulu.

Prosesornya juga berbeda. SPOJ menjalankan kiriman untuk kebanyakan soal di sebuah klaster yang dinamainya *Cube*, yang dibangun dari prosesor Intel Xeon E3-1220 v5 [5]. Prosesor rancangan tahun 2015 itu mendukung instruksi vektor AVX2 tetapi tidak AVX-512 [6]. Laptop Anda mungkin lebih tua, lebih baru, atau berjenis lain sama sekali: laptop Apple sejak 2020 memakai prosesor ARM yang tidak menjalankan satu pun instruksi itu. Ingat hal ini untuk Subbab 5.7.

5.4.2 Menanyai kompiler tentang dirinya

Cara tercepat untuk mengetahui fakta-fakta yang penting adalah menanyai kompilernya. **Kode 5.3** mencetak nama kompiler, standar bahasa yang dipakainya, ukuran tipe-tipe bilangan bulat, apakah `char` bertanda, dan apakah tipe perluasan 128-bit `__int128` tersedia.

Kode 5.3 `platform_probe.cpp`: mencetak fakta tentang kompiler dan mesin yang menentukan apakah sebuah program berperilaku sama di tempat lain.

```cpp
// Mencetak fakta tentang kompiler dan mesin ini yang menentukan apakah
// program yang ditulis di sini berperilaku sama di juri.
#include <climits>
#include <cstdio>

int main() {
    std::printf("compiler     : ");
#if defined(__clang__)
    std::printf("clang %d.%d\n", __clang_major__, __clang_minor__);
#elif defined(__GNUC__)
    std::printf("g++ %d.%d\n", __GNUC__, __GNUC_MINOR__);
#elif defined(_MSC_VER)
    std::printf("MSVC %d\n", _MSC_VER);
#else
    std::printf("unknown\n");
#endif
    std::printf("__cplusplus  : %ldL\n", (long)__cplusplus);
    std::printf("sizeof int   : %zu\n", sizeof(int));
    std::printf("sizeof long  : %zu\n", sizeof(long));
    std::printf("sizeof llong : %zu\n", sizeof(long long));
    std::printf("sizeof ptr   : %zu\n", sizeof(void*));
    std::printf("char signed  : %s\n", CHAR_MIN < 0 ? "yes" : "no");
#ifdef __SIZEOF_INT128__
    std::printf("__int128     : available\n");
#else
    std::printf("__int128     : not available\n");
#endif
    std::printf("LONG_MAX     : %ld\n", LONG_MAX);
    return 0;
}
```

Baris-baris `#if defined(...)` dievaluasi oleh *praprosesor* sebelum kompilasi: setiap kompiler mendefinisikan makro pengenalnya sendiri, seperti `__GNUC__` untuk g++ dan `__clang__` untuk clang, dan hanya cabang yang cocok yang dikompilasi. Makro `__cplusplus` menyimpan standar yang sedang dipakai sebagai bilangan: 201402 untuk C++14, 201703 untuk C++17, 202002 untuk C++20. Di mesin Linux kami, dikompilasi dengan `g++ -std=c++17 -O2`, program mencetak:

```
compiler     : g++ 13.3
__cplusplus  : 201703L
sizeof int   : 4
sizeof long  : 8
sizeof llong : 8
sizeof ptr   : 8
char signed  : yes
__int128     : available
LONG_MAX     : 9223372036854775807
```

Kode 5.4 mengajukan pertanyaan serupa kepada Python.

Kode 5.4 `platform_probe.py`: versi, implementasi, dan batas Python yang bisa berbeda antara mesin Anda dan juri.

```python
# Pasangan Python dari platform_probe.cpp: apa yang bisa berbeda di juri.
import platform
import struct
import sys

print("python       :", platform.python_version())
print("implementation:", platform.python_implementation())
print("pointer bits :", struct.calcsize("P") * 8)
print("int max      : unlimited (sys.maxsize = %d)" % sys.maxsize)
print("recursion    :", sys.getrecursionlimit())
print("int->str cap :", getattr(sys, "get_int_max_str_digits", lambda: "none")())
```

```
python       : 3.11.15
implementation: CPython
pointer bits : 64
int max      : unlimited (sys.maxsize = 9223372036854775807)
recursion    : 1000
int->str cap : 4300
```

Baris terakhir mudah terlewat padahal penting: sejak Python 3.11, konversi antara `int` dan `str` menolak bilangan lebih dari 4300 digit dan memunculkan `ValueError`, sebuah perlindungan terhadap serangan penolakan layanan [7]. Program yang mencetak 1000! dengan `print` berjalan di Python 3.10 tetapi gagal di 3.11 ke atas, kecuali batasnya dinaikkan dengan `sys.set_int_max_str_digits`. Bab 8 bertemu batas ini lagi pada soal dengan bilangan sejuta digit.

5.4.3 Ukuran tipe bilangan bulat

Standar C++ hanya menetapkan ukuran minimum tipe bilangan bulatnya; ukuran sebenarnya dipilih oleh *model data* setiap platform [8]. **Tabel 5.2** menunjukkan tiga kombinasi yang kemungkinan besar akan Anda temui.

Tabel 5.2 Ukuran bilangan bulat dalam dua model data 64-bit; baris tebal adalah yang berbeda.

Tipe	Linux dan macOS, 64-bit (LP64)	Windows, 64-bit (LLP64)	Dipakai untuk
`int`	32 bit	32 bit	nilai di bawah sekitar 2.1×10^9
`long`	**64 bit**	**32 bit**	hindari dalam kode lomba
`long long`	64 bit	64 bit	nilai sampai sekitar 9.2×10^{18}
pointer, `size_t`	64 bit	64 bit	ukuran dan indeks kontainer
`__int128`	ada di g++ dan clang	tidak ada di MSVC	hasil kali antara sampai 10^{38}

Baris `long` itulah penyebab malam Sinta yang terbuang di Subbab 5.1. Microsoft mendokumentasikan bahwa `long` berukuran 4 byte pada kompilernya di setiap platform Windows [9], dan MinGW-w64, g++ untuk Windows yang dipasang di Bab 1, mengikuti model data yang sama agar dapat memanggil Windows itu sendiri. Aturan untuk kode lomba karenanya sederhana: tulis `int` bila nilainya di bawah dua miliar, `long long` bila bisa lebih, dan jangan pernah `long` polos.

5.5 Flag yang Menemukan Bug

5.5.1 Peringatan

Peringatan (*warning*) adalah kompiler yang berkata "ini sah, tetapi mungkin bukan yang Anda maksud". Flag `-Wall` dan `-Wextra` menyalakan peringatan-peringatan yang ingin dilihat pemrogram berpengalaman [10]. Versi pertama **Kode 5.1** mengabaikan nilai kembalian `scanf`; dikompilasi dengan `-Wall -Wextra`, g++ memperingatkannya:

```
fctrl.cpp:10:19: warning: ignoring return value of 'int scanf(const char*, ...)'
declared with attribute 'warn_unused_result' [-Wunused-result]
```

Dalam lomba itu bukan bencana, karena masukan juri tertata rapi, tetapi memeriksa nilai kembalian tidak memakan biaya apa pun dan membuat program berhenti dengan rapi pada berkas yang terpotong, alih-alih berputar pada nilai basi. Versi yang dicetak di **Kode 5.1** memeriksanya, dan terkompilasi tanpa satu peringatan pun. Jadikan itu kebiasaan: kompilasi

dengan `-Wall -Wextra` di rumah dan perlakukan setiap peringatan sebagai pertanyaan yang harus Anda jawab.

5.5.2 Sanitizer

Sebagian bug tidak dapat dilihat dengan membaca teks; bug itu hanya muncul saat program berjalan, dengan nilai-nilai tertentu. **Sanitizer** adalah mode pemeriksaan di mana kompiler menambahkan uji ke dalam kode mesin, sehingga program berhenti dengan laporan tepat di baris tempat bug terjadi [10]. Dua di antaranya layak diketahui:

- `-fsanitize=undefined` (UBSan) melaporkan perilaku tak terdefinisi seperti luapan bertanda, pembagian dengan nol, atau pergeseran bit yang terlalu jauh;
- `-fsanitize=address` (ASan) melaporkan operasi baca dan tulis di luar larik, pemakaian memori yang sudah dibebaskan, dan galat memori sejenis.

Program yang disanitasi berjalan beberapa kali lebih lambat dan memakai lebih banyak memori, jadi Anda memakainya di rumah, tidak pernah di juri. Sanitizer bekerja dengan g++ dan clang di Linux, dengan clang di macOS, dan di bawah WSL di Windows; kompiler MinGW-w64 tidak menyertakannya.

Untuk melihat UBSan bekerja, **Kode 5.5** menghitung Persamaan (5.1) dengan cara yang paling jelas, memakai pangkat 5 yang terus membesar.

Kode 5.5 `fctrl_mulloop.cpp`: rumus yang sama dengan pangkat 5 yang membesar; benar di dalam batasan soal, perilaku tak terdefinisi tepat di luarnya.

```cpp
// Hitungan yang sama, ditulis dengan pangkat 5 yang membesar. Tampak polos;
// untuk N dekat 2^31 pangkatnya meluap dari int 32-bit (perilaku tak terdefinisi).
#include <cstdio>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        int n;
        if (std::scanf("%d", &n) != 1) break;
        int zeros = 0;
        for (int p = 5; p <= n; p *= 5)   // p *= 5 bisa melewati INT_MAX
            zeros += n / p;
        std::printf("%d\n", zeros);
    }
    return 0;
}
```

Pada berkas contoh, program ini mencetak enam baris yang persis sama dengan **Kode 5.1**:

```
1
6
31
249
249999989
249999998
```

Di dalam batasan FCTRL program ini bahkan benar: untuk $N \le 10^9$ loop berhenti di $p = 5^{13} = 1\,220\,703\,125$, yang lebih besar daripada N dan masih di bawah batas `int` $2^{31}-1 = 2\,147\,483\,647$. Namun program ini tidak punya ruang aman. Untuk setiap N dari 5^{13} sampai $2^{31}-1$, langkah `p *= 5` harus menghitung $5^{14} = 6\,103\,515\,625$, yang tidak muat dalam `int`. Itu luapan bertanda, dan luapan bertanda adalah perilaku tak terdefinisi [11]. Kami memberi program berkas dua baris dengan $N = 10^9$ dan $N = 2^{31}-1$. Dikompilasi biasa, program tidak

mencetak apa pun dan terus berjalan sampai batas waktu lima detik skrip uji kami menghentikannya: pangkat yang meluap berubah menjadi bilangan negatif, tetap "paling besar N", dan loop tidak pernah selesai. Satu konstanta yang keliru dalam spesifikasi masukan, soal yang menulis $N < 2^{31}$ alih-alih $N \le 10^9$, akan mengubah program ini menjadi *Time limit exceeded*. Dikompilasi dengan UBSan,

```
g++ -std=c++17 -O2 -fsanitize=undefined -fno-sanitize-recover=all \
    -o mul_san fctrl_mulloop.cpp
```

masukan yang sama langsung berhenti, lengkap dengan baris, kolom, dan nilai-nilainya:

```
fctrl_mulloop.cpp:12:35: runtime error: signed integer overflow:
1220703125 * 5 cannot be represented in type 'int'
```

(Di layar laporan itu satu baris; di sini dipatahkan agar muat di halaman.) Perhatikan juga bahwa jawaban untuk $N = 10^9$, yang sudah dihitung program sebelum luapan, tidak tampak dalam laporan: program dihentikan sebelum penyangga keluarannya ditulis. Ketika program yang mogok mencetak lebih sedikit daripada yang Anda harapkan, belum tentu ia berhenti lebih awal daripada dugaan Anda.

5.5.3 Pengoptimasi menganggap perilaku tak terdefinisi tidak pernah terjadi

Mengapa loop yang rusak berjalan selamanya alih-alih berhenti? Karena kompiler boleh menganggap bahwa program yang benar tidak pernah meluap, dan ia memakai anggapan itu untuk mempercepat kode. **Kode 5.6** menunjukkan akibatnya dalam program terkecil yang dapat kami tulis.

Kode 5.6 `ub_check.cpp`: syarat loop `i > 0` bergantung pada luapan, sehingga banyaknya langkah bergantung pada tingkat optimasi.

```cpp
// Luapan bertanda = perilaku tak terdefinisi: pengoptimasi boleh menganggapnya tak
// pernah terjadi, jadi sumber yang sama bisa mencetak angka lain di -O0 dan -O2.
#include <climits>
#include <cstdio>

int main() {
    volatile int start = INT_MAX - 2500000;  // volatile: tak diketahui kompiler
    int steps = 0;
    for (int i = start; i > 0 && steps < 100; i += 1000000)
        steps++;              // i > 0 "tak mungkin gagal" bila tak ada luapan
    std::printf("steps = %d\n", steps);
    unsigned int u = UINT_MAX;
    u = u + 1;                // aritmetika unsigned berputar: terdefinisi
    std::printf("UINT_MAX + 1 = %u\n", u);
    return 0;
}
```

Variabel `start` dideklarasikan `volatile`, yang memaksa kompiler membacanya saat program berjalan, sehingga kompiler tidak dapat menghitung jawabannya lebih dahulu. Loop dimulai 2,5 juta di bawah `int` terbesar dan menambahkan satu juta per langkah. Jika dinalar dengan bilangan bulat sungguhan, i tetap positif selamanya, dan loop hanya dihentikan oleh `steps < 100`. Jika dinalar dengan aritmetika 32-bit, penjumlahan ketiga berputar menjadi bilangan negatif dan loop berhenti setelah 3 langkah. Dikompilasi dengan `-O2`, program mencetak:

```
steps = 100
UINT_MAX + 1 = 0
```

Dikompilasi dengan `-O0`, tanpa optimasi, sumber yang sama mencetak `steps = 3`. Kami mencoba kedua kompiler:

Tabel 5.3 Keluaran `ub_check.cpp` dengan berbagai kompiler dan tingkat optimasi.

Kompiler dan flag	`steps` yang dicetak
g++ 13.3, `-O0`	3
g++ 13.3, `-O1`	100
g++ 13.3, `-O2`	100
clang 18.1, `-O0`	3
clang 18.1, `-O2`	100
g++ 13.3, `-O2 -fsanitize=undefined`	berhenti dengan "signed integer overflow: 2146983647 + 1000000"

Tabel 5.3 memperlihatkan inti persoalannya. Mulai `-O1`, kedua kompiler bernalar "i mulai positif dan hanya bertambah, dan luapan tidak mungkin terjadi pada program yang benar, jadi `i > 0` selalu benar", menghapus uji itu, dan menjalankan loop 100 kali. Tak satu pun kompiler yang salah; programnyalah yang salah. Paruh kedua **Kode 5.6** memperlihatkan pembandingnya: aritmetika *unsigned* didefinisikan berputar modulo 2^{32}, sehingga `UINT_MAX + 1` bernilai 0 di setiap kompiler pada setiap tingkat.

Bagi Anda, akibatnya sangat praktis. Program dengan perilaku tak terdefinisi dapat lolos uji Anda di `-O0`, bawaan tombol "Run" di banyak editor, lalu gagal di juri pada `-O2`, atau sebaliknya. Ujilah dengan flag yang dipakai juri, dan jalankan uji stres Anda (Bab 8) sekali dengan sanitizer menyala.

5.5.4 Mode debug pustaka standar

Satu mode pemeriksaan lagi khusus untuk g++: mengompilasi dengan `-D_GLIBCXX_DEBUG` mengganti kontainer pustaka standar dengan versi yang memeriksa setiap indeks dan iterator [12]. `v[i]` yang di luar jangkauan lalu menghentikan program dengan pesan yang jelas, alih-alih diam-diam membaca memori tetangganya. Bab 8 memakainya pada bug sungguhan; untuk sekarang, tambahkan ke daftar sakelar yang Anda pakai di rumah dan tidak pernah di juri.

5.6 Flag yang Membuat Kode Lebih Cepat

5.6.1 Tingkat optimasi

Tanpa perintah, g++ menerjemahkan setiap baris sumber seharfiah mungkin, agar debugger dapat mengikuti program baris demi baris. Itulah tingkat optimasi `-O0`. Tingkat yang lebih tinggi mengizinkan kompiler menyimpan nilai di register alih-alih di memori, membuang perhitungan yang hasilnya tidak pernah dipakai, memindahkan kerja keluar dari loop, mengganti pemanggilan fungsi dengan badan fungsinya (*inlining*), dan, mulai `-O2` pada versi GCC terbaru, memproses beberapa elemen larik dengan satu instruksi *vektor* (*vektorisasi*) [13]. `-O3` menambahkan transformasi yang lebih agresif, dan `-march=native` meminta kompiler memakai setiap instruksi milik prosesor tempat ia berjalan.

Juri mengompilasi dengan optimasi; kebanyakan memakai `-O2`. Untuk melihat seberapa besar pengaruh tingkat optimasi, kami mengukur waktu **Kode 5.7**, sejumlah kerja sederhana yang tetap: untuk setiap dari 20.000 ambang, hitung berapa dari 20.000 nilai yang berada di bawahnya, total 4×10^8 perbandingan.

Kode 5.7 `flags_bench.cpp`: 4×10^8 perbandingan yang kecepatannya hanya bergantung pada flag kompiler.

```cpp
// Beban kerja tetap yang kecepatannya bergantung pada flag kompiler:
// untuk tiap ambang dari Q ambang, hitung nilai larik di bawahnya (Q * N langkah).
// Checksum dicetak ke stdout; waktunya ke stderr.
#include <chrono>
#include <cstdint>
#include <cstdio>
#include <vector>

int main() {
    const int N = 20000, Q = 20000;
    std::vector<int> a(N), q(Q);
    std::uint32_t s = 12345;                  // pembangkit mungil, sama di mana pun
    for (int& x : a) { s = s * 1103515245u + 12345u; x = (int)(s >> 8) % 1000000; }
    for (int& x : q) { s = s * 1103515245u + 12345u; x = (int)(s >> 8) % 1000000; }
    auto start = std::chrono::steady_clock::now();
    long long checksum = 0;
    for (int j = 0; j < Q; j++) {
        int below = 0;
        for (int i = 0; i < N; i++) below += a[i] < q[j];  // tanpa cabang: vektor
        checksum += below;
    }
    auto stop = std::chrono::steady_clock::now();
    std::printf("checksum %lld\n", checksum);
    std::fprintf(stderr, "time %.3f s\n",
                 std::chrono::duration<double>(stop - start).count());
    return 0;
}
```

Dua rincian membuat uji ini jujur. Bilangan-bilangannya berasal dari pembangkit acak mungil yang ditulis lengkap di dalam program, sehingga setiap kompiler di setiap platform melihat data yang persis sama; dan program mencetak checksum, total semua hitungan, sehingga kita dapat memastikan setiap build menghitung hal yang sama. Waktunya dikirim ke aliran galat, `stderr`, agar keluaran normal tetap identik:

```
checksum 200865372
```

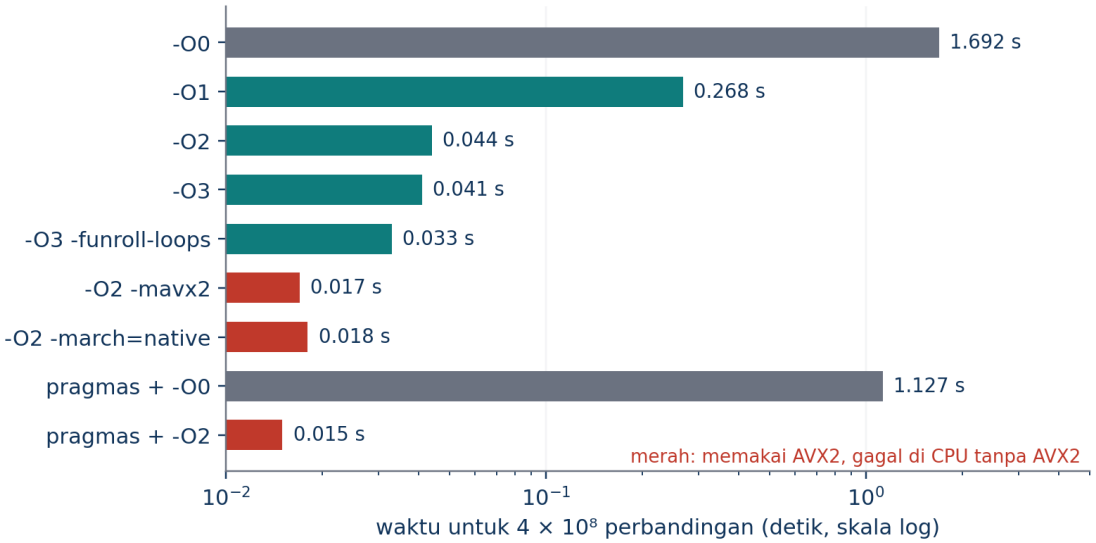
Perhatikan juga baris `below += a[i] < q[j];`. Perbandingan itu bernilai 1 atau 0, dan menambahkannya menghindari `if` di dalam loop terdalam; loop tanpa percabangan persis jenis loop yang cocok untuk instruksi vektor.

5.6.2 Apa yang kami ukur

Kami mengompilasi **Kode 5.7** dengan sembilan pengaturan berbeda dan mengambil yang tercepat dari tiga kali jalan untuk masing-masing, di mesin yang sedang tidak dipakai pekerjaan lain; rangkaian pertama, yang dijalankan saat pekerjaan lain sedang memakai prosesor, memberikan waktu sekitar dua kali lebih lama di setiap baris, dengan rasio yang sama. **Tabel 5.4** dan **Gambar 5.1** menunjukkan hasilnya; checksum-nya 200865372 setiap kali.

Tabel 5.4 Waktu `flags_bench.cpp` dengan sembilan pengaturan (tercepat dari tiga kali jalan, g++ 13.3, Xeon 2,1 GHz).

Flag	Waktu (s)	Berjalan di prosesor x86-64 mana pun?
`-O0`	1.692	ya
`-O1`	0.268	ya
`-O2`	0.044	ya
`-O3`	0.041	ya
`-O3 -funroll-loops`	0.033	ya
`-O2 -mavx2`	0.017	hanya dengan AVX2
`-O2 -march=native`	0.018	hanya pada model prosesor ini
`-O0` + kedua pragma **Kode 5.8**	1.127	hanya dengan AVX2
`-O2` + kedua pragma **Kode 5.8**	0.015	hanya dengan AVX2



Gambar 5.1 4×10^8 perbandingan yang sama dengan sembilan pengaturan flag, pada skala logaritmik; batang merah memakai instruksi AVX2.

Empat pelajaran menonjol.

1. **Tingkat optimasi mengalahkan segalanya.** Dari `-O0` ke `-O2` program menjadi 38 kali lebih cepat, tanpa satu pun perubahan pada sumber. Mengukur waktu program yang dibangun tanpa optimasi tidak berkata apa-apa tentang kinerjanya di juri. Jika tombol run di editor Anda mengompilasi dengan `-O0`, solusi yang "butuh dua detik" di rumah mungkin hanya butuh seperdua puluh detik di juri, dan Anda bisa membuang gagasan yang benar tanpa alasan.
2. **`-O3` bukan sakelar ajaib.** Di sini ia menambah kecepatan 7% dibandingkan `-O2`; pada program lain ia tidak menambah apa-apa atau sedikit merugikan. Menambahkan unrolling memberi seperlima lagi. Efek-efek ini kecil dibandingkan pilihan algoritma.
3. **Instruksi vektor yang lebih lebar membayar pada loop seperti ini**, dua setengah kali lebih cepat daripada `-O2`, tetapi hanya di prosesor yang memilikinya.
4. **Pragma tidak menggantikan tingkat optimasi di baris perintah.** Pragma di **Kode 5.8** dengan baris perintah `-O0` hanya 1,5 kali lebih cepat daripada `-O0` polos, karena bagian-bagian pustaka standar yang didefinisikan sebelum pragma tetap dikompilasi tanpa optimasi.

5.6.3 Optimasi dari dalam sumber

Karena Anda tidak dapat mengubah flag juri, GCC mengizinkan Anda meminta optimasi dari dalam berkas. `#pragma GCC optimize("O3,unroll-loops")` mengatur opsi optimasi, dan `#pragma GCC target("avx2")` mengizinkan instruksi dari sebuah perluasan prosesor, keduanya untuk fungsi-fungsi yang didefinisikan setelah baris itu [14]. **Kode 5.8** adalah **Kode 5.7** dengan kedua pragma itu ditambahkan.

Kode 5.8 `flags_bench_pragma.cpp`: uji kecepatan dengan pragma optimize dan pragma target, diletakkan sesudah include.

```
// flags_bench.cpp dengan dua pragma: pragma optimize menaikkan
// tingkat optimasi, pragma target mengizinkan instruksi AVX2.
// CPU tanpa AVX2 menghentikan program ini dengan "Illegal instruction".
// Pragma diletakkan SESUDAH include: bila di awal, g++ 13 menolak
// meng-inline fungsi bantu pustaka ("target specific option mismatch").
#include <chrono>
#include <cstdint>
#include <cstdio>
#include <vector>
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2")

int main() {
    const int N = 20000, Q = 20000;
    std::vector<int> a(N), q(Q);
    std::uint32_t s = 12345;
    for (int& x : a) { s = s * 1103515245u + 12345u; x = (int)(s >> 8) % 1000000; }
    for (int& x : q) { s = s * 1103515245u + 12345u; x = (int)(s >> 8) % 1000000; }
    auto start = std::chrono::steady_clock::now();
    long long checksum = 0;
    for (int j = 0; j < Q; j++) {
        int below = 0;
        for (int i = 0; i < N; i++) below += a[i] < q[j];
        checksum += below;
    }
    auto stop = std::chrono::steady_clock::now();
    std::printf("checksum %lld\n", checksum);
    std::fprintf(stderr, "time %.3f s\n",
                 std::chrono::duration<double>(stop - start).count());
    return 0;
}
```

```
checksum 200865372
```

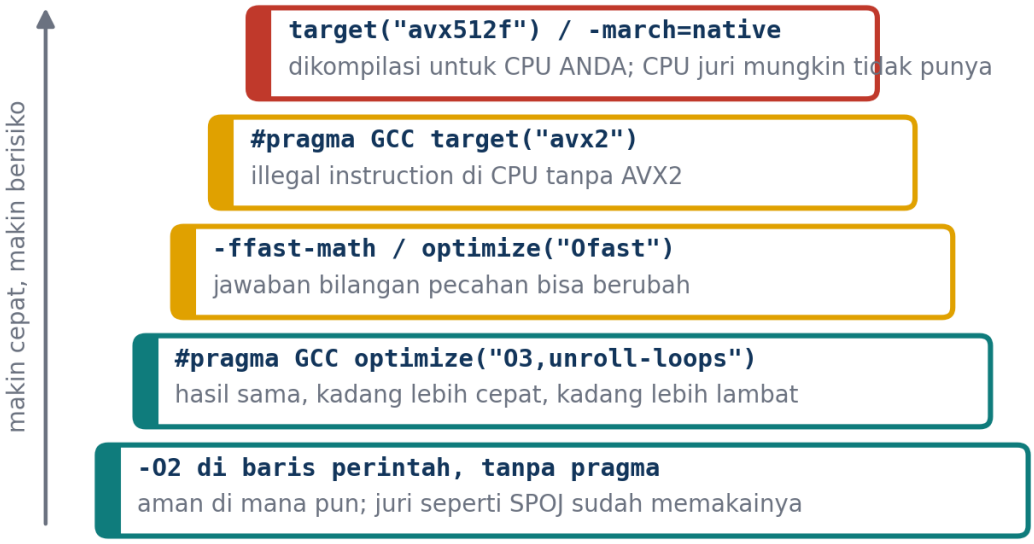
Komentar di awal listing mencatat sesuatu yang kami pelajari dengan cara yang tidak enak. Blog biasanya meletakkan pragma di baris-baris pertama berkas, sebelum include. Kami mula-mula melakukan itu, dan g++ 13.3 menolak mengompilasi program:

```
error: inlining failed in call to 'always_inline' 'std::allocator<...>::~allocator()
noexcept [with _Tp = int]': target specific option mismatch
```

Dengan pragma target di awal, fungsi-fungsi kecil milik pustaka dikompilasi untuk AVX2 sementara bagian lain yang memanggilnya tidak, dan kompiler tidak mau meng-inline kode yang dikompilasi untuk himpunan instruksi lebih besar ke dalam kode untuk himpunan yang lebih kecil. Muncul tidaknya galat ini bergantung pada versi kompiler dan header yang Anda sertakan, dan itu sendiri sebuah pelajaran: pragma yang disalin dari templat orang lain dapat mengubah program yang diterima menjadi galat kompilasi di kompiler lain. Memindahkan pragma ke sesudah include, seperti di **Kode 5.8**, menerapkannya hanya pada fungsi-fungsi Anda sendiri dan terkompilasi bersih dengan setiap kompiler yang kami coba.

5.7 Tangga Risiko Pragma

Pragma adalah tukar-menukar: sedikit kecepatan untuk sedikit risiko. **Gambar 5.2** menyusun pilihan-pilihan yang umum sebagai tangga, dari anak tangga terbawah yang selalu aman sampai yang teratas, yang untung-untungan.



Gambar 5.2 Tangga risiko pragma: setiap anak tangga ke atas membeli kecepatan di sebagian mesin dan menambah satu cara gagal di mesin lain.

Anak tangga 1: `-O2` milik juri, tanpa tambahan. Inilah yang dipakai setiap solusi yang diterima di buku ini. Ia portabel menurut definisinya.

Anak tangga 2: `#pragma GCC optimize("O3,unroll-loops")`. Pragma ini hanya mengubah *cara* kompiler menerjemahkan, bukan instruksi mana yang boleh dipakainya, sehingga program yang benar tetap benar dan berjalan di setiap prosesor. Keuntungannya tidak pasti, seperti diperlihatkan **Tabel 5.4**, dan manual GCC menggambarkan opsi-opsi ini sebagai alat untuk menyetel fungsi tertentu, bukan sakelar umum [14]. Tidak berbahaya, tetapi jarang menjadi pembeda antara diterima dan tidak.

Anak tangga 3: `-ffast-math` atau `optimize("Ofast")`. Keduanya mengizinkan kompiler menyusun ulang aritmetika bilangan pecahan seolah-olah itu aritmetika bilangan real yang eksak, dan menganggap tidak ada nilai yang tak hingga atau "bukan bilangan" [13]. Untuk kode bilangan bulat keduanya tidak mengubah apa pun. Untuk kode bilangan pecahan keduanya dapat mengubah digit-digit terakhir hasil, dan pemeriksaan seperti `if (x != x)` untuk "bukan bilangan" bisa dihapus sama sekali. Pada soal geometri yang membandingkan jarak dengan toleransi 10^{-9}, itu risiko *Wrong answer* yang nyata.

Anak tangga 4: `#pragma GCC target("avx2")` dan sejenisnya. Kompiler kini boleh memancarkan instruksi AVX2. Di prosesor tanpa AVX2, program dihentikan sistem operasi dengan sinyal SIGILL, "illegal instruction", yang dilaporkan juri sebagai runtime error. Prosesor Cube milik SPOJ memiliki AVX2 [6], jadi di SPOJ anak tangga ini kebetulan berhasil; di juri lain, atau di klaster yang lebih tua, belum tentu.

Anak tangga 5: `target("avx512f")` atau `-march=native`. AVX-512 tidak ada pada Xeon E3-1220 v5 [6], dan `-march=native` berarti "apa pun yang dimiliki mesin tempat saya mengompilasi", yang

tidak bermakna bila mesin yang mengompilasi bukan mesin tempat Anda menguji. Inilah persisnya kegagalan Bima di Subbab 5.1: pragma itu membuat program lebih cepat di laptopnya dan mematikannya di juri. Di laptop Apple berprosesor ARM, pragma target x86 bahkan tidak terkompilasi, yang setidaknya gagal lebih awal.

Tangga ini menyarankan satu aturan yang bisa diterapkan dalam hitungan detik. **Naiklah hanya bila algoritmanya sudah benar dan waktu terukur sudah mendekati batas, dan jangan pernah melewati anak tangga 2 tanpa mengetahui prosesor juri.** Solusi yang membutuhkan anak tangga 4 untuk lolos biasanya solusi dengan kompleksitas yang salah, dan tabel anggaran di Bab 3 adalah tempat mencari yang lebih baik daripada sebuah pragma.

5.8 Daftar Periksa Portabilitas

5.8.1 C++

Kegagalan di Subbab 5.1 termasuk dalam daftar pendek yang mencakup hampir semua masalah portabilitas dalam kode lomba. **Tabel 5.5** mengumpulkannya.

Tabel 5.5 Fitur yang rusak ketika program C++ pindah ke platform lain, dan apa yang sebaiknya ditulis.

Fitur	Di mana rusak	Pengganti yang portabel
`#include <bits/stdc++.h>`	clang dengan libc++ (bawaan di macOS) [15], MSVC	sertakan header standar yang Anda pakai
`long` untuk nilai di atas 2^{31}	Windows (`long` 32-bit)	`long long`, dicetak dengan `%lld`
`__int128`	MSVC; mencetaknya perlu kode sendiri di mana pun	pecah perhitungannya, atau pakai hanya bila g++/clang terjamin
larik panjang variabel `int a[n];`	MSVC; `n` besar meluapkan stack di mana pun	`std::vector<int> a(n);`
`gets`, `itoa`, `strrev`, `conio.h`	dihapus sejak C++14 atau tidak pernah standar	`std::getline`, `std::to_string`, `std::reverse`
`char` berisi byte di atas 127	Linux ARM, tempat `char` tak bertanda	`unsigned char` atau `int`
fitur C++20 (`<=>`, ranges, `std::format`)	juri yang mengompilasi C++14 atau C++17	padanan C++17, atau periksa menunya
rekursi yang dalam	Windows (stack 1 MB), sebagian juri	stack eksplisit (Bab 6)
luapan bertanda, variabel tanpa inisialisasi	kompiler mana pun pada tingkat mana pun	tipe 64-bit, inisialisasi semuanya, uji dengan sanitizer

Baris pertama layak dikomentari, karena `bits/stdc++.h` muncul di templat lomba yang tak terhitung banyaknya. Header ini adalah header internal pustaka standar GCC, libstdc++, yang menyertakan segalanya, dan banyak templat lomba yang dipublikasikan dimulai dengannya [16]; ia praktis dan sedikit memperlambat kompilasi. Juri memakai g++, jadi header ini berhasil di SPOJ. Ia gagal di Mac, yang kompiler bawaannya memakai libc++ milik LLVM, dan di kompiler Microsoft. Jika Anda berbagi kode dengan orang lain, atau memakai Mac, tuliskan include satu per satu; biayanya hanya beberapa baris.

5.8.2 Python

Program Python lebih portabel daripada program C++, karena interpreter menyembunyikan prosesor, tetapi tiga perbedaan tetap penting.

- **Versinya.** Python di juri bisa beberapa versi di belakang milik Anda. Sintaks seperti pernyataan `match` (3.10), exception group (3.11), atau sintaks generik yang baru (3.12) adalah galat sintaks di interpreter yang lebih tua, yang dilaporkan juri sebagai runtime error. Jika ragu, tulis Python yang diterima 3.8, seperti **Kode 5.2**.
- **Batasnya.** Batas rekursi secara bawaan 1000, dan sejak 3.11 konversi bilangan bulat ke string dibatasi 4300 digit [7]. Keduanya pengaturan, bukan hukum alam, tetapi program harus mengubahnya secara eksplisit.
- **Implementasinya.** Sebagian juri menawarkan PyPy, implementasi Python yang lebih cepat, di samping CPython. PyPy menjalankan loop berkali-kali lebih cepat, tetapi punya perilaku memori sendiri dan versinya tertinggal dari CPython. Ujilah dengan yang akan Anda kirimi.

5.8.3 Skrip pemeriksa sebelum mengirim

Membaca daftar periksa itu baik; menjalankannya lebih baik. **Kode 5.9** mengompilasi setiap program C++ di folder `cpp` bab ini lima kali: sebagai C++14, sebagai C++17 dengan peringatan dijadikan galat, sebagai C++20, dengan kedua sanitizer, dan dengan clang.

Kode 5.9 `portability_check.py`: mengompilasi setiap program dengan beberapa standar, kompiler, dan mode pemeriksaan lalu melaporkan kegagalannya.

```
# Kompilasi setiap program di ../cpp dengan beberapa kompiler, standar, dan
# mode pemeriksaan, lalu laporkan kombinasi yang gagal. Jalankan sebelum kirim.
import glob
import os
import shutil
import subprocess

MODES = [
    ("g++", ["-std=c++14", "-O2"]),
    ("g++", ["-std=c++17", "-O2", "-Wall", "-Wextra", "-Werror"]),
    ("g++", ["-std=c++20", "-O2"]),
    ("g++", ["-std=c++17", "-O1", "-fsanitize=address,undefined"]),
    ("clang++", ["-std=c++17", "-O2", "-Wall", "-Wextra"]),
]

here = os.path.dirname(os.path.abspath(__file__))
sources = sorted(glob.glob(os.path.join(here, "..", "cpp", "*.cpp")))
out = os.path.join(here, "build")
os.makedirs(out, exist_ok=True)
for compiler, flags in MODES:
    if shutil.which(compiler) is None:
        print("%-9s %-44s skipped (not installed)" % (compiler, " ".join(flags)))
        continue
    bad = []
    for src in sources:
        exe = os.path.join(out, "probe")
        r = subprocess.run([compiler] + flags + ["-o", exe, src],
                           capture_output=True, text=True)
        if r.returncode != 0:
            bad.append(os.path.basename(src))
    status = "FAIL: " + ", ".join(bad) if bad else "all %d compile" % len(sources)
    print("%-9s %-44s %s" % (compiler, " ".join(flags), status))
```

Dijalankan dari folder `tests` dengan `python3 portability_check.py`, skrip ini mencetak:

```
g++       -std=c++14 -O2                              all 6 compile
g++       -std=c++17 -O2 -Wall -Wextra -Werror        all 6 compile
g++       -std=c++20 -O2                              all 6 compile
g++       -std=c++17 -O1 -fsanitize=address,undefined  all 6 compile
clang++   -std=c++17 -O2 -Wall -Wextra                all 6 compile
```

Setiap program bab ini terkompilasi di setiap mode. Skrip yang sama menemukan galat pragma di Subbab 5.6.3 sebelum pragmanya kami pindahkan, dan menandai peringatan `scanf` dari Subbab 5.5.1 di baris `-Werror`. Di Windows dengan MSYS2, baris sanitizer gagal, karena MinGW-w64 tidak punya sanitizer, dan baris clang dilewati kecuali clang terpasang; baris-baris lainnya bekerja seperti di sini.

5.9 Mengujinya

Rumus di Subbab 5.3 sudah dibuktikan, tetapi bukti berbicara tentang rumus, bukan tentang program. Program masih bisa membagi dengan bilangan yang salah, mencetak tipe yang salah, atau membaca masukan dengan cara yang salah. Bab 8 mengembangkan metode yang kita pakai untuk setiap soal di buku ini; berikut cicipan pertamanya.

Gagasannya adalah menulis program kedua yang *jelas* benar, selambat apa pun, lalu membandingkan keduanya pada banyak masukan acak. Untuk FCTRL, program yang jelas benar menghitung $N!$ secara eksak dengan bilangan bulat Python yang tak terbatas, lalu menghitung nol di ujung bentuk desimalnya, karakter demi karakter. **Kode 5.10** adalah program itu.

Kode 5.10 `brute_fctrl.py`: brute force untuk FCTRL membangun $N!$ dan menghitung nolnya; hanya N kecil yang layak.

```
# Brute force untuk FCTRL: bangun N! persis lalu hitung nol di belakangnya.
# Hanya untuk N kecil, dan justru itu intinya: jelas benar.
import sys

data = sys.stdin.read().split()
t = int(data[0])
for x in data[1:1 + t]:
    f = 1
    for i in range(2, int(x) + 1):
        f *= i
    s = str(f)
    print(len(s) - len(s.rstrip("0")))
```

Program ini hanya sanggup menangani N kecil (menghitung 10^9! mustahil), jadi generator `gen_fctrl.py` menghasilkan satu sampai delapan nilai hingga 400. Untuk N besar kita memakai pendapat kedua, `brute_fctrl_big.py`, yang mengevaluasi Persamaan (5.1) dengan pangkat 5 yang membesar dalam bilangan bulat eksak Python, perhitungan yang berbeda dari loop pembagian di **Kode 5.1**. Pengikat uji `stress.py`, yang ditampilkan dan dijelaskan di Bab 8, menjalankan generator, brute force, dan kandidat untuk 300 benih acak:

```
python3 stress.py "python3 gen_fctrl.py" "python3 brute_fctrl.py" \
    build/fctrl 300
python3 stress.py "python3 gen_fctrl.py big" "python3 brute_fctrl_big.py" \
    build/fctrl 300
```

```
OK: 300 random tests agree
OK: 300 random tests agree
```

Program Python juga cocok pada ke-300 uji. Generator "big" sengaja menyertakan $N = 2^{31} - 1$, tepat di luar batasan, dan menjalankannya terhadap `fctrl_mulloop` memunculkan kembali loop tanpa akhir dari Subbab 5.5.2 pada benih ketiga:

```
MISMATCH on seed 3
input:
4
2147483647
244140625
671862058
650257552
expected: 536870902 61035156 167965506 162564382
received: TIMEOUT (more than 5 s)
```

Hasil seperti itulah gunanya uji stres: program yang lolos setiap uji buatan tangan, dengan jurang tersembunyi satu masukan jauhnya.

5.10 Menggunakannya Sungguhan

Semua isi bab ini menjadi kebiasaan begitu dituliskan ke dalam perkakas Anda. Tiga investasi kecil membuatnya otomatis.

Perintah kompilasi yang selalu Anda pakai. Putuskan sekali flag apa yang Anda pakai di rumah dan simpan dalam skrip, alias, atau pengaturan editor. Perintah harian yang baik meniru juri dan menambahkan peringatan:

```
g++ -std=c++17 -O2 -Wall -Wextra -o prog prog.cpp
```

dan perintah "curiga" yang baik menambahkan pemeriksaan:

```
g++ -std=c++17 -O1 -g -Wall -Wextra -fsanitize=address,undefined \
    -D_GLIBCXX_DEBUG -o prog prog.cpp
```

Di VS Code, masukkan keduanya ke berkas `.vscode/tasks.json` di folder lomba Anda sebagai dua build task, sehingga **Terminal > Run Build Task** menjalankan perintah harian; task bawaan ekstensi C/C++ mengompilasi hanya dengan `-g`, yang berarti `-O0`, dan akan menyesatkan setiap pengukuran waktu Anda.

Templat tanpa kejutan. Jika Anda menyimpan berkas yang disalin di awal setiap soal, buatlah portabel: include satu per satu, `long long` alih-alih `long`, tanpa pragma target, masukan dan keluaran cepat dari Bab 4. Apa pun yang lebih eksotis dimasukkan dengan sengaja, untuk satu soal, dengan komentar yang menjelaskan alasannya.

Pemeriksaan terakhir sebelum mengirim. Untuk soal dengan batas ketat atau tipe yang tidak lazim, jalankan `portability_check.py` atau setidaknya build sanitizer pada uji stres Anda, dan ukur waktu masukan terbesar dengan perintah harian. Seluruh rutinitas itu makan waktu semenit, dan semenit itulah yang mengubah "jalan di rumah" menjadi "jalan".

Cara berpikir yang sama berlaku di luar lomba. Seorang pemilik toko di Sidoarjo yang menulis program kecil untuk menjumlahkan penjualan harian di laptop Windows-nya, lalu kelak menjalankannya di server Linux yang murah, akan bertemu persis masalah `long` di **Tabel 5.2** begitu total dalam rupiah melewati dua miliar, dan bulan yang ramai bisa mencapainya. Menulis `long long` sejak awal, dan menguji di kedua mesin, adalah seluruh obatnya.

5.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

- **Mengukur waktu program yang dibangun tanpa optimasi.** Build `-O0` bisa hampir 40 kali lebih lambat daripada build `-O2` milik juri (**Tabel 5.4**). Selalu ukur dengan flag juri.
- **Mengandalkan standar bawaan kompiler.** GCC 16 mengompilasi C++20 secara bawaan, kompiler yang lebih tua C++14 atau C++17. Berikan `-std=` secara eksplisit, sesuai menu juri.

- **Memakai `long` untuk nilai 64-bit.** Ukurannya 32 bit di Windows. Pakai `long long` dan `%lld`.
- **Menyalin templat pragma tanpa memahaminya.** Pragma target bisa mogok di prosesor juri, dan pragma yang diletakkan sebelum include bisa menggagalkan kompilasi. Naiki tangga di **Gambar 5.2** satu anak tangga setiap kali, dan hanya dengan pengukuran yang membenarkannya.
- **Yakin program benar karena memberi jawaban benar di satu tingkat optimasi.** Perilaku tak terdefinisi dapat berubah bersama tingkatnya (**Tabel 5.3**). Jalankan sanitizer paling tidak sekali.
- **Membesarkan pangkat sampai melewati N.** `for (p = 5; p <= n; p *= 5)` meluap bila n dekat batas tipenya. Bagilah N, seperti yang diizinkan Persamaan (5.2).
- **Menulis Python hanya untuk versi sendiri.** Sintaks baru menjadi runtime error di juri yang lebih tua; batas konversi 4300 digit menjadi galat di juri yang lebih baru. Ketahui Python mana yang Anda kirimi.

5.12 Ringkasan Bab

- Program Anda dikompilasi dan dijalankan di mesin juri, dengan kompiler, standar, dan flag milik juri. Hanya teks sumbernya yang milik Anda.
- Perbedaan datang dalam tiga jenis: bahasa yang diterima (standar, header, perluasan), ukuran tipe (`long` 32 bit di Windows), dan perilaku tak terdefinisi, yang bisa memberi hasil berbeda dengan kompiler dan tingkat optimasi yang berbeda.
- FCTRL menghitung faktor 5 dari $N!$ dengan rumus Legendre (Persamaan (5.1)); membagi N berulang-ulang (Persamaan (5.2)) mengevaluasinya tanpa risiko luapan, dalam sekitar 10^6 operasi untuk berkas terbesar (Persamaan (5.3)).
- `-Wall -Wextra` memperlihatkan kode yang mencurigakan, `-fsanitize=undefined,address` menangkap luapan dan galat memori saat berjalan, `-D_GLIBCXX_DEBUG` memeriksa indeks kontainer. Pakai di rumah, bukan di juri.
- Tingkat optimasi sangat berpengaruh: 38 kali antara `-O0` dan `-O2` pada uji kami. `-O3`, unrolling, dan pragma mengubah jauh lebih sedikit, dan pragma target bisa mogok di prosesor tanpa instruksi yang diizinkannya.
- Daftar periksa pendek (**Tabel 5.5**) dan skrip yang mengompilasi dengan beberapa mode (**Kode 5.9**) menangkap hampir setiap masalah portabilitas sebelum juri menemukannya.

5.13 Soal Latihan

1. Dengan tangan, hitung $Z(1000)$ memakai Persamaan (5.1), lalu N terkecil dengan $Z(N) = 1000$. Periksa keduanya dengan `fctrl.cpp`.
2. Adakah N dengan $Z(N) = 5$? Jelaskan nilai-nilai mana yang dilompati Z dan mengapa.
3. Tulis ulang `fctrl_mulloop.cpp` agar tetap memakai pangkat yang membesar tetapi tidak mungkin meluap untuk nilai `int` N mana pun, tanpa memakai tipe 64-bit. Jelaskan mengapa loop Anda aman.
4. Kompilasi `ub_check.cpp` dengan `-O2 -fwrapv` lalu jalankan. Apa yang diubah `-fwrapv`, dan mengapa itu bukan alasan yang baik untuk mengandalkan luapan bertanda dalam kode lomba?
5. Templat seorang teman diawali `#pragma GCC target("avx2,bmi,bmi2,popcnt,lzcnt")` dan `#include <bits/stdc++.h>`. Daftarlah setiap platform dari bab ini tempat program yang benar dengan templat ini bisa gagal, dan sebutkan cara gagalnya di masing-masing platform.

6. Program Python **Kode 5.2** harus mencetak 100000! secara persis (semua digitnya) di juri yang menjalankan Python 3.12. Tuliskan baris-baris yang akan Anda tambahkan, dan jelaskan apa yang terjadi tanpanya.

5.14 Penyelesaian Soal Latihan

1. Suku-sukunya $\lfloor 1000/5 \rfloor = 200$, $\lfloor 1000/25 \rfloor = 40$, $\lfloor 1000/125 \rfloor = 8$, $\lfloor 1000/625 \rfloor = 1$, lalu 0, jadi $Z(1000) = 249$, yang juga baris keempat keluaran contoh. Untuk bagian kedua, Z tumbuh kira-kira $N/4$, jadi cobalah N di sekitar 4000: $Z(4000) = 800 + 160 + 32 + 6 + 1 = 999$. Z hanya berubah di kelipatan 5, dan $Z(4005) = 801 + 160 + 32 + 6 + 1 = 1000$, jadi N terkecil adalah 4005. Program mencetak 999 untuk 4004 dan 1000 untuk 4005.

2. Tidak ada. $Z(24) = 4$ dan $Z(25) = 6$: di $N = 25$ bilangan 25 membawa dua faktor 5 sekaligus, sehingga Z melompat 2 dan nilai 5 terlewati. Secara umum Z melompat sebesar k di kelipatan 5^k yang bukan kelipatan 5^{k+1}, dan di situ melewati $k - 1$ nilai.

3. Uji sebelum mengalikan: lanjutkan selama $p \le N/5$, yaitu selama mengalikan dengan 5 tidak mungkin melampaui N.

```
int zeros = 0;
for (int p = 5; ; p *= 5) {
    zeros += n / p;
    if (p > n / 5) break;   // 5 * p akan melampaui n: berhenti sebelum dihitung
}
```

Jika $p \le \lfloor N/5 \rfloor$ maka $5p \le N \le 2^{31} - 1$, jadi perkalian berikutnya muat; jika tidak, $5p > N$ dan setiap suku berikutnya 0. Untuk $N < 5$ suku pertama 0 dan loop langsung berhenti, dan itu benar.

4. `-fwrapv` meminta g++ mendefinisikan luapan bertanda sebagai berputar, seperti aritmetika unsigned. Pengoptimasi lalu tidak lagi boleh menghapus uji `i > 0`, dan program mencetak `steps = 3` di setiap tingkat. Itu bukan alasan untuk mengandalkan luapan, karena juri tidak mengompilasi dengan `-fwrapv`, dan karena nilai yang berputar tetaplah nilai yang salah: di FCTRL pangkat yang berputar menjadi negatif, dan loop tetap berperilaku salah.

5. Di macOS dengan Apple clang, `bits/stdc++.h` tidak ada: galat kompilasi. Di laptop Apple berprosesor ARM, nama target x86 tidak dikenal: galat kompilasi. Dengan sebagian versi g++ dan header, pragma target sebelum include memberi galat inlining seperti di Subbab 5.6.3: galat kompilasi. Di juri yang prosesornya tidak memiliki salah satu perluasan yang disebut, program bisa dihentikan dengan SIGILL: runtime error, tetapi hanya pada uji yang mencapai kode pemakai instruksi itu, sehingga tampak seperti bug yang bergantung pada data. Di mesin Cube milik SPOJ perluasan-perluasan itu ada, jadi templat itu kebetulan berhasil di sana.

6. Tambahkan `import sys` dan, sebelum mencetak, `sys.set_int_max_str_digits(0)` (0 berarti tanpa batas), dijaga untuk versi yang lebih tua:

```
if hasattr(sys, "set_int_max_str_digits"):
    sys.set_int_max_str_digits(0)
```

100000! memiliki 456.574 digit. Tanpa panggilan itu, `str()` atau `print()` atas bilangan itu memunculkan `ValueError: Exceeds the limit (4300 digits) for integer string conversion`, dan juri melaporkan runtime error. Penjaga `hasattr` membuat program tetap bekerja di Python 3.10 ke bawah, tempat fungsi itu tidak ada dan tidak ada batasnya.

Daftar Pustaka

[1] Sphere Online Judge, "FCTRL - Factorial." [Online]. Available: https://www.spoj.com/problems/FCTRL/ (accessed Sep. 2026).

[2] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 6th ed., revised by D. R. Heath-Brown and J. H. Silverman. Oxford, UK: Oxford University Press, 2008.

[3] hkktr, "polski-spoj: solutions for POLSKI SPOJ problems," README, GitHub. [Online]. Available: https://github.com/hkktr/polski-spoj (accessed Sep. 2026).

[4] GCC Steering Committee, "GCC 16.1 Released," gcc-announce mailing list, Apr. 30, 2026. [Online]. Available: https://gcc.gnu.org/pipermail/gcc-announce/2026/000190.html

[5] Wikipedia, "SPOJ." [Online]. Available: https://en.wikipedia.org/wiki/SPOJ (accessed Sep. 2026).

[6] Intel Corporation, "Intel Xeon Processor E3-1220 v5 (8M Cache, 3.00 GHz): Product Specifications." [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/88172/intel-xeon-processor-e31220-v5-8m-cache-3-00-ghz/specifications.html (accessed Sep. 2026).

[7] Python Software Foundation, "Built-in Types: Integer string conversion length limitation," *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/stdtypes.html#int-max-str-digits

[8] cppreference.com, "Fundamental types." [Online]. Available: https://en.cppreference.com/w/cpp/language/types (accessed Sep. 2026).

[9] Microsoft, "Data Type Ranges," *Microsoft C++ Language Reference*, Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/cpp/cpp/data-type-ranges (accessed Sep. 2026).

[10] Free Software Foundation, "Options to Request or Suppress Warnings" and "Program Instrumentation Options," *Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html (accessed Sep. 2026).

[11] cppreference.com, "Arithmetic operators: Overflows." [Online]. Available: https://en.cppreference.com/w/cpp/language/operator_arithmetic (accessed Sep. 2026).

[12] Free Software Foundation, "Debug Mode: Using," *The GNU C++ Library Manual*. [Online]. Available: https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug_mode_using.html (accessed Sep. 2026).

[13] Free Software Foundation, "Options That Control Optimization," *Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html (accessed Sep. 2026).

[14] Free Software Foundation, "Function Specific Option Pragmas," *Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Function-Specific-Option-Pragmas.html (accessed Sep. 2026).

[15] LLVM Project, "libc++ C++ Standard Library." [Online]. Available: https://libcxx.llvm.org/ (accessed Sep. 2026).

[16] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Singapore: Lulu, 2020.

Bab 6

Memori dan Pilihan Struktur Data dalam Batas Byte

Setiap juri memberi program Anda dua anggaran: detik dan byte. Bab 3 mengajari Anda membelanjakan yang pertama; bab ini membahas yang kedua. Di akhir bab Anda akan mampu memperkirakan, sebelum menulis kode apa pun, berapa byte yang dibutuhkan sebuah struktur data; menyebutkan harga sebenarnya satu elemen di kontainer C++ dan tipe Python yang umum, berdasarkan pengukuran dan bukan tebakan; menyelesaikan PRIME1 dari SPOJ dengan saringan bersegmen yang hanya butuh seratus kilobyte, padahal saringan biasa butuh satu gigabyte; menyelesaikan LKS, knapsack yang tabel versi buku teksnya akan memenuhi empat gigabyte, dengan satu baris saja sebesar delapan megabyte; menjelaskan mengapa program rekursif yang benar bisa mogok pada masukan yang dalam, lalu menulis ulang agar tidak mungkin mogok; dan mengenali tiga cara diam-diam memori ikut memakan waktu.

6.1 Koper dengan Batas Berat

Pak Harun menjual kain batik dan tenun Makassar di sebuah pasar mingguan di Surabaya. Ia terbang dengan maskapai berbiaya rendah yang tiketnya hanya mencakup satu tas kabin seberat 7 kilogram; bagasi tercatat harus dibayar lagi, dan tas yang ketahuan kelebihan berat di pintu naik pesawat biayanya lebih mahal lagi. Bertahun-tahun ia menjadi ahli berkemas. Ia tidak berkemas menurut harga barang; ia berkemas menurut *beratnya*. Ia tahu bahwa selembar sarung terlipat beratnya sekitar 400 gram, bahwa kotak kayu seukuran itu tiga kali lebih berat dan tidak melindungi apa pun yang tak bisa dilindungi kertas tisu, dan bahwa gantungan plastik tempat pemasoknya mengirim kemeja lebih berat daripada kemejanya. Sebelum berangkat ia menimbang tasnya di timbangan kamar mandi, karena mengetahuinya di pintu naik pesawat sudah terlambat.

Batas memori sebuah juri adalah 7 kilogram milik Pak Harun. Pernyataan soal menyebutkan berapa memori yang boleh dipakai program Anda, dan bila program memakai lebih, juri menghentikannya dan melaporkan *Memory limit exceeded*, atau sama seringnya runtime error, karena alokasi yang melewati batas itu gagal. Seperti maskapai, juri tidak peduli apa yang Anda niatkan untuk dibawa; ia menimbang apa yang benar-benar Anda bawa. Dan seperti Pak Harun, pemrogram yang baik tahu berat setiap barang: bahwa `set<int>` berisi sejuta bilangan empat puluh kali lebih berat daripada `vector<int>` berisi bilangan yang sama, bahwa list Python berisi bilangan bulat sepuluh kali lebih berat daripada `array`, dan bahwa sebagian struktur, seperti gantungan baju itu, lebih mahal daripada isinya.

Di SPOJ batas memori bawaannya 1536 MB [1], yang terdengar murah hati, dan untuk kebanyakan soal memang demikian. Namun soal-soal di bab ini justru soal yang struktur datanya yang paling jelas membutuhkan beberapa gigabyte, dan juri lain lebih ketat: 256 MB umum di banyak platform lomba. Yang lebih penting, memori bukan sekadar batas. Setiap byte yang disentuh program harus berpindah dari memori utama ke prosesor, dan memindahkan byte memakan waktu. Program yang berkemas ringan biasanya juga program yang berjalan cepat.

6.2 Gagasannya dengan Kata-Kata Sederhana

6.2.1 Tempat program menyimpan datanya

Memori program yang sedang berjalan terbagi atas beberapa wilayah dengan aturan berbeda. Dua di antaranya penting bagi kode lomba.

> **DEFINISI.** Satu *byte* adalah 8 bit; 1 KB (kibibyte) adalah 1024 byte, 1 MB adalah 1024 KB, dan 1 GB adalah 1024 MB. *Stack* adalah wilayah berukuran tetap yang cukup kecil (biasanya 1 MB di Windows dan 8 MB di Linux) yang menyimpan variabel lokal fungsi-fungsi yang sedang berjalan; setiap pemanggilan menambah satu *frame* dan setiap kembalian membuangnya. *Heap* adalah wilayah besar tempat kontainer seperti `std::vector` dan semua objek Python mendapat memorinya; heap tumbuh sesuai permintaan sampai batas memori. *Memori puncak* sebuah program adalah jumlah terbesar yang pernah dipegangnya pada satu saat; juri mengukur puncak ini.

Perbedaan ini penting karena kedua wilayah gagal dengan cara yang berbeda. Meminta heap lebih dari batasnya membuat alokasi gagal: di C++ `new` melempar `std::bad_alloc`, dan program berhenti dengan runtime error kecuali ia menangkapnya. Meluapkan stack tidak memberi peringatan apa pun: program menyentuh memori di luar ujung stack dan dimatikan dengan *segmentation fault*. Subbab 6.6 menunjukkan keduanya.

6.2.2 Rumus penimbangan

Kebanyakan memori program lomba berada di beberapa larik atau kontainer besar, dan masing-masing berharga kira-kira

$$\text{byte} \approx n \times b \tag{6.1}$$

dengan n banyaknya elemen dan b byte per elemen. Rumusnya sepele; pekerjaannya adalah mengetahui b, yang tidak selalu sama dengan ukuran nilai yang Anda simpan. `vector<int>` berharga 4 byte per elemen, tepat ukuran sebuah `int`. `set<int>` menyimpan `int` yang sama di dalam sebuah *simpul* dengan tiga pointer dan sebuah penanda warna, dan harganya 40. Subbab 6.3 mengukur b untuk kontainer yang kemungkinan besar Anda pakai.

Dua angka layak dihafal untuk pemeriksaan yang sebaiknya Anda lakukan sebelum menulis kode:

- 10^6 elemen 4 byte kira-kira 4 MB; 10^8 elemen 4 byte kira-kira 400 MB;
- 10^9 apa pun tidak akan muat.

Jika Persamaan (6.1) memberi lebih dari separuh batas, berhenti dan carilah struktur yang menyimpan lebih sedikit, karena program juga memerlukan memori untuk masukan, keluaran, pustaka standar, dan interpreter.

6.2.3 Memori juga memakan waktu

Tiga efek membuat pilihan memori berpengaruh pada batas waktu juga.

1. **Alokasi.** Membuat satu simpul untuk setiap elemen, seperti yang dilakukan `set` dan `list`, memanggil pengalokasi memori sekali per elemen; membuat satu larik berisi sejuta elemen memanggilnya sekali.
2. **Cache.** Prosesor menyimpan memori yang baru dipakai di *cache* yang kecil dan cepat. Larik bersifat bersebelahan, sehingga membaca elemen i ikut membawa elemen $i + 1$, $i + 2$, ... ke

dalam cache tanpa biaya. Simpul-simpul pohon atau linked list tersebar, dan setiap langkah bisa menunggu memori utama, yang kira-kira seratus kali lebih lambat daripada cache tercepat [2].

3. **Inisialisasi.** Mengosongkan larik berisi n elemen memerlukan n langkah. Algoritma yang mengalokasikan satu gigabyte baru untuk setiap kasus uji menghabiskan waktunya untuk mengosongkan, bukan menghitung.

Aturan praktis yang mengikutinya adalah aturan yang akan diberikan Pak Harun: bawa lebih sedikit, dan kemas rapat-rapat.

Satu gambaran membantu mengingat efek kedua. Bayangkan seorang pelayan warung yang harus mengambil sepuluh piring. Jika piring-piring itu bertumpuk di satu rak, ia mengambil setumpuk sekaligus dalam satu kali jalan. Jika setiap piring diletakkan di meja yang berbeda di seluruh ruangan, ia harus berjalan sepuluh kali. Larik adalah tumpukan di satu rak; pohon dan linked list adalah piring yang tersebar di meja-meja. Prosesor, seperti si pelayan, selalu membawa "setumpuk" memori sekaligus, biasanya 64 byte, jadi data yang berdampingan datang hampir gratis, sedangkan data yang tersebar menuntut satu perjalanan untuk setiap potongnya.

6.3 Harga Sebenarnya Satu Elemen

6.3.1 Mengukurnya di C++

Menebak ukuran sebuah kontainer tidak dapat diandalkan, karena ukurannya bergantung pada implementasi pustaka. Mengukurnya mudah. Kontainer C++ mendapat memorinya melalui sebuah *allocator*, objek dengan dua fungsi: `allocate(n)`, yang mengembalikan ruang untuk n elemen, dan `deallocate(p, n)`, yang mengembalikannya. **Kode 6.1** mendefinisikan allocator `Counting` yang melakukan persis apa yang dilakukan allocator standar sambil mencatat total byte yang sedang terpakai. Setiap kontainer dibangun dengan sejuta nilai sementara totalnya diamati.

Kode 6.1 `memsize.cpp`: allocator penghitung mengukur byte heap yang dipakai setiap kontainer untuk 10^6 nilai.

```cpp
// Berapa byte sebenarnya harga satu elemen? Allocator penghitung mencatat
// setiap byte heap yang diminta kontainer selama memuat N = 1.000.000 nilai.
#include <bitset>
#include <cstdio>
#include <functional>
#include <list>
#include <map>
#include <set>
#include <unordered_set>
#include <vector>

static long long liveBytes = 0;             // byte heap yang sedang terpakai

template <class T>
struct Counting {                           // allocator C++11 paling ringkas
    using value_type = T;
    Counting() = default;
    template <class U> Counting(const Counting<U>&) {}
    T* allocate(std::size_t n) {
        liveBytes += (long long)(n * sizeof(T));
        return std::allocator<T>().allocate(n);
    }
    void deallocate(T* p, std::size_t n) {
        liveBytes -= (long long)(n * sizeof(T));
        std::allocator<T>().deallocate(p, n);
    }
};
template <class T, class U>
bool operator==(const Counting<T>&, const Counting<U>&) { return true; }
template <class T, class U>
bool operator!=(const Counting<T>&, const Counting<U>&) { return false; }

const int N = 1000000;

void report(const char* name, long long bytes) {
    std::printf("%-22s %6.2f bytes/element\n", name, (double)bytes / N);
}
```

Kode 6.1 (Lanjutan) *memsize.cpp: allocator penghitung mengukur byte heap yang dipakai setiap kontainer untuk 10^6 nilai.*

```
int main() {
    { std::vector<char, Counting<char>> v(N); report("vector<char>", liveBytes); }
    { std::vector<bool, Counting<bool>> v(N); report("vector<bool>", liveBytes); }
    { std::vector<int, Counting<int>> v(N); report("vector<int>", liveBytes); }
    { std::vector<long long, Counting<long long>> v(N);
      report("vector<long long>", liveBytes); }
    { std::vector<int, Counting<int>> v;
      for (int i = 0; i < N; i++) v.push_back(i);
      report("vector<int> push_back", liveBytes); }
    report("bitset<N>", (long long)sizeof(std::bitset<N>));
    { std::list<int, Counting<int>> l;
      for (int i = 0; i < N; i++) l.push_back(i);
      report("list<int>", liveBytes); }
    { std::set<int, std::less<int>, Counting<int>> s;
      for (int i = 0; i < N; i++) s.insert(i);
      report("set<int>", liveBytes); }
    { std::map<int, int, std::less<int>, Counting<std::pair<const int, int>>> m;
      for (int i = 0; i < N; i++) m[i] = i;
      report("map<int,int>", liveBytes); }
    { std::unordered_set<int, std::hash<int>, std::equal_to<int>, Counting<int>> s;
      for (int i = 0; i < N; i++) s.insert(i);
      report("unordered_set<int>", liveBytes); }
    struct Padded { char tag; int value; char flag; };  // 6 byte data
    struct Packed { int value; char tag; char flag; };  // data sama, urutan diubah
    std::printf("sizeof(Padded) = %zu, sizeof(Packed) = %zu\n",
                sizeof(Padded), sizeof(Packed));
    return 0;
}
```

Setiap pengukuran berada di dalam pasangan kurung kurawalnya sendiri, sehingga kontainer dihancurkan, dan memorinya dikembalikan, sebelum kontainer berikutnya dibangun; karena itulah `liveBytes` dapat dibaca langsung. Allocator ditulis dalam bentuk paling ringkas yang diizinkan C++11: sebuah `value_type`, konstruktor konversi, `allocate`, `deallocate`, dan dua operator pembanding. `std::bitset` tidak dapat menerima allocator, jadi untuknya kami mencetak `sizeof`, yang merupakan seluruh ukurannya. Dikompilasi dengan g++ 13.3, program mencetak:

```
vector<char>             1.00 bytes/element
vector<bool>             0.12 bytes/element
vector<int>              4.00 bytes/element
vector<long long>        8.00 bytes/element
vector<int> push_back    4.19 bytes/element
bitset<N>                0.12 bytes/element
list<int>               24.00 bytes/element
set<int>                40.00 bytes/element
map<int,int>            40.00 bytes/element
unordered_set<int>      27.58 bytes/element
sizeof(Padded) = 12, sizeof(Packed) = 8
```

6.3.2 Membaca angka-angkanya

Lima baris pertama adalah kontainer *bersebelahan*, yang menyimpan nilai-nilainya berdampingan. Harganya sama dengan harga nilainya: 1 byte untuk `char`, 4 untuk `int`, 8 untuk `long long`. Baris `vector<int> push_back` sedikit di atas 4, karena vector yang tumbuh lewat `push_back` menggandakan kapasitasnya setiap kali penuh; setelah sejuta kali push, kapasitasnya $2^{20} = 1\,048\,576$, dan $1\,048\,576 \times 4/10^6 = 4.19$ byte per elemen. Memanggil `reserve(n)` lebih dahulu, atau membuat vector langsung dengan ukuran akhirnya, menghapus kelebihan 5% itu. Selama setiap penggandaan, blok lama dan blok baru ada bersamaan, sehingga *puncak* vector

yang tumbuh bisa tiga kali ukuran yang dicapainya di akhir; untuk larik besar, tetapkan ukurannya sekali.

`vector<bool>` dan `bitset` menyimpan satu bit per elemen, delapan kali lebih hemat daripada `char`. Keduanya pilihan tepat untuk tabel besar berisi penanda ya/tidak, dengan dua catatan. `vector<bool>` adalah kasus khusus dalam standar: elemennya bukan objek `bool` sungguhan, sehingga `&v[i]` bukan `bool*`, dan kode yang mengambil referensi ke elemen, atau membaca dan menulis elemen berbeda dari beberapa thread, berperilaku lain daripada vector lainnya [3]. Selain itu, membaca satu bit memerlukan pergeseran dan penyaringan, sehingga `vector<char>` sering sedikit lebih cepat bila memorinya cukup.

Empat baris terakhir adalah kontainer *berbasis simpul*. Setiap elemen tinggal di simpul yang dialokasikan terpisah:

- simpul `list<int>` berisi dua pointer (berikut dan sebelumnya, masing-masing 8 byte) dan `int`-nya, dibulatkan ke atas menjadi 24 byte;
- simpul `set<int>` atau `map<int,int>` adalah simpul pohon merah-hitam: tiga pointer (induk, kiri, kanan), sebuah warna, dan nilainya, 40 byte;
- simpul `unordered_set<int>` berisi satu pointer dan nilainya (16 byte), ditambah larik bucket masing-masing 8 byte, sekitar 1,45 bucket untuk setiap elemen pada ukuran ini.

Angka-angka ini adalah byte yang *diminta* kontainer. Pengalokasi sistem menambahkan catatan administrasinya sendiri pada setiap blok, biasanya 8 sampai 16 byte, sehingga harga sebenarnya sebuah simpul lebih besar lagi: `set<int>` berisi sejuta elemen menempati sekitar 48 MB, dua belas kali `vector<int>`. Terakhir, baris penutup menunjukkan bahwa satu `struct` pun bisa memboroskan tempat. `int` di dalam `Padded` harus dimulai di alamat yang habis dibagi 4, jadi kompiler menyisipkan tiga byte kosong setelah `char` pertama, dan tiga lagi di ujung agar larik struct itu tetap sejajar. Meletakkan medan terbesar lebih dahulu, seperti pada `Packed`, menghemat sepertiga.

6.3.3 Mengukurnya di Python

Di Python setiap nilai adalah *objek* dengan kepala (*header*), dan list menyimpan pointer ke objek, bukan nilainya sendiri. **Kode 6.2** memakai modul standar `tracemalloc`, yang menghitung setiap byte yang dialokasikan interpreter [4].

Kode 6.2 `memsize.py`: `tracemalloc` mengukur byte per elemen kontainer Python yang memuat 10^6 nilai.

```python
# Harga satu elemen di Python: tracemalloc menghitung setiap byte yang
# dialokasikan CPython selama kontainer berisi N = 1.000.000 nilai dibangun.
import array
import tracemalloc

N = 1000000

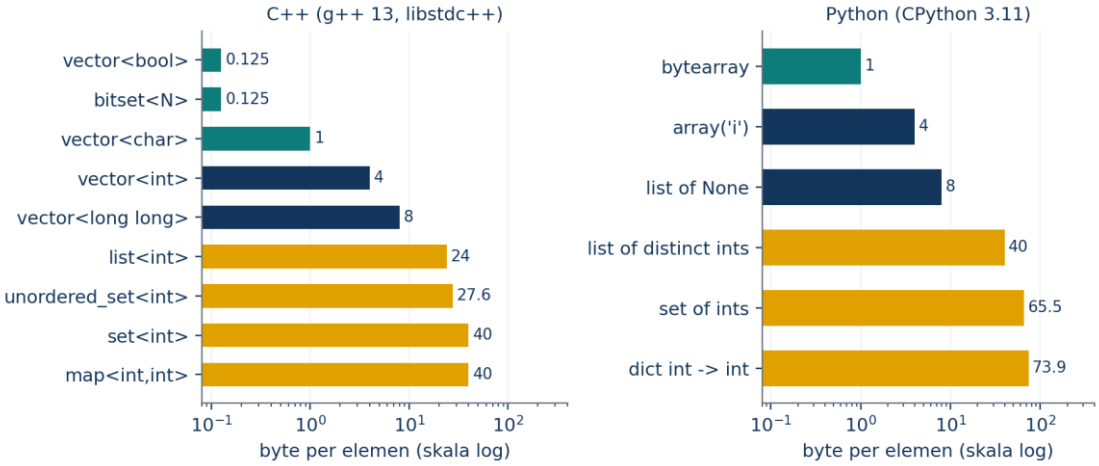

def measure(name, build):
    tracemalloc.start()
    obj = build()                       # biarkan tetap hidup selama diukur
    used, _peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    del obj
    print("%-28s %6.2f bytes/element" % (name, used / N))


measure("bytearray", lambda: bytearray(N))
measure("array('i')", lambda: array.array("i", [0]) * N)
measure("list of None", lambda: [None] * N)
measure("list of small ints (0..255)", lambda: [i % 256 for i in range(N)])
measure("list of distinct ints", lambda: list(range(10**6, 10**6 + N)))
measure("set of ints", lambda: set(range(10**6, 10**6 + N)))
measure("dict int -> int", lambda: {i: i for i in range(10**6, 10**6 + N)})
```

```
bytearray                     1.00 bytes/element
array('i')                    4.00 bytes/element
list of None                  8.00 bytes/element
list of small ints (0..255)   8.45 bytes/element
list of distinct ints        40.00 bytes/element
set of ints                  65.55 bytes/element
dict int -> int              73.94 bytes/element
```

List berisi `None` memperlihatkan harga list itu sendiri: satu pointer 8 byte per elemen. Bila elemennya bilangan bulat yang berbeda-beda, setiap bilangan juga menjadi objek `int` berukuran 28 byte, dibulatkan ke 32 oleh pengalokasi, sehingga totalnya 40. Bilangan bulat kecil dari -5 sampai 256 adalah objek bersama yang dibuat CPython sekali saja, dan karena itu list bilangan kecil hanya sedikit lebih mahal daripada pointer-pointernya. `set` atau `dict` menyimpan tabel hash dengan slot kosong agar pencarian tetap cepat, dan harganya 65 sampai 74 byte per bilangan. Di puncak tabel, `bytearray` dan `array` menyimpan nilai mentah, 1 dan 4 byte masing-masing, persis seperti vector C++. Pelajaran untuk Python terus terang saja: list berisi sejuta bilangan bulat beratnya 40 MB, dan sepuluh juta beratnya 400 MB. Untuk tabel besar berisi bilangan kecil atau penanda, pakailah `bytearray` atau `array`.

Gambar 6.1 menempatkan kedua pengukuran berdampingan, pada skala logaritmik.



Gambar 6.1 Byte per elemen untuk 10^6 nilai, diukur dengan allocator penghitung di C++ dan dengan `tracemalloc` di Python; skalanya logaritmik.

Tabel 6.1 meringkas pengukuran itu menjadi aturan yang dapat Anda terapkan sambil membaca pernyataan soal.

Tabel 6.1 Aturan praktis memilih kontainer menurut beratnya (g++ 13 dengan libstdc++, CPython 3.11).

Kebutuhan	Pilihan C++	Byte/elemen	Pilihan Python	Byte/elemen
penanda ya/tidak per bilangan	`vector<char>`, `vector<bool>`	1, 0.125	`bytearray`	1
bilangan di bawah 2^{31}	`vector<int>`	4	`array('i')`	4
bilangan di bawah 2^{63}	`vector<long long>`	8	`array('q')`	8
himpunan terurut dengan sisip/hapus	`set<int>`	40 (+ pengalokasi)	list terurut + `bisect`	8 sampai 40
pencarian menurut kunci	urutkan + binary search, atau `unordered_map`	8, atau 30 sampai 50	`dict`	sekitar 74

6.4 PRIME1: Rentang yang Terlalu Besar untuk Disimpan

6.4.1 Soalnya

Soal SPOJ PRIME1, "Prime Generator" [5], memberi paling banyak 10 kasus uji, masing-masing sepasang bilangan m dan n dengan $1 \le m \le n \le 10^9$ dan $n - m \le 10^5$. Untuk setiap pasangan, program harus mencetak setiap bilangan prima p dengan $m \le p \le n$, satu per baris, dengan satu baris kosong di antara kasus-kasus uji. Batas waktunya 6 detik dan batas memorinya 1536 MB.

Bilangan *prima* adalah bilangan bulat lebih besar dari 1 yang pembaginya hanya 1 dan dirinya sendiri: 2, 3, 5, 7, 11, dan seterusnya. Cara klasik mendaftar bilangan prima sampai batas L adalah *saringan Eratosthenes*: tuliskan semua bilangan dari 2 sampai L, dan untuk setiap bilangan p yang belum dicoret, coret kelipatannya $p^2, p^2 + p, p^2 + 2p, \ldots$. Yang tersisa tanpa coretan adalah bilangan prima.

6.4.2 Menimbang solusi yang paling jelas

Solusi yang paling jelas menyaring sampai 10^9 sekali lalu mencetak rentang yang diminta. Menurut Persamaan (6.1), dengan satu penanda per bilangan,

$$10^9 \times 1 \text{ byte} \approx 954 \text{ MB}, \qquad 10^9 \times 4 \text{ byte} \approx 3.7 \text{ GB} \tag{6.2}$$

masing-masing sebagai `char` dan sebagai `int`. Versi `int` jauh di atas batas; versi `char` nyaris menyelinap di bawah 1536 MB milik SPOJ, tetapi akan gagal di juri 256 MB mana pun, dan mencoret satu miliar penanda memakan beberapa detik sebelum baris pertama dicetak. `vector<bool>` akan membutuhkan 119 MB, yang muat, tetapi tetap menghabiskan waktunya menyaring 999.900.000 bilangan yang tidak ditanyakan siapa pun. Rentang-rentang yang diminta seluruhnya berisi paling banyak $10 \times 100\,001$ bilangan; masalahnya adalah bagaimana hanya menyaring bilangan-bilangan itu.

6.4.3 Gagasannya: saringan kecil dan sebuah jendela

Dua fakta membuatnya mungkin.

Fakta 1. Setiap bilangan komposit $x \le n$ memiliki faktor prima $p \le \sqrt{n}$. Memang, jika $x = a \times b$ dengan $1 < a \le b$, maka $a \times a \le a \times b = x \le n$, jadi $a \le \sqrt{n}$, dan setiap faktor prima a paling besar a. Karena $\sqrt{10^9} < 31\,623$, bilangan prima sampai 31.623 cukup untuk mengenali setiap bilangan komposit sampai 10^9. Banyaknya hanya 3401.

Fakta 2. Untuk mencoret kelipatan p di dalam *jendela* $[m, n]$, kita tidak memerlukan bilangan-bilangan di bawah m. Kelipatan p pertama yang paling sedikit m adalah

$$f(p) = \left\lceil \frac{m}{p} \right\rceil \cdot p = \left\lfloor \frac{m + p - 1}{p} \right\rfloor \cdot p \tag{6.3}$$

dengan $\lceil x \rceil$ berarti x dibulatkan ke atas; bentuk kedua hanya memakai pembagian bulat. Dari situ, kelipatannya adalah $f(p), f(p) + p, f(p) + 2p, \ldots$ sampai n. Kita mulai dari $\max(f(p), p^2)$ alih-alih $f(p)$, karena dua alasan: kelipatan di bawah p^2 memiliki faktor prima yang lebih kecil dan toh dicoret olehnya, dan, lebih penting lagi, p sendiri bisa berada di dalam jendela dan tidak boleh dicoret.

Jadi saringan bersegmen bekerja dengan dua larik kecil: saringan berisi 31.624 penanda yang menemukan 3401 bilangan prima kecil satu kali, dan jendela berisi $n - m + 1 \le 100\,001$ penanda, satu per bilangan dalam kasus uji yang sedang dikerjakan, yang dipakai ulang untuk setiap kasus. **Gambar 6.2** memperlihatkan gambarannya.



Gambar 6.2 Saringan bersegmen: bilangan prima kecil sampai $\sqrt{10^9}$ mencoret kelipatannya di dalam jendela $[m, n]$; bilangan di luar jendela tidak pernah disimpan.

Berapa banyak kerjanya? Untuk setiap bilangan prima kecil p, jendela menerima sekitar $(n-m)/p$ coretan, sehingga totalnya

$$\sum_{p \le \sqrt{n}} \frac{n-m}{p} + \pi(\sqrt{n}) \approx (n-m)\ln\ln\sqrt{n} + \pi(\sqrt{n}) \tag{6.4}$$

memakai taksiran klasik bahwa jumlah $1/p$ atas bilangan prima sampai x tumbuh seperti $\ln\ln x$ [6], dengan $\pi(x)$ banyaknya bilangan prima sampai x (suku kedua menghitung bilangan prima yang loopnya tidak mengerjakan apa-apa). Dengan $n - m = 10^5$ dan $\ln\ln 31\,623 \approx 2.3$, itu sekitar 230.000 coretan dan 3401 awal loop per kasus uji: sangat kecil. Memorinya: $31\,624 + 100\,001$ byte penanda dan 3401×4 byte bilangan prima, sekitar 140 KB seluruhnya, dibandingkan 954 MB pada Persamaan (6.2).

6.4.4 Contoh yang dikerjakan dengan tangan

Ambil jendela [24,30]. Jendela ini berisi 7 bilangan, dan $\sqrt{30} < 6$, sehingga hanya bilangan prima 2, 3, dan 5 yang berperan.

- $p = 2$: menurut Persamaan (6.3), $f(2) = \lfloor 25/2 \rfloor \times 2 = 24$, dan $\max(24,4) = 24$. Coret 24, 26, 28, 30.
- $p = 3$: $f(3) = \lfloor 26/3 \rfloor \times 3 = 24$. Coret 24, 27, 30.
- $p = 5$: $f(5) = \lfloor 28/5 \rfloor \times 5 = 25$. Coret 25, 30.
- $p = 7$: $7^2 = 49 > 30$, berhenti.

Bilangan yang tidak tercoret hanya 29, dan 29 memang satu-satunya bilangan prima antara 24 dan 30.

6.4.5 Program-programnya

Kode 6.3 adalah program C++.

Kode 6.3 `prime1.cpp`: saringan bersegmen; memorinya bergantung pada $\sqrt{n}$ dan $n - m$, tidak pernah pada n.

```cpp
// PRIME1 (SPOJ): semua prima di [m, n], n <= 10^9, n - m <= 10^5.
// Saringan bersegmen: memori bergantung pada sqrt(n) dan n - m, bukan pada n.
#include <cstdio>
#include <vector>

int main() {
    const int LIMIT = 31623;                 // 31623^2 > 10^9
    std::vector<char> composite(LIMIT + 1, 0);
    std::vector<int> small;                  // bilangan prima sampai sqrt(10^9)
    for (int i = 2; i <= LIMIT; i++) {
        if (composite[i]) continue;
        small.push_back(i);
        for (long long j = (long long)i * i; j <= LIMIT; j += i) composite[j] = 1;
    }
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    for (int tc = 0; tc < t; tc++) {
        long long m, n;
        if (std::scanf("%lld %lld", &m, &n) != 2) break;
        if (tc > 0) std::printf("\n");       // baris kosong di antara kasus uji
        std::vector<char> crossed(n - m + 1, 0);  // crossed[k] mewakili m + k
        for (int p : small) {
            long long pp = (long long)p * p;
            if (pp > n) break;
            long long first = (m + p - 1) / p * p;   // kelipatan pertama >= m
            if (first < pp) first = pp;              // sisanya dicoret prima lain
            for (long long x = first; x <= n; x += p) crossed[x - m] = 1;
        }
        for (long long x = m; x <= n; x++)
            if (x >= 2 && !crossed[x - m]) std::printf("%lld\n", x);
    }
    return 0;
}
```

Tiga rincian layak dilihat dua kali. Hasil kali `(long long)i * i` dan `(long long)p * p` dihitung dalam 64 bit: $31\,623^2$ muat dalam `int`, tetapi kebiasaan ini tidak memakan biaya dan soal berikutnya mungkin tidak semurah hati itu. Jendelanya adalah `vector<char>` lokal yang dibuat untuk setiap kasus uji, sehingga selalu bersih; larik global harus dibersihkan secara manual, dan lupa membersihkannya adalah bug klasik di Bab 7. Bilangan 1 dikecualikan oleh uji `x >= 2`, karena tidak ada yang mencoretnya: 1 sama sekali tidak memiliki faktor prima.

Kode 6.4 adalah program Python. Alih-alih loop yang mencoret satu bilangan setiap kali, program ini menugaskan satu irisan utuh: `alive[first - m::p] = bytes(k)` mengisi setiap penanda ke-p mulai `first` dengan 0 dalam satu operasi yang berjalan dengan kecepatan C di dalam interpreter. Panjang `k` dari objek `bytes` berisi nol itu harus tepat sama dengan banyaknya posisi dalam irisan, dan `len(range(first, n + 1, p))` menghitungnya tanpa membangun rentangnya.

Kode 6.4 `prime1.py`: saringan bersegmen pada jendela `bytearray`, mencoret satu irisan utuh sekaligus.

```python
# PRIME1 (SPOJ): semua prima di [m, n] dengan saringan bersegmen pada bytearray.
# Penugasan irisan mencoret satu barisan aritmetika utuh dengan kecepatan C.
import sys


def small_primes(limit):
    flags = bytearray([1]) * (limit + 1)
    flags[0] = flags[1] = 0
    for i in range(2, int(limit ** 0.5) + 1):
        if flags[i]:
            flags[i * i::i] = bytes(len(range(i * i, limit + 1, i)))
    return [i for i in range(limit + 1) if flags[i]]


def main():
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    small = small_primes(31623)
    blocks = []
    for tc in range(t):
        m, n = int(data[1 + 2 * tc]), int(data[2 + 2 * tc])
        alive = bytearray([1]) * (n - m + 1)     # alive[k] mewakili m + k
        for p in small:
            if p * p > n:
                break
            first = max(p * p, (m + p - 1) // p * p)
            if first <= n:
                alive[first - m::p] = bytes(len(range(first, n + 1, p)))
        for x in range(m, min(2, n + 1)):        # 1 bukan bilangan prima
            alive[x - m] = 0
        blocks.append("\n".join(str(m + k) for k in range(n - m + 1) if alive[k]))
    sys.stdout.write("\n\n".join(b for b in blocks) + "\n")


if __name__ == "__main__":
    main()
```

Kami menguji dengan tiga jendela buatan sendiri: satu yang memuat bilangan prima terkecil dan bilangan 1, satu di puncak rentang, dan satu yang kita kerjakan dengan tangan.

```
3
1 20
999999937 1000000000
24 30
```

```
2
3
5
7
11
13
17
19

999999937

29
```

Program Python mencetak baris-baris yang sama. Bilangan 999999937 adalah bilangan prima terbesar di bawah 10^9, dan jendela-jendela dipisahkan baris kosong sebagaimana diminta soal. Pada masukan terberat yang diizinkan soal, sepuluh jendela masing-masing 100.001 bilangan di puncak rentang, program C++ memerlukan 0,01 detik dan program Python 0,05 detik, dan keduanya bertahan di sekitar 10 MB, yang sebagian besar adalah program dan interpreter itu sendiri.

6.5 LKS: Ketika Tabel Tidak Muat

6.5.1 Soalnya

Soal SPOJ LKS, "Large Knapsack" [7], adalah soal knapsack 0/1 klasik dengan bilangan besar. Sebuah knapsack berkapasitas K, dan ada N barang, barang ke-i bernilai v_i dan berbobot w_i. Pilih sekumpulan barang, masing-masing paling banyak sekali, yang total bobotnya paling besar K dan total nilainya sebesar mungkin; cetak nilai itu. Batasannya $K \le 2\,000\,000$ dan $N \le 500$, nilai dan bobot sampai 10^7, dan soal menjamin bahwa setiap hasil antara muat dalam bilangan bulat bertanda 32-bit. Batas waktunya 2 detik.

Koper Pak Harun adalah soal ini: $K = 7000$ gram, barang-barangnya adalah lembaran kain yang bisa ia bawa, dan nilainya adalah harga jual yang ia harapkan.

6.5.2 Solusi buku teks dan beratnya

Bab 9 mengembangkan pemrograman dinamis secara mendalam; di sini kita hanya memerlukan rekurens bakunya. Misalkan $B_i(c)$ nilai terbaik yang memakai hanya i barang pertama dengan kapasitas c. Barang ke-i bisa ditinggalkan, dan nilai terbaiknya $B_{i-1}(c)$, atau diambil, yang memerlukan $c \ge w_i$ dan memberi $B_{i-1}(c - w_i) + v_i$. Jadi

$$B_i(c) = \max(B_{i-1}(c),\ B_{i-1}(c - w_i) + v_i), \qquad B_0(c) = 0 \tag{6.5}$$

dengan pilihan kedua hanya ada bila $c \ge w_i$, dan jawabannya $B_N(K)$ [8]. Buku teks menyimpan seluruh tabel, satu baris per barang dan satu kolom per kapasitas. **Kode 6.5** adalah program itu.

Kode 6.5 `lks_table.cpp`: knapsack dengan tabel penuh; benar, tetapi uji terbesar membutuhkan sekitar 4 GB.

```cpp
// LKS dengan tabel penuh ala buku teks: table[i][c] untuk setiap barang i dan
// kapasitas c. Benar, dan baik untuk K kecil -- tetapi uji terbesar butuh
// (N + 1) * (K + 1) int, sekitar 4 GB, jauh di atas 1536 MB milik juri.
#include <cstdio>
#include <vector>

int main() {
    int K, N;
    if (std::scanf("%d %d", &K, &N) != 2) return 0;
    std::vector<std::vector<int>> table(N + 1, std::vector<int>(K + 1, 0));
    for (int i = 1; i <= N; i++) {
        int value, weight;
        if (std::scanf("%d %d", &value, &weight) != 2) return 0;
        for (int c = 0; c <= K; c++) {
            table[i][c] = table[i - 1][c];                  // lewati barang i
            if (c >= weight && table[i - 1][c - weight] + value > table[i][c])
                table[i][c] = table[i - 1][c - weight] + value;  // ambil barang itu
        }
    }
    std::printf("%d\n", table[N][K]);
    return 0;
}
```

Timbanglah dengan Persamaan (6.1): tabel itu berisi $(N + 1)(K + 1)$ entri 4 byte, jadi pada batas-batasnya ia membutuhkan

$$501 \times 2\,000\,001 \times 4 \text{ byte} = 4\,008\,002\,004 \text{ byte} \approx 3.73 \text{ GB} \tag{6.6}$$

dua setengah kali 1536 MB milik SPOJ. Pada masukan kecil program ini benar sepenuhnya. Berkas di bawah adalah contoh yang dikerjakan dengan tangan di Subbab 6.5.4: kapasitas 10 dan empat barang, setiap baris memberi nilai dan bobot.

```
10 4
6 5
5 4
4 3
7 6
```

```
12
```

Subbab 6.7 memeriksanya terhadap brute force pada ratusan masukan acak. Untuk melihat apa yang terjadi di juri pada ukuran penuh, kami meniru batasnya di Linux dengan `ulimit -v 1572864`, yang membatasi memori program pada 1536 MB, lalu menjalankan uji terbesar yang dihasilkan `gen_lks_max.py` ($K = 2\,000\,000, N = 500$):

```
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted
```

Alokasi tabel gagal, `std::bad_alloc` dilempar, tidak ada yang menangkapnya, dan program berhenti paksa: di juri, runtime error atau *Memory limit exceeded*, bergantung pada di mana batasnya ditegakkan.

6.5.3 Satu baris sudah cukup

Perhatikan Persamaan (6.5) sekali lagi: baris i hanya dihitung dari baris $i-1$. Baris $0,1,\ldots,i-2$ tidak pernah dibaca lagi. Jadi kita dapat menyimpan satu baris `best` saja dan menimpanya di tempat, asalkan, ketika kita menghitung `best[c]` yang baru, nilai `best[c - w]` yang kita baca masih yang *lama*, dari baris $i-1$.

Itu bergantung pada urutan loop. Nilai baru di c membaca nilai lama di indeks lebih kecil $c-w$. Jika kita menelusuri kapasitas dari tinggi ke rendah, dari K turun ke w, maka pada saat kita menulis `best[c]`, entri `best[c - w]` belum ditimpa pada putaran ini, karena $c-w<c$ baru dikunjungi kemudian. Entri itu masih menyimpan baris $i-1$, dan Persamaan (6.5) dihitung dengan tepat. Jika kita berjalan dari rendah ke tinggi, `best[c - w]` sudah berisi baris i, yang mungkin sudah memuat barang i, dan barang itu bisa terambil dua kali, tiga kali, sebanyak yang muat: itu menghitung soal lain, knapsack *tak terbatas*. Bab 9 kembali membahas trik arah ini; di sini nilainya adalah memori. **Kode 6.6** adalah programnya.

Kode 6.6 `lks.cpp`: knapsack dengan satu baris berisi $K+1$ nilai, dikunjungi dari kapasitas tinggi ke rendah.

```cpp
// LKS (SPOJ): knapsack 0/1 berkapasitas K <= 2.000.000 dan N <= 500 barang.
// Tabel penuh butuh (N + 1) * (K + 1) int, sekitar 4 GB. Satu baris berisi
// K + 1 int (8 MB) cukup jika kapasitas dikunjungi dari tinggi ke rendah.
#include <cstdio>
#include <vector>

int main() {
    int K, N;
    if (std::scanf("%d %d", &K, &N) != 2) return 0;
    std::vector<int> best(K + 1, 0);   // best[c]: nilai maks dengan bobot <= c
    for (int i = 0; i < N; i++) {
        int value, weight;
        if (std::scanf("%d %d", &value, &weight) != 2) return 0;
        for (int c = K; c >= weight; c--) {      // turun: tiap barang sekali
            int take = best[c - weight] + value;
            if (take > best[c]) best[c] = take;
        }
    }
    std::printf("%d\n", best[K]);
    return 0;
}
```

Memorinya $(K + 1) \times 4$ byte, 8 MB pada batasnya, 500 kali lebih kecil daripada Persamaan (6.6), dan waktunya sama, $N \times K = 10^9$ langkah sederhana. Pada uji terbesar program ini memerlukan 0,48 detik di mesin kami (yang terbaik dari tiga kali jalan; sampai 1,2 detik ketika pekerjaan lain berbagi prosesor), di dalam batas 2 detik; loop terdalamnya menelusuri memori secara berurutan, yang dihargai cache.

6.5.4 Contoh yang dikerjakan dengan tangan

Ambil kapasitas $K = 10$ dan empat barang dengan (nilai, bobot) = (6, 5), (5, 4), (4, 3), dan (7, 6). **Tabel 6.2** menunjukkan baris setelah setiap barang.

Tabel 6.2 Baris tunggal `best` pada `lks.cpp` setelah setiap barang, untuk kapasitas 10.

Setelah barang	$c=0$	1	2	3	4	5	6	7	8	9	10
(awal)	0	0	0	0	0	0	0	0	0	0	0
(6, 5)	0	0	0	0	0	6	6	6	6	6	6
(5, 4)	0	0	0	0	5	6	6	6	6	11	11
(4, 3)	0	0	0	4	5	6	6	9	10	11	11
(7, 6)	0	0	0	4	5	6	7	9	10	11	12

Ikuti satu entri. Ketika barang (5, 4) datang, loop mencapai $c = 9$ lebih dahulu dan menghitung $\max(6,\text{best}[5] + 5) = \max(6,6 + 5) = 11$: barang (6, 5) dan barang (5, 4) bersama-sama. Karena kita berjalan turun, `best[5]` masih berisi 6, nilai *sebelum* barang (5, 4). Seandainya kita berjalan naik, `best[5]` sudah diperbarui, dan `best[9]` bisa saja dibangun dari barang (5, 4) dua kali. Jawaban akhirnya best[10] = 12, barang (5, 4) dan (7, 6).

Berkas masukan yang sama memberi jawaban yang sama dengan program satu baris, seperti halnya dengan tabel penuh:

```
10 4
6 5
5 4
4 3
7 6
```

```
12
```

6.5.5 Python tidak dapat lolos LKS

Kode 6.7 adalah algoritma satu baris yang sama dalam Python. Untuk menghindari loop Python atas kapasitas, program membangun baris yang digeser dengan irisan dan menggabungkan kedua baris dengan `map(max, ...)`, yang sebagian besar berjalan di dalam kode C interpreter. Karena ruas kanan dihitung lengkap sebelum penugasan, ia hanya membaca nilai lama, dan masalah arah di Subbab 6.5.3 lenyap.

Kode 6.7 `lks.py`: knapsack satu baris dalam Python; benar, tetapi jauh terlalu lambat untuk uji terbesar.

```python
# LKS (SPOJ) dalam Python: DP satu baris yang sama. Benar, tetapi N * K bisa
# 10^9 langkah; walau memakai irisan list, tetap terlalu lambat untuk uji terbesar.
import sys


def main():
    data = sys.stdin.buffer.read().split()
    k, n = int(data[0]), int(data[1])
    best = [0] * (k + 1)
    for i in range(n):
        value, weight = int(data[2 + 2 * i]), int(data[3 + 2 * i])
        if weight > k:
            continue
        # baru[c] = max(best[c], best[c - weight] + value), untuk semua c >= weight,
        # dihitung dari baris LAMA, jadi tiap barang dipakai paling banyak sekali
        shifted = [b + value for b in best[:k + 1 - weight]]
        best[weight:] = map(max, best[weight:], shifted)
    print(best[k])


if __name__ == "__main__":
    main()
```

Program ini mencetak 12 pada contoh dan cocok dengan brute force pada setiap uji acak. Namun ia tidak cukup cepat. Pada uji sepersepuluh ukuran maksimum ($K = 200\,000$, $N = 500$), ia memerlukan 8,8 detik, dibandingkan 0,05 detik untuk program C++; uji penuh, sepuluh kali lebih besar, akan memerlukan jauh lebih dari satu menit. $N \times K = 10^9$ langkah dasar memang di luar jangkauan interpreter dalam 2 detik, trik apa pun yang kita pakai, dan inilah salah satu soal di buku ini yang harus diselesaikan dengan C++. Pelajarannya memperluas tabel anggaran Bab 3: di Python, perhitungkan sekitar 10^7 operasi sederhana per detik, lebih sedikit bila setiap operasi membuat objek.

6.6 Stack Lebih Kecil daripada Dugaan Anda

6.6.1 Graf yang berbentuk garis

Rekursi adalah cara alami menulis penelusuran *depth-first* (DFS), salah satu penjelajahan graf yang paling dasar (Bab 12 membahasnya secara lengkap): untuk mengunjungi sebuah simpul, tandai simpul itu lalu kunjungi setiap tetangganya yang belum ditandai. Setiap panggilan rekursif menambahkan satu frame ke stack, berisi variabel lokalnya dan alamat untuk kembali, dan frame itu tetap di sana sampai panggilan kembali. Untuk graf berbentuk garis, $1 - 2 - 3 - \cdots - n$, rekursi masuk sedalam n panggilan sebelum panggilan pertama kembali. **Kode 6.8** membangun graf seperti itu dan menjelajahinya dengan dua cara: secara rekursif, dan dengan stack eksplisit berisi simpul-simpul yang disimpan dalam `vector` di heap.

Kode 6.8 `deep_dfs.cpp`: penelusuran depth-first pada lintasan berisi n simpul, secara rekursif dan dengan stack eksplisit.

```
// Penelusuran depth-first pada lintasan 1 - 2 - ... - n, ditulis dua cara.
// Rekursi menyimpan satu frame stack per tingkat; stack eksplisit menyimpan satu
// int per tingkat di heap. Pemakaian: deep_dfs n recursive|iterative
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <vector>

std::vector<std::vector<int>> adj;
std::vector<char> seen;

int dfsRecursive(int u) {                   // mengembalikan banyak simpul terlihat
    seen[u] = 1;
    int count = 1;
    for (int v : adj[u])
        if (!seen[v]) count += dfsRecursive(v);
    return count;
}

int dfsIterative(int start) {
    std::vector<int> stack{start};          // tinggal di heap, tumbuh bebas
    seen[start] = 1;
    int count = 0;
    while (!stack.empty()) {
        int u = stack.back();
        stack.pop_back();
        count++;
        for (int v : adj[u])
            if (!seen[v]) { seen[v] = 1; stack.push_back(v); }
    }
    return count;
}

int main(int argc, char** argv) {
    int n = argc > 1 ? std::atoi(argv[1]) : 1000000;
    bool recursive = argc > 2 && std::strcmp(argv[2], "recursive") == 0;
    adj.assign(n + 1, {});
    seen.assign(n + 1, 0);
    for (int u = 1; u < n; u++) {         // sisi u - (u + 1)
        adj[u].push_back(u + 1);
        adj[u + 1].push_back(u);
    }
    int visited = recursive ? dfsRecursive(1) : dfsIterative(1);
    std::printf("%s DFS visited %d nodes\n",
                recursive ? "recursive" : "iterative", visited);
    return 0;
```

Kode 6.8 (Lanjutan) *deep_dfs.cpp: penelusuran depth-first pada lintasan berisi n simpul, secara rekursif dan dengan stack eksplisit.*

```
}
```

Dijalankan tanpa argumen, program membangun lintasan berisi sejuta simpul dan memakai versi iteratif:

```
iterative DFS visited 1000000 nodes
```

6.6.2 Di mana rekursi patah

Kami menjalankan kedua versi dengan tiga ukuran stack, yang diatur dengan perintah Linux `ulimit -s` (dalam kilobyte). **Tabel 6.3** menunjukkan hasilnya.

Tabel 6.3 DFS rekursif dan iteratif pada sebuah lintasan, dengan berbagai ukuran stack (g++ 13.3, `-O2`).

n	Stack	Rekursif	Iteratif
100000	8 MB (bawaan Linux)	visited 100000 nodes	visited 100000 nodes
1000000	8 MB (bawaan Linux)	Segmentation fault	visited 1000000 nodes
100000	1 MB (bawaan Windows)	Segmentation fault	visited 100000 nodes

Aritmetika menjelaskannya. Setiap frame `dfsRecursive` menyimpan parameter, penghitung, keadaan loop, beberapa register yang diselamatkan, dan alamat kembali. Dengan menaikkan n sedikit demi sedikit, kami mendapati rekursi masih bekerja pada 160.000 tingkat dan mogok pada 180.000 dengan stack 8 MB, jadi satu frame memakan sekitar $8\,388\,608/170\,000 \approx 49$ byte. Sejuta frame membutuhkan sekitar 49 MB, enam kali ukuran stack, sedangkan seratus ribu frame membutuhkan sekitar 5 MB, yang muat di 8 MB tetapi tidak di 1 MB yang diberikan Windows kepada thread utama sebuah program secara bawaan [9]. Versi iteratif menyimpan satu `int` untuk setiap simpul yang menunggu di dalam vector di heap, sehingga kedalamannya hanya dibatasi oleh batas memori.

Bagaimana dengan juri? Sebagian juri menjalankan program dengan stack yang lebih besar daripada bawaan desktop, dan sebagian menyamakan stack dengan batas memori; tetapi sedikit yang mendokumentasikannya, dan program yang mengandalkannya tidak portabel. Bila kedalaman bisa mencapai 10^5 atau lebih, tulislah versi iterasinya.

Di rumah Anda dapat memperbesar stack untuk meniru perilaku sebuah juri atau untuk menjalankan solusi acuan yang rekursif: di Linux dan macOS dengan `ulimit -s unlimited` (atau ukuran dalam kilobyte) di terminal sebelum menjalankan program, dan di Windows dengan MinGW-w64 dengan menautkan program memakai `-Wl,--stack,268435456`, yang meminta stack sebesar 256 MB. Keduanya hanya mengubah mesin Anda sendiri; tak satu pun tersedia di juri, dan justru karena itulah stack eksplisit menjadi jawaban yang portabel.

6.6.3 Kisah yang sama di Python

Python melindungi dirinya lebih awal. Interpreternya menghitung kedalaman panggilan Python dan memunculkan `RecursionError` begitu melewati batas rekursi, yang secara bawaan 1000. Versi Python program ini, `deep_dfs.py` di folder bab, berhenti pada lintasan berisi 100.000 simpul dengan

```
RecursionError: maximum recursion depth exceeded
```

`sys.setrecursionlimit(10**6)` menaikkan penghitungnya tetapi tidak menaikkan stack sungguhan milik kode C di bawahnya, sehingga rekursi yang cukup dalam kemudian meruntuhkan seluruh interpreter alih-alih memunculkan eksepsi. Jawaban yang kokoh sama dengan di C++: stack eksplisit. Versi Python yang iteratif menangani sejuta simpul dalam 0,58 detik dan 170 MB, dibandingkan 0,09 detik dan 58 MB untuk C++; selisih memorinya adalah 40 byte per bilangan bulat di **Tabel 6.1**, dibayar beberapa kali dalam list ketetanggaan.

Angka terakhir itu menunjuk pada satu berat lagi. Bahkan program C++ memakai 58 MB untuk graf dengan dua juta entri ketetanggaan, karena `vector<vector<int>>` memberi setiap simpul vector kecilnya sendiri: 24 byte untuk objek vector dan satu blok heap terpisah untuk elemen-elemennya. Untuk graf dengan jutaan simpul, alternatif yang ringkas menyimpan tetangga semua simpul berurutan dalam satu larik, bersama larik kedua yang mencatat di mana daftar setiap simpul dimulai: dua alokasi alih-alih sejuta.

6.7 Mengujinya

Setiap program di bab ini diperiksa dengan cara yang diuraikan Bab 8: generator menulis masukan acak kecil, brute force yang jelas benar menghitung keluaran yang diharapkan, dan pengikat uji stres membandingkannya.

- Untuk PRIME1, brute force menguji setiap bilangan dalam rentang dengan pembagian percobaan, membagi dengan setiap d yang memenuhi $d \times d \le x$. Generator membuat jendela sampai 300 bilangan, kadang dimulai dari 1, 2, atau 3, kadang di mana saja sampai 10^9.
- Untuk LKS, brute force mencoba semua 2^N himpunan bagian dari paling banyak 12 barang, yang sama sekali tidak memerlukan rekurens maupun trik arah. **Kode 6.9** adalah program itu.

Kode 6.9 `brute_lks.py`: brute force LKS mencoba setiap himpunan bagian barang.

```python
# Brute force untuk LKS: coba semua 2^N himpunan bagian barang.
import sys

data = sys.stdin.read().split()
k, n = int(data[0]), int(data[1])
items = [(int(data[2 + 2 * i]), int(data[3 + 2 * i])) for i in range(n)]
best = 0
for mask in range(1 << n):
    value = weight = 0
    for i in range(n):
        if mask >> i & 1:
            value += items[i][0]
            weight += items[i][1]
    if weight <= k:
        best = max(best, value)
print(best)
```

Empat kali uji stres, masing-masing dengan 300 uji acak, memberikan:

```
python3 stress.py "python3 gen_prime1.py" "python3 brute_prime1.py" build/prime1
python3 stress.py "python3 gen_prime1.py" "python3 brute_prime1.py" \
    "python3 ../python/prime1.py"
python3 stress.py "python3 gen_lks.py" "python3 brute_lks.py" build/lks
python3 stress.py "python3 gen_lks.py" "python3 brute_lks.py" \
    "python3 ../python/lks.py"
```

```
OK: 300 random tests agree
OK: 300 random tests agree
OK: 300 random tests agree
OK: 300 random tests agree
```

`lks_table.cpp` juga cocok dengan brute force pada 200 uji acak, yang menegaskan klaim Subbab 6.5: tabel penuh itu benar dan hanya terlalu berat. Masukan-masukan terbesar diukur waktunya secara terpisah dengan `measure.py`, yang melaporkan waktu nyata dan memori puncak sebuah eksekusi; angka-angka yang dikutip di Subbab 6.4 dan 6.5 berasal dari sana.

6.8 Menggunakannya Sungguhan

Kebiasaan yang diajarkan bab ini adalah timbangan kamar mandi Pak Harun: timbang sebelum bepergian. Dalam praktik, ia berupa lembar kerja tiga baris yang Anda isi sambil membaca batasan soal, sebelum menulis kode apa pun.

1. **Daftarkan struktur-struktur besar** yang dibutuhkan gagasan Anda, beserta ukurannya dalam besaran masukan: “satu penanda per bilangan sampai n”, “tabel $N \times K$”, “daftar tetangga per simpul”.

2. **Timbang masing-masing** dengan Persamaan (6.1) dan byte per elemen dari **Tabel 6.1**, pada masukan terbesar.
3. **Bandingkan dengan batasnya**, sisakan separuhnya sebagai cadangan. Bila sebuah struktur tidak muat, ajukan dua pertanyaan bab ini: bisakah saya hanya menyimpan sebuah *jendela* darinya (PRIME1)? bisakah saya hanya menyimpan *baris terakhir* (LKS)?

Lembar kerja yang sama berguna di luar lomba. Sebuah koperasi petani durian dan buah di Jember yang ingin menyimpan satu catatan per pohon per hari selama lima tahun, 12.000 pohon, memiliki $12\,000 \times 365 \times 5 \approx 2.2 \times 10^7$ catatan. Sebagai list Python berisi dictionary, beberapa ratus byte per catatan, itu beberapa gigabyte; sebagai baris berisi beberapa bilangan dalam `array` atau tabel basis data, itu beberapa ratus megabyte; sebagai total harian per pohon yang diperbarui di tempat, seperti baris tunggal LKS, itu hanya beberapa kilobyte. Mana yang tepat bergantung pada pertanyaan yang ingin diajukan para petani, tetapi beratnya sebaiknya diketahui sebelum pilihan dibuat.

Hitungan yang sama juga menolong ketika program itu kelak dipindahkan ke tempat yang lebih sempit. Aplikasi pencatat di ponsel murah, misalnya, mungkin hanya diberi beberapa ratus megabyte oleh sistem operasinya, dan aplikasi yang melewati batas itu ditutup paksa tanpa pesan yang jelas, persis seperti program yang mendapat *Memory limit exceeded*. Kebiasaan menimbang struktur data sebelum menulis kode, yang dilatih di soal-soal SPOJ, adalah kebiasaan yang sama yang membuat aplikasi seperti itu tetap berjalan di ponsel petani, bukan hanya di laptop pemrogramnya.

6.9 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

- **Menetapkan ukuran larik menurut rentang nilai, bukan ukuran masukan.** "$n \le 10^9$" tidak berarti "larik berukuran 10^9". Carilah jendela, pemampatan nilai, atau rumus.
- **Menyimpan tabel yang hanya dibaca satu baris ke belakang.** Jika baris i hanya bergantung pada baris $i-1$, simpan satu atau dua baris; periksa arah loop bila Anda menyimpan satu.
- **Memakai kontainer berbasis simpul untuk data massal.** `set<int>` atau `map<int,int>` berisi sejuta elemen kira-kira sepuluh kali lebih berat daripada `vector` terurut. Bila data tidak berubah setelah dibaca, urutkan sebuah vector dan pakai binary search.
- **Menumbuhkan vector besar dengan `push_back`.** Penggandaan dapat membuat puncaknya tiga kali ukuran akhir vector itu. Buatlah langsung dengan ukurannya, atau panggil `reserve`.
- **List bilangan di Python.** List berisi 10^7 bilangan bulat beratnya 400 MB. Pakailah `bytearray` atau `array`.
- **Rekursi yang dalam.** DFS rekursif pada 10^6 simpul mogok dengan stack 8 MB, dan pada 10^5 di Windows. Tulislah stack eksplisit.
- **Larik lokal yang besar.** `int a[1000000];` di dalam fungsi berarti 4 MB di stack: baik-baik saja di Linux, mogok di Windows. Jadikan `vector` atau variabel global.
- **Mengalokasikan per kasus uji tanpa perlu.** Satu gigabyte baru per kasus uji dibersihkan setiap kali. Alokasikan sekali untuk kasus terbesar, atau alokasikan tepat sebanyak yang dibutuhkan setiap kasus, seperti yang dilakukan `prime1.cpp`.

6.10 Ringkasan Bab

- Juri menimbang memori puncak program Anda; stack kecil (1 sampai 8 MB), heap besar sampai batasnya, dan masing-masing gagal dengan cara yang berbeda.

- Berat sebuah struktur adalah $n \times b$ (Persamaan (6.1)), dan b harus diukur: 1, 4, atau 8 byte pada kontainer bersebelahan, 24 sampai 40 pada kontainer berbasis simpul, 40 sampai 74 pada list, set, dan dictionary Python berisi bilangan bulat (**Gambar 6.1**).
- Saringan biasa untuk PRIME1 seberat 954 MB (Persamaan (6.2)); saringan bersegmen menyimpan 3401 bilangan prima kecil dan jendela berisi 100.001 penanda, sekitar 140 KB, dan berjalan dalam sekitar $(n-m)\ln\ln\sqrt{n}$ langkah (Persamaan (6.4)).
- Tabel LKS seberat 3,73 GB (Persamaan (6.6)); karena setiap baris Persamaan (6.5) hanya membaca baris sebelumnya, satu baris berukuran 8 MB sudah cukup, ditelusuri dari kapasitas tinggi ke rendah. Python terlalu lambat untuk 10^9 langkahnya.
- Kedalaman rekursi juga memori: sejuta panggilan bersarang meluapkan stack 8 MB. Stack eksplisit di heap menghapus batas itu, di C++ maupun di Python.
- Struktur yang lebih ringan biasanya lebih cepat, karena alokasinya lebih sedikit, pemakaian cache-nya lebih baik, dan pembersihannya lebih sedikit.

6.11 Soal Latihan

1. Sebuah soal memberi $n \le 5 \times 10^6$ bilangan bulat di bawah 10^9 dan menanyakan berapa banyak nilai yang berbeda. Timbanglah tiga solusi C++: `set<int>`, `unordered_set<int>`, dan `vector<int>` terurut yang diikuti penghitungan perubahan. Mana yang akan Anda kirim ke juri 64 MB?
2. Untuk jendela [1,12], jalankan saringan bersegmen dengan tangan seperti di Subbab 6.4.4. Baris mana di `prime1.cpp` yang mencegah bilangan prima kecil 2, 3, 5, 7, dan 11 ikut dicoret?
3. Ubah `prime1.cpp` agar jendelanya berupa `vector<bool>`. Apakah keluarannya berubah? Ukur waktunya pada masukan terbesar dengan `measure.py` dan jelaskan hasilnya.
4. Di `lks.cpp`, balik loop terdalam agar berjalan dari `weight` naik sampai `K`. Apa yang dicetak program untuk contoh di Subbab 6.5.4, dan soal apa yang kini diselesaikannya?
5. Uji LKS terbesar memiliki $K = 2 \times 10^6$ tetapi, dalam generator kami, total bobot semua barang hanya sekitar 10^6. Perbaiki `lks.cpp` agar loop terdalam tidak pernah mengunjungi kapasitas yang lebih besar daripada total bobot barang yang sudah dibaca, lalu ukur keuntungannya.
6. Tulis DFS rekursif `deep_dfs.py` agar dapat menangani $n = 10^6$ di Python tanpa stack eksplisit, memakai `sys.setrecursionlimit` dan `threading.stack_size`. Mengapa stack eksplisit tetap jawaban yang lebih baik?

6.12 Penyelesaian Soal Latihan

1. Dengan **Tabel 6.1**: `set<int>` membutuhkan sekitar $5 \times 10^6 \times 40 = 200$ MB ditambah beban pengalokasi, `unordered_set<int>` sekitar $5 \times 10^6 \times 28 = 140$ MB, dan vector terurut $5 \times 10^6 \times 4 = 20$ MB. Hanya vector yang muat dalam 64 MB. Pengurutan berbiaya $O(n \log n)$, sekitar 10^8 langkah dasar, jauh di bawah satu detik, dan setelah diurutkan nilai-nilai yang berbeda dihitung dengan satu penelusuran yang membandingkan setiap elemen dengan elemen sebelumnya (atau dengan `std::unique`).

2. $\sqrt{12} < 4$, jadi hanya $p = 2$ dan $p = 3$ yang mencoret sesuatu. Untuk $p = 2$: $f(2) = \lfloor 2/2 \rfloor \times 2 = 2$, tetapi awalnya $\max(2,4) = 4$: coret 4, 6, 8, 10, 12. Untuk $p = 3$: $f(3) = 3$, awal $\max(3,9) = 9$: coret 9, 12. Untuk $p = 5$, $25 > 12$: berhenti. Bilangan tak tercoret mulai 2 adalah 2, 3, 5, 7, 11, dan 1 dikecualikan oleh `x >= 2`. Baris `if (first < pp) first = pp;` melindungi bilangan prima kecil: tanpanya, $p = 2$ akan mulai di $f(2) = 2$ dan mencoret 2 sendiri.

3. Keluarannya tidak berubah: hanya penyimpanan penandanya yang berubah. Kami mengukur 0,010 sampai 0,015 detik untuk versi `vector<bool>` dibandingkan 0,007 sampai 0,009 untuk versi `vector<char>`: sedikit lebih buruk, karena setiap akses ke elemen `vector<bool>` memerlukan pergeseran dan penyaringan, dan dengan jendela yang hanya berisi 100.001 penanda, penghematan memorinya (86 KB) tidak berarti. `vector<bool>` membayar bila versi byte tidak muat, bukan dalam keadaan lain.

4. Program mencetak 13. Dengan loop naik, `best[c - weight]` sudah diperbarui untuk barang yang sedang diproses, sehingga satu barang dapat terhitung beberapa kali: program kini menyelesaikan knapsack *tak terbatas*, tempat setiap barang boleh diambil berapa kali pun. Untuk contoh itu pilihan tak terbatas terbaik adalah barang (4, 3) dua kali dan barang (5, 4) sekali, bobot 10 dan nilai 13. Kembalikan arah loop untuk memperoleh lagi jawaban 0/1, yaitu 12.

5. Simpan jumlah berjalan `total` dari bobot yang sudah dibaca dan jalankan loop `for (int c = std::min(K, total); c >= weight; c--)`, dengan menambahkan `weight` ke `total` sebelum loop. Kapasitas di atas total bobot tidak mungkin memuat sesuatu yang lebih baik daripada total bobot itu sendiri, dan `best[c]` untuk *c* seperti itu sama dengan `best[total]`; jadi di akhir jawabannya `best[std::min(K, total)]`. Pada uji terbesar kami, tempat ke-500 bobot bersama-sama berjumlah 951.080, loop tidak pernah berjalan di atas total itu, dan selama barang-barang pertama dibaca ia berjalan jauh lebih rendah lagi; waktunya turun dari 0,48 menjadi 0,16 detik. Program harus mencetak `best[min(K, total)]`, bukan `best[K]`, karena entri-entri yang tinggi tidak pernah diisi.

6. Jalankan penelusuran di thread baru dengan stack yang besar:

```
import sys, threading
sys.setrecursionlimit(1 << 21)
threading.stack_size(512 * 1024 * 1024)
t = threading.Thread(target=main)
t.start()
t.join()
```

Cara ini berhasil di Linux dan biasanya di Windows, tetapi bergantung pada seberapa besar stack yang diizinkan sistem operasi bagi sebuah thread, memakai memori jauh lebih banyak daripada stack eksplisit, dan di sebagian juri pembuatan thread dibatasi. Stack eksplisit tidak bergantung pada apa pun: ia hanyalah memori heap biasa, yang hanya dibatasi oleh batas memori.

Daftar Pustaka

[1] Wikipedia, “SPOJ.” [Online]. Available: https://en.wikipedia.org/wiki/SPOJ (accessed Sep. 2026).

[2] U. Drepper, “What Every Programmer Should Know About Memory,” Red Hat, Inc., Nov. 2007. [Online]. Available: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

[3] cppreference.com, “std::vector.” [Online]. Available: https://en.cppreference.com/w/cpp/container/vector_bool (accessed Sep. 2026).

[4] Python Software Foundation, “tracemalloc: Trace memory allocations,” *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/tracemalloc.html

[5] Sphere Online Judge, “PRIME1 - Prime Generator.” [Online]. Available: https://www.spoj.com/problems/PRIME1/ (accessed Sep. 2026).

[6] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 6th ed., revised by D. R. Heath-Brown and J. H. Silverman. Oxford, UK: Oxford University Press, 2008.

[7] Sphere Online Judge, “LKS - Large Knapsack.” [Online]. Available: https://www.spoj.com/problems/LKS/ (accessed Sep. 2026).

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[9] Microsoft, "/STACK (Stack allocations)," *Linker options*, Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/cpp/build/reference/stack-stack-allocations (accessed Sep. 2026).

Bab 7

Jebakan Implementasi Klasik Penyebab Wrong Answer

Vonis paling menjengkelkan di sebuah juri adalah *Wrong answer* untuk program yang gagasannya benar. Algoritmanya benar, pembuktiannya kokoh, contoh-contohnya lolos, dan juri tetap berkata tidak. Bab ini adalah buku panduan lapangan tentang kekeliruan implementasi kecil yang menjadi penyebab kebanyakan vonis semacam itu. Di akhir bab Anda akan mampu mengenali tujuh keluarga kekeliruan itu beserta gejala yang ditinggalkan masing-masing; menyelesaikan empat soal SPOJ sungguhan, EIGHTS, FCTRL2, ADDREV, dan CANDY, yang masing-masing dipilih karena memuat satu jebakan klasik, dan membuktikan kebenaran setiap solusinya; meramalkan hasil pembagian bulat, sisa bagi, dan aritmetika bilangan pecahan di C++ dan di Python, yang ternyata berbeda; serta menangkap setiap kekeliruan di bab ini dengan uji stres yang sama yang dikembangkan sepenuhnya di Bab 8.

7.1 Sepuluh Meter Pagar, Sebelas Tiang

Mas Dedi adalah tukang kayu di sebuah desa di luar Klaten. Seorang tetangga memintanya memasang pagar bambu sepanjang sepuluh meter di depan halamannya, dengan satu tiang setiap meter. Mas Dedi sudah seratus kali mengerjakan pekerjaan semacam ini, tetapi pada kali pertama ia membeli sepuluh tiang. Sepuluh meter, satu tiang per meter, sepuluh tiang: penalarannya terdengar sempurna. Menjelang sore ia berdiri di meter terakhir tanpa tiang di tangan, karena pagar sepuluh ruas memiliki *sebelas* tiang, satu di setiap ujung setiap ruas. Tukang kayu menyebutnya kesalahan tiang pagar, dan para pemrogram meminjam nama itu untuk loop yang berjalan satu kali terlalu sedikit atau satu kali terlalu banyak.

Sepupunya, Mbak Wati, menjaga kasir di toko kelontong keluarga. Kalkulator lamanya menampilkan delapan digit. Suatu malam, sambil menjumlahkan kiriman barang sebulan, ia mengetik total sembilan puluh juta rupiah ditambah lima belas juta, dan layar menampilkan huruf “E” serta sebuah angka yang tak bermakna. Kalkulator itu tidak salah berhitung; jawabannya saja yang tidak muat dalam delapan digitnya. `int` di C++ adalah kalkulator Mbak Wati dengan layar yang sedikit lebih besar: sekitar dua ribu juta, lalu sampah. Tidak seperti kalkulator itu, `int` bahkan tidak menampilkan huruf “E”.

Baik Mas Dedi maupun Mbak Wati tidak salah memahami persoalannya. *Metode* mereka benar; satu rincian pelaksanaannya yang keliru. Itulah persis keadaan peserta lomba yang gagasannya benar tetapi mendapat *Wrong answer*, dan obatnya pun sama: kenali segelintir tempat persembunyian rincian semacam itu, lalu periksa tempat-tempat itu dengan sengaja. Tukang kayu yang pernah kena menghitung tiang sebagai banyaknya ruas ditambah satu; pemilik toko yang pernah melihat huruf “E” menjumlahkan angka besar dalam dua bagian. Bab ini memberi Anda versi pemrogram dari kebiasaan-kebiasaan itu.

7.2 Gagasannya dengan Kata-Kata Sederhana

> **DEFINISI.** *Jebakan implementasi* adalah kekeliruan dalam menerjemahkan gagasan yang benar menjadi kode, yang membuat program memberi keluaran salah pada sebagian masukan yang sah sementara tetap tampak benar pada masukan lainnya. Ini bukan algoritma yang salah: memperbaikinya hanya mengubah satu atau dua baris, bukan pendekatannya. *Gejala* sebuah jebakan adalah pola masukan tempat program itu gagal.

Gejala itu penting, karena juri hanya memberi tahu bahwa *sebagian* uji gagal. Mengetahui keluarga kekeliruan mana yang menghasilkan gejala mana membantu Anda menebak tempat mencarinya. **Tabel 7.1** mendaftar tujuh keluarga yang dibahas bab ini, beserta gejala yang biasanya membongkar masing-masing dan subbab yang membahasnya.

Tabel 7.1 Tujuh keluarga jebakan implementasi, gejala yang ditinggalkan masing-masing, dan tempat bab ini membahasnya.

Keluarga	Gejala khas	Subbab
Luapan bilangan bulat	benar pada masukan kecil, salah (sering negatif) pada masukan besar	7.3
Bilangan melampaui semua tipe bawaan	benar sampai suatu ambang, lalu salah atau nol	7.4
Format bilangan: nol di depan dan di belakang	benar kecuali bila hasil diawali atau diakhiri 0	7.5
Keadaan yang bocor antarkasus uji	kasus uji pertama benar, berikutnya salah	7.6
Pembagian, sisa bagi, pembulatan	salah hanya untuk bilangan negatif atau tepat setengah	7.7
Perbandingan dan pencetakan bilangan pecahan	salah di digit terakhir, atau pada nilai yang “pas”	7.7
Selisih satu dan bentuk masukan	salah di ujung rentang, atau pada spasi yang tak lazim	7.8

Setiap keluarga di bawah disertai soal SPOJ sungguhan atau demonstrasi kecil, program yang benar dalam C++ dan Python, dan, bila membantu, program rusak yang dengan mudah bisa ditulis orang yang cermat sekalipun. Program-program rusak itu sengaja kita simpan: melihat keluaran salahnya secara persis adalah cara agar gejalanya menjadi mudah dikenali.

7.3 Luapan: EIGHTS

7.3.1 Soalnya

Soal SPOJ EIGHTS, “Triple Fat Ladies” [1], meminta bilangan bulat positif terkecil ke-k yang pangkat tiganya berakhir dengan digit 888. Masukan memberi banyaknya pertanyaan t lalu t nilai k, dengan $1 \le k \le 2 \times 10^{12}$; jawabannya dijamin di bawah 2^{63}. Batas waktunya sekitar 1,2 detik.

Mencari ke atas sambil menguji pangkat tiga tidak ada harapannya untuk k di sekitar 2×10^{12}, jadi kita pahami dulu bilangan-bilangannya.

7.3.2 Gagasannya, beserta bukti

Mari kita cari semua n di bawah 1000 yang pangkat tiganya berakhir 888, dengan brute force: bilangan itu 192, 442, 692, dan 942. Periksa yang pertama: $192^3 = 7\,077\,888$. Keempat bilangan itu berjarak 250, yang menyiratkan bahwa jawaban-jawabannya membentuk *barisan aritmetika*: mulai dari 192 dan tambah 250 setiap kali. Rumusnya akan menjadi

$$a_k = 192 + 250\,(k - 1) \tag{7.1}$$

Pola yang terlihat pada empat bilangan barulah dugaan, belum fakta. Berikut buktinya. "Pangkat tiganya berakhir 888" berarti $n^3 \equiv 888 \pmod{1000}$, dengan $a \equiv b \pmod{m}$ menyatakan bahwa a dan b bersisa sama bila dibagi m. Karena $1000 = 8 \times 125$ dan 8 serta 125 tidak memiliki faktor persekutuan, syarat itu berlaku tepat ketika ia berlaku modulo 8 dan modulo 125 secara terpisah (inilah teorema sisa Cina [2]).

- **Modulo 8.** $888 = 111 \times 8$, jadi kita perlu $n^3 \equiv 0 \pmod{8}$. Jika n ganjil, n^3 ganjil dan tidak habis dibagi 8. Jika n genap, $n = 2m$ dan $n^3 = 8m^3$ habis dibagi 8. Jadi syaratnya: *n genap*.
- **Modulo 125.** $888 = 7 \times 125 + 13$, jadi kita perlu $n^3 \equiv 13 \pmod{125}$. Mencoba ke-125 sisa memperlihatkan bahwa tepat satu yang cocok, $n \equiv 67$. (Ada juga alasannya: di antara 100 sisa yang tidak habis dibagi 5, pemangkatan tiga adalah korespondensi satu-satu, karena 3 tidak memiliki faktor persekutuan dengan 100 [2]; dan kelipatan 5 memiliki pangkat tiga yang habis dibagi 125, bukan 13.)

Menggabungkan keduanya: $n \equiv 67 \pmod{125}$ dan n genap. Bilangan-bilangan yang $\equiv 67 \pmod{125}$ adalah 67,192,317,442, ..., berganti-ganti ganjil dan genap; yang genap adalah 192,442,692, ..., yaitu tepat bilangan-bilangan yang $\equiv 192 \pmod{250}$. Jadi jawabannya $192{,}192 + 250{,}192 + 500$, ..., dan Persamaan (7.1) terbukti. Setiap pertanyaan dijawab dengan satu perkalian.

7.3.3 Seberapa besar bilangannya?

Jebakannya ada pada ukuran. Untuk k terbesar,

$$a_k \le 192 + 250 \times (2 \times 10^{12} - 1) < 5 \times 10^{14} \tag{7.2}$$

jauh di atas `int` 32-bit terbesar, $2^{31} - 1 \approx 2.1 \times 10^9$, dan dengan nyaman di bawah `long long` 64-bit terbesar, $2^{63} - 1 \approx 9.2 \times 10^{18}$. Bahkan k sendiri tidak muat dalam `int`. **Kode 7.1** mengerjakan semuanya dalam `long long`.

Kode 7.1 `eights.cpp`: rumus Persamaan (7.1), dihitung dengan aritmetika 64-bit.

```
// EIGHTS (SPOJ): bilangan positif ke-k yang pangkat tiganya berakhir 888.
// Bilangan itu 192, 442, 692, ... = 192 + 250 (k - 1), k <= 2 * 10^12.
#include <cstdio>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        long long k;                          // k sendiri butuh 64 bit
        if (std::scanf("%lld", &k) != 1) break;
        std::printf("%lld\n", 192 + 250 * (k - 1));  // k long long: hitungan 64-bit
    }
    return 0;
}
```

Komentar di baris `printf` menunjuk pada satu aturan aritmetika C++ yang menjadi penyebab banyak luapan: tipe sebuah ekspresi ditentukan oleh *operand*-nya, bukan oleh variabel yang menampung hasilnya [3]. Karena `k` bertipe `long long`, `250 * (k - 1)` dihitung dalam 64 bit. Seandainya `k` bertipe `int`, hasil kalinya dihitung dalam 32 bit dan meluap *sebelum* disimpan di mana pun, bahkan ke dalam variabel `long long` sekalipun. `long long x = a * b;` dengan dua operand `int` adalah bentuk klasik kekeliruan ini; tulislah `(long long)a * b`.

Bilangan bulat Python tidak berukuran tetap, jadi **Kode 7.2** sama sekali tidak memerlukan kewaspadaan.

Kode 7.2 `eights.py`: rumus yang sama dalam Python, yang bilangan bulatnya tidak pernah meluap.

```python
# EIGHTS (SPOJ): bilangan ke-k yang kubiknya berakhir 888 = 192 + 250 (k - 1).
# Bilangan bulat Python tak pernah meluap, jadi cukup rumusnya saja.
import sys

data = sys.stdin.buffer.read().split()
t = int(data[0])
print("\n".join(str(192 + 250 * (int(k) - 1)) for k in data[1:1 + t]))
```

Pada berkas uji kami, yang pertanyaan terakhirnya $k = 10^{12}$,

```
4
1
2
3
1000000000000
```

kedua program mencetak:

```
192
442
692
249999999999942
```

7.3.4 Program yang sama dengan `int`

Kode 7.3 adalah versi yang ditulis peserta yang terburu-buru.

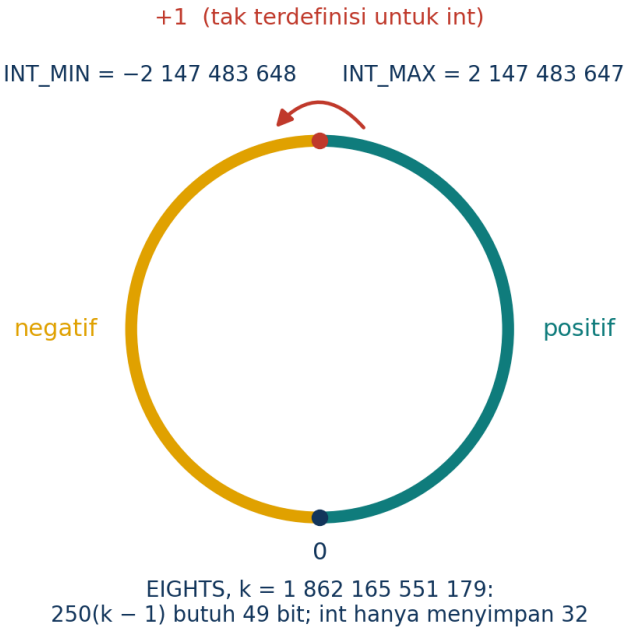
Kode 7.3 `eights_int.cpp`: rumus dengan `int`; kompilasi bersih, contoh benar, jawaban salah untuk k besar.

```cpp
// Rumus yang sama dengan int: terkompilasi bersih, lolos contoh kecil,
// dan mencetak sampah begitu 250 * (k - 1) tidak muat lagi dalam 32 bit.
#include <cstdio>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        int k;                                  // salah: k bisa 2 * 10^12
        if (std::scanf("%d", &k) != 1) break;
        std::printf("%d\n", 192 + 250 * (k - 1));
    }
    return 0;
}
```

```
192
442
692
-1456365626
```

Tiga baris pertama benar, dan justru baris-baris seperti itulah yang biasanya ada di contoh sebuah pernyataan soal. Baris keempat salah dalam dua hal sekaligus: `scanf("%d")` tidak dapat menyimpan 10^{12} dalam `int` (nilai yang disimpannya tidak ditentukan), dan perkaliannya meluap. g++ mengompilasi program dengan `-Wall -Wextra` tanpa satu peringatan pun, karena tidak ada yang salah dalam bentuk teksnya; hanya nilainya yang terlalu besar. **Gambar 7.1** memperlihatkan apa yang terjadi pada nilai 32-bit yang melewati maksimumnya: ia melompat ke nilai paling negatif dan terus menghitung dari sana. Untuk tipe unsigned perputaran itu terdefinisi; untuk `int` itu perilaku tak terdefinisi (Bab 5), dan bilangan negatif hanyalah apa yang biasanya terjadi.



Gambar 7.1 `int` 32-bit sebagai lingkaran: satu langkah melewati `INT_MAX` mendarat di `INT_MIN`; rumus EIGHTS untuk k besar membutuhkan 49 bit.

Gejala "benar pada masukan kecil, salah dan sering negatif pada masukan besar" adalah tanda tangan luapan. Bila Anda melihat bilangan negatif di tempat yang seharusnya hanya berisi bilangan positif, carilah perkalian antar-`int`.

7.3.5 Mengujinya

Brute force untuk EIGHTS tidak boleh memakai Persamaan (7.1); jika memakainya, ia hanya mengulang anggapan program. **Kode 7.4** menelusuri 1,2,3, ... dan menyimpan setiap bilangan yang pangkat tiganya berakhir 888, yaitu pernyataan soal yang diubah menjadi kode.

Kode 7.4 `brute_eights.py`: brute force untuk EIGHTS mencari ke atas dan tidak memerlukan rumus.

```
# Brute force untuk EIGHTS: telusuri 1, 2, 3, ... dan simpan setiap bilangan
# yang pangkat tiganya berakhir 888. Tanpa rumus, tanpa bukti -- k kecil saja.
import sys

data = sys.stdin.read().split()
t = int(data[0])
wanted = [int(x) for x in data[1:1 + t]]
found, x = [], 0
while len(found) < max(wanted):
    x += 1
    if x ** 3 % 1000 == 888:
        found.append(x)
print("\n".join(str(found[k - 1]) for k in wanted))
```

Program ini cepat untuk $k \leq 40$, rentang generator kecil, dan di sana baik `eights` maupun `eights_int` cocok dengannya pada 300 uji acak: uji kecil tidak dapat melihat luapan. Untuk k besar, pendapat keduanya adalah Persamaan (7.1) yang dievaluasi dalam bilangan bulat eksak Python, yang kini dibenarkan oleh bukti dan uji kecil bersama-sama. Dengan mode `big` generator, uji stres membongkar versi `int` pada benih pertama:

```
MISMATCH on seed 1
input:
2
1862165551179
1682777914929
expected: 465541387794692 420694478732192
received: 1292646660 -1862878304
```

Pelajarannya umum: **uji stres dengan masukan kecil tidak dapat menemukan luapan.** Setiap uji stres membutuhkan mode kedua yang memakai nilai terbesar yang diizinkan batasan, dibandingkan dengan perhitungan independen yang tidak mungkin meluap.

7.4 Melampaui Semua Tipe Bawaan: FCTRL2

7.4.1 Soalnya

Soal SPOJ FCTRL2, "Small factorials" [4], memberi paling banyak 100 nilai n dengan $1 \le n \le 100$ dan meminta $n!$ dicetak utuh. Batas 1 detik terbilang longgar; kesulitannya hanya pada ukuran. 100! memiliki 158 digit.

7.4.2 Seberapa jauh `unsigned long long`?

Tipe bilangan bulat bawaan terbesar di C++ standar adalah `unsigned long long`, 64 bit, dengan maksimum $2^{64} - 1 \approx 1.8 \times 10^{19}$. Karena $20! \approx 2.4 \times 10^{18}$ dan $21! \approx 5.1 \times 10^{19}$, tipe ini memuat $n!$ hanya sampai $n = 20$. **Kode 7.5** adalah program yang melupakan hal ini.

Kode 7.5 `fctrl2_ull.cpp`: faktorial dalam tipe bawaan terbesar; benar sampai 20! dan diam-diam salah mulai 21!.

```cpp
// FCTRL2 dengan tipe bawaan terbesar. Benar sampai 20!, diam-diam salah
// mulai 21!, karena aritmetika unsigned berputar modulo 2^64.
#include <cstdio>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        int n;
        if (std::scanf("%d", &n) != 1) break;
        unsigned long long f = 1;
        for (int i = 2; i <= n; i++) f *= i;
        std::printf("%llu\n", f);
    }
    return 0;
}
```

Dengan berkas yang meminta empat faktorial, yaitu dari 5, 20, 21, dan 30:

```
4
5
20
21
30
```

```
120
2432902008176640000
14197454024290336768
9682165104862298112
```

Dua baris pertama benar. Dua baris terakhir adalah 21! dan 30! yang direduksi modulo 2^{64}: aritmetika unsigned berputar menurut definisinya, jadi tidak ada perilaku tak terdefinisi dan

tidak ada peringatan, hanya bilangan salah yang tampak masuk akal. Uji stres di Subbab 7.4.5 menemukan sesuatu yang lebih aneh lagi: untuk $n = 73$ dan $n = 98$ program mencetak 0. Alasannya sepotong kecil teori bilangan. Banyaknya faktor 2 di dalam $n!$ diberikan oleh penghitungan berlapis yang sama dengan rumus Legendre untuk faktor lima di Bab 5,

$$v_2(n!) = \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \left\lfloor \frac{n}{8} \right\rfloor + \cdots \tag{7.3}$$

Untuk $n = 66$ nilainya $33 + 16 + 8 + 4 + 2 + 1 = 64$. Jadi mulai 66!, hasil kalinya kelipatan 2^{64}, dan sisanya modulo 2^{64} tepat 0. Program yang mencetak 0 untuk 73! jelas salah; program yang mencetak `14197454024290336768` untuk 21! tidak jelas apa-apa, dan itulah sebabnya jebakan ini bertahan sampai juri menemukannya.

7.4.3 Digit dalam larik, seperti di kertas

Perbaikannya adalah melakukan apa yang dilakukan orang dengan pensil dan kertas: simpan digit-digit bilangan dalam sebuah daftar dan kalikan digit demi digit, dengan menyimpan ke kolom berikutnya. Kita menyimpan digit dalam `vector<int>` mulai dari yang *paling tidak* signifikan sampai yang paling signifikan, karena dengan begitu simpanan yang menciptakan digit depan baru cukup menjadi `push_back` di ujung, bukan penyisipan di depan.

Untuk mengalikan bilangan berdigit $d_0, d_1, \ldots, d_{L-1}$ (jadi bilangannya $\sum_j d_j\, 10^j$) dengan faktor kecil f, telusuri dari d_0 ke atas dengan simpanan c yang mulai dari 0:

$$x_j = d_j \cdot f + c, \qquad d_j \leftarrow x_j \bmod 10, \qquad c \leftarrow \lfloor x_j/10 \rfloor \tag{7.4}$$

dan bila digitnya habis, tambahkan sisa simpanan digit demi digit. Inilah metode bersusun dalam pembahasan Knuth tentang aritmetika presisi jamak [5]. Dengan $f \le 100$ dan c tidak pernah melebihi f, setiap x_j paling besar $9 \times 100 + 100 = 1000$, yang mudah muat dalam `int`.

Contoh yang dikerjakan: kalikan 120 (digit 0,2,1, yang paling tidak signifikan lebih dahulu) dengan $f = 6$.

- $j = 0$: $x = 0 \times 6 + 0 = 0$, digit 0, simpanan 0.
- $j = 1$: $x = 2 \times 6 + 0 = 12$, digit 2, simpanan 1.
- $j = 2$: $x = 1 \times 6 + 1 = 7$, digit 7, simpanan 0.

Digitnya 0,2,7, yaitu $720 = 6!$. **Kode 7.6** adalah programnya.

Kode 7.6 `fctrl2.cpp`: faktorial sampai 158 digit, disimpan mulai dari digit paling tidak signifikan dan dikalikan seperti di kertas.

```cpp
// FCTRL2 (SPOJ): n! persis, untuk n <= 100 (158 digit).
// Digit disimpan dari yang paling tidak signifikan dalam vector, seperti di kertas.
#include <cstdio>
#include <vector>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        int n;
        if (std::scanf("%d", &n) != 1) break;
        std::vector<int> digits{1};          // bilangan 1
        for (int f = 2; f <= n; f++) {
            int carry = 0;
            for (int& d : digits) {          // kalikan setiap digit dengan f
                int x = d * f + carry;
                d = x % 10;
                carry = x / 10;
            }
            while (carry) { digits.push_back(carry % 10); carry /= 10; }
        }
        for (int i = (int)digits.size() - 1; i >= 0; i--)
            std::putchar('0' + digits[i]);
        std::putchar('\n');
    }
    return 0;
}
```

```
120
2432902008176640000
51090942171709440000
265252859812191058636308480000000
```

Biayanya: menghitung $n!$ mengalikan bilangan paling banyak 158 digit dengan setiap faktor sampai n, jadi paling banyak 100×158 langkah digit per pertanyaan dan 1.6×10^6 untuk 100 pertanyaan. Pencetakan menelusuri vector dari belakang, digit paling signifikan lebih dahulu.

Di Python soal ini cukup satu baris, karena bilangan bulat Python eksak di ukuran berapa pun dan pustaka standarnya menghitung faktorial secara langsung; **Kode 7.7** adalah seluruh programnya.

Kode 7.7 `fctrl2.py`: FCTRL2 dalam Python, dengan `math.factorial`.

```python
# FCTRL2 (SPOJ): n! untuk n <= 100. Bilangan bulat Python eksak di ukuran apa pun.
import math
import sys

data = sys.stdin.read().split()
t = int(data[0])
print("\n".join(str(math.factorial(int(x))) for x in data[1:1 + t]))
```

158 digit jauh di bawah batas konversi 4300 digit milik Python yang dibahas di Bab 5, jadi pencetakan di sini tidak memerlukan kewaspadaan khusus.

7.4.4 Gejalanya

"Benar sampai suatu ambang, lalu salah, dan kadang 0 yang mencurigakan" adalah tanda tangan nilai yang telah melampaui tipenya. Ambangnya memberi tahu tipe yang dipakai: 12 untuk faktorial dalam `int` 32-bit ($13! > 2^{31}$), 20 untuk 64 bit, 33 untuk tipe perluasan 128-bit bertanda `__int128`. Bila jawaban sebuah soal bisa lebih dari 18 atau 19 digit, tidak ada tipe bawaan yang cukup, dan Anda memerlukan digit dalam larik atau Python.

7.4.5 Mengujinya

Brute force menghitung $n!$ dengan loop biasa di Python, dan generator mengambil n dari 1 sampai 100. `fctrl2.cpp` dan `fctrl2.py` cocok dengannya pada 300 uji acak; `fctrl2_ull.cpp` gagal pada benih pertama, dengan $n = 73$ dan $n = 98$, mencetak `0 0`.

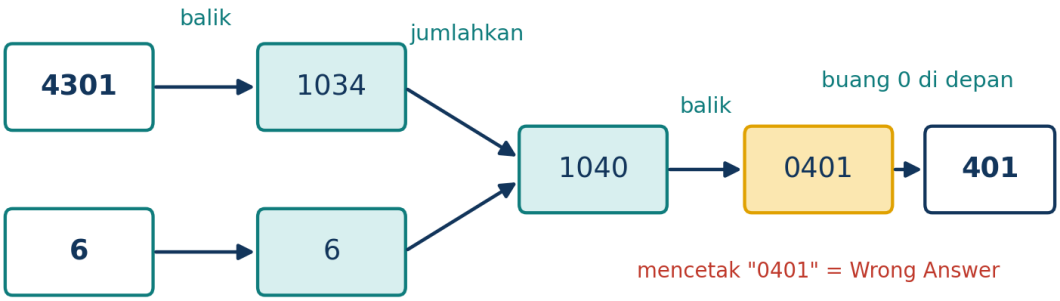
7.5 Nol yang Harus Hilang: ADDREV

7.5.1 Soalnya

Soal SPOJ ADDREV, "Adding Reversed Numbers" [6], bermain-main dengan bilangan yang ditulis terbalik. Membalik bilangan berarti menulis digitnya dalam urutan terbalik dan membuang nol yang berakhir di depan: kebalikan 1200 adalah 21, bukan 0021. Masukan memberi N pasang bilangan, masing-masing sudah *dibalik* (jadi tak satu pun berakhir dengan 0, karena bilangan yang dibalik tidak menyisakan nol di ujungnya). Untuk setiap pasangan, program harus membalik kedua bilangan kembali, menjumlahkannya, lalu mencetak kebalikan jumlah itu, lagi-lagi tanpa nol di depan.

7.5.2 Tempat jebakannya

Aritmetikanya sepele; nolnya tidak. Ambil pasangan 4301 dan 6, contoh pada **Gambar 7.2**. Dibalik, keduanya menjadi 1034 dan 6; jumlahnya 1040; dibalik, jumlah itu menjadi "0401", dan jawaban yang benar adalah 401. Program yang membalik string jumlah lalu mencetaknya mencetak `0401`, yang dibandingkan juri karakter demi karakter dengan `401`: *Wrong answer*.



Gambar 7.2 ADDREV pada pasangan 4301 dan 6: jumlah 1040 dibalik menjadi "0401", dan nol di depan harus dibuang sebelum dicetak.

Ada jebakan kedua yang lebih senyap di masukan. Jika kita membalik *string* 4301 dan 6, kita mendapat "1034" dan "6", dan itu benar. Namun jika sebuah bilangan bisa berakhir 0, katakanlah "430", kebalikannya "034", dan program yang memperlakukannya sebagai string juga harus membuang nol itu. Soal menjanjikan bahwa masukan tidak berakhir 0, tetapi jumlahnya bisa: itulah persis 1040 di atas. Aturan yang aman adalah membuang nol di depan setelah *setiap* pembalikan, bukan hanya di akhir.

7.5.3 Program-programnya

Kode 7.8 bekerja pada string digit, sehingga tidak peduli seberapa panjang bilangannya, dan menerapkan satu fungsi, `reverseNumber`, di setiap tempat sebuah bilangan dibalik. Penjumlahannya adalah penjumlahan bersusun seperti di Subbab 7.4, dari digit terakhir ke digit pertama, dengan simpanan.

Kode 7.8 `addrev.cpp`: pembalikan yang selalu membuang nol di depan, dan penjumlahan bersusun pada string digit.

```cpp
// ADDREV (SPOJ): balik dua bilangan, jumlahkan, balik hasil jumlahnya.
// Bekerja pada string digit, jadi panjang berapa pun boleh; nol ditangani sengaja.
#include <algorithm>
#include <iostream>
#include <string>

// Balik string digit lalu buang nol yang berakhir di depan.
std::string reverseNumber(std::string s) {
    std::reverse(s.begin(), s.end());
    std::size_t first = s.find_first_not_of('0');
    return first == std::string::npos ? "0" : s.substr(first);
}

// Jumlahkan dua string desimal tak negatif, cara bersusun.
std::string add(const std::string& a, const std::string& b) {
    std::string sum;
    int i = (int)a.size() - 1, j = (int)b.size() - 1, carry = 0;
    while (i >= 0 || j >= 0 || carry) {
        int d = carry;
        if (i >= 0) d += a[i--] - '0';
        if (j >= 0) d += b[j--] - '0';
        sum.push_back(char('0' + d % 10));
        carry = d / 10;
    }
    std::reverse(sum.begin(), sum.end());
    return sum;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n;
    if (!(std::cin >> n)) return 0;
    while (n--) {
        std::string a, b;
        if (!(std::cin >> a >> b)) break;
        std::cout << reverseNumber(add(reverseNumber(a), reverseNumber(b))) << '\n';
    }
    return 0;
}
```

Kasus khusus di `reverseNumber` layak dilirik: jika string berisi nol semua, `find_first_not_of('0')` mengembalikan `npos`, dan fungsi mengembalikan "0" alih-alih string kosong. Masukan soal ini positif, jadi kasus itu tidak pernah terjadi di juri, tetapi fungsi yang bisa mengembalikan string kosong untuk bilangan yang sah adalah jebakan yang menunggu soal berikutnya.

Di Python, **Kode 7.9** menyerahkan pembuangan nol kepada `int()`: mengubah string yang dibalik menjadi bilangan bulat lalu kembali ke string membuang nol di depan secara otomatis.

Kode 7.9 `addrev.py`: ADDREV dalam Python; `int()` atas string yang dibalik membuang nol di depan.

```python
# ADDREV (SPOJ): balik, jumlahkan, balik. int() dari string terbalik membuang
# nol di depan yang berasal dari nol di belakang, jadi tak perlu kasus khusus.
import sys


def reverse_number(s):
    return int(s[::-1])


def main():
    data = sys.stdin.read().split()
    n = int(data[0])
    out = []
    for i in range(n):
        a, b = data[1 + 2 * i], data[2 + 2 * i]
        total = reverse_number(a) + reverse_number(b)
        out.append(str(reverse_number(str(total))))
    print("\n".join(out))


if __name__ == "__main__":
    main()
```

Berkas uji kami memuat pasangan pada **Gambar 7.2**, sebuah pasangan yang jumlahnya tidak berakhir 0, dan dua pasangan yang jumlahnya 1000 dan 10:

```
4
4301 6
21 31
999 1
5 5
```

```
401
52
1
1
```

Untuk pasangan 999 dan 1 jumlahnya 1000, yang kebalikannya “0001” harus dicetak sebagai 1; untuk 5 dan 5 jumlahnya 10, dicetak sebagai 1.

7.5.4 Mengujinya

Generator acak jarang menghasilkan jumlah yang berakhir beberapa nol, padahal justru kasus-kasus itulah yang menarik. Karena itu `gen_addrev.py` membangun 40% pasangannya dengan sengaja: ia memilih total bulat seperti 3000 atau 30000 lalu menetapkan bilangan kedua sebagai total dikurangi bilangan pertama. Brute force mengikuti pernyataan soal secara harfiah, langkah demi langkah, dengan list berisi karakter. Kedua program cocok dengannya pada 300 uji acak. Versi yang mencetak jumlah terbalik tanpa membuang nol gagal hampir di setiap benih.

Pelajaran umum dari ADDREV menyangkut *format*. Setiap kali jawaban dicetak sebagai teks, tanyakan apa karakter pertama dan terakhirnya: nol di depan, tanda minus pada nol (“-0”), spasi di belakang, baris baru terakhir yang hilang, “1e+06” padahal yang diharapkan “1000000”. Juri membandingkan teks, bukan bilangan.

7.6 Keadaan yang Bocor Antarkasus Uji: CANDY

7.6.1 Soalnya

Soal SPOJ CANDY, “Candy I” [7], menggambarkan bungkus-bungkus permen berbagai ukuran yang harus dibuat sama dengan memindahkan permen dari satu bungkus ke bungkus lain, satu

permen per langkah. Masukan berisi beberapa blok; setiap blok diawali banyaknya bungkus N ($1 < N \le 10\,000$), diikuti N ukuran bungkus (masing-masing di bawah 1000), dan masukan diakhiri baris berisi -1. Untuk setiap blok, cetak banyak langkah tersedikit yang membuat semua bungkus sama, atau -1 jika mustahil.

7.6.2 Gagasannya, beserta bukti

Misalkan bungkus-bungkus itu berisi $c_1, \ldots, c_N$ permen, dengan total $S = \sum c_i$. Langkah-langkah tidak mengubah S, jadi jika semua bungkus akhirnya sama, masing-masing berisi S/N; jika N tidak membagi S, itu mustahil dan jawabannya -1. Jika tidak, misalkan $A = S/N$. Bungkus dengan $c_i > A$ harus kehilangan $c_i - A$ permen, dan setiap langkah mengeluarkan satu permen dari satu bungkus, jadi paling sedikit $\sum_{c_i > A} (c_i - A)$ langkah diperlukan. Sebanyak itu pula sudah cukup: pindahkan permen satu per satu dari bungkus mana pun yang di atas A ke bungkus mana pun yang di bawah A; karena kelebihan dan kekurangan berjumlah sama, selalu ada bungkus di bawah A selama masih ada yang di atas. Jadi

$$\text{langkah} = \sum_{i:\, c_i > A} (c_i - A), \qquad A = \frac{S}{N} \tag{7.5}$$

Untuk blok 3, 5, 2, 6: $S = 16$, $A = 4$, kelebihannya $5 - 4 = 1$ dan $6 - 4 = 2$, jadi 3 langkah.

7.6.3 Kebocorannya

Kode 7.10 adalah program yang benar.

Kode 7.10 `candy.cpp`: langkah-langkah Persamaan (7.5), dengan setiap variabel sebuah blok dibuat baru di dalam loop.

```cpp
// CANDY (SPOJ): blok-blok bungkus sampai baris -1. Untuk setiap blok,
// banyak langkah satu-permen tersedikit agar semua bungkus sama, atau -1.
#include <cstdio>
#include <vector>

int main() {
    int n;
    while (std::scanf("%d", &n) == 1 && n != -1) {
        std::vector<int> packet(n);
        long long total = 0;                // baru untuk setiap blok
        for (int& c : packet) {
            if (std::scanf("%d", &c) != 1) return 0;
            total += c;
        }
        if (total % n != 0) { std::printf("-1\n"); continue; }
        long long target = total / n, moves = 0;
        for (int c : packet)
            if (c > target) moves += c - target;   // tiap kelebihan pindah sekali
        std::printf("%lld\n", moves);
    }
    return 0;
}
```

Berkas uji kami memuat tiga blok: 4, 7, 2 (total 13, tidak habis dibagi 3); 3, 5, 2, 6 (contoh di atas); dan 1, 9 (total 10, target 5).

```
3
4
7
2
4
3
5
2
6
2
1
9
-1
```

Program mencetak satu baris per blok:

```
-1
3
4
```

Kode 7.11 berbeda dalam satu hal: total berjalannya adalah variabel global, dideklarasikan sekali di luar loop, sebagaimana banyak program mendeklarasikan lariknya.

Kode 7.11 `candy_leak.cpp`: program yang sama dengan total dideklarasikan di luar loop; setiap blok setelah yang pertama mewarisi permen dari blok-blok sebelumnya.

```cpp
// CANDY dengan total berjalan di luar loop: blok 1 benar,
// setiap blok berikutnya mewarisi permen dari blok-blok sebelumnya.
#include <cstdio>
#include <vector>

long long total = 0;                        // global: tidak pernah diatur ulang

int main() {
    int n;
    while (std::scanf("%d", &n) == 1 && n != -1) {
        std::vector<int> packet(n);
        for (int& c : packet) {
            if (std::scanf("%d", &c) != 1) return 0;
            total += c;
        }
        if (total % n != 0) { std::printf("-1\n"); continue; }
        long long target = total / n, moves = 0;
        for (int c : packet)
            if (c > target) moves += c - target;
        std::printf("%lld\n", moves);
    }
    return 0;
}
```

Pada berkas yang sama program ini mencetak:

```
-1
-1
-1
```

Jawaban pertama benar, dan itu pun hanya karena blok pertama dimulai dari nol. Blok kedua melihat total $13 + 16 = 29$, yang tidak habis dibagi 4, dan blok ketiga $29 + 10 = 39$, yang tidak habis dibagi 2. Uji stres menemukan kasus ketika kebocoran itu bahkan tidak menghasilkan -1:

```
MISMATCH on seed 3
input:
6
10
2
5
9
7
9
2
8
4
-1
expected: 7 2
received: 7 0
```

Blok kedua (8 dan 4) mewarisi 42 permen dari blok pertama; "target"-nya menjadi $54/2 = 27$, tidak ada bungkus di atasnya, dan program dengan yakin mencetak 0 langkah.

Tanda tangan keluarga ini tidak mungkin keliru begitu Anda mengenalnya: **kasus uji pertama benar dan kasus berikutnya salah**, sering dengan selisih yang terus membesar. Di SPOJ, tempat kebanyakan soal mengemas banyak kasus uji dalam satu berkas, inilah salah satu penyebab *Wrong answer* yang paling umum. Obatnya, menurut urutan kesukaan:

1. **Deklarasikan variabel per kasus uji di dalam loop**, seperti **Kode 7.10**. Variabel itu dibuat baru untuk setiap kasus uji, dan tidak ada yang bisa bocor.
2. Jika sebuah larik besar harus global (agar tidak dialokasikan ulang, atau karena terlalu besar untuk stack), **atur ulang di awal setiap kasus uji**, dan atur ulang hanya bagian yang dipakai kasus itu; mengosongkan larik 10^6 elemen untuk setiap dari 10^5 kasus uji kecil adalah masalah batas waktu tersendiri.
3. Tempatkan seluruh penyelesaian satu kasus uji dalam fungsi `solve()` yang variabel lokalnya mati saat fungsi kembali.

Versi Python, **Kode 7.12**, mengikuti obat pertama dan juga memperlihatkan cara baku membaca masukan yang diakhiri nilai penanda: baca semua token sekaligus lalu telusuri dengan sebuah indeks.

Kode 7.12 `candy.py`: CANDY dalam Python, membaca semua token sekaligus dan berhenti pada penanda -1.

```python
# CANDY (SPOJ): baca semua bilangan sekaligus lalu telusuri blok-bloknya.
import sys


def main():
    data = sys.stdin.read().split()
    pos, out = 0, []
    while pos < len(data):
        n = int(data[pos])
        pos += 1
        if n == -1:
            break
        packet = [int(x) for x in data[pos:pos + n]]
        pos += n
        total = sum(packet)                  # baru untuk setiap blok
        if total % n:
            out.append("-1")
            continue
        target = total // n
        out.append(str(sum(c - target for c in packet if c > target)))
    print("\n".join(out))


if __name__ == "__main__":
    main()
```

Brute force untuk CANDY tidak memakai Persamaan (7.5): ia benar-benar memindahkan satu permen setiap kali dari bungkus terpenuh ke bungkus terkosong sampai semuanya sama, dan menghitung langkahnya. Kedua program yang benar cocok dengannya pada 300 uji acak.

7.7 Pembagian, Sisa Bagi, dan Bilangan Pecahan

7.7.1 Dua bahasa, dua jawaban

Berapa −7 dibagi 2? Matematika berkata −3.5; pembagian bulat harus membulatkannya, dan C++ serta Python membulatkan ke arah yang berbeda. **Kode 7.13** dan **Kode 7.14** mengajukan pertanyaan yang sama dalam kedua bahasa.

Kode 7.13 `arith_demo.cpp`: pembagian bulat, sisa bagi, pembulatan, dan perbandingan bilangan pecahan di C++.

```cpp
// Pembagian bulat, sisa bagi, dan floating point: jawaban versi C++.
// Bandingkan dengan arith_demo.py -- beberapa baris berbeda.
#include <cmath>
#include <cstdio>

int main() {
    std::printf("-7 / 2   = %d\n", -7 / 2);       // memotong ke arah nol
    std::printf("-7 %% 2   = %d\n", -7 % 2);      // tanda mengikuti yang dibagi
    std::printf("((-7 %% 5) + 5) %% 5 = %d\n", ((-7 % 5) + 5) % 5);  // mod aman
    double x = 0.1 + 0.2;
    std::printf("0.1 + 0.2 == 0.3 : %s\n", x == 0.3 ? "true" : "false");
    std::printf("0.1 + 0.2 printed with %%.17g : %.17g\n", x);
    bool close = std::fabs(x - 0.3) < 1e-9;          // bandingkan dengan toleransi
    std::printf("|x - 0.3| < 1e-9 : %s\n", close ? "true" : "false");
    std::printf("(int)(0.29 * 100) = %d\n", (int)(0.29 * 100));
    std::printf("llround(0.29 * 100) = %lld\n", std::llround(0.29 * 100));
    std::printf("printf %%.2f of 2.675 = %.2f\n", 2.675);
    return 0;
}
```

```
-7 / 2   = -3
-7 % 2   = -1
((-7 % 5) + 5) % 5 = 3
0.1 + 0.2 == 0.3 : false
0.1 + 0.2 printed with %.17g : 0.30000000000000004
|x - 0.3| < 1e-9 : true
(int)(0.29 * 100) = 28
llround(0.29 * 100) = 29
printf %.2f of 2.675 = 2.67
```

Kode 7.14 `arith_demo.py`: pertanyaan yang sama dalam Python; beberapa jawabannya berbeda dari C++.

```python
# Pertanyaan yang sama dalam Python. Pembagian bulat ke bawah, % ikut pembagi.
import math

print("-7 // 2  =", -7 // 2)                  # membulatkan ke arah minus tak hingga
print("-7 % 2   =", -7 % 2)                   # tanda mengikuti pembagi
print("int(-7 / 2) =", int(-7 / 2))           # pemotongan gaya C++, bila perlu
print("math.fmod(-7, 2) =", math.fmod(-7, 2))
x = 0.1 + 0.2
print("0.1 + 0.2 == 0.3 :", x == 0.3)
print("0.1 + 0.2 printed with repr :", repr(x))
print("math.isclose(x, 0.3) :", math.isclose(x, 0.3))
print("int(0.29 * 100) =", int(0.29 * 100))
print("round(0.29 * 100) =", round(0.29 * 100))
print("round(2.5), round(3.5) =", round(2.5), round(3.5))
print("'%.2f' % 2.675 =", "%.2f" % 2.675)
```

```
-7 // 2  = -4
-7 % 2   = 1
int(-7 / 2) = -3
math.fmod(-7, 2) = -1.0
0.1 + 0.2 == 0.3 : False
0.1 + 0.2 printed with repr : 0.30000000000000004
math.isclose(x, 0.3) : True
int(0.29 * 100) = 28
round(0.29 * 100) = 29
round(2.5), round(3.5) = 2 4
'%.2f' % 2.675 = 2.67
```

Tabel 7.2 merangkum perbedaannya.

Tabel 7.2 Pembagian bulat, sisa bagi, dan pembulatan di C++ dan Python.

Ekspresi	C++	Python	Mengapa
−7 dibagi 2 (bilangan bulat)	`-7 / 2` bernilai −3	`-7 // 2` bernilai −4	C++ memotong ke arah nol, Python membulatkan ke bawah
sisa −7 dibagi 2	`-7 % 2` bernilai −1	`-7 % 2` bernilai 1	sisa mengikuti hasil bagi
membulatkan 2,5	`llround(2.5)` bernilai 3	`round(2.5)` bernilai 2	Python membulatkan setengah ke genap
memotong 0.29 × 100	`(int)` memberi 28	`int()` memberi 28	0.29 × 100 sedikit di bawah 29

Pembagian bulat C++ memotong ke arah nol, dan sisanya bertanda sama dengan bilangan yang dibagi, sehingga `(a / b) * b + a % b == a` selalu berlaku [3]. `//` di Python membulatkan ke bawah (ke arah minus tak hingga), dan `%`-nya bertanda sama dengan pembagi, yang menjaga identitas yang sama dengan pembulatan yang lain [8]. Keduanya konsisten; keduanya hanya berbeda. Jebakannya muncul di dua tempat.

Sisa bagi bilangan negatif. Banyak soal meminta jawaban "modulo m", dan ekspresi C++ seperti `(a - b) % m` bernilai negatif bila $a < b$. Perbaikan bakunya memetakan sisa apa pun ke $0, 1, \ldots, m-1$:

$$r = ((a \bmod m) + m) \bmod m \tag{7.6}$$

dihitung dengan `%` milik C++ sebagai mod dalam dan mod luar. Untuk $a = -7$ dan $m = 5$: $-7 \,\%\, 5 = -2$, $-2 + 5 = 3$, $3 \,\%\, 5 = 3$, dan memang $-7 = -2 \times 5 + 3$. `%` milik Python sudah memberi hasil ini.

Menerjemahkan antarbahasa. Rumus yang ditulis untuk Python, `(x + k) // 2` untuk `x` yang mungkin negatif, memberi jawaban lain bila disalin ke C++ sebagai `(x + k) / 2`. Bila bilangan negatif mungkin muncul, periksa setiap pembagian.

7.7.2 Bilangan pecahan

`double` menyimpan bilangan dalam biner dengan 53 bit signifikan, sekitar 16 digit desimal. Kebanyakan pecahan desimal, termasuk 0,1, 0,2, dan 0,3, tidak memiliki bentuk biner yang eksak, sama seperti 1/3 tidak memiliki bentuk desimal yang eksak. Jadi `0.1 + 0.2` adalah double terdekat ke 0.30000000000000004, dan itu tidak sama dengan double terdekat ke 0,3: kedua program mencetak `false` [9]. Akibatnya bagi kode lomba adalah tiga aturan.

1. **Jangan pernah membandingkan double hasil perhitungan dengan `==`.** Bandingkan dengan toleransi ε, seperti dalam demo:

$$|x - y| < \varepsilon \tag{7.7}$$

dengan ε dipilih menurut soal: jika jawaban diperiksa sampai 10^{-6}, hitunglah dengan $\varepsilon = 10^{-9}$. Python menyediakan `math.isclose`, yang memakai toleransi relatif.

2. **Jangan memotong double yang seharusnya bilangan bulat.** 0.29×100 adalah 28.999999999999996, dan `(int)` atau `int()` memotongnya menjadi 28. Bulatkan ke bilangan bulat terdekat, dengan `llround` atau `round`. Lebih baik lagi, hindari double sama sekali: harga dalam rupiah, atau jumlah dalam sen, tempatnya di bilangan bulat.
3. **Ketahui cara keluaran Anda dibulatkan.** `printf("%.2f", 2.675)` mencetak 2.67, bukan 2.68, karena double yang tersimpan sedikit di bawah 2,675. Bila juri membandingkan dengan toleransi, hal ini tidak penting; bila juri membandingkan teks secara persis, hitunglah dengan bilangan bulat atau lakukan pembulatannya sendiri.

Peringatan Bab 5 tentang `-ffast-math` juga berlaku di sini: dengan flag itu, bahkan uji toleransi pun bisa dioptimasi dengan cara yang mengejutkan.

7.8 Selisih Satu dan Bentuk Masukan

7.8.1 Menghitung tiang

Pagar Mas Dedi memiliki bentuk matematika yang tepat. Banyaknya bilangan bulat dalam rentang tertutup dari L sampai R, keduanya termasuk, adalah

$$|\{L, L+1, \ldots, R\}| = R - L + 1 \tag{7.8}$$

bukan $R - L$: pagar sepuluh meter dari tiang 0 sampai tiang 10 membutuhkan $10 - 0 + 1 = 11$ tiang. Hampir setiap kesalahan selisih satu adalah kerancuan antara rentang *tertutup* $[L, R]$, yang

memuat kedua ujungnya, dan rentang *setengah terbuka* $[L, R)$, yang memuat L tetapi tidak R, dengan $R - L$ elemen. C++ dan Python sama-sama menyukai rentang setengah terbuka: `for (int i = 0; i < n; i++)`, `range(0, n)`, `v.begin()` sampai `v.end()`, `s[a:b]`. Kebanyakan pernyataan soal memakai rentang tertutup: "cetak semua bilangan prima p dengan $m \leq p \leq n$". Penerjemahannya itulah tempat kekeliruan terjadi.

Tiga kebiasaan mencegah kebanyakan kesalahan ini.

- **Tuliskan rentangnya dengan kata-kata sebelum menulis loop**: "dari m sampai n, keduanya termasuk". Maka `x <= n`, bukan `x < n`. Program PRIME1 di Bab 6 melakukan persis hal ini.
- **Uji kedua ujungnya.** Setiap rentang memiliki elemen pertama dan terakhir; masukan terkecil (1 elemen, $L = R$) dan terbesar adalah tempat kesalahan selisih satu tampak.
- **Pertahankan satu konvensi di dalam program Anda.** Jika Anda memakai rentang setengah terbuka di mana-mana, seperti yang dilakukan merge sort Python di Bab 8 pada paruh-paruhnya, panjangnya selalu `hi - lo` dan titik tengahnya membelah dengan bersih.

7.8.2 Masukan tidak selalu seperti kelihatannya

Beberapa jebakan tinggal di pembacaan masukan.

- **Banyaknya kasus uji.** Sebagian soal memberi t lebih dahulu, sebagian diakhiri nilai penanda (-1 di CANDY, 42 di soal klasik pertama SPOJ, TEST), sebagian berakhir begitu saja di ujung berkas. Membaca dengan cara yang salah akan melewatkan uji atau membaca sampah. Di C++, `while (cin >> x)` dan `while (scanf("%d", &x) == 1)` berhenti dengan rapi di ujung berkas; di Python, membaca semua token sekaligus, seperti yang dilakukan setiap program Python di bab ini, membuat ketiga bentuk itu mudah.
- **Baris kosong dan spasi.** INVCNT di Bab 8 memisahkan kasus-kasus uji dengan baris kosong. Pembacaan berbasis token (`cin >>`, `scanf`, `split()`) melewatinya secara otomatis; pembacaan berbasis baris (`getline`, `input()`) tidak. Pilihlah token, kecuali soalnya memang tentang baris.
- **Akhir baris.** Berkas yang dibuat di Windows mengakhiri barisnya dengan dua karakter `\r\n`. Program yang membaca baris dengan `getline` di Linux menyisakan `\r` di ujung setiap baris, dan perbandingan seperti `line == "YES"` gagal. Juri membuat berkasnya di Linux, tetapi berkas uji buatan Anda sendiri, yang diketik di Windows, mungkin tidak; pembacaan berbasis token mengabaikan masalah ini.
- **Baris terakhir.** Sebagian berkas berakhir tanpa baris baru terakhir. `getline` tetap mengembalikan baris terakhir itu; kode yang membaca karakter demi karakter sampai `'\n'` bisa berputar selamanya atau kehilangan baris itu.

7.8.3 Bentuk keluaran

Juri membandingkan keluaran Anda dengan keluaran yang diharapkan, biasanya token demi token, kadang karakter demi karakter. Jebakan keluaran adalah bayangan cermin jebakan masukan: baris kosong yang hilang di antara kasus uji (PRIME1 memintanya), "Case 1:" padahal pernyataan soal berkata "Case #1:", `YES` padahal tertulis `Yes`, `-0.00` padahal yang diharapkan `0.00`. Format keluaran di pernyataan soal adalah spesifikasi, bukan saran; bacalah dua kali.

7.8.4 Pembanding yang berkata "kurang dari atau sama dengan"

Satu kekeliruan lagi termasuk keluarga selisih satu, karena ia kerancuan yang sama antara $<$ dan $\leq$ di tempat yang baru. `std::sort` menerima fungsi pembanding, dan seorang peserta yang ingin mengurutkan pasangan menurut elemen pertamanya menulis

```
sort(v.begin(), v.end(), [](const P& a, const P& b) { return a.first <= b.first; });
```

Standar C++ mensyaratkan pembanding berperilaku seperti "benar-benar lebih kecil": untuk elemen yang sama, ia harus menjawab *false* ke kedua arah [3]. Pembanding dengan `<=` menjawab *true* untuk `cmp(x, x)`, yang melanggar aturan itu, dan perilaku program menjadi tak terdefinisi. Dalam praktik, dengan banyak kunci yang sama, pengurutan bisa membaca di luar larik lalu mogok, atau meninggalkan elemen dalam urutan yang salah. Perbaikannya satu karakter, `<` alih-alih `<=`; jika elemen pertama yang sama harus diurutkan menurut hal lain, bandingkan elemen kedua ketika elemen pertamanya sama. `sorted` di Python menerima fungsi `key` alih-alih pembanding dan tidak memiliki jebakan ini, satu alasan lagi untuk memilih `key=` ketika program Python mengurutkan catatan-catatan.

Kesalahan ini licik karena jarang terlihat pada masukan kecil: dengan sedikit kunci yang sama, implementasi pengurutan sering kebetulan memberikan hasil yang benar. Ia baru muncul pada masukan besar dengan banyak nilai kembar, persis jenis masukan yang dimiliki juri dan jarang dimiliki contoh soal. Uji stres dengan generator yang sengaja membuat banyak kunci kembar, misalnya nilai-nilai yang diambil dari rentang 1 sampai 3 saja, akan menemukannya jauh lebih cepat daripada generator dengan nilai yang tersebar lebar.

7.9 Mengujinya

Setiap soal di bab ini diuji stres dengan cara yang sama: generator, brute force yang mengikuti pernyataan soal tanpa kecerdikan, dan pengikat uji `stress.py` dari Bab 8, 300 uji acak per putaran. **Tabel 7.3** mencatat hasilnya untuk program-program yang benar dan untuk yang sengaja dirusak.

Tabel 7.3 Hasil uji stres untuk program-program bab ini (300 uji acak per putaran).

Program	Generator	Hasil
`eights.cpp`, `eights.py`	k kecil dan k besar	OK: 300 dari 300, kedua mode
`eights_int.cpp`	k kecil	OK: 300 dari 300 (luapan tak terlihat)
`eights_int.cpp`	k besar	tidak cocok pada benih 1
`fctrl2.cpp`, `fctrl2.py`	$1 \le n \le 100$	OK: 300 dari 300
`fctrl2_ull.cpp`	$1 \le n \le 100$	tidak cocok pada benih 1 (mencetak 0 untuk 73!)
`addrev.cpp`, `addrev.py`	pasangan dengan jumlah bulat	OK: 300 dari 300
`candy.cpp`, `candy.py`	1 sampai 4 blok	OK: 300 dari 300
`candy_leak.cpp`	1 sampai 4 blok	tidak cocok pada benih 3

Dua baris layak ditekankan. Baris `eights_int` dengan k kecil lolos sepenuhnya: uji stres hanya menemukan apa yang dapat dihasilkan generatornya. Dan setiap program rusak tertangkap dalam tiga benih pertama, yang memang lazim: jebakan bukanlah peristiwa langka, melainkan sistematis, dan beberapa ratus uji acak biasanya cukup untuk membongkarnya.

7.10 Menggunakannya Sungguhan

Tujuh keluarga di **Tabel 7.1** membentuk daftar periksa yang hanya memakan dua menit sebelum setiap pengiriman.

1. **Ukuran.** Berapa nilai terbesar yang dapat dicapai setiap variabel, jumlah, atau hasil kali? Apakah semuanya bertipe yang dapat memuatnya, dan apakah setiap perkalian dilakukan dalam tipe itu?
2. **Melampaui tipe.** Bisakah jawaban lebih dari 18 digit? Maka pakai digit dalam larik, atau Python.
3. **Format.** Apa karakter pertama dan terakhir setiap nilai yang dicetak? Nol di depan, tanda, presisi, pemisah, baris baru terakhir.
4. **Keadaan.** Apakah setiap variabel per kasus uji dibuat di dalam loop kasus uji, atau diatur ulang di awalnya?
5. **Pembagian.** Adakah yang negatif yang dibagi atau direduksi modulo m?
6. **Bilangan pecahan.** Adakah `==` yang diterapkan pada double? Adakah double yang dipotong menjadi bilangan bulat?
7. **Rentang dan masukan.** Tertutup atau setengah terbuka? Apa masukan terkecil dan terbesar, dan apa yang dilakukan program di setiap ujung? Bagaimana masukan berakhir?

Daftar yang sama berguna bagi siapa pun yang mengubah aturan menjadi program. Toko kelontong yang menghitung potongan 2,5% atas harga dalam rupiah bertemu keluarga 5 dan 6 sekaligus: $2.5\% \times 19\,900 = 497.5$, dan apakah pembeli membayar 19.402 atau 19.403 bergantung pada aturan pembulatan yang harus dinyatakan program, bukan diwarisi dari `round`. Lembar gaji yang menjumlahkan gaji setahun seratus karyawan bertemu keluarga 1 di kolom 32-bit mana pun. Dan laporan bulanan koperasi yang tanpa sengaja memulai setiap bulan dari total bulan lalu bertemu keluarga 4. Tak satu pun dari ini soal algoritma; semuanya nyata.

Satu kebiasaan lagi membuat daftar itu menjadi milik Anda sendiri. Simpan buku catatan kecil, di kertas atau di berkas teks di samping solusi-solusi Anda, dan tulis satu baris setiap kali sebuah kiriman gagal karena alasan dari bab ini: soalnya, keluarganya, dan baris yang salah. Setelah beberapa bulan, buku catatan itu memperlihatkan keluarga mana yang paling sering menjebak *Anda*, dan keluarga-keluarga itulah yang layak diperiksa pertama kali dalam setiap daftar periksa. Peserta berpengalaman bukanlah orang yang tidak pernah membuat kekeliruan ini; mereka adalah orang yang tahu kekeliruan mana yang biasa mereka buat, dan karena itu tahu harus mencari di mana lebih dahulu.

7.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Bab ini sendiri sudah merupakan katalog kesalahan, jadi daftar di sini pendek: kesalahan-kesalahan tingkat atas yang membiarkan kesalahan lain lolos.

- **Mempercayai contoh.** Contoh dipilih untuk menjelaskan soal, bukan untuk menguji program Anda; ukurannya kecil dan bersahabat. Contoh EIGHTS adalah jenis masukan yang diloloskan `eights_int.cpp`.
- **Menguji stres hanya dengan masukan kecil.** Masalah luapan dan kinerja hanya muncul di puncak batasan. Selalu tambahkan mode “big” ke generator.
- **Membiarkan brute force berbagi anggapan dengan program cepat.** Brute force EIGHTS yang memakai Persamaan (7.1) tidak memeriksa apa-apa. Buat brute force mengikuti pernyataan soal.
- **Memperbaiki satu gejala, bukan seluruh keluarganya.** Setelah menemukan satu luapan, periksa setiap perkalian lain di program; setelah menemukan satu kebocoran, periksa setiap variabel global.

- **Mengabaikan pola vonis.** "Wrong answer on test 1" dan "wrong answer on test 17" adalah petunjuk yang berbeda: yang pertama biasanya berarti masalah format atau pembacaan, yang kedua masalah ukuran atau tepi.

7.12 Ringkasan Bab

- Gagasan yang benar tetap bisa mendapat *Wrong answer* melalui tujuh keluarga kekeliruan (**Tabel 7.1**), masing-masing dengan gejala yang dapat dikenali.
- EIGHTS: bilangan yang pangkat tiganya berakhir 888 adalah $192 + 250(k-1)$ (Persamaan (7.1)), dibuktikan dengan memecah 1000 menjadi 8×125; jawabannya mencapai 5×10^{14} (Persamaan (7.2)), jadi setiap langkah harus 64-bit, dan tipe sebuah hasil kali adalah tipe operand-nya.
- FCTRL2: `unsigned long long` memuat faktorial hanya sampai 20! dan mencetak 0 mulai 66! (Persamaan (7.3)); digit dalam larik, dikalikan dengan simpanan (Persamaan (7.4)), menangani 158 digit, dan Python tidak memerlukan kewaspadaan.
- ADDREV: setiap pembalikan harus membuang nol di depan; keluaran "0401" alih-alih "401" adalah *Wrong answer*.
- CANDY: banyak langkah adalah total kelebihan di atas rata-rata (Persamaan (7.5)); total yang dideklarasikan di luar loop uji bocor ke blok-blok berikutnya, gejala klasik "kasus pertama benar, berikutnya salah".
- Pembagian C++ memotong dan Python membulatkan ke bawah; sisa bagi bilangan negatif memerlukan Persamaan (7.6); double dibandingkan dengan toleransi (Persamaan (7.7)) dan dibulatkan, bukan dipotong.
- Rentang dari L sampai R inklusif memiliki $R - L + 1$ elemen (Persamaan (7.8)); format masukan dan keluaran adalah spesifikasi.

7.13 Soal Latihan

1. Tanpa menjalankan apa pun, carilah bilangan ke-5 dan ke-1000 yang pangkat tiganya berakhir 888, lalu periksa bahwa a_{1000}^3 berakhir 888 dengan hanya menghitung tiga digit terakhirnya.
2. Bilangan mana yang pangkat tiganya berakhir 111? Pakai metode Subbab 7.3.2 dan berikan rumus untuk bilangan ke-k.
3. Ekspresi `long long area = w * h;` dengan `int w = 100000, h = 100000;` mencetak luas yang salah. Berikan dua perbaikan satu baris yang berbeda dan jelaskan mengapa `long long area = (long long)(w * h);` bukan salah satunya.
4. Ubah `fctrl2.cpp` agar menyimpan *empat* digit desimal per `int` (basis 10.000) alih-alih satu. Berapa nilai antara terbesar di Persamaan (7.4) sekarang, dan bagaimana pencetakan harus berubah?
5. Sebuah soal meminta jumlah n bilangan modulo $10^9 + 7$, dengan bilangan-bilangan yang bisa negatif (sampai -10^9). Tulis loop penjumlahannya dalam C++ agar hasil yang dicetak selalu berada di $0, \ldots, 10^9 + 6$.
6. Sebuah program membaca daftar harga seperti `19.90` sebagai double, mengalikan masing-masing dengan 100, lalu memotongnya untuk memperoleh sen. Untuk harga mana di antara 0.29, 0.57, 1.15, 19.90 program ini memberi banyak sen yang salah? Sarankan cara membaca harga yang tidak mungkin salah.

7.14 Penyelesaian Soal Latihan

1. Menurut Persamaan (7.1), $a_5 = 192 + 250 \times 4 = 1192$ dan $a_{1000} = 192 + 250 \times 999 = 249\,942$. Hanya tiga digit terakhir yang menentukan tiga digit terakhir pangkat tiganya: $249\,942 \equiv 942 \pmod{1000}$, dan $942^2 = 887\,364 \equiv 364$, lalu $364 \times 942 = 342\,888 \equiv 888$. Jadi a_{1000}^3 berakhir 888, sebagaimana mestinya.

2. Modulo 8: $111 \equiv 7 \pmod 8$, dan pemangkatan tiga mempermutasikan sisa-sisa ganjil modulo 8 ($1^3 = 1$, $3^3 = 27 \equiv 3$, $5^3 = 125 \equiv 5$, $7^3 = 343 \equiv 7$), jadi $n \equiv 7 \pmod 8$. Modulo 125: 111 tidak habis dibagi 5, jadi menurut argumen di Subbab 7.3.2 ada tepat satu penyelesaian $n^3 \equiv 111 \pmod{125}$; mencoba sisa-sisanya memberi $n \equiv 96$, dan memang $96^3 = 884\,736 = 7077 \times 125 + 111$. Menggabungkan $n \equiv 7 \pmod 8$ dan $n \equiv 96 \pmod{125}$ memberi $n \equiv 471 \pmod{1000}$: $471^3 = 104\,487\,111$. Jadi bilangan-bilangannya 471,1471,2471, ..., dan yang ke-k adalah $471 + 1000(k-1)$. Berbeda dengan 888, periodenya 1000 penuh, karena modulo 8 hanya ada satu penyelesaian, bukan empat.

3. `w * h` dihitung dalam `int` karena kedua operand-nya `int`, meluap, dan baru kemudian dikonversi. Perbaikannya: `long long area = (long long)w * h;` (satu operand yang dikonversi sebelum perkalian membuat seluruh hasil kali menjadi 64-bit), atau deklarasikan `w` dan `h` sebagai `long long`. Cast `(long long)(w * h)` mengonversi hasil *setelah* perkalian yang meluap, sudah terlambat.

4. Dengan basis $B = 10\,000$, Persamaan (7.4) menjadi $x_j = d_j f + c$, $d_j = x_j \bmod B$, $c = \lfloor x_j / B \rfloor$. Dengan $d_j \le 9999$, $f \le 100$, dan $c \le 100$, x_j terbesar adalah $9999 \times 100 + 100 = 1\,000\,000$, yang muat dalam `int`. Pencetakan: "digit" paling signifikan dicetak biasa, setiap yang lain dengan tepat empat digit termasuk nol di depan, `printf("%04d", d)`; lupa mengisi nol itu adalah jebakan baru ala ADDREV. Loopnya mengerjakan seperempat langkah.

5. Terapkan Persamaan (7.6) setelah setiap penjumlahan:

```
const long long MOD = 1000000007;
long long sum = 0;
for (long long x : values)
    sum = ((sum + x) % MOD + MOD) % MOD;   // sisa yang aman, di setiap langkah
```

`sum + x` paling kecil -10^9 dan paling besar sekitar 2×10^9, yang muat dalam `long long`; `%` bagian dalam bisa negatif, dan menambahkan `MOD` sebelum `%` bagian luar membuatnya tak negatif.

6. Keempatnya. 0.29×100 memberi 28.999999999999996 dan terpotong menjadi 28; $0.57 \times 100 = 56.99999999999999$ memberi 56; $1.15 \times 100 = 114.99999999999999$ memberi 114; dan $19.90 \times 100 = 1989.9999999999998$ memberi 1989. Masing-masing kurang satu sen. Cara membaca yang aman: bulatkan alih-alih memotong (`llround(x * 100)`), atau lebih baik lagi, jangan pernah membuat double sama sekali: baca harga sebagai teks, pisahkan di titik desimal, dan hitung `rupiah * 100 + cents` dengan bilangan bulat.

Daftar Pustaka

[1] Sphere Online Judge, "EIGHTS - Triple Fat Ladies." [Online]. Available: https://www.spoj.com/problems/EIGHTS/ (accessed Sep. 2026).

[2] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 6th ed., revised by D. R. Heath-Brown and J. H. Silverman. Oxford, UK: Oxford University Press, 2008.

[3] cppreference.com, "Arithmetic operators." [Online]. Available: https://en.cppreference.com/w/cpp/language/operator_arithmetic (accessed Sep. 2026).

[4] Sphere Online Judge, "FCTRL2 - Small factorials." [Online]. Available: https://www.spoj.com/problems/FCTRL2/ (accessed Sep. 2026).

[5] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.

[6] Sphere Online Judge, "ADDREV - Adding Reversed Numbers." [Online]. Available: https://www.spoj.com/problems/ADDREV/ (accessed Sep. 2026).

[7] Sphere Online Judge, "CANDY - Candy I." [Online]. Available: https://www.spoj.com/problems/CANDY/ (accessed Sep. 2026).

[8] Python Software Foundation, "Built-in Types: Numeric Types," *The Python Standard Library*, Python 3.14 documentation. [Online]. Available: https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

[9] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, Mar. 1991.

Bab 8

Men-debug TLE dan WA: Stress Test dan Pemeriksa Brute-Force

Juri hampir tidak memberi tahu apa-apa. Ia tidak memperlihatkan uji yang gagal, jawaban yang diharapkannya, atau baris yang lambat; ia hanya berkata *Wrong answer* atau *Time limit exceeded*, lalu menunggu. Bab ini membahas cara menemukan bug itu juga, di mesin Anda sendiri, sebelum maupun sesudah vonis. Di akhir bab Anda akan mampu menulis tiga program kecil sebuah uji stres, yaitu generator, brute force, dan pengikat uji (*harness*), lalu memakainya untuk menemukan jawaban yang salah dalam hitungan menit; memperkecil masukan yang gagal sampai bug-nya tampak jelas; membangun masukan kasus terburuk dan mengukur bagaimana waktu sebuah program tumbuh, untuk meramalkan batas waktu sebelum juri melakukannya; memakai asersi, mode debug pustaka C++, dan address sanitizer untuk menghentikan program tepat di baris tempat ia keliru; dan menerapkan semua ini pada dua soal SPOJ sungguhan, INVCNT dan PALIN, masing-masing dengan bug yang kita tanam, temukan, dan perbaiki di depan Anda.

8.1 Menghitung Uang Dua Kali

Bu Endang adalah bendahara sebuah *arisan*, kelompok tabungan bergilir beranggotakan tiga puluh ibu di sebuah desa dekat Sleman. Setiap bulan setiap anggota menyetor, seorang anggota membawa pulang seluruh uang, dan Bu Endang harus memastikan jumlahnya tepat. Ia menghitungnya dua kali, dengan dua cara berbeda. Pertama ia menjumlahkan setoran di buku catatannya dengan kalkulator, anggota demi anggota. Lalu ia memilah uang kertas di meja menurut pecahannya, menghitung setiap tumpukan, dan mengalikan: sekian lembar seratus ribuan, sekian lembar lima puluh ribuan, dan seterusnya. Jika kedua total sama, ia yakin; dua metode yang berbeda jarang membuat kesalahan yang sama. Jika berbeda, ia tidak menghitung ulang semuanya. Ia membelah buku catatan menjadi dua paruh dan uangnya menjadi dua kelompok anggota yang sama, mencari paruh yang totalnya berbeda, lalu membelah lagi, sampai yang ia pandangi tinggal satu setoran anggota dan satu tumpukan kecil uang.

Tanpa menyebut namanya, Bu Endang telah menemukan metode bab ini. Total buku catatan adalah program cepat Anda, yang ingin Anda kirim. Hitungan tumpukan adalah *brute force*, metode kedua yang begitu sederhana sehingga hampir pasti benar, walaupun lambat. Membandingkan keduanya pada banyak masukan adalah *uji stres* (*stress test*). Membelah perbedaan menjadi paruh-paruh sampai cukup kecil untuk dilihat adalah *memperkecil* (*shrinking*). Dan alasan metode ini berhasil sama bagi dia dan bagi Anda: kesalahan dalam satu metode kecil kemungkinannya terulang, secara kebetulan, dalam metode yang sama sekali berbeda.

Pemrogram lebih membutuhkan metode ini daripada bendahara, karena juri memberi keterangan lebih sedikit lagi daripada uang yang kurang. Juri tidak pernah memperlihatkan uji yang gagal. Peserta yang mendapat *Wrong answer* lalu sekadar membaca ulang kodenya seperti bendahara yang menghitung ulang buku catatan dengan kalkulator yang sama: ia akan membuat kesalahan yang sama lagi. Metode kedualah yang menemukannya.

8.2 Gagasannya dengan Kata-Kata Sederhana

> **DEFINISI.** *Kandidat* adalah program yang ingin Anda kirim. *Brute force* (atau solusi acuan) adalah program untuk soal yang sama yang ditulis agar jelas benar, bukan agar cepat, misalnya dengan mencoba setiap kemungkinan. *Generator* adalah program yang menulis masukan acak yang sah, ditentukan oleh sebuah bilangan yang disebut *benih* (*seed*), sehingga benih yang sama selalu memberi masukan yang sama. *Uji stres* menjalankan generator untuk banyak benih dan membandingkan keluaran kandidat dengan keluaran brute force pada setiap masukan, berhenti pada perbedaan pertama. *Masukan gagal minimal* adalah masukan gagal yang tidak dapat diperkecil lagi tanpa berhenti gagal.

Mengapa masukan acak yang kecil bisa menemukan bug yang ditemukan uji besar milik juri? Karena kebanyakan bug tidak berkaitan dengan ukuran. Selisih satu, simpanan yang terlupa, perbandingan yang keliru, atau kasus khusus yang terlewat gagal juga pada *sebagian* masukan kecil, biasanya masukan dengan dua atau tiga elemen atau dua atau tiga digit, dan beberapa ratus masukan kecil acak memuat sebagian besar bentuk-bentuk kecil itu. Pengamatan ini memiliki nama dalam rekayasa perangkat lunak, *hipotesis cakupan kecil* (*small scope hypothesis*): kebanyakan bug dapat diperlihatkan pada instans yang kecil [1]. Peserta berpengalaman mengandalkannya setiap hari, dan buku-buku panduan pemrograman kompetitif menganjurkan rutinitas yang persis sama, yaitu solusi acuan yang lambat dan uji acak [2]. Pengecualiannya justru bug yang berkaitan dengan ukuran: luapan, yang membutuhkan nilai besar (Bab 7), dan kelambatan, yang membutuhkan masukan besar. Untuk keduanya, uji stres memerlukan jenis masukan kedua, yaitu yang terbesar dan terburuk yang diizinkan batasan, serta cara mengukur waktu. Bab ini membangun keduanya.

Tabel 8.1 merangkum apa yang disiratkan setiap vonis dan perkakas mana di bab ini yang dipakai untuknya.

Tabel 8.1 Apa yang biasanya berarti dari setiap vonis, dan perkakas pertama yang sebaiknya diraih.

Vonis	Penyebab yang paling mungkin	Perkakas pertama
Wrong answer, uji kecil lolos	kekeliruan logika pada suatu bentuk kecil	uji stres acak (Subbab 8.3 sampai 8.5)
Wrong answer, hanya uji besar	luapan, percabangan yang bergantung pada ukuran	masukan terbesar melawan jawaban independen (Subbab 8.5.3)
Time limit exceeded	kompleksitas yang salah, atau langkah kuadratik tersembunyi	pengukuran waktu pada masukan yang membesar dan kasus terburuk (Subbab 8.6, 8.7.4)
Runtime error	indeks di luar jangkauan, stack meluap, pembagian dengan nol	mode debug, sanitizer, asersi (Subbab 8.8)

8.3 Pengikat Uji

Ketiga program sebuah uji stres berukuran kecil, dan pengikat ujinya yang terkecil. **Kode 8.1** adalah yang dipakai di seluruh buku ini. Ia menerima tiga perintah, menjalankan generator dengan benih 1, 2, 3, …, memberikan setiap masukan kepada brute force dan kepada kandidat, lalu membandingkan keluaran keduanya token demi token.

Kode 8.1 `stress.py`: menjalankan generator, brute force, dan kandidat untuk banyak benih dan berhenti pada perbedaan pertama.

```
# Penguji stres: jalankan generator, brute force, dan kandidat pada banyak
# masukan acak dan berhenti pada perbedaan pertama.
# Pemakaian: python3 stress.py GEN BRUTE CAND [iterasi] [benih_pertama]
#   GEN, BRUTE, CAND ialah perintah bertanda kutip, mis. "python3 gen.py" "./fast".
#   Generator menerima benih (seed) sebagai argumen terakhir.
import shlex
import subprocess
import sys


def run(cmd, text_in=""):
    try:
        r = subprocess.run(shlex.split(cmd), input=text_in, capture_output=True,
                           text=True, timeout=5)
    except subprocess.TimeoutExpired:
        return "TIMEOUT (more than 5 s)"
    if r.returncode != 0:
        raise RuntimeError("%s exited with %d\n%s" % (cmd, r.returncode, r.stderr))
    return r.stdout


def main():
    gen, brute, cand = sys.argv[1:4]
    iterations = int(sys.argv[4]) if len(sys.argv) > 4 else 300
    first = int(sys.argv[5]) if len(sys.argv) > 5 else 1
    for seed in range(first, first + iterations):
        test = run("%s %d" % (gen, seed))
        want = run(brute, test).split()
        got = run(cand, test).split()
        if want != got:
            print("MISMATCH on seed %d" % seed)
            print("input:\n" + test.rstrip())
            print("expected: " + " ".join(want)[:200])
            print("received: " + " ".join(got)[:200])
            with open("failing_input.txt", "w") as f:
                f.write(test)
            sys.exit(1)
    print("OK: %d random tests agree" % iterations)


if __name__ == "__main__":
    main()
```

Beberapa keputusan di dalamnya disengaja.

- **Perintah, bukan bahasa.** Ketiga argumen adalah perintah bergaya shell, dipecah dengan `shlex.split`, sehingga campuran apa pun berhasil: generator Python, brute force C++, dan kandidat Python, atau sebaliknya.
- **Benih, bukan keacakan.** Generator menerima benih sebagai argumen terakhirnya dan harus menghasilkan masukan yang sama untuk benih yang sama. Kasus gagal lalu dapat diulang hanya dengan benihnya, dan begitu pula cara membagikan kegagalan kepada seorang teman.
- **Perbandingan token.** `split()` mengabaikan perbedaan spasi dan pergantian baris, seperti kebanyakan juri. Untuk soal yang jurinya membandingkan teks secara persis, bandingkan `stdout` secara langsung.
- **Batas waktu.** Kandidat yang berputar selamanya akan membuat pengikat uji macet; setelah 5 detik eksekusinya dicatat sebagai `TIMEOUT` dan dihitung sebagai perbedaan. Bab 5 memakai ini untuk menangkap loop tanpa akhir.
- **Masukan yang gagal disimpan** di `failing_input.txt`, siap diberikan kepada debugger atau kepada kandidat yang ditambahi pencetakan.

Jalankan dari terminal di folder `tests` bab ini, di VS Code atau di mana saja: `python3 stress.py "python3 gen.py" build/brute build/fast 300` di Linux dan macOS, `py stress.py "py gen.py" build\brute.exe build\fast.exe 300` di Windows.

Generator yang baik mengikuti empat aturan. Ia hanya menulis masukan yang *sah*, persis seperti yang digambarkan pernyataan soal, karena kalau tidak, perbandingannya tak bermakna. Ia menjaga masukan tetap *kecil*, sehingga brute force cepat dan kegagalan mudah dibaca. Ia membiarkan ukurannya dipilih, sehingga kegagalan dapat diperkecil. Dan ia membidik *bentuk-bentuk berbahaya*: nilai berulang, nilai di ujung rentang, urutan terurut dan terbalik, digit 9 dan 0. **Kode 8.2** adalah generator untuk soal pertama bab ini.

Kode 8.2 `gen_invcnt.py`: masukan INVCNT acak dengan nilai berbeda dan baris kosong sesuai format masukan aslinya; ukuran larik terbesar dapat dipilih.

```
# Masukan INVCNT acak: nilai berbeda, larik mungil, baris kosong seperti di SPOJ.
# Pemakaian: python3 gen_invcnt.py [max_n] seed   (bawaan max_n = 6)
import random
import sys

rng = random.Random(int(sys.argv[-1]))
max_n = int(sys.argv[1]) if len(sys.argv) > 2 else 6
t = rng.randint(1, 3)
print(t)
for _ in range(t):
    n = rng.randint(1, max_n)
    print()
    print(n)
    print("\n".join(map(str, rng.sample(range(1, 20), n))))
```

Generator ini menulis baris kosong sebelum setiap kasus uji, sebagaimana format masukan INVCNT (Subbab 8.4.1), sehingga cara kandidat membaca masukan ikut teruji, dan ia menerima argumen pertama opsional `max_n`, yang dipakai Subbab 8.5.2 untuk memperkecil kegagalan.

8.3.1 Memeriksa generatornya sendiri

Generator juga sebuah program, dan ia bisa keliru. Generator yang sesekali menulis masukan yang tidak sah, dua nilai yang sama di INVCNT atau bilangan dengan nol di depan di PALIN, menghasilkan "kegagalan" yang sama sekali bukan bug kandidat, dan seorang peserta bisa kehilangan satu jam men-debug program yang sebenarnya benar. Dua kebiasaan murah mencegahnya.

Pertama, **bacalah beberapa masukan yang dibangkitkan** sebelum uji stres pertama: `python3 gen_invcnt.py 1`, `python3 gen_invcnt.py 2`, dan seterusnya. Lima benih, sepuluh detik, dan kebanyakan kekeliruan dalam generator langsung tampak oleh mata: baris kosong yang hilang, hitungan n yang tidak cocok dengan banyaknya nilai, atau nilai di luar rentang yang diizinkan.

Kedua, untuk hal-hal yang kurang kasatmata, tulislah sebuah **validator**: program pendek yang membaca masukan dan memeriksa setiap batasan dalam pernyataan soal, tidak mencetak apa pun bila masukannya sah dan mencetak pesan bila tidak. Untuk INVCNT, validator memeriksa bahwa banyaknya kasus uji cocok dengan t, bahwa setiap larik berisi tepat n nilai, dan bahwa nilai-nilainya bilangan bulat positif yang berbeda sampai 10^7. Menjalankan validator pada beberapa ratus masukan yang dibangkitkan, dalam loop yang sama dengan uji stres, hanya memakan beberapa detik. Para pembuat soal memvalidasi berkas uji mereka sendiri dengan cara persis seperti ini; peserta yang melakukan hal yang sama tidak akan pernah memburu bug yang diciptakan oleh masukannya sendiri.

Validator memiliki kegunaan kedua. Bila uji stres gagal, masukan yang gagal itu sudah diperiksa, sehingga perbedaannya pasti berasal dari salah satu dari kedua program; dan bila brute force dan kandidat berbeda pada masukan yang ditolak validator, kesalahannya ada pada generator. Dengan begitu, pertanyaan "salah siapa?" selalu punya jawaban yang cepat, dan waktu Anda dipakai untuk memperbaiki hal yang benar-benar rusak.

Kebiasaan ini terasa berlebihan untuk soal sekecil INVCNT, tetapi nilainya segera terasa pada soal yang batasannya rumit, misalnya graf yang harus terhubung, pohon tanpa siklus, atau string yang hanya boleh memuat huruf tertentu. Di soal-soal seperti itu, generator yang keliru adalah hal yang biasa, dan validator adalah satu-satunya cara untuk mengetahuinya sebelum waktu terbuang.

8.4 INVCNT: Menghitung Inversi

8.4.1 Soalnya

Soal SPOJ INVCNT, "Inversion Count" [3], memberi larik A berisi n bilangan bulat positif yang berbeda, dengan $n \le 200\,000$ dan nilai sampai 10^7, lalu menanyakan berapa banyak *inversi* di dalamnya: pasangan posisi $i < j$ dengan $A[i] > A[j]$, yaitu pasangan yang urutannya salah. Masukan diawali banyaknya kasus uji t; setiap kasus uji didahului sebuah baris kosong dan memberi n diikuti n nilai, satu per baris. Batas waktunya sekitar 3,6 detik.

Untuk larik 7,2,9,4 inversinya (7,2), (7,4), dan (9,4), jadi jawabannya 3. Larik terurut tidak memiliki inversi; larik dalam urutan terbalik memiliki satu inversi untuk setiap pasangan, sebanyak mungkin:

$$I_{\max}(n) = \binom{n}{2} = \frac{n(n-1)}{2} \tag{8.1}$$

Untuk $n = 200\,000$ nilainya $19\,999\,900\,000$, hampir sepuluh kali `int` terbesar. Penghitungnya harus 64-bit.

8.4.2 Brute force-nya

Brute force memeriksa setiap pasangan, persis seperti bunyi definisi. **Kode 8.3** adalah programnya.

Kode 8.3 `invcnt_brute.cpp`: brute force memeriksa setiap pasangan; jelas benar dan berwaktu kuadratik.

```cpp
// Brute force INVCNT: periksa setiap pasangan (i, j) dengan i < j. Waktu O(n^2),
// jelas benar -- acuan untuk memeriksa program yang cepat.
#include <cstdio>
#include <vector>

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        int n;
        if (std::scanf("%d", &n) != 1) break;
        std::vector<long long> a(n);
        for (auto& x : a)
            if (std::scanf("%lld", &x) != 1) return 0;
        long long inversions = 0;
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j < n; j++)
                if (a[i] > a[j]) inversions++;
        std::printf("%lld\n", inversions);
    }
    return 0;
}
```

Program ini melakukan $\binom{n}{2}$ perbandingan, sekitar 2×10^{10} untuk larik terbesar, jauh terlalu lambat untuk juri tetapi seketika untuk larik berisi paling banyak 6 elemen yang ditulis generator. Ia brute force yang baik karena tidak ada yang bisa keliru: dua loop dan satu perbandingan, langsung dari definisinya.

8.4.3 Gagasan cepatnya: hitung sambil menggabung

Merge sort mengurutkan larik dengan mengurutkan kedua paruhnya secara rekursif lalu *menggabungkannya*: berulang kali bandingkan elemen terkecil yang tersisa di setiap paruh dan pindahkan yang lebih kecil ke keluaran [4]. Inversi dapat dihitung selama penggabungan tanpa biaya tambahan.

Setiap inversi (i, j) termasuk salah satu dari tiga jenis: kedua posisinya di paruh kiri, keduanya di paruh kanan, atau i di kiri dan j di kanan. Dua jenis pertama dihitung oleh panggilan-panggilan rekursif. Mengurutkan sebuah paruh tidak mengubah elemen-elemen mana yang ada di dalamnya, jadi tidak mengubah banyaknya inversi *silang*, yang kita hitung saat menggabung. Misalkan paruh kiri sudah terurut sebagai $L[\text{lo}..\text{mid}]$, paruh kanan sebagai $R[\text{mid}+1..\text{hi}]$, dan penggabungan hendak mengambil $R[j]$ karena lebih kecil daripada $L[i]$, elemen terkecil yang masih menunggu di paruh kiri. Maka setiap elemen yang masih menunggu di paruh kiri, $L[i], L[i+1], \ldots, L[\text{mid}]$, lebih besar daripada $R[j]$ (semuanya paling kecil $L[i]$), dan setiap elemen kiri yang sudah diambil lebih kecil daripada $R[j]$ (ia diambil karena paling besar sama dengan suatu elemen kanan yang tidak lebih besar daripada $R[j]$). Jadi $R[j]$ membentuk inversi silang dengan tepat elemen-elemen kiri yang menunggu, dan hitungannya bertambah

$$\text{mid} - i + 1 \tag{8.2}$$

yaitu banyaknya elemen di $L[i..\text{mid}]$, tertutup di kedua ujungnya. Setiap elemen kanan diambil tepat sekali, sehingga setiap inversi silang terhitung tepat sekali.

Waktunya memenuhi rekurens merge sort: dua paruh ditambah penggabungan linear,

$$T(n) = 2\,T(n/2) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n \log n) \tag{8.3}$$

sekitar $200\,000 \times 18 \approx 3.6 \times 10^6$ langkah untuk larik terbesar. **Kode 8.4** adalah programnya.

Kode 8.4 `invcnt.cpp`: penghitungan inversi dengan merge sort, memakai penghitung 64-bit.

```cpp
// INVCNT (SPOJ): hitung pasangan i < j dengan a[i] > a[j], n <= 200.000.
// Merge sort: saat menggabung dua paruh terurut, elemen yang diambil dari
// paruh kanan melompati setiap elemen yang masih menunggu di paruh kiri.
#include <cstdio>
#include <vector>

long long sortCount(std::vector<int>& a, std::vector<int>& tmp, int lo, int hi) {
    if (hi - lo < 1) return 0;                    // satu elemen: tidak ada pasangan
    int mid = lo + (hi - lo) / 2;                 // paruh [lo, mid], [mid+1, hi]
    long long count = sortCount(a, tmp, lo, mid) + sortCount(a, tmp, mid + 1, hi);
    int i = lo, j = mid + 1, k = lo;
    while (i <= mid && j <= hi) {
        if (a[i] <= a[j]) {
            tmp[k++] = a[i++];
        } else {
            count += mid - i + 1;                 // a[i..mid] semuanya > a[j]
            tmp[k++] = a[j++];
        }
    }
    while (i <= mid) tmp[k++] = a[i++];
    while (j <= hi) tmp[k++] = a[j++];
    for (k = lo; k <= hi; k++) a[k] = tmp[k];
    return count;
}

int main() {
    int t;
    if (std::scanf("%d", &t) != 1) return 0;
    while (t--) {
        int n;
        if (std::scanf("%d", &n) != 1) break;
        std::vector<int> a(n), tmp(n);
        for (int& x : a)
            if (std::scanf("%d", &x) != 1) return 0;
        std::printf("%lld\n", n > 0 ? sortCount(a, tmp, 0, n - 1) : 0LL);
    }
    return 0;
}
```

Kedalaman rekursinya hanya $\log_2 n \approx 18$, jadi stack di Bab 6 tidak perlu dikhawatirkan. **Kode 8.5** melakukan hal yang sama secara bottom-up di Python: ia menggabungkan potongan selebar 1, 2, 4, … sampai satu potongan mencakup seluruh larik, dengan rentang setengah terbuka `[lo, mid)` dan `[mid, hi)`, sehingga hitungan yang ditambahkan adalah `mid - i`, bentuk Python dari Persamaan (8.2).

Kode 8.5 `invcnt.py`: merge sort bottom-up yang menghitung inversi, tanpa rekursi.

```python
# INVCNT (SPOJ) dalam Python: merge sort bottom-up yang menghitung inversi.
# Tanpa rekursi, jadi tanpa batas rekursi; sekitar 18 putaran untuk n = 200.000.
import sys


def count_inversions(a):
    n, width, count = len(a), 1, 0
    src, dst = a[:], [0] * len(a)
    while width < n:
        for lo in range(0, n, 2 * width):
            mid, hi = min(lo + width, n), min(lo + 2 * width, n)
            i, j, k = lo, mid, lo
            while i < mid and j < hi:
                if src[i] <= src[j]:
                    dst[k] = src[i]
                    i += 1
                else:
                    count += mid - i          # src[i:mid] semuanya dilompati src[j]
                    dst[k] = src[j]
                    j += 1
                k += 1
            dst[k:hi] = src[i:mid] if i < mid else src[j:hi]
        src, dst = dst, src
        width *= 2
    return count


def main():
    data = sys.stdin.buffer.read().split()
    t, pos, out = int(data[0]), 1, []
    for _ in range(t):
        n = int(data[pos])
        a = list(map(int, data[pos + 1:pos + 1 + n]))
        pos += 1 + n
        out.append(str(count_inversions(a)))
    print("\n".join(out))


if __name__ == "__main__":
    main()
```

Contoh kami memuat dua kasus uji, larik di Subbab 8.4.1 dan larik enam nilai dalam urutan terbalik, yang harus memberi $\binom{6}{2} = 15$:

```
2

4
7
2
9
4

6
60
50
40
30
20
10
```

```
3
15
```

Brute force dan program Python mencetak dua bilangan yang sama.

8.5 Memburu Jawaban yang Salah

8.5.1 Program dengan dua kekeliruan

Program yang ditulis tergesa-gesa jarang tampak seperti **Kode 8.4** pada kali pertama. Berkas `invcnt_bug.cpp` di folder bab ini adalah program yang sama dengan dua kekeliruan yang sudah sering kami lihat dibuat peserta: penghitung dan nilai kembalian fungsinya bertipe `int`, dan hitungan yang ditambahkan di setiap langkah adalah `mid - i` alih-alih `mid - i + 1`, bentuk setengah terbuka dari Persamaan (8.2) yang keliru dipakai pada rentang tertutup. Perbedaannya dengan **Kode 8.4**, sebagaimana diperlihatkan `diff`:

```
<     count += mid - i + 1;       // a[i..mid] semuanya > a[j]
---
>     count += mid - i;           // keliru: + 1 terlupa

<     long long sortCount(...)    // dan: long long count = ...
---
>     int sortCount(...)          // dan: int count = ...
```

Pada contoh, program ini mencetak `0` dan `8` alih-alih 3 dan 15. Di juri sungguhan Anda hanya akan melihat *Wrong answer*.

8.5.2 Menemukan dan memperkecilnya

Uji stres menemukan perbedaannya pada benih pertama:

```
python3 stress.py "python3 gen_invcnt.py" build/invcnt_brute build/invcnt_bug 300
```

```
MISMATCH on seed 1
input:
1

5
3
9
4
16
13
expected: 2
received: 0
```

Larik 3,9,4,16,13 sudah kecil, tetapi kita bisa lebih baik, dan masukan yang lebih kecil membuat penyebabnya lebih mudah dilihat. Argumen pertama generator membatasi ukurannya; dengan larik berisi paling banyak 2 elemen,

```
python3 stress.py "python3 gen_invcnt.py 2" build/invcnt_brute build/invcnt_bug 300
```

```
MISMATCH on seed 6
input:
3

1
16

2
2
1

1
19
expected: 0 1 0
received: 0 0 0
```

kasus uji kedua, larik 2,1, memiliki satu inversi, dan program melaporkan tidak ada. Sekarang telusuri dengan tangan: paruhnya [2] dan [1], jadi lo = mid = 0 dan hi = 1; penggabungan membandingkan 2 dengan 1, mengambil 1 dari paruh kanan, dan menambahkan `mid - i` $= 0 - 0 = 0$. Bagian kiri yang menunggu, $L[0..0]$, berisi satu elemen, bukan nol: rumusnya selisih satu. Dengan $+1$ dikembalikan, uji stres berjalan bersih:

```
OK: 300 random tests agree
```

Inilah seluruh metodenya dalam bentuk mini: kegagalan acak, diperkecil sampai bentuk terkecil, ditelusuri dengan tangan. Memperkecil dengan membatasi ukuran adalah strategi paling sederhana; strategi lain adalah menghapus elemen dari masukan gagal satu per satu, mempertahankan setiap penghapusan yang masih gagal, sampai tidak ada penghapusan tunggal yang mempertahankan kegagalan. Sebagian perkakas pengujian melakukannya secara otomatis; dengan tangan, jarang lebih dari beberapa menit.

8.5.3 Bug yang tak terlihat oleh uji kecil

Program yang sudah diperbaiki, `invcnt_int.cpp`, masih memiliki penghitung `int`, dan ia lolos ke-300 uji acak: larik 6 elemen paling banyak memiliki 15 inversi. Di sinilah mode kedua uji stres diperlukan: masukan terbesar, dibandingkan dengan jawaban yang dapat kita hitung secara independen. Persamaan (8.1) memberi jawaban persis untuk larik terbalik. `gen_invcnt_max.py` menulis larik terbalik berisi 200.000 elemen dan satu larik acak:

```
reversed, invcnt.cpp     : 19999900000
reversed, invcnt_int.cpp : -1474936480
random,   invcnt.cpp     : 10009896365
random,   invcnt_int.cpp : 1419961773
```

Larik terbalik langsung memperlihatkan luapannya: 19 999 900 000 adalah apa yang dikatakan Persamaan (8.1), dan banyaknya pasangan yang negatif mustahil. Larik acak adalah kasus yang lebih berbahaya: versi `int` mencetak bilangan positif yang masuk akal tetapi salah. Tanpa pemeriksaan independen Anda mungkin mempercayainya. Untuk larik acak, pemeriksanya adalah program yang benar itu sendiri, yang kini dipercaya karena cocok dengan brute force pada masukan kecil dan dengan Persamaan (8.1) pada masukan ekstrem.

8.6 Memburu Batas Waktu

8.6.1 Ukur, jangan menebak

Batas waktu adalah pernyataan tentang masukan terbesar, jadi program harus diukur waktunya pada masukan terbesar. **Kode 8.6** adalah stopwatch kecil yang menjalankan perintah apa pun pada sebuah berkas masukan dan melaporkan waktu nyata dan, di Linux dan macOS, memori puncaknya.

Kode 8.6 `measure.py`: menjalankan perintah pada berkas masukan dan melaporkan waktu serta memori puncaknya.

```
# Jalankan program pada berkas masukan, laporkan waktu nyata dan memori puncak.
# Pemakaian: python3 measure.py BERKAS_MASUKAN PERINTAH...
# Memori puncak butuh modul resource (Linux, macOS); di Windows dilewati.
import subprocess
import sys
import time

try:
    import resource
except ImportError:
    resource = None


def main():
    path, cmd = sys.argv[1], sys.argv[2:]
    with open(path, "rb") as f:
        start = time.perf_counter()
        r = subprocess.run(cmd, stdin=f, stdout=subprocess.DEVNULL)
        elapsed = time.perf_counter() - start
    line = "%-28s %7.3f s" % (" ".join(cmd)[-28:], elapsed)
    if resource is not None:
        peak = resource.getrusage(resource.RUSAGE_CHILDREN).ru_maxrss
        if sys.platform == "darwin":
            peak //= 1024                    # macOS melaporkan byte, Linux KiB
        line += "  %8.1f MiB" % (peak / 1024)
    if r.returncode != 0:
        line += "  exit code %d" % r.returncode
    print(line)


if __name__ == "__main__":
    main()
```

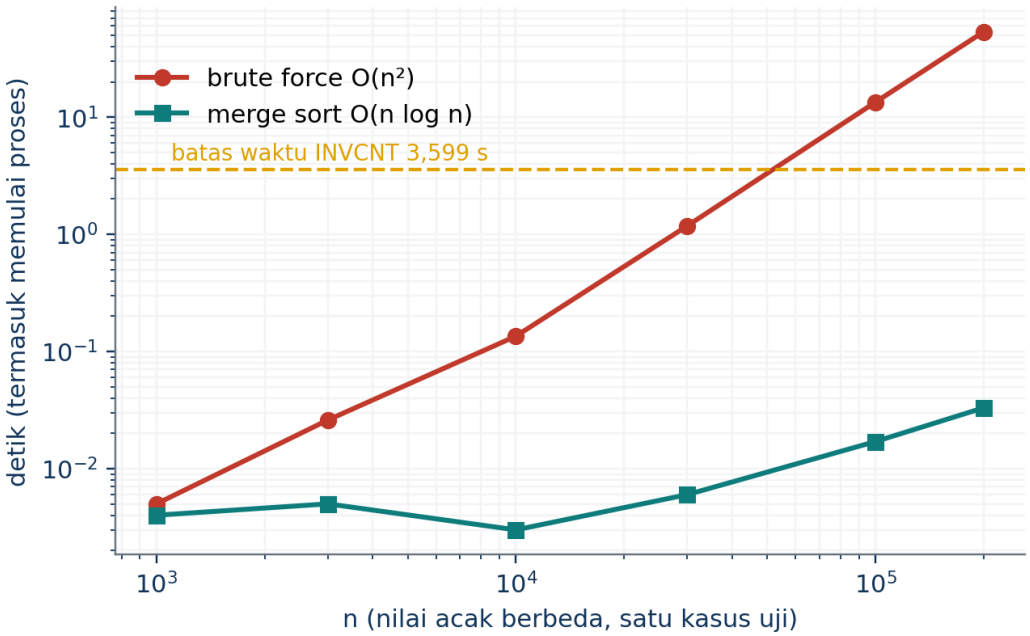
Memori puncak diperoleh dari `resource.getrusage`, yang melaporkan ukuran residen terbesar dari proses-proses anak yang telah selesai; modul ini tidak ada di Windows, dan di sana kolom itu dihilangkan saja. Selalu kompilasi program dengan flag juri (Bab 5) sebelum mengukur waktunya.

8.6.2 Bagaimana waktu tumbuh?

Satu pengukuran memberi tahu apakah program cepat pada satu masukan. Beberapa pengukuran pada masukan yang membesar memberi tahu *bagaimana* waktunya tumbuh, dan karena itu bagaimana kinerjanya pada masukan yang tidak dapat Anda jalankan. `timing_invcnt.py` menjalankan kedua program INVCNT pada larik acak yang ukurannya membesar, tiga kali masing-masing, dan mengambil eksekusi tercepat. **Tabel 8.2** dan **Gambar 8.1** menunjukkan hasilnya.

Tabel 8.2 Waktu kedua program INVCNT pada larik acak yang membesar (tercepat dari tiga kali jalan, September 2026).

n	Brute force (s)	Merge sort (s)
1000	0.005	0.004
3000	0.026	0.005
10000	0.135	0.003
30000	1.182	0.006
100000	13.406	0.017
200000	53.467	0.033



Gambar 8.1 Brute force dan program merge sort pada larik acak, dengan skala logaritmik; garis putus-putus adalah batas waktu INVCNT.

Angka-angka untuk n kecil didominasi oleh waktu memulai proses, beberapa milidetik yang tidak ada hubungannya dengan algoritma. Mulai 10.000, brute force memperlihatkan wataknya. Jika waktu sebuah program tumbuh seperti n^k, mengalikan n dengan faktor r mengalikan waktunya dengan r^k, sehingga eksponennya dapat ditaksir dari dua pengukuran:

$$k \approx \frac{\log(t(n_2)/t(n_1))}{\log(n_2/n_1)} \tag{8.4}$$

Dari 30.000 ke 100.000, $k \approx \log(13.406/1.182)/\log(100\,000/30\,000) = \log 11.34/\log 3.33 \approx 2.02$: kuadratik, sebagaimana dijanjikan dua loop bersarangnya. Taksiran itu meramalkan sekitar $13.4 \times 2^2 \approx 54$ detik untuk $n = 200\,000$, dan pengukuran memberi 53,467 detik, lima belas kali batas INVCNT. Program merge sort tumbuh begitu lambat sehingga waktu memulai proses masih mendominasi di 100.000; di 200.000 ia memerlukan 0,033 detik. Pada grafik logaritmik seperti **Gambar 8.1**, hukum pangkat tampak sebagai garis lurus yang kemiringannya k, sehingga perbedaannya terlihat sekilas.

Kegunaan praktis Persamaan (8.4) adalah ramalan. Bila program memerlukan 0,3 detik pada sepersepuluh ukuran maksimum dan tumbuh secara kuadratik, ia akan memerlukan sekitar 30 detik pada ukuran maksimum: Anda sudah tahu vonisnya sebelum mengirim.

8.6.3 Python pada masukan terbesar

Program Python di **Kode 8.5** memerlukan 0,33 detik pada satu larik acak berisi 200.000 elemen. INVCNT tidak menyebutkan banyaknya kasus uji; jika berkas juri memuat beberapa larik berukuran maksimum, Python tidak akan lolos dalam 3,6 detik, sedangkan program C++ memerlukan 0,033 detik per larik. Inilah posisi khas Python pada soal SPOJ dengan n dalam ratusan ribu dan kerja $\Theta(n \log n)$ yang ditulis sebagai loop Python: benar, dan bergantung pada belas kasihan banyaknya kasus uji.

8.7 PALIN: Jawaban Salah dan Langkah Kuadratik Tersembunyi

8.7.1 Soalnya

Soal SPOJ PALIN, "The Next Palindrome" [5], memberi t bilangan bulat positif K sampai 1.000.000 digit, tanpa nol di depan, dan meminta untuk masing-masing *palindrom* terkecil yang benar-benar lebih besar dari K: bilangan yang dibaca sama dari depan maupun dari belakang, seperti 7, 11, 818, atau 12321. Halaman soal menyebutkan batas waktu sebagai rentang dari 2 sampai 9 detik, dan memperingatkan bahwa masukan dan keluarannya besar.

Sejuta digit menutup kemungkinan aritmetika pada bilangan: masukannya string, dan jawabannya harus dibangun sebagai string dalam waktu linear.

8.7.2 Gagasannya

Misalkan K memiliki n digit. Dua kasus bersifat khusus atau mudah.

- **Semua sembilan.** Jika $K = 99 \ldots 9$, tidak ada palindrom n digit yang lebih besar, dan jawabannya $10 \ldots 01$ dengan $n + 1$ digit.
- **Selain itu**, jawabannya memiliki n digit. Palindrom n digit ditentukan oleh *paruh kiri*-nya, yaitu $\lceil n/2 \rceil$ digit pertamanya, dan dua palindrom yang sama panjang dibandingkan persis seperti paruh kiri mereka dibandingkan.

Misalkan M adalah palindrom yang diperoleh dengan *mencerminkan* paruh kiri K ke paruh kanannya. Setiap palindrom dengan paruh kiri yang lebih kecil lebih kecil daripada K (ia sudah lebih kecil di paruh kirinya). Jadi jika $M > K$, maka M jawabannya. Jika $M \leq K$, jawabannya harus memiliki paruh kiri yang lebih besar, dan paruh kiri lebih besar yang terkecil adalah paruh kiri itu ditambah 1, lalu dicerminkan lagi. Menambahkan 1 pada paruh kiri sama dengan menambahkan 1 pada sebuah bilangan: mulai dari digit tengah; selama digitnya 9, jadikan 0 dan geser satu digit ke kiri; lalu naikkan digit itu satu. Paruh kiri tidak mungkin seluruhnya sembilan di sini (jika demikian, M semuanya sembilan, dan $M \leq K$ akan memaksa K juga semuanya sembilan), jadi simpanannya selalu berhenti.

Karena kedua string memiliki n digit, membandingkan M dan K sebagai string, karakter demi karakter, sama dengan membandingkannya sebagai bilangan. Setiap langkah menyentuh setiap digit sebanyak tetapan kali, sehingga seluruh algoritma linear dalam n.

Dua contoh yang dikerjakan. Untuk $K = 12921$ paruh kirinya 129, dan pencerminan memberi $M = 12921 = K$, tidak lebih besar. Naikkan paruh kirinya: digit tengah 9 menjadi 0 dengan simpanan, 2 menjadi 3, sehingga 130, dan pencerminan memberi 13031. Untuk $K = 808$, $M = 808 = K$; digit tengah 0 menjadi 1, dan jawabannya 818.

8.7.3 Program-programnya

Kode 8.7 `palin.cpp`: palindrom berikutnya dalam waktu linear pada string digit.

```cpp
// PALIN (SPOJ): palindrom terkecil yang benar-benar lebih besar dari K,
// dengan K sampai 1.000.000 digit. Waktu linear pada string digit.
#include <iostream>
#include <string>

std::string nextPalindrome(std::string s) {
    int n = (int)s.size();
    if (s.find_first_not_of('9') == std::string::npos)   // 9...9 -> 10...01
        return "1" + std::string(n - 1, '0') + "1";
    std::string m = s;
    for (int i = 0; i < n / 2; i++) m[n - 1 - i] = m[i];  // cerminkan kiri ke kanan
    if (m > s) return m;               // panjang sama: urutan string = urutan angka
    int i = (n - 1) / 2;               // naikkan paruh kiri satu, mulai dari tengah
    while (m[i] == '9') { m[i] = '0'; i--; }             // simpanan pindah ke kiri
    m[i]++;
    for (int k = 0; k < n / 2; k++) m[n - 1 - k] = m[k];  // cerminkan lagi
    return m;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int t;
    if (!(std::cin >> t)) return 0;
    std::string k;
    while (t-- && std::cin >> k) std::cout << nextPalindrome(k) << '\n';
    return 0;
}
```

Kode 8.7 mengikuti gagasan itu baris demi baris. Versi Python, **Kode 8.8**, memiliki satu alasan tambahan untuk tetap memakai string: sejak Python 3.11, `int()` menolak mengonversi string lebih dari 4300 digit (Bab 5), sehingga mengubah K menjadi bilangan bulat bukan hanya lambat tetapi juga memunculkan galat pada uji-uji besar milik juri. Fungsi bantu `bump` menambahkan 1 pada string digit dengan mengganti sembilan-sembilan di ujungnya menjadi nol dan menaikkan digit sebelumnya.

Kode 8.8 `palin.py`: palindrom berikutnya dalam Python, hanya pada string.

```python
# PALIN (SPOJ) dalam Python, pada string: int() menolak mengonversi string
# lebih dari 4300 digit (Python 3.11 ke atas), dan toh akan lambat.
import sys


def next_palindrome(s):
    n = len(s)
    if s.count("9") == n:                                # 9...9 -> 10...01
        return "1" + "0" * (n - 1) + "1"
    left = s[:(n + 1) // 2]
    m = left + left[:n // 2][::-1]                       # cerminkan kiri ke kanan
    if m > s:
        return m
    left = bump(left)
    return left + left[:n // 2][::-1]


def bump(digits):                    # tambah 1 pada string digit yang tak semuanya 9
    body = digits.rstrip("9")
    nines = len(digits) - len(body)
    return body[:-1] + chr(ord(body[-1]) + 1) + "0" * nines


def main():
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    out = [next_palindrome(x.decode()) for x in data[1:1 + t]]
    sys.stdout.write("\n".join(out) + "\n")


if __name__ == "__main__":
    main()
```

Contoh kami memuat kedua contoh yang dikerjakan dan tiga kasus kecil: 7 (palindrom berikutnya 8), 9 (semua sembilan: 11), dan 199 (pencerminan memberi 191, yang lebih kecil, dan menaikkan paruh kiri 19 memberi 20 dan jawaban 202).

```
5
7
9
808
12921
199
```

```
8
11
818
13031
202
```

Brute force-nya, **Kode 8.9**, melakukan apa yang dikatakan pernyataan soal tanpa gagasan apa pun: hitung naik dari $K + 1$ sampai bilangannya terbaca sama dari belakang. Ia tidak ada harapan untuk sejuta digit dan seketika untuk lima digit.

Kode 8.9 `brute_palin.py`: brute force PALIN menghitung naik sampai bertemu palindrom.

```python
# Brute force untuk PALIN: hitung naik sampai muncul palindrom.
import sys

data = sys.stdin.read().split()
for x in data[1:1 + int(data[0])]:
    k = int(x) + 1
    while str(k) != str(k)[::-1]:
        k += 1
    print(k)
```

Generator `gen_palin.py` menulis bilangan 1 sampai 5 digit, mengambil setiap digit dari sebuah kumpulan yang diberi tambahan sembilan dan nol, dan mengubah 30% di antaranya menjadi palindrom, karena kasus-kasus yang menarik justru "K sudah palindrom" dan "simpanan melewati beberapa sembilan".

8.7.4 Jawaban salah: simpanan yang hilang

`palin_bug.cpp` berbeda dari **Kode 8.7** di satu baris: ia menaikkan digit tengah tanpa loop simpanan, seolah-olah digit tengah tidak mungkin 9.

```
<     while (m[i] == '9') { m[i] = '0'; i--; }            // simpanan ke kiri
<     m[i]++;
---
>     m[i]++;                              // keliru: tanpa simpanan saat m[i] '9'
```

Pada contoh ia mencetak `8 11 818 12:21 1:1`, dan uji stres menemukannya pada benih pertama:

```
python3 stress.py "python3 gen_palin.py" "python3 brute_palin.py" build/palin_bug
```

```
MISMATCH on seed 1
input:
2
88999
2289
expected: 89098 2332
received: 88:88 2332
```

Karakter sesudah '9' dalam tabel karakter adalah ':', sehingga "naikkan digit 9" menghasilkan titik dua. Kekeliruan yang sama dalam bahasa dengan digit numerik akan menghasilkan angka 10 di satu posisi, atau 0 yang diam-diam salah; titik dua itu gejala yang kebetulan lantang. Dengan loop simpanan dikembalikan, uji stres melaporkan `OK: 300 random tests agree` untuk `palin.cpp` maupun `palin.py`.

8.7.5 Batas waktu yang tak terlihat oleh uji acak

Varian ketiga, `palin_slow.cpp`, benar: ia lolos setiap uji stres. Bedanya terletak pada cara ia membangun paruh kanan yang dicerminkan setelah paruh kiri dinaikkan: alih-alih menyalin digit ke tempatnya, ia membangun string baru dengan menyisipkan setiap digit di *depan*.

```
<     for (int k = 0; k < n / 2; k++) m[n - 1 - k] = m[k];  // cerminkan lagi
<     return m;
---
>     std::string left = m.substr(0, (n + 1) / 2), right;
>     for (int k = 0; k < n / 2; k++) right.insert(0, 1, left[k]);  // lambat
>     return left + right;
```

Setiap penyisipan di depan string menggeser setiap karakter yang sudah ada satu tempat ke kanan, sehingga membangun paruh kanan berisi h karakter memerlukan $1 + 2 + \cdots + h \approx h^2/2$ perpindahan karakter. Untuk $n = 10^6$, $h = 500\,000$ dan itu sekitar 1.25×10^{11} perpindahan: langkah kuadratik yang tersembunyi di dalam algoritma linear.

Kami mengukur waktu program-program itu pada dua masukan sejuta digit. Yang pertama acak; yang kedua, `7` diikuti 499.999 angka tiga dan 500.000 angka delapan, dibangun agar pencerminan memberi bilangan yang lebih kecil sehingga jalur lambat harus dijalankan. **Tabel 8.3** menunjukkan hasilnya.

Tabel 8.3 Waktu tiga program PALIN pada masukan sejuta digit.

Masukan (10^6 digit)	palin.cpp	palin.py	palin_slow.cpp
digit acak	0.004 s	0.021 s	0.011 s
7, lalu angka 3, lalu angka 8 (memaksa kenaikan)	0.005 s	0.014 s	2.471 s

Pada masukan acak, program lambat itu secepat yang lain, karena untuk kebanyakan bilangan acak cerminannya sudah lebih besar dan cabang lambat tidak pernah dicapai. Hanya masukan yang *dirancang* untuk memaksa kenaikan yang membongkarnya. Pelajaran ini sama pentingnya dengan uji stres itu sendiri: **kasus terburuk harus dibangun, bukan diharapkan muncul.** Untuk setiap kandidat, tanyakan masukan mana yang membuat setiap loop berjalan paling lama, dan tulislah generator untuk masukan itu. Untuk PALIN jawabannya "pencerminan memberi bilangan yang lebih kecil"; untuk pengurutan sering "sudah terurut" atau "semuanya sama"; untuk tabel hash, "semua kunci bertabrakan".

Pengukuran itu juga memperlihatkan pertumbuhan kuadratik: pada masukan padanannya yang 100.000 digit, program lambat yang sama memerlukan 0,011 detik, sebagian besar untuk memulai proses, sehingga digit yang sepuluh kali lebih banyak memakan waktu lebih dari dua ratus kali lipat; dengan h^2 langkah, bagian algoritmanya saja tumbuh seratus kali lipat.

8.8 Perkakas Lain untuk Runtime Error

8.8.1 Berhenti di baris yang keliru

Runtime error di juri, atau jawaban salah yang bergantung pada sampah di memori, biasanya berasal dari operasi baca atau tulis di luar larik. C++ secara bawaan tidak memeriksa indeks, sehingga program seperti itu bisa berjalan dan mencetak bilangan yang masuk akal. **Kode 8.10** adalah contoh empat baris: loop jumlah awalan yang membaca `a[i]` padahal seharusnya `a[i - 1]`, sehingga pada langkah terakhirnya membaca satu elemen melewati ujung.

Kode 8.10 `bounds_demo.cpp`: loop jumlah awalan yang membaca satu elemen melewati ujung vector.

```cpp
// Loop jumlah awalan yang membaca satu elemen melewati ujung vector.
// -O2 biasa mencetak sebuah angka; -D_GLIBCXX_DEBUG atau -fsanitize=address
// program tepat pada baris yang salah.
#include <cstdio>
#include <vector>

int main() {
    std::vector<int> a = {4, 1, 3, 2, 5};
    std::vector<long long> prefix(a.size() + 1, 0);
    for (std::size_t i = 1; i <= a.size(); i++)
        prefix[i] = prefix[i - 1] + a[i];   // keliru: seharusnya a[i - 1]
    std::printf("sum of all = %lld\n", prefix[a.size()]);
    return 0;
}
```

Dikompilasi biasa dengan `-O2`, program mencetak jumlah yang salah tanpa peringatan apa pun; jumlah 4, 1, 3, 2, 5 adalah 15.

```
sum of all = 11
```

Dikompilasi dengan `-D_GLIBCXX_DEBUG`, pustaka standar g++ memeriksa setiap indeks [6], dan program berhenti pada akses buruk pertama dengan pesan yang menyebutkan masalahnya (laporannya dipersingkat di sini):

```
/usr/include/c++/13/debug/vector:442:
Error: attempt to subscript container with out-of-bounds index 5, but
container only holds 5 elements.
```

Dikompilasi dengan `-fsanitize=address` dan `-g`, address sanitizer [7] melaporkan baris sumber yang tepat:

```
ERROR: AddressSanitizer: heap-buffer-overflow on address 0x503000000054
READ of size 4 at 0x503000000054 thread T0
    #0 0x562df89a2a2f in main bounds_demo.cpp:11
```

Baris 11 adalah badan loop itu. Kedua perkakas mengubah bug yang tak terlihat menjadi program yang berhenti dengan alamat yang jelas; jalankan uji stres Anda sekali dengan masing-masing perkakas, seperti anjuran Bab 5.

8.8.2 Asersi

Asersi menyatakan sesuatu yang harus benar di satu titik program: `assert(0 <= i && i < n);` atau `assert(total % n == 0);`. Jika salah, program berhenti dan mencetak berkas, baris, dan syaratnya. Asersi hampir tidak memakan biaya, mendokumentasikan apa yang Anda yakini, dan mengubah jawaban salah yang senyap menjadi runtime error yang lantang selama uji stres. Asersi aktif kecuali program dikompilasi dengan `-DNDEBUG`, jadi biarkan tetap ada selama pengujian; di juri, asersi yang gagal menjadi runtime error, yang memberi tahu Anda lebih banyak daripada jawaban salah.

8.8.3 Cetak ke aliran galat

Ketika Anda menambahkan pencetakan untuk memahami masukan yang gagal, cetaklah ke `stderr` (`std::cerr` atau `fprintf(stderr, ...)` di C++, `print(..., file=sys.stderr)` di Python). Juri dan pengikat uji hanya membandingkan keluaran standar, sehingga baris-baris debug di `stderr` tidak mengubah vonis bila Anda lupa menghapus satu, walaupun tetap memakan waktu pada masukan besar.

8.8.4 Membelah berkas berisi banyak kasus uji

Kadang Anda memang memiliki berkas yang gagal, berkas besar dengan ratusan kasus uji, misalnya dari generator seorang teman atau dari uji yang dipublikasikan setelah lomba, dan keluaran program berbeda dari keluaran yang diharapkan di suatu tempat. Jika bug-nya kebocoran antarkasus uji (Bab 7), setiap kasus setelah suatu titik mungkin salah, dan kasus salah yang pertama itulah yang menarik: di situlah keadaan yang bocor pertama kali berpengaruh. Memeriksa kasus satu per satu memang berhasil, tetapi lambat bila setiap pemeriksaan memerlukan pengamatan manual.

Kegagalan itu memiliki sifat yang berguna: begitu kebocoran merusak keadaan, keadaan itu tetap rusak. Jadi predikat "k kasus uji pertama semuanya dijawab benar" bernilai benar sampai suatu k^* dan salah sesudahnya, dan k^* dapat ditemukan dengan *binary search*: jalankan program pada k kasus pertama untuk k di tengah rentang yang belum diketahui, bandingkan, dan simpan paruh yang memuat perubahan. Dengan 500 kasus uji, sekitar $\log_2 500 \approx 9$ kali eksekusi menemukan kasus gagal pertama, alih-alih sampai 500 kali. Pembelahan yang sama berlaku pada *masukan* satu kasus uji besar selama kegagalannya bertahan begitu muncul, dan itulah metode Bu Endang di Subbab 8.1 yang diterapkan pada sebuah berkas. Gagasan ini juga yang mendasari `git bisect`, yang menemukan commit yang memasukkan sebuah bug dengan membelah riwayatnya.

Pembelahan memerlukan kegagalan yang monoton. Jika kasus uji 17 gagal karena isinya sendiri dan kasus uji 18 lolos, predikatnya tidak monoton, dan penelusuran linear, atau sekadar membandingkan keluaran baris demi baris dengan `diff`, adalah perkakas yang tepat.

8.8.5 Bila jawabannya tidak tunggal

Sebagian soal menerima salah satu dari beberapa jawaban yang benar: "cetak lintasan terpendek mana saja", "cetak susunan sah mana saja". Membandingkan dengan keluaran brute force lalu gagal bahkan untuk program yang benar. Perbaikannya adalah *pemeriksa* (*checker*): program yang membaca masukan dan keluaran kandidat lalu memverifikasi sifat-sifat jawabannya, misalnya bahwa lintasan yang dicetak memakai sisi yang benar-benar ada, dimulai dan diakhiri di simpul yang tepat, dan panjangnya sama dengan yang ditemukan brute force. Pengikat ujinya tetap sama; hanya perbandingannya yang diganti dengan pemanggilan pemeriksa.

8.9 Menggunakannya Sungguhan

Rutinitas bab ini muat dalam selembar kartu indeks.

Pada wrong answer:

1. Tulis brute force dari pernyataan soal, tanpa kecerdikan, dan generator untuk masukan kecil yang sah yang membidik bentuk-bentuk berbahaya.
2. Jalankan uji stres. Bila gagal, perkecil masukannya dan telusuri kasus terkecil dengan tangan.
3. Bila lolos, tambahkan masukan terbesar dan bandingkan dengan jawaban independen: sebuah rumus, metode cepat kedua, atau kasus ekstrem yang jawabannya Anda ketahui.

Pada batas waktu:

1. Bangun masukan terburuk yang diizinkan batasan, bukan hanya masukan acak; tanyakan masukan mana yang membuat setiap loop berjalan paling lama.
2. Ukur waktu program dengan flag juri pada ukuran yang membesar dan taksir eksponennya dengan Persamaan (8.4).
3. Jika eksponennya terlalu besar, ubah algoritmanya; jika eksponennya benar tetapi tetapannya terlalu besar, cari langkah kuadratik tersembunyi (penyisipan di depan, `erase` di dalam loop, penyambungan string di dalam loop, pengosongan larik besar per kasus uji) sebelum meraih flag-flag di Bab 5.

Pada runtime error: kompilasi dengan `-D_GLIBCXX_DEBUG` dan dengan `-fsanitize=address,undefined` lalu jalankan ulang uji stres; periksa kedalaman rekursi seperti di Bab 6.

Sepatah kata tentang waktu. Seluruh rutinitas ini terdengar panjang, tetapi dalam praktik brute force untuk soal SPOJ yang khas hanya sepuluh baris, generatornya sepuluh baris lagi, dan pengikat ujinya cukup ditulis sekali untuk seumur hidup. Lima belas menit yang dipakai untuk semua itu biasanya menghemat lebih dari lima belas menit memandangi kode, dan dalam lomba ia menghemat sesuatu yang lebih berharga: keyakinan untuk mengirim, atau kepastian bahwa Anda belum boleh mengirim. Dengan pengalaman, Anda akan mulai menulis brute force *lebih dahulu*, sebelum solusi yang cepat, karena brute force juga memeriksa pemahaman Anda atas pernyataan soal pada contoh-contohnya. Jika brute force Anda tidak mencetak keluaran contoh, masalahnya bukan pada algoritma, melainkan pada cara Anda membaca soal, dan itu jauh lebih baik diketahui sebelum menulis solusi yang rumit.

Simpan ketiga program itu di samping solusi Anda, dalam folder `tests` seperti di buku ini, dan biaya memakainya kembali hampir nol. Kebiasaan yang sama berguna jauh dari lomba. Dua hitungan Bu Endang adalah uji stres atas buku catatannya; koperasi yang memindahkan pembukuannya dari buku besar kertas ke spreadsheet dapat mempertahankan keduanya selama beberapa bulan dan membandingkan total bulanannya, memperkecil setiap selisih sampai ke minggu, hari, dan entrinya; dan siapa pun yang menulis rumus di spreadsheet dapat memeriksanya terhadap hitungan tangan pada tiga atau empat baris sederhana sebelum mempercayainya pada tiga ribu baris.

8.10 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

- **Brute force yang berbagi gagasan dengan kandidat.** Jika keduanya memakai rumus yang sama, keduanya sepakat dalam kesalahan rumus itu. Brute force sebaiknya mengikuti pernyataan soal, bukan solusinya.
- **Generator yang menulis masukan tidak sah.** Perbedaan pada masukan yang dilarang soal bukanlah bug. Periksa generator terhadap batasan: nilai berbeda di INVCNT, tanpa nol di depan di PALIN, tanpa nol di belakang di ADDREV.
- **Hanya uji acak kecil.** Uji itu tidak dapat melihat luapan atau kelambatan. Tambahkan masukan terbesar dan kasus terburuk yang dibangun.
- **Mengukur waktu build tanpa optimasi, atau hanya masukan acak.** Ukur dengan `-O2` pada masukan terburuk.
- **Tidak menyimpan benih.** Kegagalan yang tidak dapat Anda ulang adalah kegagalan yang tidak dapat Anda perbaiki. Generator harus deterministik untuk benih tertentu.
- **Men-debug masukan gagal yang besar.** Perkecil dulu. Kegagalan dua elemen menjelaskan dirinya sendiri; kegagalan dua ratus elemen jarang.
- **Berhenti setelah perbaikan pertama.** Setelah memperbaiki satu bug, jalankan ulang seluruh uji stres, lalu masukan terbesar: perbaikan pertama bisa menyingkap bug berikutnya, seperti yang terjadi pada `invcnt_int.cpp`.

8.11 Ringkasan Bab

- Juri hanya memperlihatkan vonis, jadi bug harus ditemukan di mesin Anda sendiri dengan metode kedua: brute force yang jelas benar, dibandingkan dengan kandidat pada banyak masukan yang dibangkitkan (**Kode 8.1**).
- Kebanyakan bug muncul pada masukan kecil (hipotesis cakupan kecil); memperkecil masukan gagal sampai bentuk terkecilnya membuat penyebabnya tampak, seperti larik dua elemen yang membongkar selisih satu pada Persamaan (8.2).
- Bug yang berkaitan dengan ukuran memerlukan masukan lain: larik terbesar melawan jawaban independen seperti Persamaan (8.1) menangkap luapan; kasus terburuk yang dibangun menangkap kelambatan tersembunyi yang terlewat oleh masukan acak (**Tabel 8.3**).
- Pengukuran waktu pada masukan yang membesar dan Persamaan (8.4) mengungkap bagaimana waktu tumbuh dan meramalkan vonis pada ukuran yang tidak dapat Anda jalankan; brute force INVCNT kuadratik, merge sort $\Theta(n \log n)$ (Persamaan (8.3)).
- `-D_GLIBCXX_DEBUG`, address sanitizer, dan asersi menghentikan program di baris tempat ia keliru.

- INVCNT diselesaikan dengan menghitung inversi silang selama penggabungan; PALIN dengan mencerminkan paruh kiri dan, bila perlu, menaikkannya dengan simpanan, dalam waktu linear pada string digit.

8.12 Soal Latihan

1. Hitung inversi 5,1,4,2,3 dengan tangan, pertama dengan mendaftar pasangannya lalu dengan mengikuti penggabungan di **Kode 8.4** dan menambahkan Persamaan (8.2) di setiap langkah. Periksa dengan program.
2. Berkas `invcnt_bug.cpp` memiliki dua kekeliruan. Andaikan hanya kekeliruan `int` yang ada. Berapa n terkecil yang membuat larik terbalik menyebabkan program mencetak jawaban salah? Berikan nilai yang dicetak.
3. Tuliskan masukan gagal sekecil mungkin untuk `palin_bug.cpp`, dan jelaskan mengapa kumpulan digit generator "0999912345678" menemukan masukan seperti itu lebih cepat daripada digit yang seragam.
4. Sebuah kandidat untuk PALIN memberi jawaban benar tetapi, untuk masukan dengan banyak digit genap, mencerminkan paruh kiri dengan indeks yang selisih satu. Bilangan mana di antara lima bilangan contoh yang akan membongkar kekeliruan itu, dan mana yang tidak?
5. Tulis generator `gen_palin_worst.py` yang menulis satu bilangan n digit yang memaksa `palin_slow.cpp` mengambil jalur lambatnya, untuk setiap $n \geq 2$. Jelaskan mengapa bilangan Anda memaksa kenaikan.
6. Ubah `stress.py` agar, setelah menemukan masukan gagal untuk soal dengan satu larik per kasus uji, ia mencoba menghapus elemen satu per satu dan mempertahankan setiap penghapusan yang masih gagal. Apa yang dijamin prosedur ini tentang masukan yang dihasilkannya di akhir?

8.13 Penyelesaian Soal Latihan

1. Pasangan yang urutannya salah adalah (5, 1), (5, 4), (5, 2), (5, 3), (4, 2), (4, 3): 6 inversi. Dengan penggabungan: larik dibelah menjadi [5,1,4] dan [2,3] (lo = 0, mid = 2, hi = 4). Bagian kiri dibelah menjadi [5,1] dan [4]; [5,1] digabung menjadi [1,5] dengan menambahkan $0 - 0 + 1 = 1$ ketika 1 diambil. Menggabungkan [1,5] dengan [4]: 1 diambil dari kiri, lalu 4 lebih kecil daripada 5, menambahkan $1 - 1 + 1 = 1$; total berjalan 2. [2,3] tidak menambah apa-apa. Penggabungan terakhir [1,4,5] dengan [2,3]: 1 dari kiri; 2 lebih kecil daripada 4, menambahkan $2 - 1 + 1 = 2$; 3 lebih kecil daripada 4, menambahkan 2 lagi; total $2 + 2 + 2 = 6$. Program mencetak 6.

2. Larik terbalik memiliki $n(n-1)/2$ inversi menurut Persamaan (8.1); `int` meluap bila nilai ini melebihi $2^{31} - 1 = 2\,147\,483\,647$. $n(n-1)/2 > 2\,147\,483\,647$ pertama kali berlaku untuk $n = 65\,537$, dengan hitungan $2\,147\,516\,416$; nilai yang berputar adalah $2\,147\,516\,416 - 2^{32} = -2\,147\,450\,880$. (Secara ketat, luapan itu perilaku tak terdefinisi; inilah nilai yang dihasilkannya di kompiler-kompiler yang umum.)

3. $K = 9$ semuanya sembilan dan ditangani terpisah, dan bilangan satu digit di bawah 9 tidak pernah menimbulkan simpanan. Masukan gagal terkecil adalah $K = 191$: pencerminan memberi 191, tidak lebih besar; digit tengah 9 dinaikkan tanpa simpanan menjadi ':', dan program mencetak `1:1` alih-alih 202. (99 semuanya sembilan; untuk dua digit, $K = 19$ dicerminkan menjadi 11 lalu 1 dinaikkan menjadi 2, memberi 22, dengan benar.) Kumpulan itu memuat digit 9 empat kali dari tiga belas, sehingga digit tengah acak bernilai 9 dengan peluang sekitar 0,31 alih-alih 0,1, dan bentuk-bentuk yang gagal muncul tiga kali lebih sering.

4. Hanya bilangan dengan banyak digit genap yang dapat membongkarnya: 12921, 808, 199, 7, dan 9 semuanya berpanjang ganjil, jadi tak satu pun dari kelimanya. Itulah sebabnya generator mengambil panjang dari 1 sampai 5: separuh bilangan acaknya berpanjang genap.

5. Tulis 7, lalu $\lceil n/2 \rceil - 1$ angka tiga, lalu $\lfloor n/2 \rfloor$ angka delapan. Paruh kirinya 733 … 3; pencerminan menaruh 3 … 37 di paruh kanan, yang digit pertamanya 3 lebih kecil daripada angka 8 di posisi yang sama pada K (untuk $n = 2$ paruh kirinya hanya 7, cerminannya 77 dan K bernilai 78). Jadi $M < K$, kenaikan terpaksa terjadi, dan penyisipan lambat berjalan atas seluruh paruh kanan. Digit pertamanya, 7, menjamin tidak ada nol di depan dan bahwa bilangan itu tidak semuanya sembilan.

6. Setelah uji stres gagal, ulangi: untuk setiap posisi p larik, bentuk masukan tanpa elemen p (sesuaikan n), jalankan brute force dan kandidat, dan jika keduanya masih berbeda, simpan masukan yang lebih kecil itu lalu mulai lagi; berhenti bila tidak ada penghapusan tunggal yang gagal. Hasilnya *1-minimal*: menghapus elemen mana pun membuat kegagalan hilang. Belum tentu itu masukan gagal terkecil secara keseluruhan, tetapi biasanya cukup kecil untuk ditelusuri dengan tangan, dan prosedurnya paling banyak memerlukan n^2 kali eksekusi.

Daftar Pustaka

[1] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, rev. ed. Cambridge, MA, USA: MIT Press, 2012.

[2] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 2nd ed. Cham, Switzerland: Springer, 2020.

[3] Sphere Online Judge, "INVCNT - Inversion Count." [Online]. Available: https://www.spoj.com/problems/INVCNT/ (accessed Sep. 2026).

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[5] Sphere Online Judge, "PALIN - The Next Palindrome." [Online]. Available: https://www.spoj.com/problems/PALIN/ (accessed Sep. 2026).

[6] Free Software Foundation, "Debug Mode: Using," *The GNU C++ Library Manual*. [Online]. Available: https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug_mode_using.html (accessed Sep. 2026).

[7] Free Software Foundation, "Program Instrumentation Options," *Using the GNU Compiler Collection (GCC)*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html (accessed Sep. 2026).

Bab 9

Pola Pemrograman Dinamis

Banyak soal SPOJ tampak mustahil pada pandangan pertama: cara memilih, membagi, berjalan, atau menggabungkan terlalu banyak untuk dicoba satu per satu oleh komputer mana pun. Namun, satu keluarga besar di antaranya takluk pada satu gagasan, yaitu *dynamic programming* (pemrograman dinamis): nyatakan jawaban untuk masukan besar melalui jawaban untuk masukan yang lebih kecil, hitung setiap jawaban kecil sekali saja, lalu simpan dalam sebuah tabel. Di akhir bab ini Anda akan mampu menemukan *state* (keadaan) sebuah soal, menuliskan rekurensinya beserta kasus dasarnya, memilih urutan pengisian tabel, lalu mengubah semuanya menjadi program yang ringkas dan cepat dalam C++17 dan Python 3. Anda akan mengerjakan enam soal SPOJ sungguhan, masing-masing mewakili pola yang terus muncul: deretan pilihan (FARIDA), menghitung banyak cara (ACODE), lintasan pada grid (BYTESM2), barang dengan anggaran terbatas (KNAPSACK), dua string (EDIST), dan interval (MIXTURES). Setiap program diperiksa terhadap *brute force* (pencarian menyeluruh) pada ratusan masukan acak dan diukur waktunya pada masukan terbesar yang diizinkan soal.

9.1 Hari Pasar Bu Wati

Bu Wati berjualan buah di pasar-pasar mingguan di sekitar desanya: rambutan dan manggis pada bulan-bulan hujan, mangga dan durian ketika musimnya tiba. Setiap hari ada pasar di dekat situ, dan setiap pasar berbeda. Pasar Senin di dekat terminal sepi; pasar Kamis di dekat pabrik ramai kalau para pekerja baru gajian; pasar Sabtu didatangi keluarga dari kota. Dari buku catatannya ia tahu kira-kira berapa yang bisa ia dapat di setiap pasar selama empat belas hari ke depan.

Ada satu syarat. Sesudah hari berjualan, ia memerlukan satu hari penuh di rumah untuk memetik, memilah, dan mengemas buah segar. Jadi ia tidak pernah bisa berjualan dua hari berturut-turut. Hari pasar mana yang harus ia pilih agar penghasilannya selama dua minggu itu paling besar?

Gagasan yang paling jelas ternyata gagal. “Ambil hari terbaik, lalu hari terbaik yang masih boleh, dan seterusnya” terdengar masuk akal, tetapi dengan penghasilan 5, 6, dan 5 pada tiga hari berurutan, mengambil 6 menutup kedua angka 5: padahal 6 lebih kecil daripada 10. “Ambil selang sehari” gagal pada 1, 9, 1, 1, 9, karena rencana terbaik mengambil kedua angka 9 yang berjarak tiga hari. Mencoba semua rencana memang benar, tetapi tidak sanggup menghadapi ukuran besar: empat belas hari saja sudah memberi 987 rencana yang sah, dan pasar harian selama setahun memberi sekitar 10^{76} rencana.

Pertanyaan Bu Wati, kata demi kata, adalah soal SPOJ FARIDA (Princess Farida) [1]: sederet monster, masing-masing menjaga sejumlah koin, dengan aturan bahwa Anda tidak boleh mengambil koin dari dua monster yang bertetangga. Bentuk yang sama muncul ketika menyusun jadwal seorang pekerja yang perlu hari istirahat, memilih petak tanam yang tidak bersebelahan dalam satu musim, dan ratusan pilihan sehari-hari lainnya. Bab ini menyelesaikannya, beserta lima kerabatnya, dengan satu cara berpikir.

9.2 Gagasannya dengan Kata-Kata Sederhana

9.2.1 Pertanyaan yang lebih kecil dengan bentuk yang sama

Perhatikan hari pasar terakhir, hari ke-n. Bu Wati berjualan pada hari itu atau tidak.

- Jika ia **tidak** berjualan pada hari ke-n, rencana terbaiknya adalah rencana terbaik untuk hari 1 sampai $n-1$.
- Jika ia **berjualan** pada hari ke-n, ia mendapat c_n pada hari itu, tidak boleh berjualan pada hari $n-1$, dan untuk hari 1 sampai $n-2$ ia kembali memakai rencana terbaik yang ada.

Kedua kemungkinan mengajukan pertanyaan yang sama dengan pertanyaan awal, hanya untuk deretan hari yang lebih pendek. Tulis $best(i)$ untuk penghasilan terbesar yang bisa diperoleh dengan hanya memakai hari 1 sampai i. Maka

$$best(i) = \max(best(i-1),\ best(i-2)+c_i), \qquad best(0)=0,\ best(1)=c_1 \tag{9.1}$$

Persamaan (9.1) adalah seluruh penyelesaiannya. Bentuk ini disebut *rekurensi*: setiap nilai didefinisikan melalui nilai-nilai dengan argumen yang lebih kecil, dan kedua *kasus dasar* menghentikan rantainya. Tidak diperlukan pilihan "rakus" apa pun; rekurensi cukup membandingkan dua kemungkinan untuk hari terakhir, dan setiap kemungkinan dijawab tepat oleh versi yang lebih kecil dari pertanyaan yang sama.

> **DEFINISI.** *State* (keadaan) adalah deskripsi singkat sebuah submasalah, misalnya "i hari pertama". *Nilai* sebuah state adalah jawaban terbaik (atau banyaknya cara, atau biaya terkecil) untuk submasalah itu. *Rekurensi* menyatakan nilai sebuah state melalui nilai state-state yang lebih kecil; *kasus dasar* memberikan nilai state terkecil secara langsung. *Dynamic programming* (DP) menghitung nilai setiap state yang diperlukan tepat satu kali, menyimpannya, dan membacanya kembali setiap kali diperlukan lagi [2], [3].

9.2.2 Mengerjakannya dengan tangan

Ambil lima hari pasar dengan penghasilan $c = (4,1,1,9,2)$. Isi $best(i)$ dari kiri ke kanan dengan Persamaan (9.1):

- $best(0) = 0$ dan $best(1) = 4$.
- $best(2) = \max(4,\ 0+1) = 4$: melewatkan hari 2 lebih baik.
- $best(3) = \max(4,\ 4+1) = 5$: berjualan pada hari 1 dan 3.
- $best(4) = \max(5,\ 4+9) = 13$: berjualan pada hari 4, ditambah hasil terbaik hari 1 dan 2.
- $best(5) = \max(13,\ 5+2) = 13$: melewatkan hari 5 lebih baik.

Jawabannya 13: hari 1 dan hari 4. Setiap baris hanya memakai dua baris di atasnya, sehingga pekerjaannya satu langkah pendek per hari. Untuk n hari itu berarti waktu $O(n)$, bukan rencana yang banyaknya eksponensial seperti yang akan dicoba *brute force*.

9.2.3 Lima pertanyaan untuk setiap DP

Setiap penyelesaian DP dalam bab ini, dan hampir setiap DP yang akan Anda tulis, menjawab lima pertanyaan yang sama. Menjawabnya dengan urutan ini adalah cara paling andal untuk bergerak dari pernyataan soal ke program yang benar.

1. **State.** Apa submasalahnya, dan beberapa bilangan apa yang menggambarkannya? Di sini: panjang awalan i.
2. **Rekurensi.** Bagaimana nilai sebuah state diperoleh dari state yang lebih kecil? Pikirkan *keputusan terakhir*: di sini, apakah hari ke-i dipakai.
3. **Kasus dasar.** State mana yang dapat dijawab tanpa rekurensi? Di sini: $i = 0$ dan $i = 1$.
4. **Urutan.** Dalam urutan apa state harus diisi agar setiap nilai sudah siap sebelum dibaca? Di sini: i menaik.
5. **Jawaban.** State mana, atau gabungan state mana, yang menjadi jawaban? Di sini: $best(n)$.

Waktu jalan sebuah DP adalah (banyaknya state) × (pekerjaan per state). Di sini hasilnya $n \times O(1)$. Bab 3 mengajari Anda membaca batasan lebih dulu; dengan DP Anda memperkirakan banyaknya state dan pekerjaan per state, lalu membandingkan hasil kalinya dengan kira-kira 10^8 langkah sederhana per detik yang diberikan juri.

> **WAWASAN KUNCI:** DP bukan rumus yang cerdik. DP adalah disiplin menamai submasalah dengan tepat. Begitu state-nya benar, rekurensi biasanya muncul dengan sendirinya lewat pertanyaan "apa keputusan terakhirnya?", dan programnya hanyalah gelung yang mengisi tabel dengan urutan yang benar.

9.3 Percobaan Pertama: Rekursi Polos

Program yang paling langsung untuk Persamaan (9.1) adalah fungsi rekursif yang memanggil dirinya untuk $i - 1$ dan $i - 2$. Program ini benar dan cocok sebagai pemeriksaan pertama, tetapi lambat karena suatu alasan yang layak Anda lihat sendiri. **Kode 9.1** menghitung rekurensi yang sama dengan tiga cara: rekursi polos, rekursi dengan *memo* (tabel nilai yang sudah dihitung, dengan -1 berarti "belum dihitung"), dan tabel yang diisi dari kiri ke kanan. Program ini juga menghitung banyaknya pemanggilan fungsi.

Kode 9.1 `dp_intro.cpp`: rekursi polos, rekursi dengan memo, dan tabel, ketiganya menghitung Persamaan (9.1).

```cpp
// Three ways to evaluate best(i) = max(best(i-1), best(i-2) + c[i]):
// plain recursion, recursion with a memo, and a table filled left to right.
#include <algorithm>
#include <cstdio>
#include <iostream>
#include <vector>
using namespace std;

vector<long long> c;
long long calls = 0;

long long plain(int i) {  // best total over monsters 0..i-1
    calls++;
    if (i <= 0) return 0;
    if (i == 1) return c[0];
    return max(plain(i - 1), plain(i - 2) + c[i - 1]);
}

vector<long long> memo;  // -1 means "not computed yet"; real answers are >= 0

long long memoized(int i) {
    calls++;
    if (i <= 0) return 0;
    if (i == 1) return c[0];
    if (memo[i] != -1) return memo[i];
    return memo[i] = max(memoized(i - 1), memoized(i - 2) + c[i - 1]);
}

long long table(int n) {
    vector<long long> best(n + 1, 0);
    if (n >= 1) best[1] = c[0];
    for (int i = 2; i <= n; i++) best[i] = max(best[i - 1], best[i - 2] + c[i - 1]);
    return best[n];
}

int main() {
    cout << " n    plain answer      plain calls  memo calls  table answer\n";
    for (int n : {10, 20, 30, 40}) {
        c.assign(n, 0);
        for (int i = 0; i < n; i++) c[i] = (i * 37 + 11) % 50;  // fixed coins
        calls = 0;
        long long a = plain(n);
        long long plainCalls = calls;
        memo.assign(n + 1, -1);
        calls = 0;
        memoized(n);
```

Kode 9.1 (Lanjutan) *dp_intro.cpp: rekursi polos, rekursi dengan memo, dan tabel, ketiganya menghitung Persamaan (9.1).*

```cpp
        printf("%2d %15lld %16lld %11lld %13lld\n", n, a, plainCalls, calls,
               table(n));
    }
    return 0;
}
```

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o dp_intro dp_intro.cpp
$ ./dp_intro
```

```
 n    plain answer      plain calls  memo calls  table answer
10             180              177          19           180
20             310            21891          39           310
30             467          2692537          59           467
40             612        331160281          79           612
```

Ketiganya memberi jawaban yang sama, tetapi perhatikan banyaknya pemanggilan. Rekursi polos melakukan 177 pemanggilan untuk 10 hari dan lebih dari 331 juta untuk 40 hari: setiap tambahan satu hari mengalikan pekerjaan sekitar 1,6 kali. Penyebabnya, $plain(i)$ meminta $plain(i-1)$ dan $plain(i-2)$, sedangkan $plain(i-1)$ meminta $plain(i-2)$ lagi, sehingga pertanyaan kecil yang sama diajukan berulang-ulang. Banyaknya pemanggilan tumbuh seperti bilangan Fibonacci, kira-kira 1.618^n. Dengan memo, setiap state dihitung sekali dan pemanggilan berikutnya langsung kembali: $2n-1$ pemanggilan. Versi tabel sama sekali tidak memanggil fungsi, hanya menjalankan gelung.

Inilah dua cara baku menulis DP:

- **Top-down (memoisasi):** pertahankan fungsi rekursif dan tambahkan memo. Bentuknya dekat dengan rekurensi dan hanya menghitung state yang benar-benar dicapai. Harganya adalah beban pemanggilan fungsi dan kedalaman rekursi: dengan $n = 10^5$, fungsi rekursif C++ dapat meluapkan *stack*, dan Python berhenti pada batas rekursi bawaan 1000.
- **Bottom-up (tabulasi):** isi tabel dengan urutan yang menjamin setiap nilai siap sebelum diperlukan. Cara ini lebih cepat, tidak punya masalah kedalaman, dan membuat penghematan memori mudah terlihat.

Dalam kontes, tulislah *bottom-up* kalau urutannya mudah dilihat, dan simpan *top-down* untuk state yang urutannya canggung (banyak DP pada pohon atau pada himpunan bagian bermula dengan cara itu). Jagalah agar penanda memo aman: -1 aman di sini karena setiap jawaban sungguhan paling kecil 0; untuk soal yang jawabannya bisa negatif, pakailah tabel `bool` terpisah sebagai tanda "sudah dihitung".

9.4 Pola 1: Deretan Pilihan (FARIDA)

Soalnya dengan kata-kata kita. FARIDA [1] memberikan beberapa kasus uji. Setiap kasus berisi N monster dalam satu baris, $0 \le N \le 10^4$, dan monster ke-i menjaga 0 sampai 10^9 koin. Mengambil koin dari sebuah monster hanya boleh kalau Anda tidak mengambil dari monster tepat sebelumnya. Cetak banyaknya koin terbesar yang dapat dikumpulkan, dalam bentuk `Case k: x`.

Gagasannya adalah Persamaan (9.1). Karena $best(i)$ hanya memerlukan $best(i-1)$ dan $best(i-2)$, program cukup menyimpan dua bilangan, bukan seluruh tabel. Variabel `before` menyimpan $best(i-2)$ dan `last` menyimpan $best(i-1)$; sesudah setiap monster keduanya bergeser satu tempat ke kanan.

Jebakannya. Tiga hal kecil menentukan *Accepted* atau *Wrong Answer* di sini. Pertama, N boleh 0; program yang membaca koin pertama sebelum gelung, atau mengisi $best(1)$ dari elemen larik yang tidak ada, akan gagal atau *crash* pada kasus itu. Versi bergulir di bawah menanganinya tanpa pemeriksaan khusus, karena tanpa monster `last` tetap 0. Kedua, totalnya dapat mencapai $10^4 \times 10^9 = 10^{13}$, jauh di atas $2^{31}-1 \approx 2.1 \times 10^9$ milik `int`: gunakan `long long` (Bab 6). Ketiga, format keluaran: huruf C kapital, spasi, nomor kasus, titik dua, dan spasi. **Kode 9.2** adalah program C++-nya dan **Kode 9.3** program Python-nya.

Kode 9.2 `farida.cpp`: FARIDA dengan dua nilai bergulir.

```cpp
// FARIDA (Princess Farida): the most coins from a line of monsters,
// never taking from two neighbors. Linear DP with two rolling values.
#include <algorithm>
#include <iostream>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;
    for (int tc = 1; tc <= t; tc++) {
        int n;
        cin >> n;
        long long before = 0;  // best[i-2]: best total over the first i-2 monsters
        long long last = 0;    // best[i-1]: best total over the first i-1 monsters
        for (int i = 0; i < n; i++) {
            long long c;
            cin >> c;
            long long now = max(last, before + c);  // skip monster i, or take it
            before = last;
            last = now;
        }
        cout << "Case " << tc << ": " << last << '\n';
    }
    return 0;
}
```

Kode 9.3 `farida.py`: penyelesaian yang sama dalam Python, dengan membaca seluruh masukan sekaligus.

```python
"""FARIDA (Princess Farida): the most coins from a line of monsters,
never taking from two neighbors. Linear DP with two rolling values."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    pos = 1
    out = []
    for tc in range(1, t + 1):
        n = int(data[pos])
        pos += 1
        before = last = 0  # best over the first i-2 and the first i-1 monsters
        for c in map(int, data[pos : pos + n]):
            before, last = last, max(last, before + c)  # skip it, or take it
        pos += n
        out.append(f"Case {tc}: {last}")
    print("\n".join(out))


main()
```

Versi Python membaca seluruh masukan dengan `sys.stdin.buffer.read().split()`, pola cepat dari Bab 4, dan memperbarui kedua nilai bergulir dalam satu penugasan *tuple*, persis "geser satu tempat ke kanan" pada versi C++. Kita jalankan keduanya pada dua kasus uji: lima hari dari Subbab 9.2.2 dan satu kasus tanpa monster sama sekali.

```
$ printf '2\n5\n4 1 1 9 2\n0\n' | ./farida
$ printf '2\n5\n4 1 1 9 2\n0\n' | python3 ../python/farida.py
```

```
Case 1: 13
Case 2: 0
Case 1: 13
Case 2: 0
```

Keduanya mencetak 13 untuk kasus pertama, sesuai hitungan tangan, dan 0 untuk baris kosong. Pekerjaannya $O(N)$ per kasus uji dan memorinya $O(1)$ di luar masukan, sehingga kedua bahasa berada jauh di dalam batas waktu.

9.5 Pola 2: Menghitung Banyak Cara (ACODE)

Soalnya dengan kata-kata kita. Dalam ACODE (Alphacode) [4], sebuah pesan berhuruf kapital disandikan dengan menulis A sebagai 1, B sebagai 2, dan seterusnya sampai Z sebagai 26, tanpa pemisah. Diberikan string angka dengan panjang sampai 5000 digit, hitung berapa banyak pesan huruf yang dapat menghasilkannya. Masukan berisi beberapa string dan diakhiri baris yang hanya berisi 0. Soal menjamin setiap string adalah sandi yang sah dan banyaknya cara muat dalam bilangan bulat bertanda 64 bit.

State-nya. Misalkan $ways(i)$ adalah banyaknya cara membaca i digit pertama. Huruf terakhir dari bacaan semacam itu memakai digit terakhir saja (mungkin bila digit itu 1 sampai 9) atau dua digit terakhir bersama-sama (mungkin bila keduanya membentuk bilangan 10 sampai 26). Kedua kasus tidak pernah tumpang-tindih, karena diakhiri huruf dengan panjang sandi berbeda, sehingga banyaknya dijumlahkan:

$$ways(i) = [\, s_i \neq 0 \,] \cdot ways(i-1) + [\, 10 \le \overline{s_{i-1}s_i} \le 26 \,] \cdot ways(i-2), \qquad ways(0) = 1 \tag{9.2}$$

Di sini $[\ldots]$ bernilai 1 bila syarat di dalamnya benar dan 0 bila salah, sedangkan $\overline{s_{i-1}s_i}$ adalah bilangan dua digit yang dibentuk oleh digit s_{i-1} dan s_i. Kasus dasar $ways(0) = 1$ sering mengejutkan pemula: ada tepat satu cara membaca string kosong, yaitu bacaan kosong, dan tanpa itu setiap hitungan menjadi 0.

Dengan tangan, untuk string 25114:

- $ways(0) = 1$; $ways(1) = 1$ (bacaan “2”).
- $ways(2) = 1 + 1 = 2$: “2 5” dan “25”.
- $ways(3) = 2 + 0 = 2$: digit 1 sendirian; “51” bukan huruf.
- $ways(4) = 2 + 2 = 4$: digit 1 sendirian, atau “11”.
- $ways(5) = 4 + 2 = 6$: digit 4 sendirian, atau “14”.

Jebakannya. Digit 0 tidak punya huruf sendiri, jadi hanya dapat menjadi digit kedua dari 10 atau 20; faktor $[s_i \neq 0]$ mengurus hal itu. Pasangan seperti “05” atau “30” bukan huruf, sehingga pemeriksaan dua digit harus menguji rentang 10 sampai 26, bukan sekadar “paling besar 26”. Selain itu, DP penghitung tumbuh cepat: string berisi n angka 1 punya F_{n+1} bacaan (lagi-lagi bilangan Fibonacci), yang melampaui 2^{63} pada sekitar 90 digit. Soal menjamin jawabannya muat, sehingga `long long` cukup di C++; bilangan bulat Python sama sekali tidak pernah meluap. **Kode 9.4** dan **Kode 9.5** mengubah Persamaan (9.2) menjadi gelung atas awalan-awalan string.

Kode 9.4 `acode.cpp`: menghitung bacaan dengan Persamaan (9.2).

```cpp
// ACODE (Alphacode): count the ways to read a digit string as letters,
// A=1, B=2, ..., Z=26. Counting DP over prefixes; input ends with "0".
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    string s;
    while (cin >> s && s != "0") {
        int n = s.size();
        vector<long long> ways(n + 1,
                               0);  // ways[i]: readings of the first i digits
        ways[0] = 1;                // the empty prefix has one reading
        for (int i = 1; i <= n; i++) {
            if (s[i - 1] != '0') ways[i] += ways[i - 1];  // one-digit letter 1..9
            if (i >= 2) {
                int two = (s[i - 2] - '0') * 10 + (s[i - 1] - '0');
                if (two >= 10 && two <= 26)
                    ways[i] += ways[i - 2];  // letter 10..26
            }
        }
        cout << ways[n] << '\n';
    }
    return 0;
}
```

Kode 9.5 `acode.py`: hitungan yang sama dalam Python.

```python
"""ACODE (Alphacode): count the ways to read a digit string as letters,
A=1, ..., Z=26. Counting DP over prefixes; input ends with "0"."""

import sys


def count(s: str) -> int:
    ways = [0] * (len(s) + 1)  # ways[i]: readings of the first i digits
    ways[0] = 1
    for i in range(1, len(s) + 1):
        if s[i - 1] != "0":
            ways[i] += ways[i - 1]  # one-digit letter 1..9
        if i >= 2 and 10 <= int(s[i - 2 : i]) <= 26:
            ways[i] += ways[i - 2]  # two-digit letter 10..26
    return ways[len(s)]


def main() -> None:
    out = []
    for token in sys.stdin.read().split():
        if token == "0":
            break
        out.append(str(count(token)))
    print("\n".join(out))


main()
```

```
$ printf '25114\n1111111111\n3333333333\n2101\n0\n' | ./acode
$ printf '25114\n1111111111\n3333333333\n2101\n0\n' | python3 ../python/acode.py
```

```
6
89
1
1
6
89
1
1
```

Empat baris pertama berasal dari C++, empat berikutnya dari Python. String 25114 punya 6 bacaan seperti hitungan tangan; sepuluh angka 1 punya $F_{11} = 89$; sepuluh angka 3 hanya punya satu bacaan, karena tidak ada dua angka 3 yang membentuk huruf; dan 2101 tepat punya satu, B J A, karena angka 0 memaksa "10".

9.6 Pola 3: Lintasan pada Grid (BYTESM2)

Soalnya dengan kata-kata kita. Dalam BYTESM2 (Philosophers Stone) [5], sebuah lantai berisi h baris dan w kolom ubin, masing-masing dengan sejumlah batu, harus diseberangi dari baris teratas ke baris terbawah. Dari sebuah ubin Anda boleh melangkah ke ubin tepat di bawahnya atau ke salah satu dari dua ubin diagonal di bawahnya. Anda boleh mulai di mana saja pada baris teratas dan berakhir di mana saja pada baris terbawah. Kumpulkan batu sebanyak mungkin. Ada beberapa kasus uji; h dan w paling besar 100.

State-nya adalah sebuah ubin: misalkan $best(r, c)$ adalah banyaknya batu terbesar yang terkumpul pada perjalanan yang berakhir di ubin baris r, kolom c. Langkah terakhir datang dari salah satu dari tiga ubin di atasnya:

$$best(r,c) = a_{r,c} + \max(best(r-1,c-1),\ best(r-1,c),\ best(r-1,c+1)) \tag{9.3}$$

dengan $best(0, c) = a_{0,c}$ pada baris teratas, dan ubin di luar lantai cukup dikeluarkan dari maksimum. Jawabannya adalah nilai terbesar pada baris terbawah. Urutannya baris demi baris, dan setiap baris hanya memerlukan baris di atasnya, sehingga dua baris memori sudah cukup; **Kode 9.6** dan **Kode 9.7** menyimpan tepat dua baris.

Dengan tangan, pada lantai

```
3 1 7
2 8 1
4 1 6
```

baris teratas memberi (3,1,7). Baris tengah memberi $2 + \max(3,1) = 5$, $8 + \max(3,1,7) = 15$, dan $1 + \max(1,7) = 8$. Baris terbawah memberi $4 + \max(5,15) = 19$, $1 + \max(5,15,8) = 16$, dan $6 + \max(15,8) = 21$. Jawabannya 21, dari perjalanan $7 \to 8 \to 6$. Perhatikan bahwa perjalanan rakus yang mulai dari ubin teratas terbesar kebetulan berhasil di sini, tetapi gagal begitu ada ubin besar di bawah ubin kecil.

Kode 9.6 `bytesm2.cpp`: DP grid dengan dua baris.

```cpp
// BYTESM2 (Philosophers Stone): walk from the top row to the bottom row,
// each step going straight down or diagonally down; collect the most stones.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        int h, w;
        cin >> h >> w;
        vector<long long> prev(w, 0), cur(w);  // best totals ending in each column
        for (int r = 0; r < h; r++) {
            for (int col = 0; col < w; col++) {
                long long stones;
                cin >> stones;
                long long from = prev[col];  // the row above; zeros before row 0
                if (col > 0) from = max(from, prev[col - 1]);
                if (col + 1 < w) from = max(from, prev[col + 1]);
                cur[col] = stones + from;
            }
            swap(prev, cur);
        }
        cout << *max_element(prev.begin(), prev.end()) << '\n';
    }
    return 0;
}
```

Kode 9.7 `bytesm2.py`: DP yang sama dalam Python, dengan baris berbantalan sebagai pengganti pemeriksaan tepi.

```python
"""BYTESM2 (Philosophers Stone): top row to bottom row, straight down or
diagonally down at each step; collect the most stones."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    pos = 1
    out = []
    for _ in range(t):
        h, w = int(data[pos]), int(data[pos + 1])
        pos += 2
        prev = [0] * w  # best totals ending in each column of the row above
        for _ in range(h):
            row = list(map(int, data[pos : pos + w]))
            pos += w
            # pad with a very small value so column 0 and column w-1 need no test
            pad = [-1] + prev + [-1]
            prev = [row[c] + max(pad[c], pad[c + 1], pad[c + 2]) for c in range(w)]
        out.append(str(max(prev)))
    print("\n".join(out))


main()
```

Kedua program menangani tepi lantai dengan cara berbeda. Versi C++ memeriksa `col > 0` dan `col + 1 < w` sebelum melihat ke kiri atau ke kanan. Versi Python memberi bantalan -1 di kedua sisi baris sebelumnya; karena setiap total sungguhan paling kecil 0, bantalan itu tidak pernah

menang dalam maksimum, dan *list comprehension* tidak memerlukan pemeriksaan apa pun. Membantali tepi dengan nilai yang tidak mungkin terpilih adalah trik yang patut diingat untuk setiap DP grid. Pada versi C++, baris sebelum baris teratas berisi nol semua, sehingga kasus dasar $best(0, c) = a_{0,c}$ diperoleh dengan sendirinya.

```
$ printf '1\n3 3\n3 1 7\n2 8 1\n4 1 6\n' | ./bytesm2
$ printf '1\n3 3\n3 1 7\n2 8 1\n4 1 6\n' | python3 ../python/bytesm2.py
```

```
21
21
```

Keduanya mencetak 21. Pekerjaannya $O(hw)$, paling banyak 10^4 ubin per kasus uji.

9.7 Pola 4: Memilih Barang dengan Anggaran Terbatas (KNAPSACK)

Soalnya dengan kata-kata kita. KNAPSACK (The Knapsack Problem) [6] memberikan sebuah tas berkapasitas $S \leq 2000$ dan $N \leq 2000$ barang, masing-masing dengan ukuran dan nilai. Pilih barang, masing-masing paling banyak sekali, yang jumlah ukurannya paling besar S dan jumlah nilainya sebesar mungkin. Cetak nilai tersebut.

Inilah *0/1 knapsack* yang terkenal: untuk setiap barang keputusannya 0 (tinggalkan) atau 1 (ambil). Mencoba semua 2^N himpunan bagian mustahil untuk $N = 2000$, dan aturan rakus seperti "nilai per satuan ukuran terbaik lebih dulu" gagal pada soal 0/1, seperti yang akan ditunjukkan Bab 10.

State-nya memerlukan dua bilangan: berapa barang yang sudah ditimbang, dan berapa kapasitas yang boleh dipakai. Misalkan $best_i(c)$ adalah nilai terbesar dengan hanya memakai i barang pertama dan total ukuran paling besar c. Keputusan terakhir menyangkut barang ke-i, dengan ukuran s_i dan nilai v_i:

$$best_i(c) = \max(best_{i-1}(c),\ best_{i-1}(c - s_i) + v_i), \qquad best_0(c) = 0 \tag{9.4}$$

dengan pilihan kedua hanya ada bila $s_i \leq c$. Jawabannya $best_N(S)$. Ada $(N + 1)(S + 1)$ state, sekitar empat juta, dengan pekerjaan $O(1)$ masing-masing: cepat di C++. Tabel penuh bertipe `long long` akan memakan $2001 \times 2001 \times 8$ byte, sekitar 32 MB, masih dapat diterima tetapi boros, karena baris i hanya memerlukan baris $i - 1$.

9.7.1 Satu baris, dan arah yang menentukan

Trik yang lazim hanya menyimpan satu baris `best[c]` dan memperbaruinya di tempat untuk setiap barang. Sebelum barang ke-i diproses, baris itu berisi $best_{i-1}$; sesudahnya harus berisi $best_i$. Pembaruan untuk kapasitas c membaca `best[c - size]`, sel di sebelah *kiri*. Apakah sel itu masih berisi nilai lama $best_{i-1}(c - s_i)$ atau sudah berisi nilai baru $best_i(c - s_i)$ hanya bergantung pada arah gelung, seperti diperlihatkan **Gambar 9.1** untuk satu barang berukuran 2 dan bernilai 3.



Gambar 9.1 Satu barang berukuran 2 dan bernilai 3 ditambahkan ke baris yang semuanya nol. Bila menurun, setiap sel membaca nilai dari sebelum barang itu, sehingga barang dipakai paling banyak sekali; bila menaik, sel 4 membaca sel 2 yang sudah berisi barang itu, sehingga barang terhitung dua kali, dan sel 6 tiga kali.

- Bila gelung berjalan **menurun**, dari S ke s_i, setiap sel di kiri masih belum tersentuh ketika dibaca, sehingga berisi $best_{i-1}$. Inilah tepat Persamaan (9.4): setiap barang diambil paling banyak sekali.
- Bila gelung berjalan **menaik**, sel di kiri sudah diperbarui untuk barang ke-i, sehingga mungkin sudah berisi barang ke-i, dan menambahkan v_i lagi berarti mengambil barang itu untuk kedua atau ketiga kalinya. Hasilnya adalah soal lain, yaitu *unbounded knapsack*, yang membolehkan setiap barang diambil berapa kali pun.

Tidak ada arah yang membuat program *crash*, dan keduanya memberi jawaban benar pada banyak contoh kecil; itulah sebabnya arah yang salah menjadi salah satu *Wrong Answer* paling umum dalam DP. **Kode 9.8** menjalankan kedua arah pada empat barang dan memeriksanya terhadap *brute force* atas ke-16 himpunan bagian.

Kode 9.8 `knap_direction.cpp`: kedua arah gelung dibandingkan dengan brute force.

```cpp
// What the direction of the capacity loop really computes, checked against
// a brute force over all subsets.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

long long run(const vector<int>& w, const vector<long long>& v, int cap,
              bool down) {
    vector<long long> best(cap + 1, 0);
    for (size_t i = 0; i < w.size(); i++) {
        if (down) {
            for (int c = cap; c >= w[i]; c--)
                best[c] = max(best[c], best[c - w[i]] + v[i]);
        } else {
            for (int c = w[i]; c <= cap; c++)
                best[c] = max(best[c], best[c - w[i]] + v[i]);
        }
    }
    return best[cap];
}

long long brute(const vector<int>& w, const vector<long long>& v, int cap) {
    int n = w.size();
    long long ans = 0;
    for (int mask = 0; mask < (1 << n); mask++) {  // every subset of the items
        long long size = 0, value = 0;
        for (int i = 0; i < n; i++)
            if (mask >> i & 1) size += w[i], value += v[i];
        if (size <= cap) ans = max(ans, value);
    }
    return ans;
}

int main() {
    vector<int> w = {1, 3, 4, 5};
    vector<long long> v = {1, 4, 5, 7};
    int cap = 10;
    cout << "brute force over all 16 subsets: " << brute(w, v, cap) << '\n';
    cout << "capacity loop downward (0/1):    " << run(w, v, cap, true) << '\n';
    cout << "capacity loop upward:            " << run(w, v, cap, false) << '\n';
    return 0;
}
```

```
$ ./knap_direction
```

```
brute force over all 16 subsets: 13
capacity loop downward (0/1):    13
capacity loop upward:            14
```

Brute force dan gelung menurun sepakat pada 13 (ukuran 1, 4, dan 5, nilai 1, 5, dan 7). Gelung menaik melaporkan 14, yaitu dua salinan barang berukuran 5 dan bernilai 7: jawaban yang benar untuk soal *unbounded* dan salah untuk soal ini. Ingatlah aturannya dalam satu kalimat: *0/1 berarti menurun; salinan tak terbatas berarti menaik.*

9.7.2 Programnya

Kode 9.9 adalah Persamaan (9.4) dalam satu baris, dengan gelung kapasitas berjalan menurun dan nilai bertipe `long long`.

Kode 9.9 `knapsack.cpp`: 0/1 knapsack dalam satu baris, gelung kapasitas menurun.

```cpp
// KNAPSACK (The Knapsack Problem): 0/1 knapsack with a one-row table.
// The capacity loop runs DOWNWARD so that each item is used at most once.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int cap, n;
    if (!(cin >> cap >> n)) return 0;
    vector<long long> best(cap + 1, 0);  // best[c]: most value with total size <= c
    for (int i = 0; i < n; i++) {
        int size;
        long long value;
        cin >> size >> value;
        for (int c = cap; c >= size; c--)  // downward: best[c - size] is still old
            best[c] = max(best[c], best[c - size] + value);
    }
    cout << best[cap] << '\n';
    return 0;
}
```

Di Python, gelung ganda atas empat juta sel memakan beberapa detik, terlalu lambat untuk batas satu detik. **Kode 9.10** memakai gagasan lain yang lebih cepat sekaligus lebih aman: untuk setiap barang ia membangun *seluruh baris baru* dari baris lama dengan `zip`. Setiap sel baru `max(keep, took + value)` hanya membaca baris lama, sehingga masalah arah sama sekali tidak dapat muncul, dan gelungnya berjalan di dalam *list comprehension*, jauh lebih cepat daripada `for` eksplisit atas indeks.

Kode 9.10 `knapsack.py`: DP yang sama dalam Python, satu baris baru per barang.

```python
"""KNAPSACK (The Knapsack Problem): 0/1 knapsack, one whole row per item.
The new row is built from the OLD row, so each item is used at most once."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    cap, n = int(data[0]), int(data[1])
    best = [0] * (cap + 1)  # best[c]: most value with total size <= c
    for i in range(n):
        size, value = int(data[2 + 2 * i]), int(data[3 + 2 * i])
        if size > cap:
            continue
        # capacities below size keep their value; the rest may add this item
        best = best[:size] + [
            max(keep, took + value) for keep, took in zip(best[size:], best)
        ]
    print(best[cap])


main()
```

Ekspresi `zip(best[size:], best)` memasangkan `best[c]` dengan `best[c - size]` untuk setiap kapasitas c dari `size` sampai S; kapasitas di bawah `size` tidak dapat memuat barang itu dan mempertahankan nilai lamanya, yaitu irisan `best[:size]`. Barang yang lebih besar daripada seluruh tas dilewati. Kita jalankan kedua program pada empat barang di atas.

```
$ printf '10 4\n1 1\n3 4\n4 5\n5 7\n' | ./knapsack
$ printf '10 4\n1 1\n3 4\n4 5\n5 7\n' | python3 ../python/knapsack.py
```

```
13
13
```

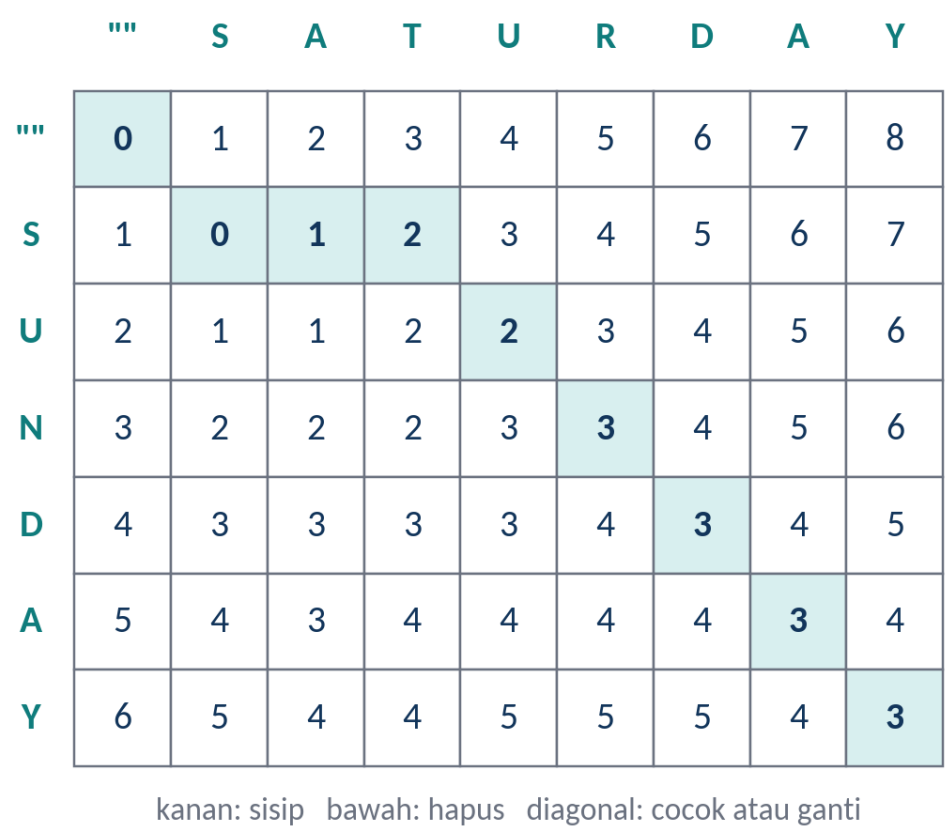
9.8 Pola 5: Dua String (EDIST)

Soalnya dengan kata-kata kita. EDIST (Edit distance) [7] memberikan sampai 10 pasang string berhuruf kapital, masing-masing paling panjang 2000 karakter. Untuk setiap pasang, cetak banyaknya operasi satu huruf paling sedikit yang mengubah string pertama menjadi string kedua, dengan operasi berupa menyisipkan huruf, menghapus huruf, atau mengganti satu huruf dengan huruf lain. Bilangan ini disebut *edit distance* (jarak sunting), atau *Levenshtein distance* [8]; pemeriksa ejaan memakainya untuk menyarankan kata yang paling dekat dengan kata yang salah ketik, dan para ahli biologi memakai kerabatnya untuk membandingkan DNA.

State-nya adalah sepasang awalan. Misalkan $d(i, j)$ adalah banyaknya operasi paling sedikit yang mengubah i huruf pertama A menjadi j huruf pertama B. Perhatikan huruf terakhirnya, A_i dan B_j. Operasi terakhir menghapus A_i, atau menyisipkan B_j, atau A_i dipasangkan dengan B_j, gratis bila keduanya sama dan dengan biaya satu penggantian bila berbeda [9]:

$$d(i, j) = \min(d(i-1, j) + 1,\ d(i, j-1) + 1,\ d(i-1, j-1) + [A_i \neq B_j]) \tag{9.5}$$

Kasus dasarnya $d(i, 0) = i$ (hapus semua i huruf) dan $d(0, j) = j$ (sisipkan semua j huruf). **Gambar 9.2** memperlihatkan tabel lengkap untuk SUNDAY dan SATURDAY, dengan satu lintasan optimal yang ditelusuri balik dari pojok. Dua langkah mendatar pada baris pertama menyisipkan A dan T, langkah diagonal mencocokkan atau mengganti, dan jawaban di pojok adalah 3: sisipkan A, sisipkan T, ganti N dengan R.



Gambar 9.2 Tabel jarak sunting untuk SUNDAY (baris) dan SATURDAY (kolom); setiap sel diperoleh dari tetangga kiri, atas, dan kiri-atasnya dengan Persamaan (9.5), dan sel yang diarsir membentuk satu lintasan termurah dengan jarak 3.

Memori. Tabel penuh punya 2001×2001 sel: empat juta `int`, 16 MB. Masih muat, tetapi, seperti pada knapsack, baris i hanya memerlukan baris $i-1$ dan sel di sebelah kirinya, sehingga dua baris berisi $m+1$ bilangan sudah cukup, dan itulah yang disimpan **Kode 9.11**. Simpan tabel penuh hanya bila Anda harus mencetak operasinya, karena penelusuran balik lintasan memerlukan setiap sel.

Kode 9.11 `edist.cpp`: jarak sunting dengan dua baris.

```cpp
// EDIST (Edit distance): fewest insertions, deletions and replacements that
// turn string A into string B. Two-string DP kept in two rows.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        string a, b;
        cin >> a >> b;
        int n = a.size(), m = b.size();
        vector<int> prev(m + 1), cur(m + 1);       // prev = row i-1, cur = row i
        for (int j = 0; j <= m; j++) prev[j] = j;  // "" -> b[0..j): j insertions
        for (int i = 1; i <= n; i++) {
            cur[0] = i;  // a[0..i) -> "": i deletions
            for (int j = 1; j <= m; j++) {
                int replace = prev[j - 1] + (a[i - 1] != b[j - 1]);
                cur[j] = min({replace, prev[j] + 1, cur[j - 1] + 1});
            }
            swap(prev, cur);
        }
        cout << prev[m] << '\n';
    }
    return 0;
}
```

Kode 9.12 `edist.py`: DP yang sama dalam Python, dengan gelung dalam yang ramping.

```python
"""EDIST (Edit distance): fewest insertions, deletions and replacements
that turn A into B. Two rows, with the inner loop kept as lean as possible."""

import sys


def distance(a: str, b: str) -> int:
    prev = list(range(len(b) + 1))  # "" -> b[:j] needs j insertions
    for i, ch in enumerate(a, 1):
        cur = [i]  # a[:i] -> "" needs i deletions
        left = i
        for diag, up, bc in zip(prev, prev[1:], b):
            x = diag if ch == bc else diag + 1  # match or replace
            if up + 1 < x:
                x = up + 1  # delete ch
            if left + 1 < x:
                x = left + 1  # insert bc
            cur.append(x)
            left = x
        prev = cur
    return prev[-1]


def main() -> None:
    data = sys.stdin.read().split()
    t = int(data[0])
    out = [str(distance(data[1 + 2 * k], data[2 + 2 * k])) for k in range(t)]
    print("\n".join(out))


main()
```

Gelung dalam versi Python sengaja tampak berbeda dari versi C++; Subbab 9.11 mengukur alasannya. Gelung itu berjalan atas `zip(prev, prev[1:], b)`, yang menyerahkan nilai kiri-atas, nilai atas, dan huruf B sekaligus tanpa pengindeksan, menyimpan tetangga kiri dalam variabel lokal, dan mengganti `min` tiga argumen dengan dua perbandingan.

```
$ printf '2\nFOOD\nMONEY\nSUNDAY\nSATURDAY\n' | ./edist
$ printf '2\nFOOD\nMONEY\nSUNDAY\nSATURDAY\n' | python3 ../python/edist.py
```

```
4
3
4
3
```

FOOD menjadi MONEY dalam 4 operasi (misalnya ganti F dengan M, ganti O kedua dengan N, ganti D dengan E, sisipkan Y), dan SUNDAY menjadi SATURDAY dalam 3 operasi pada **Gambar 9.2**.

9.9 Pola 6: Interval (MIXTURES)

Soalnya dengan kata-kata kita. Dalam MIXTURES [10], $n \le 100$ campuran berdiri berderet, masing-masing dengan warna 0 sampai 99. Dua campuran yang bertetangga dapat dicampur menjadi satu; mencampur warna a dan b menghasilkan warna $(a+b) \bmod 100$ dan $a \cdot b$ satuan asap. Campurkan semuanya menjadi satu dengan asap sesedikit mungkin. Kasus uji datang berturut-turut sampai masukan habis.

Urutan pencampuran berpengaruh. Dengan warna 30, 50, dan 70, mencampur 30 dan 50 lebih dulu menghasilkan warna 80 dan asap 1500, lalu $80 \times 70 = 5600$ lagi: total 7100. Mencampur 50 dan 70 lebih dulu menghasilkan warna 20 dan asap 3500, lalu $30 \times 20 = 600$ lagi: total 4100.

State-nya adalah sebuah interval. Apa pun urutannya, campuran i sampai j akhirnya menjadi satu campuran, dan warnanya selalu $(a_i + \cdots + a_j) \bmod 100$, apa pun urutannya; sebut saja $col(i,j)$. Pencampuran *terakhir* yang menghasilkannya menggabungkan bagian kiri $i..k$ dan bagian kanan $k+1..j$ untuk suatu titik belah k, dan setiap bagian sudah dicampur sendiri sebelumnya. Jadi

$$cost(i,j) = \min_{i \le k < j} (cost(i,k) + cost(k+1,j) + col(i,k) \cdot col(k+1,j)), \qquad cost(i,i) = 0 \tag{9.6}$$

dan jawabannya $cost(1,n)$. Bentuknya sama dengan soal perkalian rantai matriks di buku-buku teks [2]: sebuah interval, sebuah titik belah, dan biaya untuk menggabungkan kedua belahan.

Urutannya adalah pelajaran baru pola ini. $cost(i,j)$ hanya memerlukan nilai interval yang *lebih pendek*, sehingga tabel diisi menurut panjang yang menaik: pertama semua interval sepanjang 2, lalu 3, dan seterusnya. Mengisi menurut i yang menaik akan membaca $cost(k+1,j)$ sebelum nilainya dihitung. Dengan jumlah awalan (*prefix sum*), setiap $col(i,k)$ berbiaya $O(1)$; ada $O(n^2)$ interval dan $O(n)$ titik belah untuk masing-masing, sehingga pekerjaannya $O(n^3)$, sekitar 1.7×10^5 langkah untuk $n = 100$. **Kode 9.13** mengisi tabel menurut panjang; **Kode 9.14** melakukan hal yang sama dalam Python dan menghitung lebih dulu warna semua interval.

Kode 9.13 `mixtures.cpp`: DP interval yang diisi menurut panjang yang menaik.

```cpp
// MIXTURES: mix a row of colored mixtures, two neighbors at a time;
// mixing a and b gives color (a+b) mod 100 and a*b smoke. Least total smoke.
// Interval DP: cost[i][j] is the least smoke to merge mixtures i..j into one.
#include <algorithm>
#include <climits>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    while (cin >> n) {              // test cases until end of file
        vector<int> pre(n + 1, 0);  // prefix sums of the colors
        for (int i = 0; i < n; i++) {
            int x;
            cin >> x;
            pre[i + 1] = pre[i] + x;
        }
        auto color = [&](int i, int j) { return (pre[j + 1] - pre[i]) % 100; };
        vector<vector<long long>> cost(n, vector<long long>(n, 0));
        for (int len = 2; len <= n; len++) {  // shorter intervals first
            for (int i = 0; i + len - 1 < n; i++) {
                int j = i + len - 1;
                long long best = LLONG_MAX;
                for (int k = i; k < j; k++)  // the last mix joins i..k with k+1..j
                    best = min(best, cost[i][k] + cost[k + 1][j] +
                                     color(i, k) * color(k + 1, j));
                cost[i][j] = best;
            }
        }
        cout << cost[0][n - 1] << '\n';
    }
    return 0;
}
```

Kode 9.14 `mixtures.py`: DP interval yang sama dalam Python, dengan warna semua interval dihitung lebih dulu.

```python
"""MIXTURES: mix neighbors two at a time; a and b give color (a+b) mod 100
and a*b smoke. Interval DP over cost[i][j], test cases until end of file."""

import sys


def least_smoke(colors: list[int]) -> int:
    n = len(colors)
    pre = [0]
    for x in colors:
        pre.append(pre[-1] + x)
    col = [[(pre[j + 1] - pre[i]) % 100 for j in range(n)] for i in range(n)]
    cost = [[0] * n for _ in range(n)]
    for length in range(2, n + 1):  # shorter intervals first
        for i in range(n - length + 1):
            j = i + length - 1
            ci, cj = cost[i], col[i]
            cost[i][j] = min(
                ci[k] + cost[k + 1][j] + cj[k] * col[k + 1][j] for k in range(i, j)
            )
    return cost[0][n - 1]


def main() -> None:
    data = sys.stdin.buffer.read().split()
    pos, out = 0, []
    while pos < len(data):
        n = int(data[pos])
        colors = list(map(int, data[pos + 1 : pos + 1 + n]))
        pos += 1 + n
        out.append(str(least_smoke(colors)))
    print("\n".join(out))


main()
```

```
  $ printf '3\n30 50 70\n4\n10 20 30 40\n' | ./mixtures
  $ printf '3\n30 50 70\n4\n10 20 30 40\n' | python3 ../python/mixtures.py
```

```
4100
3500
4100
3500
```

Kasus uji pertama memberi 4100 seperti ditemukan di atas. Membaca “sampai masukan habis” dilakukan dengan `while (cin >> n)` di C++ dan dengan menelusuri daftar kata di Python; Bab 2 sudah memperlihatkan mengapa gelung yang menunggu baris penanda akan macet di sini.

9.10 Mengujinya: Setiap Pola terhadap Brute Force

Bab 8 memperkenalkan *stress test* (uji tekanan): bangkitkan banyak masukan kecil secara acak, hitung jawabannya dengan program lambat yang jelas benar, lalu bandingkan. Untuk DP cara ini sangat berharga, karena *bug* DP (kasus dasar yang salah, arah gelung yang salah, indeks yang meleset satu) biasanya hanya muncul pada sebagian masukan. **Tabel 9.1** mendaftar *brute force* yang dipakai untuk setiap soal. Tidak satu pun berbagi gagasan dengan DP yang diperiksanya: semuanya mencacah setiap pilihan secara langsung.

Tabel 9.1 Brute force dan masukan kecil yang dipakai untuk memeriksa keenam program DP.

Soal	Brute force dalam `tests/brute.py`	Masukan kecil dari `tests/cases.py`
FARIDA	setiap himpunan bagian monster tanpa dua tetangga	sampai 5 kasus, $0 \le N \le 10$
ACODE	rekursi atas huruf pertama, tanpa memo	sampai 5 string berisi 1 sampai 12 digit
BYTESM2	setiap perjalanan dari setiap ubin teratas	grid sampai 5×5
KNAPSACK	setiap himpunan bagian barang	sampai 10 barang, sebagian lebih besar daripada tas
EDIST	rekursi polos atas huruf pertama	string 1 sampai 6 huruf dari A, B, C
MIXTURES	setiap urutan pencampuran tetangga	sampai 6 campuran

Program pengendali, **Kode 9.15**, mengompilasi setiap program C++, menjalankannya bersama program Python pada masukan acak yang sama, lalu membandingkan kedua keluaran dengan *brute force*. Pada perbedaan pertama ia mencetak masukan yang gagal, yang cukup kecil untuk ditelusuri dengan tangan.

Kode 9.15 `tests/stress.py`: pengendali stress test untuk keenam soal bab ini.

```
"""Stress test: random small inputs; every C++ and Python solution is
compared with the brute force. Usage: python stress.py [rounds] [seed]"""

import os
import random
import subprocess
import sys

import brute
import cases

HERE = os.path.dirname(os.path.abspath(__file__))
ROOT = os.path.dirname(HERE)
EXE = ".exe" if os.name == "nt" else ""
PROBLEMS = ["farida", "acode", "bytesm2", "knapsack", "edist", "mixtures"]


def build(name: str) -> str:
    os.makedirs(os.path.join(HERE, "build"), exist_ok=True)
    exe = os.path.join(HERE, "build", name + EXE)
    src = os.path.join(ROOT, "cpp", name + ".cpp")
    subprocess.run(["g++", "-O2", "-std=c++17", "-o", exe, src], check=True)
    return exe


def run(cmd: list[str], text: str) -> str:
    done = subprocess.run(cmd, input=text, capture_output=True, text=True)
    return done.stdout.strip()


def main() -> None:
    rounds = int(sys.argv[1]) if len(sys.argv) > 1 else 200
    rng = random.Random(int(sys.argv[2]) if len(sys.argv) > 2 else 2026)
    failures = 0
    for name in PROBLEMS:
        exe = build(name)
        py = [sys.executable, os.path.join(ROOT, "python", name + ".py")]
        bad = 0
        for _ in range(rounds):
            text = getattr(cases, name)(rng)
            want = getattr(brute, name)(text)
            for who, cmd in (("C++", [exe]), ("Python", py)):
                got = run(cmd, text)
                if got != want and bad == 0:  # show the first failure only
                    print(f"{name}: {who} differs on\n{text}", end="")
                    print(f"want {want!r}, got {got!r}")
                bad += got != want
        failures += bad
        print(
            f"{name:9s} {rounds} random inputs: "
            + (
                "C++ and Python agree with brute force"
                if bad == 0
                else f"{bad} wrong answers"
            )
        )
    sys.exit(1 if failures else 0)


if __name__ == "__main__":
    main()
```

Kode 9.15 (Lanjutan) *tests/stress.py: pengendali stress test untuk keenam soal bab ini.*

```
$ cd tests
$ python3 stress.py 300
```

```
farida     300 random inputs: C++ and Python agree with brute force
acode      300 random inputs: C++ and Python agree with brute force
bytesm2    300 random inputs: C++ and Python agree with brute force
knapsack   300 random inputs: C++ and Python agree with brute force
edist      300 random inputs: C++ and Python agree with brute force
mixtures   300 random inputs: C++ and Python agree with brute force
```

Seluruh 1800 masukan acak, masing-masing berisi beberapa kasus uji, memberikan jawaban yang sama pada ketiga program. Masukan kecil itu sengaja mencakup kasus tepi yang dibahas di atas: baris tanpa monster, barang yang lebih besar daripada tas, string yang mengandung nol, grid dengan satu baris atau satu kolom.

9.11 Membuatnya Lebih Baik: Kecepatan dalam Kedua Bahasa

Benar belum berarti *Accepted*. `tests/timing.py` membangun masukan terbesar yang diizinkan setiap soal (untuk EDIST, sepuluh pasang string sepanjang 2000 huruf) dan mengukur waktu CPU setiap program. **Tabel 9.2** merangkum hasilnya pada mesin kami, sebuah PC Linux dengan g++ 13 dan Python 3.11.

```
program          C++ (s)  Python (s)  same output
farida              0.01        0.01  yes
acode               0.01        0.03  yes
bytesm2             0.01        0.01  yes
knapsack            0.00        0.19  yes
edist               0.10        3.46  yes
mixtures            0.01        0.19  yes
edist_plain         0.10        7.78  yes
```

Tabel 9.2 Masukan terbesar dan waktu CPU yang terukur; kolom terakhir adalah penilaian kami tentang seberapa longgar ruang bagi Python.

Soal	Pekerjaan pada masukan terbesar	C++ (s)	Python (s)	Python di SPOJ
FARIDA	10^4 langkah	0.01	0.01	longgar
ACODE	5000 langkah	0.01	0.03	longgar
BYTESM2	10^4 langkah	0.01	0.01	longgar
KNAPSACK	4×10^6 sel	0.00	0.19	hanya dengan pembaruan satu baris utuh
EDIST	4×10^7 sel	0.10	3.46	berisiko: juri sering lebih lambat
MIXTURES	1.7×10^6 langkah	0.01	0.19	longgar

Dua pelajaran menonjol. Pertama, C++ punya ruang lebih di mana-mana; gagasan DP, bukan bahasanya, yang membuat soal-soal ini terpecahkan. Kedua, Python memerlukan kehati-hatian tepat ketika tabelnya besar. Untuk EDIST, versi Python pertama, `python/edist_plain.py`, menulis Persamaan (9.5) secara langsung dengan indeks dan `min`; versi itu memerlukan 7,78 detik pada masukan terbesar. Gelung ramping pada **Kode 9.12** menghitung bilangan yang sama dalam 3,46 detik, kurang dari separuhnya, dengan membuang setiap pengindeksan list dan setiap pemanggilan `min` dari gelung dalam. Batas waktu EDIST cukup longgar, tetapi juri sering berjalan pada prosesor yang lebih lambat daripada PC modern; untuk tabel berisi 40 juta sel, pilihlah C++.

Tiga cara umum membuat DP lebih cepat atau lebih hemat mengikuti dari bab ini:

- **Gulirkan tabelnya.** Bila baris i hanya memerlukan baris $i - 1$, simpan dua baris (atau satu, dengan arah gelung yang tepat). FARIDA memerlukan dua bilangan, BYTESM2 dan EDIST dua baris, KNAPSACK satu baris.

- **Hitung lebih dulu apa yang diulang gelung dalam.** MIXTURES memakai jumlah awalan untuk warna; versi Python bahkan menghitung lebih dulu setiap $col(i, j)$.
- **Pindahkan gelung ke luar interpreter.** Di Python, *list comprehension* satu baris utuh dengan `zip` berjalan beberapa kali lebih cepat daripada gelung indeks, seperti diperlihatkan KNAPSACK.

9.12 Memakainya dalam Kehidupan Nyata

Keenam pola ini mencakup bagian yang mengejutkan besarnya dari soal DP yang akan Anda temui di SPOJ dan di tempat lain. **Tabel 9.3** menjajarkannya, sehingga soal baru dapat dicocokkan dengan bentuk yang sudah dikenal.

Tabel 9.3 Enam pola DP: state-nya, keputusan terakhir yang memberi rekurensi, dan urutan pengisiannya.

Pola	State	Keputusan terakhir	Urutan	Contoh
Deretan pilihan	panjang awalan i	pakai unsur ke-i atau tidak	i menaik	FARIDA
Menghitung cara	panjang awalan i	panjang potongan terakhir	i menaik	ACODE
Lintasan grid	sel (r, c)	dari mana langkah terakhir datang	baris demi baris	BYTESM2
Barang dengan anggaran	barang yang ditimbang i, anggaran c	ambil barang ke-i atau tidak	i menaik, c menurun	KNAPSACK
Dua barisan	awalan i dan j	hapus, sisip, atau cocokkan	baris demi baris	EDIST
Interval	interval $[i, j]$	titik belah terakhir k	panjang menaik	MIXTURES

Mengubah aturan berarti mengubah state, dan kelima pertanyaan tadi menunjukkan caranya. Misalkan Bu Wati mulai menjual durian di sebagian pasar, dan sesudah pasar durian ia memerlukan *dua* hari di rumah, karena durian harus dikumpulkan pada malam hari ketika buahnya jatuh. Keputusan terakhirnya tetap "berjualan pada hari ke-i atau tidak", tetapi berjualan durian pada hari ke-i kini melarang hari $i-1$ dan $i-2$, sehingga suku rekurensi itu menjadi $best(i-3)+c_i$ alih-alih $best(i-2)+c_i$, sedangkan hari-hari lainnya tetap memakai aturan lama. Programnya berubah satu baris dan menyimpan tiga nilai bergulir alih-alih dua. Bila sebaliknya ia hanya boleh berjualan paling banyak K hari dalam dua minggu itu, satu bilangan tidak lagi cukup menggambarkan sebuah submasalah: state-nya menjadi $best(i, j)$, penghasilan terbesar pada i hari pertama dengan tepat j hari pasar, sebuah tabel berisi $n \times K$ nilai. Pertanyaan "apa yang perlu diingat sebuah submasalah?" itulah yang membuat state tumbuh tepat sebesar yang diminta aturan, tidak lebih.

Ketika soal Anda sendiri datang, baik di juri maupun di tempat kerja, jalani kelima pertanyaan Subbab 9.2.3 di atas kertas sebelum menulis kode:

1. Tulis satu kalimat yang mendefinisikan state, lengkap dengan satuannya: "$best(i, c)$ adalah nilai terbesar dari i barang pertama dengan total ukuran paling besar c." Kalau kalimat ini belum bisa Anda tulis, Anda belum punya DP.
2. Tanyakan "apa keputusan terakhirnya?" dan tulis satu suku rekurensi untuk setiap kemungkinan jawabannya.
3. Tulis kasus dasarnya dan periksa pada masukan terkecil: $n = 0$, string kosong, satu ubin.
4. Gambarlah dua atau tiga sel tabel dengan panah ke sel-sel yang dibacanya, lalu pilih urutan sehingga setiap panah menunjuk ke sel yang sudah terisi.
5. Kalikan banyaknya state dengan pekerjaan per state dan bandingkan dengan batasannya.

Metode yang sama berlaku jauh di luar kontes. Toko kecil yang memutuskan pesanan mana yang diterima dengan jam kerja terbatas sedang menghadapi knapsack. Klinik yang membandingkan

dua ejaan nama pasien di dua buku register sedang menghitung jarak sunting. Petani yang merencanakan bedengan mana yang ditanami ketika bedengan bertetangga harus diistirahatkan sedang menghadapi soal Bu Wati lagi. Dalam setiap kasus programnya pendek begitu state-nya diberi nama. Untuk pola yang jauh lebih banyak (DP atas himpunan bagian dengan *bitmask*, atas digit, atas pohon), katalog Halim, Halim, dan Effendy [11] serta Laaksonen [12] adalah langkah berikutnya yang wajar; keduanya disusun, seperti bab ini, menurut bentuk state-nya.

9.13 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

- **State yang kabur.** "dp[i] adalah jawaban sampai i" menyembunyikan pertanyaannya: unsur ke-i dipakai atau tidak? Hari ke-i termasuk, atau berakhir pada hari ke-i? Tulis state sebagai satu kalimat yang tepat sebelum menulis rekurensi; kebanyakan rekurensi yang salah berasal dari state yang bermakna ganda.
- **Kasus dasar yang hilang atau salah.** Melupakan $ways(0) = 1$ membuat setiap hitungan ACODE menjadi 0; mengisi $best(1)$ dari larik kosong membuat FARIDA *crash* ketika $N = 0$. Ujilah masukan terkecil lebih dulu.
- **Mengisi dengan urutan yang salah.** Dalam DP interval, jadikan panjang sebagai gelung terluar; dalam DP apa pun, periksa bahwa setiap nilai ditulis sebelum dibaca.
- **Arah gelung yang salah pada knapsack satu baris.** Arah menaik menghitung *unbounded knapsack*. Untuk 0/1, gunakan arah menurun, atau bangun baris baru dari baris lama.
- **Luapan (*overflow*).** Jumlah banyak nilai besar (FARIDA sampai 10^{13}) dan hitungan cara (ACODE) melampaui `int`. Perkirakan nilai terbesar yang mungkin sebelum memilih tipe.
- **Penanda memo yang dapat menjadi jawaban sungguhan.** Memakai 0 untuk "belum dihitung" padahal 0 adalah jawaban yang mungkin membuat memo tidak berguna dan programnya kembali eksponensial. Gunakan -1 bila jawaban tidak negatif, atau tabel penanda tersendiri.
- **Rekursi yang terlalu dalam.** DP *top-down* atas 10^5 state dapat meluapkan *stack* C++ dan melampaui batas bawaan Python. Utamakan *bottom-up* untuk rantai yang panjang.
- **State sisa dari kasus uji sebelumnya.** Tabel yang dideklarasikan sekali lalu dipakai ulang harus diatur ulang pada setiap kasus uji (Bab 7). Mendeklarasikan baris di dalam gelung kasus uji, seperti pada semua program di sini, menghilangkan bahaya itu.

9.14 Ringkasan Bab

- *Dynamic programming* menyelesaikan soal dengan menamai submasalahnya (*state*), menyatakan setiap state melalui state yang lebih kecil (*rekurensi*), menjawab yang terkecil secara langsung (*kasus dasar*), dan mengisi tabel dengan *urutan* yang menjamin setiap nilai siap ketika dibaca.
- Rekursi polos atas sebuah rekurensi mengulang submasalah yang sama secara eksponensial; memo (*top-down*) atau tabel (*bottom-up*) menghitung setiap state sekali. **Kode 9.1** memperlihatkan 331 juta pemanggilan menyusut menjadi 79.
- Enam pola mencakup banyak soal SPOJ: deretan pilihan (FARIDA, Persamaan (9.1)), menghitung cara (ACODE, Persamaan (9.2)), lintasan grid (BYTESM2, Persamaan (9.3)), barang dengan anggaran (KNAPSACK, Persamaan (9.4)), dua string (EDIST, Persamaan (9.5)), dan interval (MIXTURES, Persamaan (9.6)).

- Waktu jalan adalah banyaknya state dikali pekerjaan per state; memori sering dapat dipangkas menjadi satu atau dua baris. Pada 0/1 knapsack satu baris, gelung kapasitas harus berjalan menurun.
- Setiap program diperiksa terhadap *brute force* pada 1800 masukan acak dan diukur waktunya pada masukan terbesar. C++ punya ruang longgar di mana-mana; Python nyaman kecuali untuk tabel terbesar, tempat operasi satu baris utuh dan gelung dalam yang ramping membantu, dan C++ tetap pilihan yang aman.

9.15 Soal Latihan

1. Dengan tangan, terapkan Persamaan (9.1) pada koin 2,7,9,3,1. Berikan setiap nilai $best(i)$, jawabannya, dan monster yang diambil. Periksa jawaban Anda dengan `farida.cpp`.
2. Dengan tangan, hitung banyaknya bacaan string ACODE 2101 dengan Persamaan (9.2). Mengapa digit 0 membuat hitungannya *lebih kecil* daripada untuk 2111?
3. Ada berapa cara membayar 10 dengan koin 1, 2, dan 5, bila urutan koin tidak diperhatikan dan setiap koin boleh dipakai berapa kali pun? Tulis rekurensi satu barisnya, sebutkan ke arah mana gelungnya harus berjalan, dan isi barisnya dengan tangan.
4. Untuk MIXTURES dengan warna 10, 20, 30, 40, program mencetak 3500. Temukan urutan pencampuran yang menghasilkan tepat 3500 asap, dan jelaskan mengapa tabel harus diisi menurut panjang interval.
5. Dengan tangan, isi tabel jarak sunting untuk KITTEN dan SITTING, lalu berikan daftar operasi yang mencapai jarak tersebut.
6. Bu Wati ingin tahu bukan hanya berapa yang dapat ia peroleh, melainkan juga hari pasar *mana* yang dipilih. Jelaskan cara memulihkan hari-hari terpilih dari tabel Persamaan (9.1), dalam bentuk pseudokode, lalu terapkan pada hari-hari di Subbab 9.2.2.

9.16 Penyelesaian Soal Latihan

1. $best(0) = 0$, $best(1) = 2$, $best(2) = \max(2,0+7) = 7$, $best(3) = \max(7,2+9) = 11$, $best(4) = \max(11,7+3) = 11$, $best(5) = \max(11,11+1) = 12$. Jawabannya 12, dari monster 1, 3, dan 5 (koin 2, 9, dan 1). Programnya sepakat:

```
$ printf '1\n5\n2 7 9 3 1\n' | ./farida
Case 1: 12
```

2. $ways(0) = 1$, $ways(1) = 1$ (B), $ways(2) = 1 + 1 = 2$ (B A, U), $ways(3) = 0 + ways(1) = 1$, karena digit 0 tidak dapat berdiri sendiri dan hanya "10" yang mungkin, dan $ways(4) = ways(3) + 0 = 1$, karena "01" bukan huruf. Satu-satunya bacaan adalah B J A. Untuk 2111 setiap digit dapat berdiri sendiri dan setiap pasangan (21, 11, 11) adalah huruf, sehingga hitungannya 1,1,2,3,5: lima bacaan. Angka 0 menghapus pilihan di kedua sisinya, karena ia harus direkatkan pada digit sebelumnya.

3. Misalkan $ways(a)$ adalah banyaknya cara membayar a dengan koin yang sudah ditimbang, $ways(0) = 1$. Dengan memproses satu nilai koin v setiap kali, $ways(a) += ways(a - v)$ untuk a dari v naik sampai 10. Gelungnya berjalan *menaik*, karena sebuah koin boleh dipakai lagi: tepat arah "unbounded" pada Subbab 9.7.1. Memproses koin satu nilai demi satu nilai, pada gelung terluar, itulah yang membuat urutan koin tidak berpengaruh (setiap kombinasi dihitung sekali, dengan urutan angka 1 dulu, lalu 2, lalu 5). Sesudah koin 1, setiap $ways(a) = 1$. Sesudah koin 2, $ways(a) = \lfloor a/2 \rfloor + 1$, sehingga $ways(0), \ldots, ways(10) = 1,1,2,2,3,3,4,4,5,5,6$. Sesudah koin 5,

dengan arah menaik, mula-mula $ways(5) = 3 + ways(0) = 4$, lalu kemudian $ways(10) = 6 + ways(5) = 6 + 4 = 10$, dengan $ways(5)$ sudah menghitung cara-cara yang memakai koin 5. Ada 10 cara.

4. Campur 10 dan 20 (asap 200, warna 30), lalu 30 ini dengan 30 berikutnya (asap 900, warna 60), lalu 60 dengan 40 (asap 2400, warna 0): $200 + 900 + 2400 = 3500$. Tabel harus diisi menurut panjang karena $cost(i, j)$ pada Persamaan (9.6) membaca $cost(k + 1, j)$, interval yang dimulai di sebelah kanan i; bila baris diisi menurut i yang menaik, nilai itu masih kosong. Setiap interval yang dibacanya lebih pendek, sehingga mengisi menurut panjang selalu aman.

```
$ printf '4\n10 20 30 40\n' | ./mixtures
3500
```

5. Dengan KITTEN sebagai baris dan SITTING sebagai kolom, baris terakhir tabel adalah 6,6,5,4,3,3,2,3 dan pojoknya berisi 3. Satu daftar termurah: ganti K dengan S, ganti E dengan I, sisipkan G. Programnya sepakat:

```
$ printf '1\nKITTEN\nSITTING\n' | ./edist
3
```

6. Simpan seluruh tabel $best(0..n)$ dan berjalanlah mundur dari ujung. Pada hari ke-i, bila $best(i) = best(i - 1)$, hari ke-i tidak diperlukan; bila tidak, hari ke-i dipilih dan penelusuran melompat ke $i - 2$. **Pseudokode 9.1** menuliskannya.

Pseudokode 9.1 Memulihkan hari-hari terpilih dari tabel FARIDA dengan berjalan mundur dari hari terakhir.

```
TRACE-DAYS(best, c, n)
1  days = empty list
2  i = n
3  while i >= 1
4      if i >= 2 and best[i] == best[i-1]
5          i = i - 1                  // day i is not needed
6      elseif i == 1 and best[1] == 0
7          i = 0                      // nothing earned on day 1
8      else
9          prepend i to days          // day i was chosen
10         i = i - 2
11 return days
```

Untuk $c = (4,1,1,9,2)$ dan $best = (0,4,4,5,13,13)$: pada $i = 5$, $best(5) = best(4)$, jadi pindah ke 4; pada $i = 4$, $13 \neq 5$, jadi hari 4 dipilih dan kita melompat ke 2; pada $i = 2$, $best(2) = best(1)$, jadi pindah ke 1; pada $i = 1$, $best(1) = 4 > 0$, jadi hari 1 dipilih. Harinya adalah 1 dan 4, seperti pada Subbab 9.2.2. "Berjalan mundur menelusuri tabel" yang sama memulihkan barang terpilih pada knapsack (simpan tabel dua dimensi yang lengkap) dan operasi pada jarak sunting (ikuti lintasan pada **Gambar 9.2**).

Daftar Pustaka

[1] Sphere Online Judge, "FARIDA - Princess Farida." [Online]. Available: https://www.spoj.com/problems/FARIDA/

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 14.

[3] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 1957.

[4] Sphere Online Judge, "ACODE - Alphacode." [Online]. Available: https://www.spoj.com/problems/ACODE/

[5] Sphere Online Judge, "BYTESM2 - Philosophers Stone." [Online]. Available: https://www.spoj.com/problems/BYTESM2/

[6] Sphere Online Judge, "KNAPSACK - The Knapsack Problem." [Online]. Available: https://www.spoj.com/problems/KNAPSACK/

[7] Sphere Online Judge, "EDIST - Edit distance." [Online]. Available: https://www.spoj.com/problems/EDIST/

[8] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[9] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974.

[10] Sphere Online Judge, "MIXTURES - Mixtures." [Online]. Available: https://www.spoj.com/problems/MIXTURES/

[11] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 1. Lulu Press, 2020.

[12] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

Bab 10

Algoritma Greedy dan Argumen Pertukaran

Algoritma *greedy* (rakus) membangun jawabannya sepotong demi sepotong dan, pada setiap langkah, mengambil potongan yang tampak terbaik saat itu, tanpa pernah mundur. Kalau berhasil, ia menghasilkan program terpendek dan tercepat di buku ini: biasanya satu pengurutan dan satu gelung. Bahayanya, algoritma greedy sering *tampak* benar padahal salah. Bab ini mengajari Anda membedakan keduanya. Di akhir bab Anda akan mampu merumuskan aturan greedy dengan tepat, membuktikannya dengan *argumen pertukaran* (ambil jawaban optimal mana pun lalu tukarkan, langkah demi langkah, menjadi jawaban greedy tanpa memperburuknya), menemukan contoh penyangkal bagi aturan yang salah dengan pemeriksa *brute force*, dan menulis penyelesaian C++17 dan Python 3 yang bersih untuk empat soal SPOJ sungguhan: BUSYMAN, FASHION, BAISED, dan GERGOVIA. Di sepanjang jalan Anda akan melihat mengapa aturan kasir "pecahan terbesar dulu" benar untuk rupiah tetapi salah untuk sistem koin lain, dan mengapa knapsack pada Bab 9 memerlukan DP sedangkan sepupunya yang pecahan tidak.

10.1 Balai Warga Pak Harun

Pak Harun mengurus balai warga di kampungnya. Setiap bulan permintaan menumpuk di buku catatannya: resepsi pernikahan dari pukul sepuluh pagi sampai empat sore, arisan ibu-ibu dari pukul satu sampai tiga, rapat karang taruna dari pukul empat sampai enam, pengajian dari pukul tujuh sampai sembilan malam, pesta ulang tahun dari pukul delapan sampai sepuluh. Balai hanya dapat menampung satu acara pada satu waktu, dan pengurus RW meminta satu hal kepadanya: tampung *sebanyak mungkin* acara, supaya sebanyak mungkin keluarga terlayani. Uang bukan soalnya; yang dihitung adalah banyaknya acara.

Ia mencoba beberapa aturan praktis. "Siapa yang mulai lebih dulu, dia yang didahulukan" memberikan balai kepada resepsi pukul sepuluh dan menutup semuanya sampai pukul empat. "Acara terpendek lebih dulu" terdengar adil, tetapi satu acara pendek di tengah sore dapat menutup dua acara yang lebih panjang yang sebenarnya muat sebelum dan sesudahnya. Setiap aturan sederhana, setiap aturan terdengar masuk akal, dan sebagian di antaranya salah. Mana yang benar, dan bagaimana Pak Harun bisa *yakin*?

Pertanyaan yang sama, dengan kegiatan sebagai pengganti acara, adalah soal SPOJ BUSYMAN (I AM VERY BUSY) [1]. Soal ini adalah contoh pertama klasik algoritma greedy [2], sekaligus contoh klasik cara membuktikannya. Begitu Anda melihat pembuktiannya, Anda akan mengenali argumen yang sama pada soal pemasangan, soal peringkat, dan soal pengangkutan, dan Anda akan tahu apa yang harus dicari ketika sebuah aturan greedy *tidak* benar.

10.2 Gagasannya dengan Kata-Kata Sederhana

Algoritma greedy membuat serangkaian pilihan. Pada setiap langkah ia menerapkan satu aturan tetap, misalnya "ambil acara yang paling cepat selesai di antara yang masih muat", dan ia tidak

pernah membatalkan pilihan. Karena itu ia cepat, tetapi juga rapuh: satu pilihan awal yang buruk tidak akan pernah bisa diperbaiki.

> **DEFINISI.** *Algoritma greedy* membangun penyelesaian langkah demi langkah, setiap kali menambahkan unsur yang terbaik menurut satu aturan lokal yang tetap, dan tidak pernah merevisi pilihan sebelumnya. Sebuah aturan greedy *benar* untuk suatu soal bila penyelesaian yang dibangunnya optimal untuk setiap masukan yang sah. *Argumen pertukaran* (*exchange argument*) membuktikan kebenaran dengan mengambil penyelesaian optimal sembarang lalu mengubahnya, satu pertukaran demi satu pertukaran, menjadi penyelesaian greedy, sambil menunjukkan bahwa tidak ada pertukaran yang memperburuknya [3].

Ada dua cara baku untuk membuktikan bahwa sebuah aturan greedy benar [2], [3]:

- **Argumen pertukaran.** Ambil penyelesaian optimal O mana pun. Bila penyelesaian itu berbeda dari penyelesaian greedy G, cari tempat pertama keduanya berbeda dan ubah O di tempat itu agar sama dengan G. Tunjukkan bahwa penyelesaian yang diubah tetap sah dan tetap optimal. Mengulang langkah ini mengubah O menjadi G, sehingga G juga optimal.
- **Greedy selalu di depan** (*greedy stays ahead*). Ukur kemajuan sesudah setiap langkah, misalnya "saat balai kembali kosong sesudah acara ke-k". Tunjukkan dengan induksi bahwa sesudah setiap langkah penyelesaian greedy paling sedikit sejauh penyelesaian lain mana pun. Maka penyelesaian greedy tidak mungkin tertinggal di akhir.

Dan ada satu cara baku untuk menunjukkan bahwa sebuah aturan greedy *salah*: satu contoh penyangkal. Bab 8 mengajari Anda menemukan contoh penyangkal secara otomatis dengan *brute force* dan uji acak; aturan greedy adalah pelanggan terbaik teknik itu.

> **WAWASAN KUNCI:** Algoritma greedy mudah ditulis dan mudah dipercaya. Sebelum mengirimkannya, buktikan dengan argumen pertukaran atau ujilah terhadap *brute force* pada ribuan masukan kecil. Kalau keduanya tidak bisa Anda lakukan, curigailah bahwa soalnya memerlukan DP.

Satu catatan tentang istilah. Dalam buku ini kata *greedy* dibiarkan dalam bentuk Inggrisnya, karena begitulah para pemrogram kompetitif di Indonesia menyebutnya sehari-hari, dan padanan "rakus" hanya dipakai untuk menjelaskan maknanya. Kata "rakus" memang tepat menggambarkan sifatnya: algoritma ini menyambar apa yang tampak paling menguntungkan di depan mata tanpa memikirkan akibatnya nanti. Sifat itu tidak selalu buruk. Seorang pedagang yang setiap pagi mendahulukan pesanan yang paling cepat selesai sering kali memang melayani pelanggan terbanyak, dan kita akan membuktikan mengapa. Tetapi sifat yang sama juga dapat menjerumuskan: pedagang yang selalu mengambil pesanan terbesar lebih dulu bisa kehabisan waktu untuk tiga pesanan kecil yang bersama-sama lebih berharga. Seluruh bab ini pada dasarnya adalah latihan untuk membedakan dua keadaan itu, dengan bukti di satu sisi dan contoh penyangkal di sisi lain. Istilah *exchange argument* kita terjemahkan menjadi "argumen pertukaran", dan *counterexample* menjadi "contoh penyangkal"; keduanya akan sering muncul. Setiap kali sebuah aturan greedy diperkenalkan dalam bab ini, kita akan mengajukan dua pertanyaan yang sama: adakah pertukaran yang membuktikannya, dan apakah *brute force* pernah membantahnya? Kebiasaan bertanya seperti itu jauh lebih berharga daripada menghafal aturan greedy mana yang benar untuk soal mana.

10.3 Percobaan Pertama: Kembalian di Kasir

Setiap kasir di Indonesia menjalankan algoritma greedy berkali-kali setiap hari. Untuk memberi kembalian 7.700 rupiah, ia mengambil uang kertas atau koin terbesar yang tidak melebihi sisa

yang masih harus dibayar, berulang-ulang: 5.000, lalu 2.000, lalu 500, lalu 200. Uang yang dipakai sehari-hari terdiri atas uang kertas 1.000, 2.000, 5.000, 10.000, 20.000, 50.000, dan 100.000 rupiah [4] serta koin 100, 200, 500, dan 1.000 rupiah [5]. Apakah "terbesar lebih dulu" selalu menghasilkan keping yang *paling sedikit*?

Untuk rupiah jawabannya ya, tetapi itu adalah sifat pecahan-pecahannya, bukan sifat algoritma greedy. Bayangkan sistem koin rekaan dengan koin 1, 3, dan 4. Untuk membayar 6, greedy mengambil 4, lalu 1, lalu 1: tiga koin. Padahal 3 + 3 hanya memakai dua. Pilihan pertama, angka 4, tampak terbaik dan ternyata salah, dan greedy tidak pernah menariknya kembali.

Kode 10.1 memeriksa kedua sistem terhadap minimum yang sebenarnya, yang dihitung dengan DP dari Bab 9: $best(a) = \min_{c \le a}(best(a-c)+1)$ atas semua koin c, dengan $best(0) = 0$. Untuk rupiah, program menguji setiap kelipatan 100 sampai 300.000.

Kode 10.1 `coin_change.cpp`: pemberian kembalian secara greedy dibandingkan dengan minimum yang dihitung DP, untuk dua sistem koin.

```
// Greedy change-making ("largest coin first") against the true minimum
// found by DP, for two coin systems: a made-up one and rupiah money.
#include <algorithm>
#include <climits>
#include <iostream>
#include <vector>
using namespace std;

int greedyCount(const vector<int>& coins, int amount) {  // coins sorted descending
    int used = 0;
    for (int c : coins) {
        used += amount / c;  // take as many of the largest coin as fit
        amount %= c;
    }
    return amount == 0 ? used : INT_MAX;
}

vector<int> fewestCoins(const vector<int>& coins, int limit) {  // DP, all amounts
    vector<int> best(limit + 1, INT_MAX);
    best[0] = 0;
    for (int a = 1; a <= limit; a++)
        for (int c : coins)
            if (c <= a && best[a - c] != INT_MAX)
                best[a] = min(best[a], best[a - c] + 1);
    return best;
}

void compare(const char* name, vector<int> coins, int unit, int limit) {
    sort(coins.rbegin(), coins.rend());
    vector<int> best = fewestCoins(coins, limit);
    int bad = 0, first = -1;
    for (int a = unit; a <= limit; a += unit) {
        if (greedyCount(coins, a) != best[a]) {
            bad++;
            if (first < 0) first = a;
        }
    }
    cout << name << ": amounts checked up to " << limit
         << ", greedy not optimal for " << bad;
    if (first >= 0)
        cout << " (first: " << first << " greedy " << greedyCount(coins, first)
             << " coins, best " << best[first] << ")";
    cout << '\n';
}
```

Kode 10.1 (Lanjutan) *coin_change.cpp: pemberian kembalian secara greedy dibandingkan dengan minimum yang dihitung DP, untuk dua sistem koin.*

```
int main() {
    compare("coins {1, 3, 4}", {1, 3, 4}, 1, 1000);
    compare("rupiah",
            {100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, 100000}, 100,
            300000);
    return 0;
}
```

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o coin_change coin_change.cpp
$ ./coin_change
```

```
coins {1, 3, 4}: amounts checked up to 1000, greedy not optimal for 249 (first: 6 greedy 3
coins, best 2)
rupiah: amounts checked up to 300000, greedy not optimal for 0
```

Dengan koin 1, 3, dan 4, greedy tidak optimal untuk 249 dari 1000 jumlah pertama, dimulai dari 6. Dengan rupiah, greedy optimal untuk setiap jumlah yang diperiksa. Sistem koin yang selalu membuat greedy optimal disebut *kanonik*. Kozen dan Zaks menunjukkan bahwa bila sebuah sistem tidak kanonik, contoh penyangkalnya pasti ada di bawah jumlah dua koin terbesarnya [6], sehingga pemeriksaan terbatas seperti milik kita dapat memutuskan persoalannya, dan Pearson kemudian memberikan uji yang cepat [7]. Untuk rupiah, pemeriksaan kita sampai 300.000 melampaui 100,000 + 50,000, sehingga kasir yang greedy terbukti benar. Untuk sistem koin baru, misalnya token arena permainan atau mata uang sebuah gim, jalankan pemeriksaan itu lebih dulu.

Mengapa rupiah berperilaku sebaik itu? Pecahannya mengikuti pola 1, 2, 5 pada setiap pangkat sepuluh: 100, 200, 500, lalu 1.000, 2.000, 5.000, dan seterusnya. Di dalam satu dekade, setiap jumlah dari satu sampai sembilan satuan dibayar oleh greedy dengan paling banyak tiga keping (misalnya $9 = 5 + 2 + 2$ dan $4 = 2 + 2$), dan tidak ada kombinasi yang lebih cerdik yang lebih baik, karena pecahan 5 adalah satu-satunya cara menutup lima satuan atau lebih dengan satu keping. Greedy lalu menangani setiap dekade sendiri-sendiri, dari atas ke bawah. Sistem rekaan 1, 3, 4 merusak keteraturan ini: angka 4 terlalu dekat dengan 3, sehingga mengambilnya dapat meninggalkan sisa yang canggung.

Ada satu jebakan lagi, dan setiap kasir mengenalnya. Pembuktian tadi mengandaikan persediaan setiap koin tidak terbatas. Ketika laci kehabisan koin 100 rupiah, greedy dapat gagal bahkan dengan rupiah: untuk memberi 600 dengan hanya tersisa koin 200 dan 500, greedy mengambil 500 lalu tidak dapat membayar sisa 100, padahal 200 + 200 + 200 berhasil. Dengan persediaan terbatas, pemberian kembalian menjadi *bounded knapsack*, dan DP dari Bab 9 kembali menjadi alatnya.

Pelajaran untuk kontes sama saja: **aturan greedy benar karena struktur soalnya, dan struktur itu harus ditunjukkan.** Empat subbab berikut menunjukkannya untuk empat soal SPOJ.

10.4 Memilih Kegiatan (BUSYMAN)

Soalnya dengan kata-kata kita. BUSYMAN [1] memberikan sampai 10 kasus uji. Setiap kasus berisi $N \leq 10^5$ kegiatan, masing-masing dengan waktu mulai dan waktu selesai antara 0 dan 10^6, dengan waktu mulai tepat sebelum waktu selesai. Seseorang hanya dapat mengerjakan satu kegiatan pada satu waktu, dan sebuah kegiatan boleh dimulai tepat ketika kegiatan sebelumnya selesai. Cetak banyaknya kegiatan terbesar yang dapat ia kerjakan.

10.4.1 Aturan yang mana?

Tiga aturan terlintas di pikiran: mulai paling awal lebih dulu, terpendek lebih dulu, dan selesai paling awal lebih dulu. Alih-alih berdebat, **Kode 10.2** menguji ketiganya terhadap *brute force* atas semua himpunan bagian, pada 10.000 himpunan acak berisi sampai tujuh kegiatan. Setiap kegiatan diperlakukan sebagai interval setengah terbuka $[s, e)$, sehingga [5,7) dan [7,9) tidak tumpang-tindih.

Kode 10.2 `greedy_keys.cpp`: tiga aturan greedy untuk pemilihan kegiatan dibandingkan dengan brute force.

```
// Three plausible greedy rules for "most non-overlapping activities",
// each checked against a brute force on 10000 random small cases.
#include <algorithm>
#include <iostream>
#include <random>
#include <vector>
using namespace std;

struct Act {
    int s, e;  // start and end; [s, e) so an end may equal the next start
};

int pickGreedy(vector<Act> a, int rule) {
    if (rule == 0) sort(a.begin(), a.end(), [](Act x, Act y) { return x.s < y.s; });
    if (rule == 1)
        sort(a.begin(), a.end(),
             [](Act x, Act y) { return x.e - x.s < y.e - y.s; });
    if (rule == 2) sort(a.begin(), a.end(), [](Act x, Act y) { return x.e < y.e; });
    vector<Act> chosen;
    for (Act x : a) {
        bool fits = true;
        for (Act y : chosen)
            if (x.s < y.e && y.s < x.e) fits = false;  // the two overlap
        if (fits) chosen.push_back(x);
    }
    return chosen.size();
}

int brute(const vector<Act>& a) {
    int n = a.size(), best = 0;
    for (int mask = 0; mask < (1 << n); mask++) {
        bool ok = true;
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j < n; j++)
                if ((mask >> i & 1) && (mask >> j & 1) && a[i].s < a[j].e &&
                    a[j].s < a[i].e)
                    ok = false;
        if (ok) best = max(best, __builtin_popcount(mask));
    }
    return best;
}
```

Kode 10.2 (Lanjutan) *greedy_keys.cpp: tiga aturan greedy untuk pemilihan kegiatan dibandingkan dengan brute force.*

```
int main() {
    const char* names[] = {"earliest start", "shortest first", "earliest end"};
    mt19937 rng(2026);
    int wrong[3] = {0, 0, 0};
    for (int round = 0; round < 10000; round++) {
        int n = rng() % 7 + 1;
        vector<Act> a(n);
        for (Act& x : a) {
            x.s = rng() % 10;
            x.e = x.s + 1 + rng() % 5;
        }
        int best = brute(a);
        for (int r = 0; r < 3; r++) wrong[r] += pickGreedy(a, r) != best;
    }
    for (int r = 0; r < 3; r++)
        cout << names[r] << ": wrong on " << wrong[r] << " of 10000 cases\n";
    return 0;
}
```

```
$ ./greedy_keys
```

```
earliest start: wrong on 1167 of 10000 cases
shortest first: wrong on 157 of 10000 cases
earliest end: wrong on 0 of 10000 cases
```

Putusannya jelas. “Mulai paling awal” gagal pada lebih dari satu kasus di antara sepuluh: kegiatan [0,10), [1,2), dan [2,3) membuatnya mengambil yang panjang dan tidak ada lagi, padahal jawabannya 2. “Terpendek lebih dulu” lebih jarang gagal, dan justru karena itu lebih berbahaya, sebab ia lolos pada kebanyakan contoh: dengan [0,5), [4,6), dan [5,10) ia mengambil [4,6) yang pendek, yang tumpang-tindih dengan kedua kegiatan lain, sehingga hasilnya 1, bukan 2. “Selesai paling awal” tidak pernah gagal. Sekarang kita buktikan bahwa ia memang tidak akan pernah gagal.

10.4.2 Mengapa selesai paling awal itu benar

Urutkan kegiatan menurut waktu selesai dan misalkan g_1 adalah kegiatan yang selesai paling awal. Ambil jadwal optimal mana pun $O = (o_1, o_2, \dots, o_k)$, tersusun menurut waktu. Ganti o_1 dengan g_1. Karena g_1 selesai tidak lebih lambat daripada kegiatan mana pun,

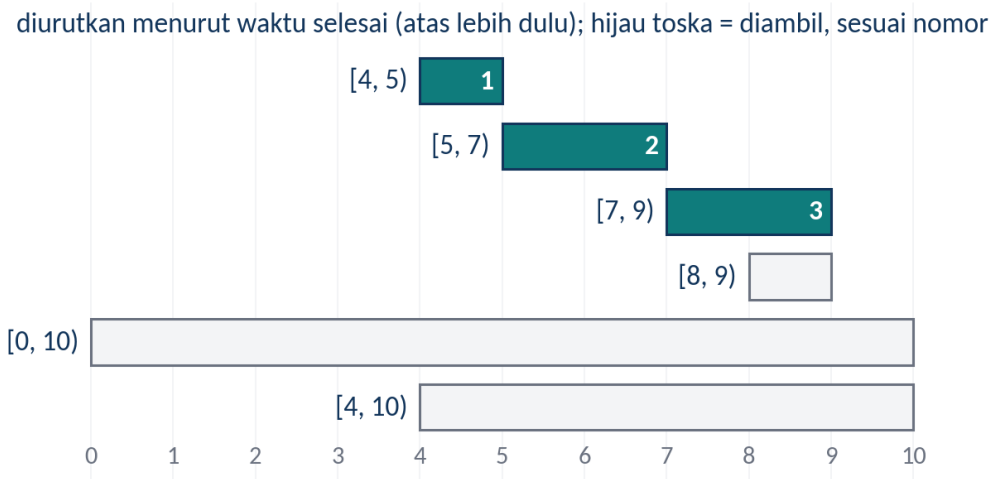
$$e(g_1) \le e(o_1) \le s(o_2) \tag{10.1}$$

sehingga g_1 tidak tumpang-tindih dengan o_2, dan jadwal baru $(g_1, o_2, \dots, o_k)$ sah dan berukuran sama, k. Jadi ada jadwal optimal yang dimulai dengan pilihan greedy. Sesudah g_1, sisa persoalannya adalah soal yang sama atas kegiatan-kegiatan yang dimulai pada atau sesudah $e(g_1)$, dan argumen yang sama berlaku untuknya. Dengan induksi, jadwal greedy memuat kegiatan sebanyak jadwal optimal. Inilah tepat argumen pertukaran dari Subbab 10.2; Persamaan (10.1) adalah satu-satunya ketaksamaan yang diperlukannya.

Fakta yang sama dapat dibaca sebagai “greedy selalu di depan”: sesudah k pilihan, jadwal greedy membebaskan orang itu tidak lebih lambat daripada jadwal lain mana pun yang berisi k kegiatan, sehingga ia selalu punya waktu tersisa paling sedikit sebanyak jadwal lain. Bila ditulis lengkap, induksinya pendek. Misalkan $g_1, g_2, \dots$ adalah kegiatan greedy dan $o_1, o_2, \dots, o_k$ kegiatan sebuah jadwal optimal, keduanya menurut urutan waktu. Untuk $k = 1$, $e(g_1) \le e(o_1)$ karena g_1 selesai paling awal di antara semuanya. Bila $e(g_j) \le e(o_j)$, maka o_{j+1} dimulai pada atau sesudah $e(o_j)$, jadi pada atau sesudah $e(g_j)$, sehingga o_{j+1} masih tersedia bagi greedy pada langkah $j + 1$;

greedy mengambil kegiatan tersedia yang selesai paling awal, sehingga $e(g_{j+1}) \leq e(o_{j+1})$. Karena itu greedy tidak pernah kehabisan kegiatan sebelum jadwal optimal kehabisan, dan ia mengambil paling sedikit k kegiatan.

Gambar 10.1 menjalankan aturan itu pada enam kegiatan. Setelah diurutkan menurut waktu selesai, greedy mengambil [4,5), lalu [5,7) yang dimulai tepat ketika yang pertama selesai, lalu [7,9); tiga kegiatan lainnya dimulai sebelum 9 dan dilewati. Jawabannya 3.



Gambar 10.1 Selesai paling awal lebih dulu pada enam kegiatan: tiga kegiatan yang diarsir diambil menurut urutan bernomor; setiap kegiatan yang dilewati dimulai sebelum orang itu kembali luang.

10.4.3 Programnya

Di C++, **Kode 10.3** menyimpan setiap kegiatan sebagai pasangan (selesai, mulai), sehingga `sort` baku mengurutkan menurut waktu selesai tanpa pembanding khusus; variabel `freeAt` adalah saat orang itu kembali luang. Perbandingan `s >= freeAt` memakai `>=`, bukan `>`, karena sebuah kegiatan boleh dimulai tepat pada saat kegiatan sebelumnya selesai. Satu karakter itulah penyebab *Wrong Answer* paling umum pada soal ini.

Kode 10.3 `busyman.cpp`: BUSYMAN, urutkan menurut waktu selesai dan ambil yang muat.

```cpp
// BUSYMAN (I AM VERY BUSY): the most activities with no two overlapping.
// Greedy: sort by end time, take every activity that starts after the last end.
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        int n;
        cin >> n;
        vector<pair<int, int>> act(n);  // stored as (end, start) to sort by end
        for (auto& [e, s] : act) cin >> s >> e;
        sort(act.begin(), act.end());
        int count = 0, freeAt = 0;  // the time at which the person is free again
        for (auto [e, s] : act) {
            if (s >= freeAt) {  // starting exactly when the last one ends is fine
                count++;
                freeAt = e;
            }
        }
        cout << count << '\n';
    }
    return 0;
}
```

Kode 10.4 `busyman.py`: greedy yang sama dalam Python.

```python
"""BUSYMAN (I AM VERY BUSY): the most activities with no two overlapping.
Greedy: sort by end time, take each activity that starts after the last end."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    pos, out = 1, []
    for _ in range(t):
        n = int(data[pos])
        starts = map(int, data[pos + 1 : pos + 1 + 2 * n : 2])
        ends = map(int, data[pos + 2 : pos + 2 + 2 * n : 2])
        pos += 1 + 2 * n
        count = free_at = 0  # the time at which the person is free again
        for e, s in sorted(zip(ends, starts)):
            if s >= free_at:  # starting exactly at the last end is fine
                count += 1
                free_at = e
        out.append(str(count))
    print("\n".join(out))


main()
```

Kode 10.4 mengambil waktu mulai dan waktu selesai dari daftar kata yang datar dengan irisan `data[pos + 1:...:2]` dan `data[pos + 2:...:2]`, lalu menggabungkannya dengan `zip` menjadi pasangan (selesai, mulai), persis seperti versi C++. Kita jalankan keduanya pada tiga kasus uji; kasus ketiga adalah kasus pada **Gambar 10.1**.

```
$ printf '3\n3\n3 9\n2 8\n6 9\n4\n1 7\n5 8\n7 8\n1 8\n6\n7 9\n0 10\n4 5\n8 9\n4 10\n5
7\n' | ./busyman
$ printf '3\n3\n3 9\n2 8\n6 9\n4\n1 7\n5 8\n7 8\n1 8\n6\n7 9\n0 10\n4 5\n8 9\n4 10\n5
7\n' | python3 ../python/busyman.py
```

```
1
2
3
1
2
3
```

Pada kasus pertama ketiga kegiatan saling tumpang-tindih, sehingga hanya satu yang mungkin. Pada kasus kedua, [1,7) selesai paling awal dan [7,8) dimulai tepat pukul 7: dua kegiatan. Kasus ketiga memberi angka 3 pada **Gambar 10.1**. Pengurutan mendominasi pekerjaannya: $O(N\log N)$ per kasus uji.

10.4.4 Bila acara punya nilai

Misalkan pengurus berubah pikiran: setiap acara kini memberi sumbangan ke kas balai, v_j rupiah untuk acara ke-j, dan Pak Harun harus memaksimalkan total sumbangan, bukan banyaknya acara. Aturan selesai paling awal langsung gagal: rapat singkat bernilai 50.000 yang selesai pukul dua belas akan menutup resepsi bernilai 2.000.000 yang berlangsung dari pukul sepuluh sampai empat. Tidak ada aturan lokal yang bertahan, karena nilai mengambil sebuah acara bergantung pada apa yang ditutupnya.

Soal *weighted interval scheduling* (penjadwalan interval berbobot) ini adalah DP bergaya Bab 9 [3]. Urutkan acara menurut waktu selesai, $e_1 \le e_2 \le \cdots \le e_N$, dan misalkan $p(j)$ adalah banyaknya acara yang selesai tidak lebih lambat daripada saat acara ke-j dimulai, yang dapat ditemukan dengan *binary search* pada waktu-waktu selesai yang sudah terurut dalam $O(\log N)$ (Bab 11). Keputusan terakhirnya adalah apakah acara ke-j ditampung:

$$best(j) = \max(best(j-1),\ v_j + best(p(j))), \qquad best(0) = 0 \tag{10.2}$$

Bentuknya sama dengan rekurensi hari pasar Bu Wati pada Subbab 9.2.1, hanya saja "hari sebelumnya" menjadi "acara terakhir yang selesai tepat waktu". Bila semua nilai sama, Persamaan (10.2) dan aturan greedy memberi banyak acara yang sama; aturan greedy adalah kasus khusus tempat argumen pertukaran berlaku. Pola ini berulang di seluruh buku: algoritma greedy sering kali adalah DP yang pilihannya sudah dibuktikan sebelumnya.

10.5 Memasangkan Dua Daftar (FASHION)

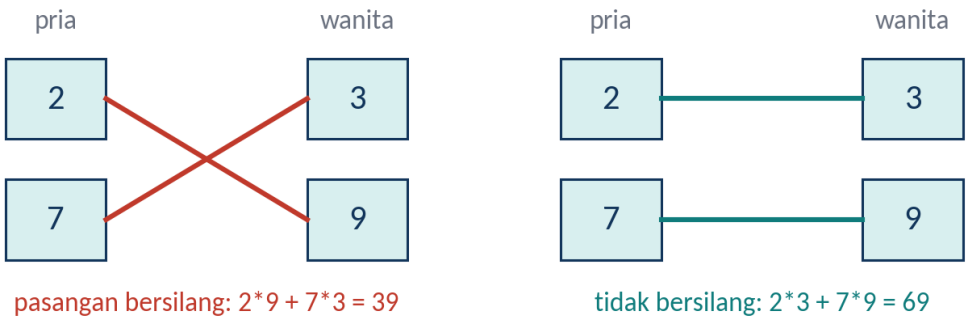
Soalnya dengan kata-kata kita. FASHION (Fashion Shows) [8] memberikan beberapa kasus uji. Setiap kasus berisi $N \le 1000$ pria dan N wanita, masing-masing dengan nilai 0 sampai 10. Pasangkan setiap pria dengan tepat satu wanita; pasangan dengan nilai x dan y menyumbang $x \cdot y$. Buat totalnya sebesar mungkin.

Aturan greedy-nya: urutkan kedua daftar lalu pasangkan menurut urutan, pria bernilai terendah dengan wanita bernilai terendah, dan seterusnya sampai yang tertinggi dengan yang tertinggi. Mengapa?

Misalkan suatu pemasangan memuat dua pasangan yang *bersilang*: pria $a_1 \le a_2$ dipasangkan berturut-turut dengan wanita b_2 dan b_1, dengan $b_1 \le b_2$. Luruskan silangnya, dengan memasangkan a_1 dengan b_1 dan a_2 dengan b_2. Totalnya berubah sebesar

$$(a_1b_1 + a_2b_2) - (a_1b_2 + a_2b_1) = (a_2 - a_1)(b_2 - b_1) \geq 0 \tag{10.3}$$

karena kedua faktornya tidak negatif. Jadi meluruskan silang tidak pernah memperkecil total. Mulailah dari pemasangan optimal mana pun dan luruskan pasangan bersilang satu per satu; setiap pertukaran mengurangi banyaknya pasangan bersilang, sehingga prosesnya berakhir, dan berakhir pada pemasangan terurut, yang karenanya juga optimal. Inilah *ketaksamaan penataan ulang* (*rearrangement inequality*) [9], dibuktikan dengan argumen pertukaran. **Gambar 10.2** memperlihatkan kasus terkecil: meluruskan 2 dan 7 terhadap 3 dan 9 menaikkan total dari 39 menjadi 69, dan Persamaan (10.3) meramalkan kenaikannya dengan tepat: $(7-2)(9-3) = 30$.



Gambar 10.2 Meluruskan dua pasangan bersilang tidak pernah menurunkan jumlah hasil kali; di sini jumlahnya naik sebesar $(7-2)(9-3) = 30$.

Kode 10.5 `fashion.cpp`: urutkan kedua daftar lalu pasangkan menurut urutan.

```cpp
// FASHION (Fashion Shows): pair men and women to maximize the sum of the
// products of their ratings. Sort both lists and pair them in order.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        int n;
        cin >> n;
        vector<int> men(n), women(n);
        for (int& x : men) cin >> x;
        for (int& x : women) cin >> x;
        sort(men.begin(), men.end());
        sort(women.begin(), women.end());
        long long sum = 0;
        for (int i = 0; i < n; i++) sum += (long long)men[i] * women[i];
        cout << sum << '\n';
    }
    return 0;
}
```

Kode 10.6 `fashion.py`: pemasangan yang sama dalam Python.

```python
"""FASHION (Fashion Shows): pair two lists of ratings to maximize the sum
of products. Sort both lists and pair them in order."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    pos, out = 1, []
    for _ in range(t):
        n = int(data[pos])
        men = sorted(map(int, data[pos + 1 : pos + 1 + n]))
        women = sorted(map(int, data[pos + 1 + n : pos + 1 + 2 * n]))
        pos += 1 + 2 * n
        out.append(str(sum(m * w for m, w in zip(men, women))))
    print("\n".join(out))


main()
```

Kode 10.5 mengubah salah satu faktor menjadi `long long` sebelum mengalikan. Dengan nilai sampai 10 dan $N \le 1000$, totalnya paling besar 10^5 dan `int` sebenarnya cukup, tetapi konversi itu tidak berbiaya dan melindungi program ketika kode yang sama dipakai ulang untuk nilai yang lebih besar; **Kode 10.6** tidak memerlukan kehati-hatian semacam itu.

```
$ printf '1\n4\n3 1 4 1\n5 9 2 6\n' | ./fashion
$ printf '1\n4\n3 1 4 1\n5 9 2 6\n' | python3 ../python/fashion.py
```

```
61
61
```

Setelah diurutkan, nilai para pria adalah 1,1,3,4 dan para wanita 2,5,6,9, dan totalnya $2 + 5 + 18 + 36 = 61$.

10.6 Tempat bagi Keinginan (BAISED)

Soalnya dengan kata-kata kita. Dalam BAISED (Biased Standings) [10], N tim (sampai sekitar 10^5) masing-masing menyebutkan peringkat dari 1 sampai N yang mereka inginkan; beberapa tim boleh menginginkan peringkat yang sama. Panitia harus memberi setiap tim peringkat yang berbeda dari 1 sampai N. *Badness* (tingkat keburukan) sebuah susunan peringkat adalah jumlah, atas semua tim, jarak antara peringkat yang diinginkan dan peringkat yang diberikan. Cetak *badness* terkecil yang mungkin. Masukan dimulai dengan banyaknya kasus uji, setiap kasus uji didahului sebuah baris kosong, dan setiap tim punya nama sebelum keinginannya.

Aturan greedy-nya: urutkan keinginan-keinginan, $w_{(1)} \le w_{(2)} \le \cdots \le w_{(N)}$, lalu berikan peringkat i kepada tim dengan keinginan terkecil ke-i. Maka

$$badness = \sum_{i=1}^{N} \left| w_{(i)} - i \right| \tag{10.4}$$

Mengapa. Argumennya adalah pertukaran pada Subbab 10.5 dengan jarak sebagai pengganti hasil kali. Ambil dua tim dengan keinginan $w_1 \le w_2$ yang menerima peringkat $p_1 > p_2$, sebuah penugasan bersilang. Menukar peringkat keduanya tidak pernah menaikkan $|w_1 - p| + |w_2 - q|$: bila dua titik pada sebuah garis dipasangkan dengan dua titik lain, pemasangan yang tidak bersilang tidak pernah lebih panjang daripada yang bersilang. (Periksalah kasus-kasusnya pada garis bilangan: bila kedua keinginan terletak di bawah kedua peringkat, kedua pemasangan

berbiaya sama; selain itu, pemasangan bersilang menempuh suatu ruas garis dua kali.) Meluruskan setiap pasangan bersilang membawa susunan optimal mana pun ke susunan terurut, sehingga Persamaan (10.4) optimal.

Jebakannya. *Badness* dapat mencapai sekitar $N^2/2 = 5 \times 10^9$ ketika semua tim menginginkan peringkat 1, yang meluapkan `int`, sehingga dijumlahkan dalam `long long` [10]. Baris kosong dan nama tim tampak menakutkan, tetapi tidak berbahaya: `cin >>` melewati semua spasi putih, termasuk baris kosong, dan nama dibaca ke dalam string yang dibuang. Di Python nama dilewati dengan mengambil setiap kata kedua. **Kode 10.7** mengurutkan keinginan dengan `sort` lalu menjumlahkan Persamaan (10.4).

Kode 10.7 `baised.cpp`: urutkan keinginan lalu jumlahkan jaraknya.

```cpp
// BAISED (Biased Standings): give every team a distinct place 1..N so that
// the sum of |preferred place - actual place| is least. Sort the wishes.
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;  // blank lines between cases are skipped by >>
    while (t--) {
        int n;
        cin >> n;
        vector<int> wish(n);
        string name;  // team names do not matter for the answer
        for (int& w : wish) cin >> name >> w;
        sort(wish.begin(), wish.end());
        long long badness = 0;  // up to about N*N/2, too big for int
        for (int i = 0; i < n; i++) badness += llabs((long long)wish[i] - (i + 1));
        cout << badness << '\n';
    }
    return 0;
}
```

Versi Python, **Kode 10.8**, memakai pengurutan yang lain. Karena setiap keinginan adalah bilangan bulat dari 1 sampai N, *counting sort* (pengurutan cacah) menyelesaikannya dalam $O(N)$: hitung berapa tim yang menginginkan setiap peringkat, lalu bagikan peringkat menurut urutan keinginan. Tim-tim dengan keinginan yang sama mendapat peringkat berurutan, persis seperti hasil pengurutan.

Kode 10.8 `baised.py`: greedy yang sama dengan counting sort.

```python
"""BAISED (Biased Standings): distinct places 1..N with the least sum of
|preferred - actual|. Sort the wishes (a counting sort, since 1 <= wish <= N)."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()  # blank lines vanish here
    t = int(data[0])
    pos, out = 1, []
    for _ in range(t):
        n = int(data[pos])
        count = [0] * (n + 2)
        for w in data[pos + 2 : pos + 2 + 2 * n : 2]:  # skip the team names
            count[int(w)] += 1
        pos += 1 + 2 * n
        place, badness = 1, 0
        for wish in range(1, n + 1):
            for _ in range(
                count[wish]
            ):  # teams with this wish take the next places
                badness += abs(wish - place)
                place += 1
        out.append(str(badness))
    print("\n".join(out))


main()
```

Kita memakai tim-tim kita sendiri, yang dinamai dengan nama burung. Pada kasus uji pertama tujuh tim menginginkan peringkat 1,2,1,1,2,3,5; pada kasus kedua, tiga tim semuanya menginginkan peringkat pertama.

```
$ printf '2\n\n7\nGaruda 1\nElang 2\nRajawali 1\nMerpati 1\nKutilang 2\nNuri 3\nKakatua
5\n\n3\nA 1\nB 1\nC 1\n' | ./baised
$ printf '2\n\n7\nGaruda 1\nElang 2\nRajawali 1\nMerpati 1\nKutilang 2\nNuri 3\nKakatua
5\n\n3\nA 1\nB 1\nC 1\n' | python3 ../python/baised.py
```

```
13
3
13
3
```

Untuk kasus pertama keinginan yang terurut adalah 1,1,1,2,2,3,5 terhadap peringkat $1, \ldots, 7$, sehingga menurut Persamaan (10.4) *badness*-nya $0 + 1 + 2 + 2 + 3 + 3 + 2 = 13$. Pada kasus kedua, ketiga tim mendapat peringkat 1, 2, dan 3: $0 + 1 + 2 = 3$.

10.7 Memindahkan Barang di Sepanjang Garis (GERGOVIA)

Soalnya dengan kata-kata kita. Dalam GERGOVIA (Wine trading in Gergovia) [11], N rumah berdiri berderet, $2 \le N \le 10^5$, dengan jarak yang sama. Rumah ke-i ingin membeli a_i botol anggur bila $a_i > 0$ atau menjual $-a_i$ botol bila $a_i < 0$, dengan $|a_i| \le 1000$, dan total permintaan sama dengan total penawaran: $a_1 + \cdots + a_N = 0$. Membawa satu botol dari sebuah rumah ke rumah tetangganya berbiaya satu satuan kerja. Carilah total kerja terkecil. Kasus uji datang berturut-turut sampai baris berisi 0.

Bayangkan keadaan yang sama di sebuah desa: sederet kios di sepanjang jalan, sebagian kelebihan beras dan sebagian kekurangan, dan seorang kuli angkut yang dibayar per karung per kios yang dilewati. Pertanyaannya adalah ongkos angkut total yang paling kecil.

Pengamatan kuncinya melihat *batas* antara rumah-rumah yang bertetangga, bukan rumahnya. Perhatikan batas antara rumah i dan rumah $i+1$, dan misalkan

$$P_i = a_1 + a_2 + \cdots + a_i$$

adalah permintaan bersih semua rumah di sebelah kirinya. Bila $P_i > 0$, sisi kiri menginginkan P_i botol lebih banyak daripada yang dimilikinya, dan botol itu hanya dapat datang melintasi batas ini dari kanan; bila $P_i < 0$, sisi kiri kelebihan $-P_i$ botol, dan botol itu harus menyeberang ke kanan. Dengan cara mana pun, paling sedikit $|P_i|$ botol melintasi batas ini, dan setiap lintasan berbiaya satu satuan. Menjumlahkan atas semua batas memberi batas bawah bagi kerja:

$$work \geq \sum_{i=1}^{N-1} |P_i| \tag{10.5}$$

Rencana greedy mencapai batas itu. Berjalanlah dari kiri ke kanan sambil membawa kelebihan atau utang berjalan: pada setiap batas, teruskan P_i saat itu ke kanan (atau ambil $-P_i$ dari kanan, yang merupakan aliran yang sama dengan arah berlawanan). Setiap batas dilintasi tepat $|P_i|$ botol, tidak pernah lebih, sehingga kerjanya tepat sama dengan ruas kanan Persamaan (10.5). Batas bawah yang tercapai adalah nilai optimal. Di sini "argumen pertukaran" berwujud batas bawah yang tercapai: rencana apa pun harus membayar paling sedikit $|P_i|$ pada setiap batas, dan rencana greedy membayar tepat sebesar itu.

Karena $P_N = 0$, menjumlahkan $|P_i|$ sampai N alih-alih $N-1$ tidak menambah apa-apa, dan program melakukannya demi kesederhanaan. Bila separuh rumah pertama masing-masing menjual 1000 botol dan separuh kedua membeli 1000 botol, $|P_i|$ naik sampai 5×10^7 dan kerjanya sampai sekitar 2.5×10^{12}, jauh di atas `int`, sehingga C++ memakai `long long`.

Kode 10.9 `gergovia.cpp`: jumlah nilai mutlak jumlah awalan.

```cpp
// GERGOVIA (Wine trading in Gergovia): houses on a line buy (+) or sell (-)
// wine; moving one bottle to a neighbor costs 1. Least total work: every
// border between house i and i+1 must carry |a[0] + ... + a[i]| bottles.
#include <cstdlib>
#include <iostream>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    while (cin >> n && n != 0) {
        long long carry = 0;  // bottles that must cross the current border
        long long work = 0;
        for (int i = 0; i < n; i++) {
            long long a;
            cin >> a;
            carry += a;
            work += llabs(carry);
        }
        cout << work << '\n';
    }
    return 0;
}
```

Kode 10.10 `gergovia.py`: jumlah yang sama dengan `itertools.accumulate`.

```python
"""GERGOVIA (Wine trading in Gergovia): the border after house i carries
|a[0] + ... + a[i]| bottles; the least total work is the sum of these."""

import sys
from itertools import accumulate


def main() -> None:
    data = sys.stdin.buffer.read().split()
    pos, out = 0, []
    while True:
        n = int(data[pos])
        if n == 0:
            break
        a = map(int, data[pos + 1 : pos + 1 + n])
        pos += 1 + n
        out.append(str(sum(abs(carry) for carry in accumulate(a))))
    print("\n".join(out))


main()
```

Kode 10.9 menyimpan jumlah awalan berjalan dalam `carry`; **Kode 10.10** membiarkan `itertools.accumulate` menghasilkan semua jumlah awalan sekaligus. Uji kita memakai dua desa: empat rumah dengan $3, -1, -4, 2$ dan enam rumah dengan $-2, 0, 5, -1, -3, 1$.

```
$ printf '4\n3 -1 -4 2\n6\n-2 0 5 -1 -3 1\n0\n' | ./gergovia
$ printf '4\n3 -1 -4 2\n6\n-2 0 5 -1 -3 1\n0\n' | python3 ../python/gergovia.py
```

```
7
10
7
10
```

Untuk desa pertama jumlah awalannya $3, 2, -2, 0$, sehingga kerjanya $3 + 2 + 2 = 7$: tiga botol melintasi batas pertama, dua batas kedua, dua batas ketiga. Ikutilah sang kuli untuk melihat bahwa 7 memang tercapai. Rumah 1 ingin 3 botol dan rumah 3 punya 4 untuk dijual; rumah 2 punya 1 untuk dijual dan rumah 4 ingin 2. Pada batas antara rumah 1 dan 2, tiga botol harus bergerak ke kiri, karena rumah 1 ingin 3 dan tidak punya sama sekali. Rumah 2 menyediakan satu di antaranya, jadi dua lagi harus melintasi batas antara rumah 2 dan 3, dari rumah 3. Rumah 3 punya 4; ia mengirim 2 ke kiri dan harus mengirim 2 sisanya ke rumah 4, melintasi batas ketiga. Botol yang melintasi ketiga batas berjumlah 3, 2, dan 2, tepat sama dengan nilai mutlak jumlah awalannya, dan tidak ada rencana yang bisa lebih sedikit, karena setiap lintasan itu dipaksakan oleh Persamaan (10.5). Untuk desa kedua jumlah awalannya $-2, -2, 3, 2, -1, 0$, dan kerjanya $2 + 2 + 3 + 2 + 1 = 10$.

10.8 Mengujinya: Setiap Greedy terhadap Brute Force

Pembuktian di atas adalah alasan sesungguhnya untuk mempercayai program-program ini, tetapi pembuktian bisa berlubang dan program bisa salah ketik. Pengendali *stress test* dari Subbab 9.10 dipakai ulang tanpa perubahan, dengan keempat soal bab ini, pembangkit acaknya dalam `tests/cases.py`, dan *brute force* pada **Tabel 10.1** dalam `tests/brute.py`. Setiap *brute force* mencoba *setiap* kemungkinan, justru hal yang diklaim tidak perlu oleh algoritma greedy.

Tabel 10.1 Brute force yang dipakai untuk memeriksa keempat program greedy.

Soal	Brute force	Masukan kecil
BUSYMAN	setiap himpunan bagian kegiatan, disimpan bila tidak ada dua yang tumpang-tindih	sampai 8 kegiatan dalam [0,12]
FASHION	setiap permutasi para wanita	sampai 7 pasangan
BAISED	setiap penugasan peringkat $1..N$	sampai 7 tim
GERGOVIA	setiap cara mengirim setiap botol ke seorang pembeli	sampai 6 rumah, paling banyak 2 botol masing-masing

Brute force GERGOVIA layak mendapat penjelasan. Ia mengubah setiap botol menjadi objek tersendiri, mendaftar posisi setiap botol yang akan dijual dan setiap botol yang akan dibeli, lalu mencoba setiap cara memasangkan botol penjual dengan botol pembeli sambil menjumlahkan jaraknya. Ia sama sekali tidak tahu tentang batas atau jumlah awalan, sehingga menjadi pemeriksaan yang benar-benar independen bagi Persamaan (10.5).

```
$ cd tests
$ python3 stress.py 300
```

```
busyman   300 random inputs: C++ and Python agree with brute force
fashion   300 random inputs: C++ and Python agree with brute force
baised    300 random inputs: C++ and Python agree with brute force
gergovia  300 random inputs: C++ and Python agree with brute force
```

Keseluruhan 1200 masukan acak memberikan jawaban yang sama. Ada gunanya merusak sebuah program dengan sengaja lalu mengamati reaksi pengendalinya: ubah `s >= freeAt` menjadi `s > freeAt` pada `busyman.cpp`, atau urutkan menurut waktu mulai alih-alih waktu selesai, lalu jalankan pengendalinya lagi. Dalam beberapa putaran saja ia mencetak sebuah masukan kecil yang gagal, biasanya berisi dua atau tiga kegiatan, bersama jawaban yang diharapkan dan jawaban yang sebenarnya keluar. Masukan kecil itulah contoh penyangkal yang kalau tidak begitu harus Anda cari dengan tangan, dan itulah yang pantas diterima oleh aturan greedy yang belum dibuktikan.

10.9 Membuatnya Lebih Baik: Kecepatan dalam Kedua Bahasa

Program-program greedy ini semuanya berbentuk "urutkan, lalu satu kali lintasan", sehingga cepat di C++. **Tabel 10.2** memberikan waktu CPU terukur pada masukan terbesar, yang dihasilkan oleh `tests/timing.py`.

```
program          C++ (s)  Python (s)  same output
busyman             0.14        0.73  yes
fashion             0.01        0.03  yes
baised              0.02        0.03  yes
gergovia            0.04        0.16  yes
```

Tabel 10.2 Masukan terbesar dan waktu CPU terukur untuk keempat program greedy.

Soal	Masukan terbesar	Pekerjaan	C++ (s)	Python (s)
BUSYMAN	10 kasus berisi 10^5 kegiatan	pengurutan, $O(N\log N)$	0.14	0.73
FASHION	100 kasus berisi 1000 pasangan	dua pengurutan	0.01	0.03
BAISED	10^5 tim	pengurutan (C++), counting sort (Python)	0.02	0.03
GERGOVIA	10 kasus berisi 10^5 rumah	satu lintasan	0.04	0.16

BUSYMAN yang perlu diawasi: batas waktunya di SPOJ ketat (sepersekian detik [1]), dan dengan sampai satu juta kegiatan dalam satu berkas, bahkan program C++ menghabiskan sebagian besar waktunya untuk membaca bilangan dan mengurutkan. Masukan cepat dari Bab 4 penting di sini,

dan Python, dengan 0,73 detik pada mesin kami, kecil kemungkinannya lolos. Dua perbaikan membantu C++ lebih jauh bila diperlukan: baca masukan dengan pembaca blok seperti pada program EKO di Bab 11, dan, karena waktu-waktunya bilangan bulat sampai 10^6, ganti pengurutan berbasis perbandingan dengan *counting sort* pada waktu selesai, yang menurunkan pekerjaan menjadi $O(N + 10^6)$.

Untuk BAISED, *counting sort* pada **Kode 10.8** itulah alasan Python secepat C++ di sini: keinginannya bilangan bulat kecil, sehingga pengurutannya berbiaya $O(N)$, bukan $O(N\log N)$. Setiap kali kunci pengurutan sebuah greedy berupa bilangan bulat kecil, ingatlah pilihan ini.

10.9.1 Bila tumpukan mana pun boleh digabung

Bab 9 menyelesaikan MIXTURES, yang hanya membolehkan campuran *bertetangga* dicampur, dengan DP interval. Ubah satu kata, dan soalnya menjadi greedy. Sebuah penggilingan padi menerima karung dari n petani dan harus menuangkan semuanya menjadi satu tumpukan, dua tumpukan setiap kali; menuang tumpukan seberat a dan b kilogram berbiaya tenaga $a + b$, dan tumpukan *mana pun* boleh dituang bersama. Rencana termurah adalah menuang dua tumpukan *terkecil* bersama-sama, setiap kali. **Pseudokode 10.1** menuliskannya dengan antrean prioritas, yang menyerahkan unsur terkecil dalam $O(\log n)$.

Pseudokode 10.1 Menggabungkan tumpukan dua demi dua dengan biaya total terkecil: selalu gabungkan dua yang terkecil.

```
MERGE-COST(a[1..n])
1  Q = priority queue containing a[1], ..., a[n]
2  total = 0
3  while Q holds more than one heap
4      x = EXTRACT-MIN(Q)
5      y = EXTRACT-MIN(Q)
6      total = total + x + y
7      INSERT(Q, x + y)
8  return total
```

Dengan tangan, untuk tumpukan seberat 4, 3, 2, dan 6 kilogram: tuang 2 dan 3 (biaya 5), lalu 4 dan 5 (biaya 9), lalu 6 dan 9 (biaya 15), total 29. Menuang dua yang terbesar lebih dulu, 4 dan 6, lalu menambahkan 2 dan 3 bergiliran, akan berbiaya $10 + 12 + 15 = 37$. Pembuktiannya kembali berupa pertukaran. Setiap rencana adalah pohon biner yang daunnya tumpukan-tumpukan asal, dan berat sebuah tumpukan dibayar sekali untuk setiap penuangan yang diikutinya, yaitu sekali untuk setiap tingkat di atas daunnya. Ambil pohon optimal dan perhatikan pasangan daun bersaudara yang paling dalam; menukar dua tumpukan terkecil ke tempat itu tidak mungkin menaikkan biaya, karena pertukaran itu memindahkan berat yang lebih besar ke atas dan berat yang lebih kecil ke bawah. Jadi ada rencana optimal yang dimulai dengan menuang dua tumpukan terkecil, dan sisa argumennya berulang pada tumpukan yang tersisa. Inilah algoritma Huffman, yang ditemukan pada 1952 untuk membangun kode prefiks terpendek bagi pemampatan teks [12], dan sampai hari ini masih ada di dalam setiap berkas ZIP. Bila diletakkan di samping MIXTURES, algoritma ini memperlihatkan seluruh tema bab ini dalam satu gambaran: biaya penggabungan yang sama, satu kebebasan dihapus, dan alat yang tepat berpindah dari greedy ke DP.

10.10 Memakainya dalam Kehidupan Nyata

Cara berpikir greedy ada di mana-mana dalam pekerjaan sehari-hari, begitu pula aturan greedy yang salah. **Tabel 10.3** mengumpulkan pertanyaan yang perlu diajukan sebelum mempercayai sebuah aturan greedy.

Tabel 10.3 Pertanyaan yang memutuskan antara algoritma greedy dan sesuatu yang lebih kuat.

Pertanyaan	Bila jawabannya ya	Contoh
Dapatkah pertukaran dua unsur dalam jawaban optimal mana pun mengubahnya menjadi jawaban greedy tanpa kerugian?	greedy benar; tuliskan pembuktiannya	BUSYMAN, FASHION, BAISED
Adakah batas bawah yang dicapai tepat oleh jawaban greedy?	greedy benar	GERGOVIA
Apakah *brute force* pada ribuan masukan kecil acak pernah berbeda pendapat?	greedy salah; simpan contoh penyangkalnya	koin 1, 3, 4; terpendek lebih dulu
Haruskah setiap barang diambil utuh, dengan anggaran (0/1)?	curigai DP, bukan greedy	KNAPSACK (Bab 9)
Bolehkah barang dibagi (pecahan)?	greedy menurut nilai per satuan sering berhasil	knapsack pecahan (Soal 4)

Beberapa pemakaian sehari-hari, kini dengan aturan yang benar:

- **Memesan satu ruang atau satu mesin.** Satu-satunya mesin USG di sebuah klinik, satu-satunya lift hidrolik di sebuah bengkel, balai warga Pak Harun: untuk melayani permintaan terbanyak, ambil permintaan yang *selesai* paling awal di antara yang masih muat. Bila permintaan punya nilai berbeda (resepsi membayar lebih daripada rapat), tujuannya bukan lagi banyaknya, dan soalnya menjadi penjadwalan interval *berbobot*, yang memerlukan DP [3].
- **Memasangkan untuk memaksimalkan hasil total.** Tugaskan pekerja terkuat ke mesin berlaju tertinggi: urutkan keduanya dan pasangkan menurut urutan, sesuai Persamaan (10.3). Untuk *meminimalkan* jumlah hasil kali (misalnya memasangkan muatan berat dengan jarak pendek), urutkan satu daftar menaik dan yang lain menurun.
- **Memindahkan stok antartoko di satu jalan.** Ongkos angkut total terkecil adalah jumlah nilai mutlak jumlah awalan, Persamaan (10.5), dan rencananya adalah meneruskan kelebihan di sepanjang jalan.
- **Menyelesaikan pesanan tepat waktu.** Seorang penjahit dengan satu mesin jahit punya pesanan dengan lama jahit dan tanggal janji yang diketahui, dan ingin pesanan yang *paling* terlambat sesedikit mungkin terlambat. Jahitlah menurut tanggal janji, *earliest deadline first* (tenggat paling awal lebih dulu). Argumen pertukarannya memandang dua pesanan bertetangga A dan B, dengan A dijahit lebih dulu padahal B punya tenggat lebih awal. Tukarkan keduanya. Pasangan itu tetap selesai pada saat yang sama, T. Sesudah ditukar, B selesai lebih awal daripada sebelumnya, sehingga keterlambatannya hanya bisa mengecil, dan A kini selesai pada T, saat B dulu selesai; karena tenggat A lebih akhir daripada tenggat B, keterlambatan A pada T lebih kecil daripada keterlambatan B dulu pada T. Jadi keterlambatan terbesar tidak dapat bertambah. Menghilangkan *inversi* semacam itu satu per satu mengubah jadwal mana pun menjadi jadwal terurut [3].
- **Memberi kembalian.** Untuk rupiah, terbesar lebih dulu adalah optimal. Untuk himpunan pecahan lain mana pun, periksalah dengan **Kode 10.1** sebelum mengandalkannya.

Untuk lebih banyak soal greedy beserta pembuktiannya, Kleinberg dan Tardos [3] memberikan pembahasan argumen pertukaran yang paling cermat, dan Laaksonen [13] mengumpulkan pola-pola greedy yang paling umum dalam kontes.

10.11 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

- **Memilih kunci berdasarkan intuisi.** Mulai paling awal dan terpendek lebih dulu sama-sama terdengar benar untuk pemilihan kegiatan dan sama-sama salah. Ujilah setiap calon kunci terhadap *brute force*; kunci yang benar biasanya kunci yang dapat Anda buktikan.
- **Perbandingan ketat alih-alih tidak ketat.** Pada BUSYMAN sebuah kegiatan boleh dimulai ketika kegiatan sebelumnya selesai: pemeriksaannya `s >= freeAt`, bukan `s > freeAt`. Carilah dalam pernyataan soal kata-kata "boleh dimulai pada saat yang sama".
- **Mempercayai greedy karena lolos contoh.** Terpendek lebih dulu hanya gagal pada 1,6% kasus kecil acak; contoh uji jarang memuat contoh penyangkalnya. Lakukan *stress test*.
- **Memakai greedy pada soal 0/1.** Nilai per satuan adalah aturan yang benar bila barang boleh dipotong, dan salah bila tidak boleh. Knapsack pada Bab 9 memerlukan DP.
- **Luapan pada penjumlahan.** BAISED dapat mencapai sekitar 5×10^9 dan GERGOVIA sekitar 2.5×10^{12}. Gunakan `long long`.
- **Tersandung format masukan.** Baris kosong di antara kasus uji dan nama sebelum bilangan tidak berbahaya dengan pembacaan berbasis token, tetapi mematikan dengan pembacaan berbasis baris yang mengharapkan bilangan pada setiap baris.
- **Mengurutkan salinan tetapi memakai yang asli.** Ketika greedy memerlukan unsur-unsur dalam urutan baru, pastikan setiap langkah berikutnya memakai urutan yang sudah diurutkan, atau simpan indeks aslinya dalam pasangan.

10.12 Ringkasan Bab

- Algoritma greedy mengambil pilihan lokal terbaik pada setiap langkah dan tidak pernah menariknya kembali. Ia cepat dan ringkas, dan benar hanya bila struktur soalnya mengizinkan.
- Argumen pertukaran membuktikan aturan greedy: tukarkan penyelesaian optimal, langkah demi langkah, menjadi penyelesaian greedy tanpa memperburuknya. Untuk pemilihan kegiatan langkah kuncinya adalah Persamaan (10.1); untuk pemasangan, keuntungan meluruskan silang Persamaan (10.3); untuk peringkat, penugasan terurut Persamaan (10.4); untuk pengangkutan, batas bawah yang tercapai Persamaan (10.5).
- Satu contoh penyangkal cukup untuk menggugurkan aturan greedy, dan *brute force* pada masukan kecil acak menemukannya dengan cepat: **Kode 10.2** menolak dua dari tiga aturan yang masuk akal, dan **Kode 10.1** memperlihatkan bahwa koin terbesar lebih dulu benar untuk rupiah tetapi salah untuk koin 1, 3, 4.
- BUSYMAN, FASHION, BAISED, dan GERGOVIA masing-masing menjadi satu pengurutan dan satu gelung dalam C++17 dan Python 3, diperiksa pada 1200 masukan acak terhadap *brute force*. C++ cepat di mana-mana; Python baik-baik saja kecuali pada batas waktu BUSYMAN yang ketat.

10.13 Soal Latihan

1. Jalankan aturan selesai paling awal dengan tangan pada kegiatan [1,4), [3,5), [0,6), [5,7), [3,9), [5,9), [6,10), [8,11), [8,12), [2,14), [12,16). Kegiatan mana yang diambil, dan berapa banyak?
2. Apakah aturan "mulai paling akhir lebih dulu" (ambil kegiatan yang mulai paling akhir, lalu yang mulai paling akhir di antara yang selesai tepat waktu, dan seterusnya, bergerak mundur sepanjang hari) benar untuk BUSYMAN? Buktikan atau berikan contoh penyangkal.
3. Sebuah sistem koin memiliki koin 1, 5, dan 12. Temukan jumlah terkecil yang membuat pemberian kembalian secara greedy tidak optimal, dan berikan kedua jawabannya.

4. Pada *fractional knapsack* (knapsack pecahan), barang boleh dipotong, dan potongan seberat x dari barang seberat w yang bernilai v berharga $x \cdot v/w$. Tunjukkan dengan argumen pertukaran bahwa mengambil barang menurut urutan nilai per kilogram adalah optimal, lalu gunakan **Kode 10.11** untuk membandingkan jawaban pecahan dan jawaban 0/1 untuk tiga barang seberat 10, 20, dan 30 kg yang bernilai 60, 100, dan 120, dengan kapasitas 50 kg.
5. Untuk FASHION, aturan apa yang memberi jumlah hasil kali *terkecil*? Buktikan dengan Persamaan (10.3).
6. Pada GERGOVIA, mengapa urutan pengangkutan botol yang sebenarnya tidak berpengaruh? Jelaskan dengan Persamaan (10.5).

10.14 Penyelesaian Soal Latihan

1. Menurut waktu selesai: [1,4), [3,5), [0,6), [5,7), [3,9), [5,9), [6,10), [8,11), [8,12), [2,14), [12,16). Ambil [1,4), luang pada 4. [3,5) dan [0,6) mulai terlalu awal. Ambil [5,7), luang pada 7. [3,9), [5,9), dan [6,10) mulai sebelum 7. Ambil [8,11), luang pada 11. [8,12) dan [2,14) mulai terlalu awal. Ambil [12,16). Empat kegiatan: [1,4), [5,7), [8,11), [12,16). Programnya sepakat:

```
$ printf '1\n11\n1 4\n3 5\n0 6\n5 7\n3 9\n5 9\n6 10\n8 11\n8 12\n2 14\n12 16\n' |
./busyman
```

```
4
```

2. Ya. Balikkan waktu: ganti setiap kegiatan $[s, e)$ dengan $[-e, -s)$. Dua kegiatan tumpang-tindih sebelum pembalikan tepat bila keduanya tumpang-tindih sesudahnya, sehingga banyaknya kegiatan terbaik tidak berubah. Pada hari yang dibalik, "mulai paling akhir lebih dulu" menjadi "selesai paling awal lebih dulu", yang sudah dibuktikan optimal pada Subbab 10.4.2. Argumen pertukarannya juga dapat ditulis langsung: kegiatan yang mulai paling akhir dapat menggantikan kegiatan terakhir dari jadwal optimal mana pun, karena ia tidak mulai lebih awal, sehingga tidak mungkin tumpang-tindih dengan kegiatan sebelumnya.

3. Jumlah 15: greedy mengambil $12 + 1 + 1 + 1$, empat koin, sedangkan $5 + 5 + 5$ memakai tiga. Jumlah yang lebih kecil semuanya baik: di bawah 12 greedy hanya memakai koin 5 dan 1, yang optimal, dan untuk 12, 13, dan 14 ia memakai 12 ditambah satu, dua, atau tiga koin 1, yang tidak dapat dikalahkan, karena tanpa koin 12 bahkan 14 memerlukan $5 + 5 + 1 + 1 + 1 + 1$. Mengubah `compare` pada **Kode 10.1** menjadi koin {1,5,12} memastikan bahwa 15 adalah kegagalan pertama.

4. Misalkan suatu muatan optimal berisi lebih sedikit barang berlaju nilai per kilogram tertinggi, r_1, daripada yang diambil greedy, sementara ia berisi sejumlah berat barang dengan laju lebih rendah $r_2 < r_1$. Tukarkan δ kilogram barang kedua dengan δ kilogram barang pertama: beratnya tidak berubah dan nilainya naik sebesar $\delta(r_1 - r_2) > 0$, sebuah kontradiksi. Jadi muatan optimal mengambil sebanyak mungkin barang berlaju terbaik, lalu barang berikutnya, dan itulah aturan greedy. **Kode 10.11** memperlihatkan angkanya:

Kode 10.11 `fractional.cpp`: knapsack pecahan menurut nilai per kilogram, berdampingan dengan jawaban 0/1 yang ditemukan brute force.

```cpp
// Exercise 10.4: fractional knapsack (goods may be cut) solved greedily by
// value per kilogram, next to 0/1 knapsack on the same goods, where it fails.
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

struct Good {
    double kg, value;
};

int main() {
    vector<Good> g = {{10, 60}, {20, 100}, {30, 120}};
    double cap = 50;
    sort(g.begin(), g.end(),
         [](Good a, Good b) { return a.value / a.kg > b.value / b.kg; });
    double room = cap, total = 0;
    for (Good x : g) {
        double take = min(room, x.kg);  // as much of the best-per-kg good as fits
        total += take / x.kg * x.value;
        room -= take;
        printf("take %4.1f kg of the %2.0f kg good\n", take, x.kg);
    }
    printf("fractional (greedy by value per kg): %.1f\n", total);
    // 0/1 version: the same greedy takes 10 and 20 kg (value 160) and stops.
    double best = 0;
    for (int mask = 0; mask < 8; mask++) {
        double kg = 0, value = 0;
        for (int i = 0; i < 3; i++)
            if (mask >> i & 1) kg += g[i].kg, value += g[i].value;
        if (kg <= cap) best = max(best, value);
    }
    printf("0/1 (brute force over 8 subsets):    %.1f\n", best);
    return 0;
}
```

```
$ ./fractional
```

```
take 10.0 kg of the 10 kg good
take 20.0 kg of the 20 kg good
take 20.0 kg of the 30 kg good
fractional (greedy by value per kg): 240.0
0/1 (brute force over 8 subsets):    220.0
```

Jawaban pecahannya 240: seluruh barang berlaju 6 dan 5 per kilogram, dan 20 dari 30 kg barang berlaju 4. Untuk barang utuh yang terbaik adalah 220 (barang 20 kg dan 30 kg), dan urutan greedy akan mengambil barang 10 kg dan 20 kg dengan nilai hanya 160 sebelum kehabisan ruang untuk barang ketiga. Argumen pertukaran memerlukan kebebasan menukar δ kilogram; barang utuh tidak mengizinkannya, dan justru itulah sebabnya 0/1 knapsack memerlukan DP.

5. Urutkan satu daftar menaik dan yang lain menurun, sehingga pria bernilai tertinggi dipasangkan dengan wanita bernilai terendah. Menurut Persamaan (10.3), meluruskan dua pasangan *menaikkan* jumlah sebesar $(a_2 - a_1)(b_2 - b_1) \geq 0$, sehingga *menyilangkannya* menurunkan jumlah sebesar itu pula. Dimulai dari pemasangan mana pun, menyilangkan setiap pasangan yang lurus tidak pernah menaikkan jumlah dan berakhir pada pemasangan bersilang penuh dengan urutan berlawanan, yang karenanya adalah minimum.

6. Persamaan (10.5) hanya bergantung pada berapa botol yang melintasi setiap batas, bukan botol yang mana atau dengan urutan apa botol itu berjalan. Rencana apa pun yang membuat tepat $|P_i|$ botol melintasi batas i, semuanya searah, berbiaya tepat sebesar batas itu. Rencana yang mengirim botol bolak-balik melintasi sebuah batas, atau lebih banyak botol daripada yang diperlukan, membayar lebih, dan hanya rencana-rencana itulah yang tidak optimal.

Daftar Pustaka

[1] Sphere Online Judge, "BUSYMAN - I AM VERY BUSY." [Online]. Available: https://www.spoj.com/problems/BUSYMAN/

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022, ch. 15.

[3] J. Kleinberg and É. Tardos, *Algorithm Design*. Boston, MA, USA: Pearson/Addison-Wesley, 2006, ch. 4 and 6.

[4] Bank Indonesia, "Bank Indonesia dan Pemerintah Meluncurkan Uang Rupiah Kertas Tahun Emisi 2022," press release, Aug. 18, 2022. [Online]. Available: https://www.bi.go.id/id/publikasi/ruang-media/news-release/Pages/sp_2421922.aspx

[5] Kompas.com, "Daftar Uang Rupiah Ini Masih Berlaku, Tak Boleh Ditolak Saat Transaksi," Oct. 8, 2024. [Online]. Available: https://www.kompas.com/tren/read/2024/10/08/183000765/daftar-uang-rupiah-ini-masih-berlaku-tak-boleh-ditolak-saat-transaksi

[6] D. Kozen and S. Zaks, "Optimal bounds for the change-making problem," *Theoretical Computer Science*, vol. 123, no. 2, pp. 377–388, 1994.

[7] D. Pearson, "A polynomial-time algorithm for the change-making problem," *Operations Research Letters*, vol. 33, no. 3, pp. 231–234, 2005.

[8] Sphere Online Judge, "FASHION - Fashion Shows." [Online]. Available: https://www.spoj.com/problems/FASHION/

[9] G. H. Hardy, J. E. Littlewood, and G. Pólya, *Inequalities*, 2nd ed. Cambridge, U.K.: Cambridge University Press, 1952, Sec. 10.2.

[10] Sphere Online Judge, "BAISED - Biased Standings." [Online]. Available: https://www.spoj.com/problems/BAISED/

[11] Sphere Online Judge, "GERGOVIA - Wine trading in Gergovia." [Online]. Available: https://www.spoj.com/problems/GERGOVIA/

[12] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[13] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

Bab 11

Binary Search pada Jawaban dan Dua Penunjuk

Sebagian soal optimasi sulit dijawab secara langsung tetapi mudah *diperiksa*: "berapa jarak terbesar yang mungkin?" itu sulit, sedangkan "apakah jarak 300 mungkin?" dapat dijawab dengan satu kali lintasan atas data. Bila jawaban pertanyaan mudah itu hanya berganti dari ya ke tidak satu kali ketika bilangannya membesar, *binary search* (pencarian biner) atas bilangan tersebut menemukan nilai terbaik dengan segelintir pemeriksaan. Teknik ini, *binary search pada jawaban*, mengubah banyak soal "maksimalkan yang terkecil" dan "minimalkan yang terbesar" menjadi program pendek. Sepupunya, teknik *dua penunjuk* (*two pointers*), menggerakkan dua indeks searah di sepanjang larik dan mengganti pencarian kuadratik atas semua sublarik dengan pencarian linear. Di akhir bab ini Anda akan mampu mengenali pertanyaan ya/tidak yang monoton di dalam sebuah soal, menulis binary search yang tidak pernah berputar selamanya dan tidak pernah meleset dari batasnya, melakukan hal yang sama pada bilangan real, membangun kembali penyelesaian yang sebenarnya setelah nilai terbaiknya diketahui, dan menyelesaikan lima soal SPOJ dalam C++17 dan Python 3: AGGRCOW, EKO, PIE, BOOKS1, dan HOTELS.

11.1 Pasar Malam Bu Ratna

Setiap Sabtu malam Bu Ratna menata pasar malam di sepanjang jalan utama kotanya. Di jalan itu ada sederet titik tempat lapak bisa berdiri, dengan jarak yang diketahui dari masjid di ujung jalan. Minggu ini empat penjual martabak mendaftar, dan setiap orang meminta hal yang sama: "Tolong jangan taruh saya persis di sebelah lapak martabak lain." Bu Ratna ingin adil. Ia memutuskan menempatkan keempat lapak martabak sehingga jarak *terkecil* antara dua lapak mana pun sebesar mungkin.

Dengan dua belas titik yang tersedia, ia bisa mencoba setiap pilihan empat titik, tetapi itu berarti 495 pilihan untuk empat lapak, dan jauh terlalu banyak untuk pasar sungguhan dengan seratus titik dan dua puluh penjual. Ada cara berpikir yang lebih baik. Alih-alih bertanya "berapa jarak terkecil yang terbaik?", Bu Ratna mengajukan pertanyaan yang lebih sederhana: "dapatkah keempat lapak ditempatkan sehingga setiap dua lapak berjarak paling sedikit 300 meter?" Pertanyaan itu mudah. Berjalanlah menyusuri jalan, taruh lapak pertama di titik pertama, lalu taruh setiap lapak berikutnya di titik pertama yang berjarak paling sedikit 300 meter dari lapak sebelumnya. Bila keempat lapak muat, jawabannya ya.

Dan pertanyaan mudah itu punya bentuk yang berguna: bila 300 meter mungkin, maka 250 meter pasti mungkin juga, dan bila 320 meter tidak mungkin, maka 400 meter juga tidak mungkin. Jawaban-jawabannya membentuk satu blok ya diikuti satu blok tidak. Menemukan tempat blok ya berakhir adalah pekerjaan binary search.

Pertanyaan Bu Ratna adalah soal SPOJ AGGRCOW (Aggressive cows) [1], dengan kandang sapi sebagai pengganti titik lapak. Soal itu membuka bab ini, lalu gagasan yang sama menyelesaikan soal penggergajian kayu, soal kue ulang tahun, dan soal kantor penyalinan.

11.2 Gagasannya dengan Kata-Kata Sederhana

11.2.1 Pertanyaan yang berganti jawaban sekali

Misalkan $ok(d)$ adalah pertanyaan ya/tidak "dapatkah lapak-lapak ditempatkan dengan setiap dua lapak berjarak paling sedikit d?". Bila $ok(d)$ benar, maka $ok(d')$ benar untuk setiap d' yang lebih kecil, karena penempatan yang menjaga jarak d juga menjaga setiap jarak yang lebih kecil. Pertanyaan dengan sifat ini disebut *monoton*.

> **DEFINISI.** Pertanyaan ya/tidak $ok(x)$ tentang sebuah bilangan x disebut *monoton* bila jawaban-jawabannya, untuk x yang menaik, berupa satu blok *ya* diikuti satu blok *tidak* (atau sebaliknya). *Binary search pada jawaban* menemukan batas antara kedua blok itu dengan menyimpan dua nilai, lo dengan $ok(lo)$ benar dan hi dengan $ok(hi)$ salah, lalu berulang kali mengajukan pertanyaan di tengah-tengahnya [2].

Pencarian itu memegang satu janji, yaitu *invarian*-nya, dari awal sampai akhir:

$$ok(lo) = \text{ya}, \qquad ok(hi) = \text{tidak}, \qquad lo < hi \tag{11.1}$$

Setiap langkah memilih mid yang terletak tegas di antara lo dan hi lalu menanyakan $ok(mid)$. Bila jawabannya ya, mid menjadi lo yang baru; bila tidak, ia menjadi hi yang baru. Dengan cara mana pun Persamaan (11.1) tetap berlaku dan selisih $hi - lo$ terbelah dua. Ketika selisihnya 1, kedua nilai bertetangga, sehingga lo adalah ya yang *terbesar* dan hi tidak yang terkecil. Banyaknya pertanyaan yang diajukan adalah

$$\left\lceil \log_2(hi_0 - lo_0) \right\rceil \tag{11.2}$$

dengan lo_0 dan hi_0 sebagai nilai awal. Untuk rentang 10^9 itu berarti 30 pertanyaan; untuk 10^{18}, 60.

Tiga hal kecil membuat templat ini aman, dan masing-masing mencegah satu *bug* klasik:

- **Mulailah dengan ya yang sungguhan dan tidak yang sungguhan.** Pilih lo_0 yang jawabannya pasti ya (untuk AGGRCOW, jarak 0 selalu berhasil) dan hi_0 yang jawabannya pasti tidak (jarak yang lebih besar daripada seluruh jalan). Dengan begitu gelung tidak perlu menebak apa yang terjadi di kedua ujung.
- **Berhentilah ketika selisihnya 1**, dengan pengujian `hi - lo > 1`. Karena mid selalu terletak tegas di antara keduanya, setiap langkah memperkecil selisih, dan gelung tidak mungkin berputar selamanya, kegagalan yang umum pada versi dengan `lo <= hi` serta `mid + 1` atau `mid - 1` yang salah tempat.
- **Hitung titik tengah sebagai `lo + (hi - lo) / 2`.** Bentuk yang lebih jelas, `(lo + hi) / 2`, dapat meluap ketika keduanya besar, sebuah *bug* yang bertahun-tahun tidak ketahuan di dalam kode pustaka yang dipakai luas [3].

> **WAWASAN KUNCI:** Binary search pada jawaban mengubah pertanyaannya. Alih-alih menghitung nilai terbaik, Anda cukup *memeriksa* sebuah nilai calon. Pemeriksaan biasanya berupa satu lintasan greedy sederhana; binary search hanya menambahkan faktor sekitar 30.

11.2.2 Templat yang sama pada larik terurut

Binary search klasik, yaitu mencari sebuah nilai dalam larik terurut, adalah templat yang sama dengan pertanyaan tertentu. Untuk larik terurut $a_0 \le a_1 \le \cdots \le a_{n-1}$ dan nilai x, ajukan kepada setiap indeks i pertanyaan: "apakah $a_i \ge x$?". Karena lariknya terurut, jawabannya berupa satu

blok tidak diikuti satu blok ya, dan ya yang pertama adalah posisi tempat x berada, atau tempat ia akan disisipkan. Posisi itulah yang dikembalikan `std::lower_bound` di C++ dan `bisect.bisect_left` di Python.

Kerjakan dengan tangan pada $a = (3,8,8,12,20)$ dengan $x = 8$. Indeks khayalan -1 dianggap tidak dan indeks khayalan 5 dianggap ya, jadi mulailah dengan $lo = -1$ dan $hi = 5$. Titik tengahnya $-1 + \lfloor 6/2 \rfloor = 2$, dan $a_2 = 8 \geq 8$, ya, jadi $hi = 2$. Berikutnya $mid = -1 + \lfloor 3/2 \rfloor = 0$, dan $a_0 = 3 \geq 8$ dijawab tidak, jadi $lo = 0$. Berikutnya $mid = 1$, dan $a_1 = 8$, ya, jadi $hi = 1$. Selisihnya 1 dan jawabannya indeks 1, angka 8 yang pertama. Dengan $x = 10$ langkah yang sama berakhir di indeks 3, tempat 10 akan disisipkan di antara 8 dan 12. Menghitung berapa kali sebuah nilai muncul lalu cukup dengan dua pencarian: unsur pertama yang $\geq x$ dan unsur pertama yang $> x$, yang selisihnya adalah banyaknya kemunculan. Satu-satunya yang berubah dari pencarian AGGRCOW adalah pertanyaannya; gelungnya, invarian Persamaan (11.1), dan banyaknya langkah Persamaan (11.2) tetap sama.

11.2.3 Bila ujung atas tidak diketahui

Kadang-kadang sebuah soal tidak memberikan batas yang nyaman untuk hi_0, atau batasnya begitu besar sehingga sebagian besar rentangnya jelas tidak berguna. Dalam keadaan itu temukan ujung atasnya dengan *penggandaan*: tanyakan $ok(1)$, $ok(2)$, $ok(4)$, $ok(8)$, dan seterusnya, sampai tidak yang pertama. Bila jawaban sebenarnya adalah A, penggandaan berhenti sesudah sekitar $\log_2 A$ pertanyaan pada nilai di bawah $2A$, dan binary search antara ya terakhir dan tidak pertama memerlukan sekitar $\log_2 A$ pertanyaan lagi. *Exponential search* (pencarian eksponensial) ini berbiaya sekitar $2\log_2 A$ pertanyaan secara keseluruhan dan tidak pernah perlu tahu ukuran rentangnya lebih dulu. Cara ini juga pilihan yang aman ketika pemeriksaan menjadi mahal untuk nilai besar, misalnya ketika memeriksa sebuah calon berarti menyimulasikan hari sebanyak itu.

Satu catatan tentang istilah. Kita membiarkan *binary search* dalam bentuk Inggrisnya, karena dalam percakapan pemrogram di Indonesia istilah itu jauh lebih lazim daripada "pencarian biner", meskipun keduanya bermakna sama. Gagasannya sudah sangat akrab dalam kehidupan sehari-hari. Ketika mencari nama dalam buku telepon atau kata dalam kamus tebal, tak seorang pun membalik halaman satu per satu dari depan: kita membuka di tengah, melihat apakah kata yang dicari berada di depan atau di belakang, lalu mengulanginya pada separuh yang tersisa. Permainan tebak angka juga begitu: "angkanya antara 1 dan 100; lebih besar atau lebih kecil dari 50?" Paling banyak tujuh pertanyaan sudah cukup. Hal yang baru dalam bab ini hanyalah bahwa yang dibelah dua bukan halaman kamus, melainkan rentang *jawaban* yang mungkin, dan yang ditanyakan pada setiap langkah bukan "lebih besar atau lebih kecil?", melainkan "apakah nilai ini masih mungkin?". Istilah *two pointers* kita sebut "dua penunjuk", dan *monotone* kita sebut "monoton". Semua istilah itu akan terasa wajar setelah Anda menulis beberapa program sendiri; yang lebih penting adalah kebiasaan mencari pertanyaan ya/tidak yang tersembunyi di balik setiap soal optimasi.

11.3 Percobaan Pertama: Coba Setiap Nilai

Cara langsung untuk menemukan ya yang terbesar adalah mencoba calon satu per satu: mulai dari 1 dan naik selama nilai berikutnya masih berhasil. **Kode 11.1** melakukannya untuk pasar malam Bu Ratna, dua belas titik antara 1 dan 1000 meter dan empat lapak martabak, lalu mengerjakan hal yang sama dengan binary search. Program ini menghitung banyaknya pemanggilan pemeriksaan.

Kode 11.1 `bsearch_demo.cpp`: pemeriksaan untuk sebuah jarak, pemindaian linear atas semua jarak, dan binary search, dengan pemeriksaan yang dihitung.

```cpp
// A yes/no question that changes its answer only once, and two ways to find
// the change: try every value, or halve the range each time.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

vector<long long> pos = {1, 40, 95, 180, 260, 333, 410, 505, 590, 700, 810, 1000};
const int COWS = 4;
int checks = 0;

bool canPlace(long long gap, int cows) {  // can every two cows be >= gap apart?
    checks++;
    int placed = 1;
    long long last = pos[0];
    for (size_t i = 1; i < pos.size(); i++)
        if (pos[i] - last >= gap)
            placed++, last = pos[i];  // first stall far enough
    return placed >= cows;
}

int main() {
    sort(pos.begin(), pos.end());
    cout << "gap   ";
    for (int g = 240; g <= 340; g += 20) cout << g << ' ';
    cout << "\nfits? ";
    for (int g = 240; g <= 340; g += 20)
        cout << (canPlace(g, COWS) ? "yes " : "no  ");
    cout << '\n';

    checks = 0;
    long long g = 1;
    while (canPlace(g + 1, COWS)) g++;  // walk up while the next gap still works
    cout << "linear scan:   largest gap " << g << " after " << checks
         << " checks\n";
```

Kode 11.1 (Lanjutan) *bsearch_demo.cpp: pemeriksaan untuk sebuah jarak, pemindaian linear atas semua jarak, dan binary search, dengan pemeriksaan yang dihitung.*

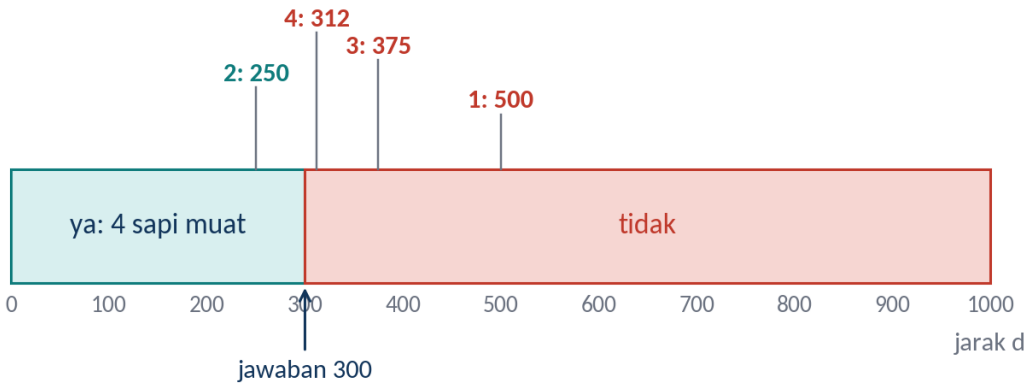
```cpp
    checks = 0;
    long long lo = 0,
              hi = pos.back() - pos.front() + 1;  // canPlace(lo), !canPlace(hi)
    while (hi - lo > 1) {
        long long mid = lo + (hi - lo) / 2;
        if (canPlace(mid, COWS))
            lo = mid;
        else
            hi = mid;
    }
    cout << "binary search: largest gap " << lo << " after " << checks
         << " checks\n";
    return 0;
}
```

```
$ g++ -std=c++17 -O2 -Wall -Wextra -o bsearch_demo bsearch_demo.cpp
$ ./bsearch_demo
```

```
gap   240 260 280 300 320 340
fits? yes yes yes yes no  no
linear scan:   largest gap 300 after 300 checks
binary search: largest gap 300 after 10 checks
```

Dua baris pertama memperlihatkan bentuk monotonnya: ya sampai 300, tidak mulai 320. Pemindaian linear memerlukan 300 pemeriksaan untuk menemukan batasnya; binary search memerlukan 10, tepat $\lceil \log_2 1000 \rceil$ menurut Persamaan (11.2). Dengan posisi sampai 10^9, pemindaian memerlukan sampai satu miliar pemeriksaan, masing-masing satu lintasan atas 10^5 kandang; binary search tetap memerlukan sekitar 30 lintasan.

Gambar 11.1 memperlihatkan empat pertanyaan pertama binary search itu. Yang pertama, pada 500, dijawab tidak; yang kedua, pada 250, ya; lalu 375 dan 312 keduanya tidak. Sesudah empat pertanyaan, jawabannya diketahui terletak antara 250 dan 312, dan enam pertanyaan lagi (281, 296, 304, 300, 302, 301) menutup selisihnya menjadi $lo = 300$, $hi = 301$.



Gambar 11.1 Jawaban atas pertanyaan "dapatkah empat lapak menjaga jarak d?" membentuk satu blok ya dan satu blok tidak; tanda bernomor adalah empat pertanyaan pertama binary search.

11.4 Memaksimalkan yang Terkecil (AGGRCOW)

Soalnya dengan kata-kata kita. AGGRCOW [1] punya beberapa kasus uji. Setiap kasus memberikan N posisi kandang pada sebuah garis, $2 \le N \le 10^5$, masing-masing antara 0 dan 10^9, dan banyaknya sapi C, $2 \le C \le N$. Tempatkan sapi-sapi di kandang yang berbeda sehingga jarak terkecil antara dua sapi sebesar mungkin, lalu cetak jarak itu.

Pemeriksaannya. Untuk jarak d, urutkan kandang dan berjalanlah dari kiri ke kanan: sapi pertama masuk ke kandang pertama, dan setiap sapi berikutnya ke kandang pertama yang berjarak paling sedikit d dari sapi sebelumnya. Bila seluruh C sapi muat, $ok(d)$ bernilai ya.

Mengapa menempatkan setiap sapi *sejauh mungkin ke kiri* itu aman? Ini argumen pertukaran kecil bergaya Bab 10. Ambil penempatan sah mana pun dengan jarak d dan bandingkan dengan penempatan greedy, sapi demi sapi. Sapi greedy pertama berada di kandang paling kiri, tidak lebih ke kanan daripada sapi pertama penempatan lain itu. Bila sapi greedy ke-j tidak lebih ke kanan daripada sapi ke-j penempatan lain, maka sapi ke-$(j+1)$ penempatan lain, yang berjarak paling sedikit d dari sapi ke-j-nya, juga berjarak paling sedikit d dari sapi greedy ke-j; jadi kandang sejauh itu ada, dan sapi greedy ke-$(j+1)$ ditempatkan tidak lebih ke kanan. Dengan induksi, greedy menempatkan setiap sapi yang ditempatkan penempatan lain, sehingga bila ada penempatan yang sah, greedy menemukan satu.

Pencariannya. Jarak 0 selalu berhasil, jadi $lo_0 = 0$. Jarak yang lebih besar daripada bentang kandang tidak pernah berhasil, jadi $hi_0 = x_{\max} - x_{\min} + 1$. Pengurutan berbiaya $O(N \log N)$ dan setiap pemeriksaan $O(N)$, sehingga totalnya $O(N \log N + N \log X)$, dengan $X \le 10^9$ sebagai bentangnya.

Kode 11.2 `aggrcow.cpp`: binary search pada jarak dengan pemeriksaan greedy.

```cpp
// AGGRCOW (Aggressive cows): put C cows in N stalls on a line so that the
// smallest distance between two cows is as large as possible.
// Binary search on the answer d with a greedy yes/no check.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

bool fits(const vector<long long>& x, int cows, long long d) {
    int placed = 1;  // the first cow always goes to the leftmost stall
    long long last = x[0];
    for (size_t i = 1; i < x.size() && placed < cows; i++)
        if (x[i] - last >= d) {
            placed++;
            last = x[i];
        }
    return placed >= cows;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        int n, c;
        cin >> n >> c;
        vector<long long> x(n);
        for (auto& v : x) cin >> v;
        sort(x.begin(), x.end());
        long long lo = 0,
                  hi = x[n - 1] - x[0] + 1;  // fits(lo) is true, fits(hi) false
        while (hi - lo > 1) {
            long long mid = lo + (hi - lo) / 2;
            if (fits(x, c, mid))
                lo = mid;
            else
                hi = mid;
        }
        cout << lo << '\n';
    }
    return 0;
}
```

Kode 11.3 `aggrcow.py`: pencarian yang sama dalam Python, dengan keluar lebih awal begitu semua sapi muat.

```python
"""AGGRCOW (Aggressive cows): make the smallest distance between two of the
C cows as large as possible. Binary search on the answer, greedy check."""

import sys


def fits(x: list[int], cows: int, d: int) -> bool:
    placed, last = 1, x[0]
    for v in x:
        if v - last >= d:  # the first stall far enough from the last cow
            placed += 1
            last = v
            if placed == cows:
                return True
    return placed >= cows


def main() -> None:
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    pos, out = 1, []
    for _ in range(t):
        n, c = int(data[pos]), int(data[pos + 1])
        x = sorted(map(int, data[pos + 2 : pos + 2 + n]))
        pos += 2 + n
        lo, hi = 0, x[-1] - x[0] + 1  # fits(lo) is true, fits(hi) is false
        while hi - lo > 1:
            mid = (lo + hi) // 2
            if fits(x, c, mid):
                lo = mid
            else:
                hi = mid
        out.append(str(lo))
    print("\n".join(out))


main()
```

Kode 11.2 menyimpan posisi dalam `long long` agar selisihnya aman, dan menghentikan pemeriksaan begitu semua sapi sudah ditempatkan. **Kode 11.3** keluar dari `fits` pada saat yang sama, dan itu penting di Python: bila banyak sapi sudah tertempatkan lebih awal, sebagian besar kandang tidak pernah dilihat. Uji kita memakai dua belas titik Bu Ratna dengan empat penjual dan sebuah jalan kedua yang lebih kecil dengan lima kandang dan tiga sapi.

```
$ printf '2\n12 4\n1\n40\n95\n180\n260\n333\n410\n505\n590\n700\n810\n1000\n5
3\n10\n1\n7\n3\n12\n' | ./aggrcow
$ printf '2\n12 4\n1\n40\n95\n180\n260\n333\n410\n505\n590\n700\n810\n1000\n5
3\n10\n1\n7\n3\n12\n' | python3 ../python/aggrcow.py
```

```
300
5
300
5
```

Untuk jalan kedua, yang terurut menjadi 1,3,7,10,12, jarak 5 menempatkan sapi di 1, 7, dan 12, sedangkan 6 hanya menempatkan dua (1 dan 7, dan tidak ada yang berjarak 6 dari 7). Jawabannya 5.

11.5 Gergaji Setinggi Mungkin (EKO)

Soalnya dengan kata-kata kita. Dalam EKO [4], seorang penebang memerlukan paling sedikit M meter kayu, $M \le 2 \times 10^9$. Ada $N \le 10^6$ pohon berderet dengan tinggi di bawah 10^9. Mesinnya

menaikkan gergaji ke ketinggian H dan memotong setiap pohon di atasnya; ia mengambil bagian di atas H. Karena menyayangi hutan, ia ingin H *setinggi* mungkin asalkan tetap mendapat paling sedikit M meter. Tinggi seluruh pohon bersama-sama lebih dari M. Cetak H terbaik.

Pertanyaan monotonnya adalah $ok(H)$: "apakah memotong pada ketinggian H memberi paling sedikit M meter?". Kayu yang terkumpul,

$$wood(H) = \sum_{i=1}^{N} \max(0,\ h_i - H) \tag{11.3}$$

hanya dapat menyusut ketika H naik, sehingga jawabannya ya sampai suatu ketinggian dan tidak di atasnya: bentuk yang sama dengan AGGRCOW. $H = 0$ memberi seluruh kayu, lebih dari M, jadi $lo_0 = 0$; pada tinggi pohon tertinggi tidak ada yang terpotong, jadi $hi_0 = h_{\max}$.

Jebakannya. Dua perangkap menunggu. Pertama, Persamaan (11.3) dapat mencapai $10^6 \times 10^9 = 10^{15}$, sehingga jumlahnya harus `long long`; dengan `int` jumlah itu meluap, berputar menjadi bilangan negatif, dan pencarian bergerak ke arah yang salah. Kedua, membaca sejuta bilangan adalah biaya nyata terhadap batas waktu soal yang ketat, sehingga **Kode 11.4** membaca masukan dalam blok 64 KB dengan `fread` dan mengurai digitnya sendiri, teknik dari Bab 4. Dengan 30 pemeriksaan atas sejuta pohon, seluruh pencarian berjumlah sekitar 3×10^7 langkah sederhana.

Kode 11.4 `eko.cpp`: binary search pada tinggi gergaji, dengan masukan per blok dan jumlah 64 bit.

```cpp
// EKO: set the saw height H as high as possible while the parts of the trees
// above H still give at least M meters of wood. Binary search on H.
// A million numbers arrive, so the input is read in large blocks (Chapter 4).
#include <cstdio>
#include <vector>
using namespace std;

static char buf[1 << 16];
size_t len = 0, at = 0;

int readChar() {
    if (at == len) {
        len = fread(buf, 1, sizeof buf, stdin);
        at = 0;
        if (len == 0) return -1;
    }
    return buf[at++];
}

long long readInt() {
    int ch = readChar();
    while (ch != -1 && (ch < '0' || ch > '9')) ch = readChar();
    long long v = 0;
    for (; ch >= '0' && ch <= '9'; ch = readChar()) v = v * 10 + (ch - '0');
    return v;
}

int main() {
    int n = readInt();
    long long m = readInt();
    vector<long long> h(n);
    long long top = 0;
    for (auto& v : h) {
        v = readInt();
        if (v > top) top = v;
    }
    long long lo = 0, hi = top;  // height lo gives enough wood, height hi does not
    while (hi - lo > 1) {
        long long mid = lo + (hi - lo) / 2, wood = 0;
        for (long long v : h)
            if (v > mid) wood += v - mid;  // can reach 10^15: needs long long
        if (wood >= m)
            lo = mid;
        else
            hi = mid;
    }
    printf("%lld\n", lo);
    return 0;
}
```

Kode 11.4 (Lanjutan) *eko.cpp: binary search pada tinggi gergaji, dengan masukan per blok dan jumlah 64 bit.*

Penyelesaian Python yang berbeda. Gelung Python atas sejuta pohon memakan sebagian detik yang cukup terasa, dan 30 gelung semacam itu jauh terlalu lambat. Karena itu **Kode 11.5** menghindari binary search. Ia mengurutkan tinggi dari yang tertinggi ke yang terendah, $h_1 \geq h_2 \geq \cdots \geq h_N$, dan menambahkan $h_{N+1} = 0$. Ketika gergaji berada di antara h_{k+1} dan h_k, tepat k pohon tertinggi yang terpotong, dan dengan $T_k = h_1 + \cdots + h_k$ kayunya adalah $T_k - kH$. H terbesar pada ruas itu yang masih memberi M meter adalah

$$H = \left\lfloor \frac{T_k - M}{k} \right\rfloor \tag{11.4}$$

asalkan nilainya paling kecil h_{k+1}. Berjalan untuk $k = 1,2,\ldots$ sampai hal itu terjadi memberi jawabannya dalam satu lintasan sesudah pengurutan.

Kode 11.5 `eko.py`: EKO tanpa binary search, dengan Persamaan (11.4) pada tinggi yang sudah terurut.

```
"""EKO: the highest saw height H that still yields at least M meters.
Python version: sort the trees once and walk down from the tallest, so no
binary search is needed (a binary search would read the million trees 30 times)."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    n, m = int(data[0]), int(data[1])
    h = sorted(map(int, data[2 : 2 + n]), reverse=True)
    h.append(0)  # the ground: cutting at 0 takes everything
    total = 0  # sum of the k tallest trees
    for k in range(1, n + 1):
        total += h[k - 1]
        # with the saw between h[k] and h[k-1], only the k tallest trees are cut:
        # wood(H) = total - k*H, and we need wood(H) >= m
        best = (total - m) // k
        if best >= h[k]:  # the best height lies in this stretch
            print(min(best, h[k - 1]))
            return


main()
```

Keduanya diuji pada enam pohon setinggi 30, 12, 25, 18, 40, dan 9, dengan kebutuhan 25 meter.

```
$ printf '6 25\n30 12 25 18 40 9\n' | ./eko
$ printf '6 25\n30 12 25 18 40 9\n' | python3 ../python/eko.py
```

```
23
23
```

Pada $H = 23$ tiga pohon tertinggi memberi $17 + 7 + 2 = 26$ meter, cukup; pada $H = 24$ ketiganya memberi $16 + 6 + 1 = 23$, kurang. Menurut Persamaan (11.4) dengan $k = 3$ dan $T_3 = 40 + 30 + 25 = 95$: $\lfloor (95 - 25)/3 \rfloor = 23$, dan $23 \geq h_4 = 18$, sehingga penelusuran berhenti di situ.

11.6 Binary Search pada Bilangan Real (PIE)

Soalnya dengan kata-kata kita. Dalam PIE [5], seorang tuan rumah punya N kue bundar setinggi 1 dengan jari-jari bilangan bulat sampai 10^4, dan F teman, $N, F \leq 10^4$. Setiap orang, yaitu F teman ditambah tuan rumah, harus mendapat potongan dengan volume yang sama, dan setiap potongan harus berasal dari satu kue (potongan tidak boleh disambung; sisa kue boleh). Cetak volume terbesar yang mungkin, dengan galat mutlak paling besar 10^{-3}.

Pertanyaan monotonnya: "dapatkah dipotong $F + 1$ potongan bervolume V?". Sebuah kue bervolume $v_i = \pi r_i^2$ menghasilkan $\lfloor v_i/V \rfloor$ potongan, sehingga

$$ok(V) \Leftrightarrow \sum_{i=1}^{N} \left\lfloor \frac{v_i}{V} \right\rfloor \geq F + 1 \tag{11.5}$$

Potongan yang lebih besar berarti potongan yang lebih sedikit, sehingga jawabannya ya sampai V terbaik dan tidak di atasnya.

Apa yang berubah pada bilangan real. Tidak ada "selisih 1" untuk berhenti, dan membandingkan `hi - lo > 1e-9` dapat berputar selamanya ketika pembulatan membuat titik tengah sama dengan salah satu ujung. Cara yang aman adalah *banyak putaran yang tetap*. Rentang dimulai dari volume kue terbesar, di bawah $\pi \cdot 10^8 \approx 3.1 \times 10^8$, dan setiap putaran membaginya dua; sesudah t putaran galatnya di bawah $3.1 \times 10^8 / 2^t$, yang kurang dari 10^{-3} begitu $t \geq 39$. Program C++ memakai 100 putaran dan program Python 60, keduanya jauh lebih banyak daripada yang diperlukan dan keduanya murah. Dua hal lagi: hitung π sebagai `acos(-1.0)`, yang tepat sampai presisi `double`, alih-alih mengetik 3.14159; dan pakailah `double`, jangan `float`, yang presisi 24 bitnya tidak sanggup menyimpan volume di sekitar 10^8 sampai tiga desimal [6].

Kode 11.6 `pie.cpp`: binary search pada bilangan real dengan banyak putaran yang tetap.

```cpp
// PIE: cut N round pies of height 1 into F+1 equal pieces, each piece from a
// single pie, as large as possible. Binary search on a real number.
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    const double PI = acos(-1.0);  // exact to double precision
    int t;
    if (!(cin >> t)) return 0;
    while (t--) {
        int n, f;
        cin >> n >> f;
        vector<double> vol(n);
        for (double& v : vol) {
            long long r;
            cin >> r;
            v = PI * r * r;
        }
        double lo = 0, hi = *max_element(vol.begin(), vol.end());
        for (int it = 0; it < 100; it++) {  // fixed rounds instead of hi - lo > eps
            double mid = (lo + hi) / 2;
            long long pieces = 0;
            for (double v : vol) pieces += (long long)(v / mid);
            if (pieces >= f + 1)
                lo = mid;  // f friends plus the host
            else
                hi = mid;
        }
        printf("%.4f\n", lo);
    }
    return 0;
}
```

Kode 11.7 `pie.py`: pencarian yang sama dalam Python.

```python
"""PIE: the largest equal piece for F+1 people, each piece from one pie.
Binary search on a real number with a fixed number of rounds."""

import math
import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    pos, out = 1, []
    for _ in range(t):
        n, f = int(data[pos]), int(data[pos + 1])
        vol = [math.pi * int(r) ** 2 for r in data[pos + 2 : pos + 2 + n]]
        pos += 2 + n
        lo, hi = 0.0, max(vol)
        for _ in range(60):  # 2**-60 of the range is far below 1e-3
            mid = (lo + hi) / 2
            if sum(int(v / mid) for v in vol) >= f + 1:
                lo = mid
            else:
                hi = mid
        out.append(f"{lo:.4f}")
    print("\n".join(out))


main()
```

Kode 11.6 menghitung potongan dalam `long long`, karena volume calon yang sangat kecil di awal pencarian membuat $\lfloor v_i/V \rfloor$ sangat besar; **Kode 11.7** mengandalkan bilangan bulat Python yang tak terbatas. Kita mencoba dua pesta: kue berjari-jari 5 dan 3 untuk lima teman, dan kue berjari-jari 1 sampai 4 untuk sepuluh teman.

```
$ printf '2\n2 5\n5 3\n4 10\n1 2 3 4\n' | ./pie
$ printf '2\n2 5\n5 3\n4 10\n1 2 3 4\n' | python3 ../python/pie.py
```

```
15.7080
7.1808
15.7080
7.1808
```

Pada pesta pertama enam orang berbagi kue bervolume $25\pi \approx 78.54$ dan $9\pi \approx 28.27$. Jawaban $5\pi \approx 15.708$ memotong kue besar menjadi tepat 5 potong dan kue kecil menjadi 1, jadi 6. Pada pesta kedua, sebelas orang dilayani dengan potongan $16\pi/7 \approx 7.1808$: kue terbesar memberi 7 potong, kue bervolume 9π memberi 3, kue 4π memberi 1, dan yang terkecil tidak memberi apa-apa. Pada kedua kasus jawabannya adalah sebuah kue yang dipotong menjadi bagian-bagian yang sama, dan dengan cara itu pula *brute force* pada Subbab 11.9 menemukannya.

11.7 Meminimalkan yang Terbesar, dan Membangun Kembali Jawabannya (BOOKS1)

Soalnya dengan kata-kata kita. Dalam BOOKS1 (Copying Books) [7], m buku dengan banyak halaman yang diketahui berdiri di rak dengan urutan tetap, $m \le 500$, masing-masing kurang dari 10^7 halaman, dan $k \le m$ penyalin akan menyalinnya. Setiap penyalin mendapat sederet buku yang berurutan, paling sedikit satu. Beban kerja seorang penyalin adalah total halamannya; pekerjaan selesai ketika penyalin tersibuk selesai. Bagilah buku-buku itu sehingga beban terbesar sekecil mungkin. Bila beberapa pembagian mencapainya, pilih yang memberi penyalin pertama beban sekecil mungkin, lalu penyalin kedua, dan seterusnya. Cetak buku-buku menurut urutan dengan `/` di antara deret milik para penyalin. Ada sekitar 200 kasus uji.

Pertanyaannya kini "minimalkan yang terbesar", bayangan cermin AGGRCOW, dan keluarannya bukan sebuah bilangan melainkan pembagian itu sendiri. Kedua hal itu baru.

Pertanyaan monotonnya. Untuk batas X, dapatkah buku-buku dibagi menjadi paling banyak k deret tanpa ada deret yang melebihi X halaman? Pemeriksaannya kembali greedy: berjalanlah dari kiri, masukkan buku ke deret saat ini selama masih dalam batas X, dan mulai deret baru ketika buku berikutnya akan membuatnya meluap. Greedy itu memakai deret sesedikit mungkin, dengan argumen "selalu di depan" yang sama seperti pada AGGRCOW: setiap deret greedy berakhir tidak lebih awal daripada deret yang bersesuaian pada pembagian sah lain mana pun. Jadi

$$ok(X) \Leftrightarrow runs(X) \leq k \tag{11.6}$$

dengan $runs(X)$ sebagai banyaknya deret yang diperlukan greedy. Batas yang lebih besar tidak pernah memerlukan deret lebih banyak, sehingga jawabannya tidak untuk X kecil dan ya mulai dari batas terbaik. Kali ini kita mencari ya yang *terkecil*, sehingga invariannya dicerminkan: lo selalu tidak, hi selalu ya, dan jawabannya adalah hi di akhir. Tidak ada batas di bawah buku tertebal yang dapat berhasil, jadi lo_0 adalah satu kurangnya dari buku tertebal; jumlah seluruh halaman selalu berhasil, jadi hi_0 adalah jumlah itu. (Bila deretnya lebih sedikit daripada penyalin, sebagian deret dapat dipecah lagi, karena setiap deret punya paling sedikit satu buku dan $k \leq m$; deret yang dipecah dua tidak pernah melampaui batas.)

Membangun kembali pembagiannya. Mengetahui batas terbaik X^* belum cukup; kita harus mencetak pembagiannya dengan aturan pemecah seri. Memberi penyalin pertama beban sekecil mungkin berarti memberi penyalin-penyalin *belakang* beban sebesar mungkin. Karena itu program mengisi deret dari *kanan*: berjalan dari buku terakhir ke buku pertama, program menambahkan buku ke deret saat ini selama masih dalam batas X^*, dan menutup deret itu (menaruh garis miring sebelumnya) ketika buku berikutnya akan membuatnya meluap, atau ketika buku yang tersisa di kiri tinggal cukup untuk memberi setiap penyalin yang tersisa satu buku. Syarat kedua itulah yang menjamin setiap penyalin mendapat paling sedikit satu buku. Mengisi dari kanan membuat deret penyalin terakhir sepanjang mungkin, sehingga halaman yang tersisa bagi yang lain sesedikit mungkin; bila diterapkan penyalin demi penyalin, cara ini meminimalkan beban penyalin pertama, lalu penyalin kedua, dan seterusnya. Karena argumen pemecah seri ini halus, *stress test* pada Subbab 11.9 membandingkan pembagian yang dicetak dengan *brute force* yang mencoba setiap pembagian dan menerapkan aturannya secara harfiah.

Kode 11.8 `books1.cpp`: binary search untuk batas terkecil, lalu pembagian yang dibangun kembali dari kanan.

```cpp
// BOOKS1 (Copying Books): split m books, in order, among k scribes so that the
// largest share of pages is least; among such splits, give the first scribe
// as little as possible, then the second, and so on. Print the split with " / ".
#include <iostream>
#include <vector>
using namespace std;

int groupsNeeded(const vector<long long>& p,
                 long long cap) {  // greedy, left to right
    int groups = 1;
    long long cur = 0;
    for (long long x : p) {
        if (cur + x > cap) groups++, cur = 0;
        cur += x;
    }
    return groups;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int cases;
    if (!(cin >> cases)) return 0;
    while (cases--) {
        int m, k;
        cin >> m >> k;
        vector<long long> p(m);
        long long lo = 0, hi = 0;  // the answer lies in (lo, hi]
        for (auto& x : p) {
            cin >> x;
            if (x > lo) lo = x;
            hi += x;
        }
        lo--;  // no share can be smaller than the thickest book
        while (hi - lo > 1) {
            long long mid = lo + (hi - lo) / 2;
            if (groupsNeeded(p, mid) <= k)
                hi = mid;
            else
                lo = mid;
        }
        long long cap = hi;
        // Fill from the right: later scribes take as much as the cap allows,
        // but leave at least one book for every scribe still to the left.
        vector<bool> slashAfter(m, false);
        int slashes = k - 1;
```

Kode 11.8 (Lanjutan) *books1.cpp: binary search untuk batas terkecil, lalu pembagian yang dibangun kembali dari kanan.*

```
        long long cur = 0;
        for (int i = m - 1; i >= 0; i--) {
            if (cur > 0 && (cur + p[i] > cap || i + 1 == slashes)) {
                slashAfter[i] = true;
                slashes--;
                cur = 0;
            }
            cur += p[i];
        }
        for (int i = 0; i < m; i++) {
            cout << p[i];
            if (i + 1 < m) cout << (slashAfter[i] ? " / " : " ");
        }
        cout << '\n';
    }
    return 0;
}
```

Kode 11.9 `books1.py`: pencarian dan pembangunan kembali yang sama dalam Python.

```
"""BOOKS1 (Copying Books): split the books in order among k scribes with the
least largest share; then give earlier scribes as little as possible."""

import sys


def groups_needed(p: list[int], cap: int) -> int:
    groups, cur = 1, 0
    for x in p:
        if cur + x > cap:
            groups, cur = groups + 1, 0
        cur += x
    return groups


def split(p: list[int], k: int) -> str:
    lo, hi = max(p) - 1, sum(p)  # the answer lies in (lo, hi]
    while hi - lo > 1:
        mid = (lo + hi) // 2
        if groups_needed(p, mid) <= k:
            hi = mid
        else:
            lo = mid
    slash_after = [False] * len(p)
    slashes, cur = k - 1, 0
    for i in range(len(p) - 1, -1, -1):  # fill from the right
        if cur > 0 and (cur + p[i] > hi or i + 1 == slashes):
            slash_after[i] = True
            slashes -= 1
            cur = 0
        cur += p[i]
    parts = []
    for i, x in enumerate(p):
        parts.append(str(x))
        if slash_after[i]:
            parts.append("/")
    return " ".join(parts)
```

Kode 11.9 (Lanjutan) *books1.py: pencarian dan pembangunan kembali yang sama dalam Python.*

```
def main() -> None:
    data = sys.stdin.buffer.read().split()
    pos, out = 1, []
    for _ in range(int(data[0])):
        m, k = int(data[pos]), int(data[pos + 1])
        out.append(split(list(map(int, data[pos + 2 : pos + 2 + m])), k))
        pos += 2 + m
    print("\n".join(out))


main()
```

Pada **Kode 11.8** syarat `i + 1 == slashes` berarti "buku $0..i$ tepat sebanyak garis miring yang masih harus dipasang", sehingga deret harus ditutup sekarang dan setiap buku yang tersisa akan menjadi satu-satunya buku seorang penyalin. **Kode 11.9** memakai pengujian yang sama. Kita memeriksa tiga rak.

```
$ printf '3\n7 3\n30 10 20 50 40 10 20\n4 4\n5 5 5 5\n6 2\n1 1 1 1 1 10\n' | ./books1
$ printf '3\n7 3\n30 10 20 50 40 10 20\n4 4\n5 5 5 5\n6 2\n1 1 1 1 1 10\n' | python3
../python/books1.py
```

```
30 10 / 20 50 / 40 10 20
5 / 5 / 5 / 5
1 1 1 1 1 / 10
30 10 / 20 50 / 40 10 20
5 / 5 / 5 / 5
1 1 1 1 1 / 10
```

Pada rak pertama, batas terkecil adalah 70: dengan 60 greedy memerlukan empat deret (30 10 20, 50, 40 10, 20). Pembagian yang dicetak memberi para penyalin 40, 70, dan 70 halaman. Penyalin pertama tidak mungkin mendapat lebih sedikit: dengan hanya 30 halaman untuk penyalin pertama, dua penyalin lainnya harus berbagi 150 halaman dengan paling banyak 70 masing-masing. Rak kedua adalah kasus ketika aturan "satu buku masing-masing" menentukan segalanya. Rak ketiga memperlihatkan bahwa buku-buku tipis dikumpulkan untuk penyalin pertama.

11.8 Dua Penunjuk (HOTELS)

Soalnya dengan kata-kata kita. Dalam HOTELS (Hotels Along the Croatian coast) [8], $N \leq 3 \times 10^5$ hotel berjajar di sepanjang pantai, masing-masing dengan harga positif di bawah 10^6, dan seorang pelancong punya $M < 2^{31}$ euro. Ia ingin menginap di sederet hotel yang berurutan, satu malam di setiap hotel, dengan membelanjakan uangnya sebanyak mungkin tanpa melampaui M. Cetak total terbesar yang tidak melebihi M (soal menjamin nilainya positif).

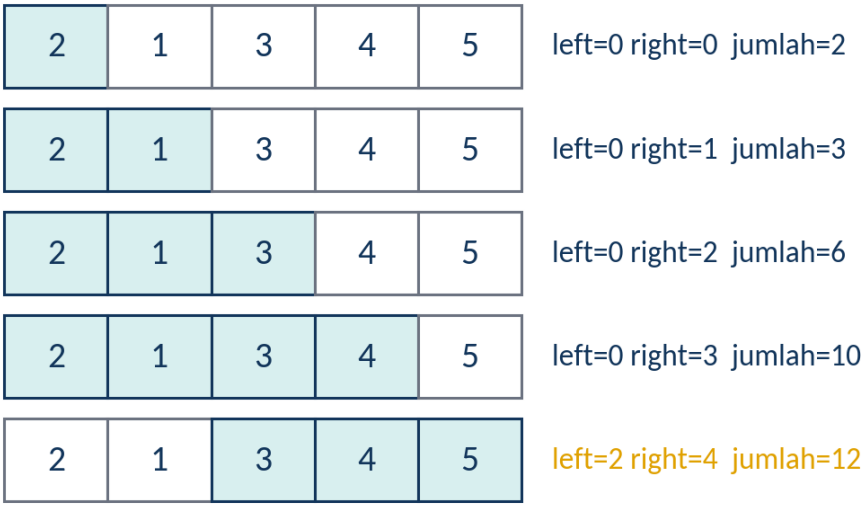
Mencoba semua $N(N+1)/2 \approx 4.5 \times 10^{10}$ deret mustahil. Binary search sebenarnya bisa (untuk setiap titik mulai, cari ujung terjauh pada jumlah awalan), tetapi ada cara linear yang lebih sederhana.

Jendelanya. Simpan sebuah jendela berisi hotel-hotel berurutan, dari indeks $left$ sampai indeks $right$, beserta jumlahnya. Majukan $right$ satu hotel setiap kali, sambil menambahkan harganya. Setiap kali jumlahnya melampaui M, buang hotel dari ujung kiri sampai jumlahnya muat lagi. Sesudah setiap langkah, jendela itu adalah deret terpanjang yang berakhir di $right$ dengan jumlah paling besar M, dan karena semua harga positif, deret terpanjang semacam itu punya jumlah terbesar. Catat jumlah terbesar yang pernah terlihat.

Mengapa $left$ hanya boleh maju? Misalkan $\ell(r)$ adalah titik mulai terkecil sehingga deret dari $\ell(r)$ sampai r berharga paling besar M. Karena harga positif, memperpanjang deret ke kanan membuatnya lebih mahal, sehingga

$$\ell(r+1) \geq \ell(r) \tag{11.7}$$

dan penunjuk kiri tidak pernah perlu mundur. Setiap penunjuk bergerak paling banyak N kali, sehingga seluruh pemindaian berbiaya paling banyak $2N$ langkah: $O(N)$. **Gambar 11.2** memperlihatkan jendela itu pada lima hotel berharga 2, 1, 3, 4, 5 dengan anggaran 12: jendela tumbuh sampai berjumlah 10, lalu menambahkan 5 membuatnya 15, sehingga 2 dan 1 dibuang, menyisakan 3, 4, 5 dengan jumlah tepat 12.



Gambar 11.2 Jendela dua penunjuk pada harga 2, 1, 3, 4, 5 dengan anggaran 12: ujung kanan maju satu langkah pada setiap baris, dan ujung kiri melompat maju hanya ketika jumlahnya melampaui anggaran.

Kode 11.10 `hotels.cpp`: jendela dua penunjuk.

```cpp
// HOTELS (Hotels Along the Croatian coast): the largest sum of consecutive
// hotel prices that is at most M. All prices are positive: two pointers.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    long long m;
    if (!(cin >> n >> m)) return 0;
    vector<long long> a(n);
    for (auto& x : a) cin >> x;
    long long sum = 0, best = 0;  // sum of the window a[left..right]
    int left = 0;
    for (int right = 0; right < n; right++) {
        sum += a[right];
        while (sum > m) sum -= a[left++];  // too expensive: drop from the left
        best = max(best, sum);
    }
    cout << best << '\n';
    return 0;
}
```

Kode 11.11 `hotels.py`: jendela yang sama dalam Python.

```python
"""HOTELS: the largest sum of consecutive prices that is at most M.
Prices are positive, so a window moved by two pointers finds it in O(N)."""

import sys


def main() -> None:
    data = sys.stdin.buffer.read().split()
    n, m = int(data[0]), int(data[1])
    a = list(map(int, data[2 : 2 + n]))
    total = best = left = 0  # total = a[left] + ... + a[right]
    for x in a:
        total += x
        while total > m:  # too expensive: drop hotels from the left
            total -= a[left]
            left += 1
        if total > best:
            best = total
    print(best)


main()
```

Kode 11.10 menyimpan jumlahnya dalam `long long`: satu harga memang di bawah 10^6, tetapi jumlah berjalan tepat sesudah sebuah harga ditambahkan dapat melampaui M, yang sendiri bisa mendekati 2^{31}, sehingga jumlah itu dapat melewati batas `int` sebelum gelung dalam menurunkannya kembali. **Kode 11.11** adalah salinan langsung gagasan itu. Kita mencoba delapan hotel dengan anggaran 20.

```
$ printf '8 20\n4 7 2 9 3 5 8 1\n' | ./hotels
$ printf '8 20\n4 7 2 9 3 5 8 1\n' | python3 ../python/hotels.py
```

```
19
19
```

Deret terbaiknya $2 + 9 + 3 + 5 = 19$; tidak ada deret harga berurutan yang berjumlah tepat 20. Menelusuri jendela langkah demi langkah memperlihatkan kedua penunjuk bekerja. Jendela

tumbuh sampai $4 + 7 + 2 = 13$; menambahkan 9 membuatnya 22, jadi 4 dibuang, menyisakan $7 + 2 + 9 = 18$, yang terbaik sejauh ini. Menambahkan 3 membuatnya 21, jadi 7 dibuang: $2 + 9 + 3 = 14$. Menambahkan 5 memberi $2 + 9 + 3 + 5 = 19$, terbaik yang baru. Menambahkan 8 membuatnya 27; membuang 2 dan 9 menyisakan $3 + 5 + 8 = 16$. Menambahkan 1 memberi 17. Secara keseluruhan penunjuk kanan bergerak 8 kali dan penunjuk kiri 4 kali, jauh lebih sedikit daripada 36 deret yang akan dijumlahkan *brute force*.

Pola yang sama menyelesaikan satu keluarga soal setiap kali *memperlebar jendela hanya dapat memperburuk dan mempersempitnya hanya dapat memperbaiki*: deret terpendek dengan jumlah paling sedikit S, deret terpanjang dengan paling banyak K nilai berbeda, banyaknya deret dengan jumlah paling besar M. Pola ini gagal ketika kemonotonan itu tidak berlaku, misalnya ketika harga boleh negatif, seperti diperlihatkan Soal 6.

11.8.1 Soal cerminnya: deret terpendek yang sudah cukup

Kerabat dekatnya menanyakan kebalikannya: deret nilai berurutan *terpendek* yang jumlahnya *paling sedikit* S. Seorang petani yang menjual mangga di lapak pinggir jalan, dengan catatan penjualan per jam dalam sebuah daftar, bisa menanyakan rentang jam terpendek yang penjualannya paling sedikit 50 kilogram, untuk memutuskan kapan pembantu kedua benar-benar diperlukan. Jendelanya bergerak dengan cara yang sama, tetapi peran memperlebar dan mempersempit bertukar. Majukan ujung kanan dan tambahkan nilainya; lalu, *selama* jumlahnya masih paling sedikit S, catat panjang jendela sebagai calon dan buang ujung kiri. Pembuangan berhenti begitu jumlahnya turun di bawah S, dan panjang terkecil yang tercatat adalah jawabannya.

Cobalah pada penjualan per jam 4,7,2,9,3,5,8,1 dengan $S = 15$. Jendela tumbuh menjadi 4,7,2,9 berjumlah 22: catat panjang 4, buang 4 (jumlah 18), catat panjang 3, buang 7 (jumlah 11) lalu berhenti. Menambahkan 3 memberi 14, belum cukup. Menambahkan 5 memberi 19: catat panjang 4 (2,9,3,5), buang 2 (jumlah 17), catat 3 (9,3,5), buang 9 (jumlah 8). Menambahkan 8 memberi 16: catat 3 (3,5,8), buang 3 (jumlah 13). Menambahkan 1 memberi 14. Panjang terpendek yang tercatat adalah 3. Sekali lagi setiap penunjuk bergerak paling banyak N kali, dan sekali lagi argumennya bertumpu pada Persamaan (11.7): dengan nilai positif, titik mulai terbaik untuk ujung berikutnya tidak pernah berada di kiri titik mulai terbaik untuk ujung yang sekarang.

11.9 Mengujinya: Setiap Pencarian terhadap Brute Force

Kelima program diperiksa persis seperti program pada Bab 9 dan 10. **Tabel 11.1** mendaftar *brute force*-nya. PIE memerlukan satu gagasan tambahan karena jawabannya bilangan real: pengendali *stress test* membandingkan bilangan yang bertitik desimal dalam batas 10^{-3} yang diizinkan soal, bukan karakter demi karakter, dan *brute force*-nya memanfaatkan fakta yang dicatat pada Subbab 11.6, bahwa potongan terbaik selalu berupa sebuah kue yang dipotong menjadi tepat j bagian yang sama, sehingga ia mencoba setiap v_i/j.

Tabel 11.1 Brute force yang dipakai untuk memeriksa kelima program bab ini.

Soal	Brute force	Masukan kecil
AGGRCOW	setiap pilihan C kandang	sampai 8 kandang, sebagian di jalan pendek
EKO	setiap tinggi gergaji dari 0 sampai pohon tertinggi	sampai 8 pohon
PIE	setiap kue dipotong menjadi $j = 1, \ldots, F + 1$ bagian sama	sampai 5 kue, 8 teman
BOOKS1	setiap cara memotong rak menjadi k deret, dengan aturan pemecah seri	sampai 8 buku
HOTELS	setiap deret hotel berurutan	sampai 10 hotel

```
$ cd tests
$ python3 stress.py 300
```

```
aggrcow   300 random inputs: C++ and Python agree with brute force
eko       300 random inputs: C++ and Python agree with brute force
pie       300 random inputs: C++ and Python agree with brute force
books1    300 random inputs: C++ and Python agree with brute force
hotels    300 random inputs: C++ and Python agree with brute force
```

Baris BOOKS1 adalah yang paling penting: baris itu memastikan, pada 300 rak acak yang masing-masing berisi beberapa kasus uji, bahwa pembagian yang dibangun kembali dari kanan mematuhi aturan pemecah seri persis seperti yang diterapkan *brute force*.

11.10 Membuatnya Lebih Baik: Kecepatan dan Bantuan Pustaka

Tabel 11.2 memperlihatkan waktu CPU terukur pada masukan terbesar.

```
program          C++ (s)  Python (s)  same output
aggrcow             0.02        0.11  yes
eko                 0.04        0.37  yes
pie                 0.01        0.04  yes
books1              0.01        0.14  yes
hotels              0.02        0.06  yes
```

Tabel 11.2 Masukan terbesar dan waktu CPU terukur untuk kelima program.

Soal	Masukan terbesar	Pekerjaan	C++ (s)	Python (s)
AGGRCOW	10^5 kandang, 1000 sapi	pengurutan + 30 pemeriksaan	0.02	0.11
EKO	10^6 pohon	30 pemeriksaan (C++), pengurutan + satu lintasan (Python)	0.04	0.37
PIE	10^4 kue, 10^4 teman	100 atau 60 putaran	0.01	0.04
BOOKS1	200 rak berisi 500 buku	masing-masing 33 pemeriksaan	0.01	0.14
HOTELS	3×10^5 hotel	satu lintasan	0.02	0.06

C++ cepat di mana-mana. Python nyaman untuk PIE, BOOKS1, dan HOTELS, di ambang batas untuk AGGRCOW bila kasus ujinya banyak, dan terlalu lambat untuk EKO, yang batas waktunya hanya sepersekian kecil detik [4]; bahkan **Kode 11.5** yang berbasis pengurutan menghabiskan 0,37 detik, sebagian besar untuk membaca dan mengurutkan sejuta bilangan.

Biarkan pustaka yang mencari. Banyak binary search sebenarnya adalah pencarian dalam larik terurut, dan kedua bahasa sudah menyediakannya. Di C++, `std::lower_bound` mengembalikan unsur pertama yang tidak lebih kecil daripada suatu nilai, `std::upper_bound` unsur pertama yang lebih besar daripadanya, dan `std::partition_point` unsur pertama tempat sebuah predikat monoton berubah menjadi salah, yang persis merupakan batas kita [9]. Di Python,

`bisect.bisect_left` dan `bisect.bisect_right` melakukan hal yang sama pada list, dan sejak Python 3.10 keduanya menerima fungsi `key` [10]. Pakailah fungsi-fungsi itu setiap kali calon-calonnya tersimpan dalam list terurut; tulislah gelung Subbab 11.2 ketika calonnya berupa rentang bilangan dan pemeriksaannya berupa perhitungan, seperti pada kelima soal di sini. Soal 5 memakai `lower_bound` untuk menemukan subbarisan menaik terpanjang dalam $O(n \log n)$, soal klasik yang menghubungkan bab ini dengan Bab 9.

11.11 Memakainya dalam Kehidupan Nyata

Tanda untuk binary search pada jawaban adalah kalimat berbentuk "cari x terbesar sehingga ..." atau "x terkecil sehingga ...", dengan pemeriksaan untuk sebuah x yang mudah dan syarat yang monoton. **Tabel 11.3** mendaftar bentuk-bentuk yang paling sering Anda temui.

Tabel 11.3 Bentuk-bentuk khas soal yang diselesaikan dengan binary search pada jawaban dan dengan dua penunjuk.

Soal meminta	Pemeriksaannya	Arah	Contoh
jarak terkecil yang terbesar	penempatan greedy dari kiri	ya terbesar	AGGRCOW
ketinggian tertinggi yang masih menghasilkan cukup	jumlah atas semua unsur	ya terbesar	EKO
bagian sama yang terbesar	hitung bagian yang diberikan setiap unsur	ya terbesar, pada bilangan real	PIE
beban terbesar yang terkecil	pengemasan greedy ke paling banyak k kelompok	ya terkecil	BOOKS1
deret berurutan terbaik dalam anggaran	dua penunjuk, bukan binary search	pemindaian linear	HOTELS

Versi sehari-harinya mudah ditemukan. Toko roti dengan tiga oven dan daftar loyang berurutan tetap ingin hari memanggang yang sesingkat mungkin: itulah BOOKS1 dengan oven sebagai penyalin. Koperasi yang harus memotong batang-batang bambu yang panjangnya berbeda-beda menjadi potongan sama panjang untuk pagar, paling sedikit F potong, mengajukan pertanyaan PIE pada garis alih-alih pada lingkaran. Perusahaan air minum yang menempatkan C stasiun pemantau di sepanjang pipa agar tersebar sejauh mungkin mengajukan pertanyaan AGGRCOW. Dalam setiap kasus, tulislah pemeriksaannya lebih dulu, ujilah pada beberapa nilai dengan tangan untuk melihat blok ya dan blok tidak, dan baru sesudah itu tambahkan pencarian di sekelilingnya. Uraian klasik Bentley tentang binary search [11] dan bab Laaksonen tentang binary search dalam kontes [12] memberikan jauh lebih banyak contoh, termasuk penanganan ujung-ujung yang cermat.

11.12 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

- **Gelung tak berujung.** `while (lo < hi)` dengan `lo = mid` berputar selamanya ketika `hi == lo + 1` dan `mid` dibulatkan ke bawah menjadi `lo`. Templat `while (hi - lo > 1)` dengan *mid* tegas di antara kedua ujung tidak mungkin berputar selamanya.
- **Nilai awal yang salah.** Bila lo_0 bukan ya yang sungguhan, atau hi_0 bukan tidak yang sungguhan, jawabannya dapat meleset satu di tepi. Pilih ujung yang pasti, meskipun rentangnya menjadi sedikit lebih besar; biayanya hanya satu langkah tambahan.
- **Pemeriksaan yang tidak monoton.** Binary search pada pertanyaan yang jawabannya ya, tidak, ya menemukan *sebuah* batas, bukan *batas yang dicari*. Cetak $ok(x)$ untuk serangkaian x kecil dan amati sebelum mencari.

- **Luapan pada titik tengah atau pada pemeriksaan.** Tulis `lo + (hi - lo) / 2`, dan jumlahkan kayu, halaman, atau harga dalam `long long`.
- **Pencarian bilangan real yang berhenti pada toleransi.** Gunakan banyak putaran yang tetap; 100 selalu cukup untuk `double`.
- **Mengembalikan ujung yang salah.** Untuk "ya terbesar" jawabannya lo; untuk "ya terkecil" jawabannya hi. Putuskan yang mana sebelum menulis gelungnya.
- **Dua penunjuk dengan nilai negatif.** Argumen jendela Persamaan (11.7) memerlukan semua nilai positif (atau tidak negatif). Dengan nilai negatif, gunakan jumlah awalan dengan struktur terurut.
- **Lupa mengurutkan.** Kandang AGGRCOW datang dengan urutan sembarang; pemeriksaan greedy memerlukan kandang yang terurut.

11.13 Ringkasan Bab

- Bila pertanyaan ya/tidak $ok(x)$ monoton, binary search menemukan batas antara ya dan tidak dengan $\lceil \log_2(hi_0 - lo_0) \rceil$ pemeriksaan (Persamaan (11.2)), sambil menjaga invarian Persamaan (11.1) dari awal sampai akhir.
- "Maksimalkan yang terkecil" (AGGRCOW) dan "minimalkan yang terbesar" (BOOKS1, Persamaan (11.6)) menjadi binary search di sekeliling pemeriksaan greedy; "ketinggian tertinggi yang masih menghasilkan cukup" (EKO, Persamaan (11.3)) adalah pencarian di sekeliling sebuah penjumlahan.
- Pada bilangan real (PIE, Persamaan (11.5)) pencarian berjalan dengan banyak putaran yang tetap dan memakai `double` serta π yang tepat.
- Begitu nilai terbaik diketahui, penyelesaiannya sendiri dapat dibangun kembali; untuk BOOKS1, satu lintasan dari kanan mematuhi aturan pemecah seri.
- Dua penunjuk mengganti pencarian kuadratik atas deret dengan pemindaian linear ketika ujung kiri tidak pernah perlu mundur (Persamaan (11.7)), seperti pada HOTELS.
- Kelima program diperiksa terhadap *brute force* pada 1500 masukan acak. C++ cepat di mana-mana; Python baik-baik saja kecuali untuk sejuta pohon EKO.

11.14 Soal Latihan

1. Jalankan binary search pada **Kode 11.2** dengan tangan untuk kandang 1, 3, 7, 10, 12 dan tiga sapi. Berikan lo, hi, dan mid pada setiap langkah, lalu hitung banyaknya pemeriksaan.
2. EKO menjanjikan bahwa semua pohon bersama-sama lebih tinggi daripada M. Apa yang salah pada **Kode 11.4** bila janji itu dilanggar, dan bagaimana Anda mendeteksinya?
3. Tulis gelung untuk x *terkecil* dengan $ok(x)$ benar, untuk pertanyaan yang jawabannya tidak, tidak, …, ya, ya. Ujung mana yang dikembalikan, dan apa yang harus benar tentang nilai-nilai awalnya?
4. Ubahlah gagasan dua penunjuk pada HOTELS untuk *menghitung* banyaknya deret hotel berurutan yang totalnya paling besar M. Terapkan dengan tangan pada harga 2, 1, 3 dengan $M = 3$.
5. Subbarisan menaik tegas terpanjang dari sebuah barisan dapat ditemukan dalam $O(n \log n)$ dengan `lower_bound`. Jelaskan gagasannya dan jalankan **Kode 11.12** pada 5,2,8,6,3,6,9,7.
6. Berikan masukan dengan sebuah harga negatif yang membuat jendela pada **Kode 11.10** memberikan jawaban salah, lalu jelaskan metode $O(N \log N)$ yang benar.

11.15 Penyelesaian Soal Latihan

1. Kandang terurut: 1, 3, 7, 10, 12, jadi $lo = 0$ dan $hi = 12$. Mula-mula $mid = 6$: sapi di 1 dan 7, dan tidak ada yang berjarak 6 dari 7, jadi tidak; $hi = 6$. Lalu $mid = 3$: sapi di 1, 7, 10, ya; $lo = 3$. Lalu $mid = 4$: sapi di 1, 7, 12, ya; $lo = 4$. Lalu $mid = 5$: sapi di 1, 7, 12, ya; $lo = 5$. Sekarang $hi - lo = 1$ dan jawabannya 5, sesudah empat pemeriksaan, $\lceil \log_2 12 \rceil = 4$ seperti diramalkan Persamaan (11.2).

2. Pencarian mengandaikan $ok(0)$ benar. Bila seluruh pohon bersama-sama memberi kurang dari M meter, $ok(0)$ salah, invariannya rusak sejak awal, dan program tetap mencetak 0, ketinggian yang tidak memberi cukup kayu. Untuk mendeteksinya, hitung $wood(0)$ sekali sebelum gelung dan laporkan "tidak mungkin" bila nilainya di bawah M. Kebiasaan yang baik: sebelum binary search apa pun, periksa bahwa nilai-nilai awal memang ya dan tidak.

3. Jaga agar $ok(lo)$ salah dan $ok(hi)$ benar. Ulangi selama `hi - lo > 1`; tetapkan `mid = lo + (hi - lo) / 2`; bila $ok(mid)$ maka $hi = mid$, bila tidak $lo = mid$. Kembalikan hi. Nilai lo awal harus nilai yang pasti tidak (boleh satu di bawah jawaban terkecil yang mungkin, meskipun nilai itu sendiri mustahil sebagai jawaban), dan nilai hi awal nilai yang pasti ya. **Kode 11.8** melakukan tepat hal ini.

4. Untuk setiap ujung kanan r, sesudah penyempitan, setiap deret yang berakhir di r dan dimulai pada atau sesudah $left$ berharga paling besar M, karena deret itu bagian dari jendela dan harganya positif; setiap deret yang dimulai sebelum $left$ lebih mahal. Jadi tambahkan $r - left + 1$ ke hitungan pada setiap langkah. Untuk 2, 1, 3 dengan $M = 3$: pada $r = 0$ jendelanya [2], tambah 1; pada $r = 1$ jendelanya [2,1] dengan jumlah 3, tambah 2; pada $r = 2$ jumlah 6 terlalu besar, buang 2 lalu 1, menyisakan [3], tambah 1. Hitungannya 4: deret [2], [1], [2,1], dan [3].

5. Simpan list `tail` dengan `tail[L]` sebagai nilai akhir terkecil yang mungkin dari subbarisan menaik sepanjang $L + 1$ yang sudah terlihat. List itu selalu menaik tegas, sehingga untuk setiap nilai baru x, `lower_bound` menemukan ekor pertama yang tidak lebih kecil daripada x: menggantinya dengan x mempertahankan subbarisan sepanjang itu dengan ujung yang lebih kecil, dan bila tidak ada, x memperpanjang subbarisan terpanjang. Panjang `tail` adalah jawabannya. Setiap nilai berbiaya satu binary search, sehingga totalnya $O(n \log n)$, dibandingkan dengan $O(n^2)$ untuk DP langsung ala Bab 9.

Kode 11.12 `lis.cpp`: subbarisan menaik tegas terpanjang dengan `lower_bound`.

```cpp
// Exercise 11.5: length of the longest strictly increasing subsequence in
// O(n log n): tail[L] is the smallest possible last value of an increasing
// subsequence of length L+1, and lower_bound finds where each value goes.
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> a = {5, 2, 8, 6, 3, 6, 9, 7};
    vector<int> tail;
    for (int x : a) {
        auto it = lower_bound(tail.begin(), tail.end(), x);  // first tail >= x
        if (it == tail.end())
            tail.push_back(x);
        else
            *it = x;
        cout << "after " << x << ": tails";
        for (int v : tail) cout << ' ' << v;
        cout << '\n';
    }
    cout << "longest strictly increasing subsequence: " << tail.size() << '\n';
    return 0;
}
```

```
$ ./lis
after 5: tails 5
after 2: tails 2
after 8: tails 2 8
after 6: tails 2 6
after 3: tails 2 3
after 6: tails 2 3 6
after 9: tails 2 3 6 9
after 7: tails 2 3 6 7
longest strictly increasing subsequence: 4
```

Jawabannya 4, misalnya 2,3,6,9 atau 2,3,6,7. Perhatikan bahwa `tail` pada umumnya bukan subbarisan dari masukan; hanya panjangnya yang bermakna.

6. Dengan harga $6, -3, 4$ dan $M = 5$, program jendela mula-mula melihat 6, yang terlalu besar, lalu mengosongkan jendela; kemudian ia menambahkan -3 dan 4 dan berakhir dengan jendela $[-3, 4]$, berjumlah 1. Jawabannya 1, padahal satu hotel berharga 4 memberi 4. Dengan nilai negatif, mempersempit jendela dapat membuatnya lebih mahal, sehingga Persamaan (11.7) gagal. Metode yang benar memakai jumlah awalan $P_0 = 0, P_1, \ldots, P_N$: deret dari $i + 1$ sampai r berharga $P_r - P_i$, dan kita mencari $P_i \geq P_r - M$ terkecil di antara $i < r$. Menyimpan jumlah-jumlah awalan sebelumnya dalam pohon pencarian seimbang (`std::set` di C++) dan menanyakan `lower_bound(P[r] - M)` untuk setiap r memberi $O(N \log N)$.

Daftar Pustaka

[1] Sphere Online Judge, "AGGRCOW - Aggressive cows." [Online]. Available: https://www.spoj.com/problems/AGGRCOW/

[2] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998, Sec. 6.2.1.

[3] J. Bloch, "Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken," Google Research Blog, Jun. 2, 2006. [Online]. Available: https://research.google/blog/extra-extra-read-all-about-it-nearly-all-binary-searches-and-mergesorts-are-broken/

[4] Sphere Online Judge, "EKO - Eko." [Online]. Available: https://www.spoj.com/problems/EKO/

[5] Sphere Online Judge, “PIE - Pie.” [Online]. Available: https://www.spoj.com/problems/PIE/

[6] *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019, 2019.

[7] Sphere Online Judge, “BOOKS1 - Copying Books.” [Online]. Available: https://www.spoj.com/problems/BOOKS1/

[8] Sphere Online Judge, “HOTELS - Hotels Along the Croatian Coast.” [Online]. Available: https://www.spoj.com/problems/HOTELS/

[9] cppreference.com, “std::partition_point,” accessed Sep. 2026. [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/partition_point

[10] Python Software Foundation, “bisect - Array bisection algorithm,” *The Python Standard Library*, Python 3.14 documentation, accessed Sep. 2026. [Online]. Available: https://docs.python.org/3/library/bisect.html

[11] J. Bentley, *Programming Pearls*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2000, Column 4.

[12] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

Bab 12

Perkakas Algoritma Graf

Banyak sekali soal juri sebenarnya adalah soal graf yang sedang menyamar: papan catur, peta kota, daftar pekerjaan yang harus dikerjakan dalam urutan tertentu, sekumpulan serangga, jaringan kabel. Bab ini memberi Anda perkakas berisi tujuh teknik graf yang mencakup sebagian besar soal SPOJ tingkat pemula dan menengah, dan setiap teknik dibangun di sekitar satu soal SPOJ yang nyata. Di akhir bab Anda akan mampu menyimpan graf dari masukan juri dalam *adjacency list* (daftar ketetanggaan); memutuskan dalam satu menit teknik mana yang diminta sebuah soal; menguji apakah sebuah graf berupa pohon dengan *union-find* (PT07Y); mencari lintasan terpendek dengan *breadth-first search* bila setiap langkah berbiaya sama (NAKANJ) dan dengan algoritma Dijkstra bila biayanya berbeda-beda (SHPATH); membagi graf menjadi dua kelompok dengan pencarian pewarnaan (BUGLIFE); mengukur lintasan terpanjang di pohon dengan dua kali pencarian (PT07Z); mengurutkan pekerjaan yang saling bergantung (TOPOSORT); dan menghubungkan semuanya semurah mungkin (MST). Setiap teknik disertai pembuktian dengan kata-kata sederhana, waktu jalannya, program lengkap dalam C++17 dan Python, keluaran nyata pada masukan buatan kita sendiri, serta *stress test* terhadap *brute force*.

12.1 Seorang Sopir, Sebuah Jadwal Pelajaran, dan Papan Catur

Pak Darto menyopiri mobil antar-jemput sebuah usaha *laundry* di Surabaya. Setiap pagi ia membawa daftar alamat, peta di kepalanya tentang jalan mana menghubungkan kampung mana, dan perkiraan berapa menit setiap ruas jalan pada jam itu. Ketika pelanggan menelepon dan bertanya, “Kira-kira berapa lama sampai sini, Pak?”, ia sedang memecahkan soal lintasan terpendek: dari posisinya sekarang, melalui jalan-jalan, ke rumah pelanggan, dengan total waktu sekecil mungkin. Ia mengandalkan perasaan, dan setelah sepuluh tahun perasaannya jitu. Sebuah program dapat melakukannya secara tepat, untuk sepuluh ribu tempat, dalam sepersekian detik.

Pada saat yang sama, di sebuah SMA di seberang kota, wakil kepala sekolah menyusun jadwal semester depan. Sebagian materi harus mendahului materi lain: pengantar fisika sebelum praktikum, aljabar dasar sebelum fungsi. Ia memegang daftar pasangan “ini sebelum itu” dan membutuhkan satu urutan yang mematuhi semuanya, atau bukti bahwa daftar itu bertentangan dengan dirinya sendiri. Itulah soal pengurutan topologis. Rekannya yang merencanakan kabel jaringan baru ingin setiap ruang kelas terhubung dengan kabel yang total panjangnya sesedikit mungkin: sebuah pohon perentang minimum.

Lalu seorang anak dua belas tahun di klub catur bertanya berapa langkah yang dibutuhkan kuda untuk berpindah dari satu sudut papan ke sudut seberangnya. Itu pun soal lintasan terpendek, tetapi istimewa, karena setiap langkah berbiaya tepat satu.

Keempat orang itu bekerja dengan sebuah **graf**: himpunan benda (tempat, mata pelajaran, ruang kelas, petak) dan himpunan hubungan antara pasangan benda itu (jalan, aturan “sebelum”, kemungkinan kabel, langkah kuda). Soal pemrograman kompetitif memakai struktur yang sama dengan cerita yang berbeda-beda. Keterampilan yang dibangun bab ini adalah melihat graf di balik cerita, mengenali pertanyaan baku mana yang sedang diajukan, dan menjawabnya dengan

teknik yang dapat Anda tulis dari ingatan tanpa *bug*. SPOJ penuh dengan soal semacam itu, dan ketujuh soal dalam bab ini termasuk yang paling terkenal di situs itu [1].

12.2 Gagasannya dengan Kata-Kata Sederhana

12.2.1 Simpul, sisi, dan istilah di sekitarnya

> **DEFINISI.** Sebuah *graf* $G = (V, E)$ adalah himpunan V berisi *simpul* (*vertex*, juga disebut *node*) dan himpunan E berisi *sisi* (*edge*), masing-masing menghubungkan dua simpul. Kita tulis $n = |V|$ untuk banyaknya simpul dan $m = |E|$ untuk banyaknya sisi. Pada graf *tak berarah* sisi $\{u, v\}$ dapat dilalui dua arah; pada graf *berarah* sisi (u, v) hanya dari u ke v. Pada graf *berbobot* setiap sisi membawa sebuah bilangan, yaitu *bobot* atau biayanya. *Lintasan* adalah barisan simpul yang setiap pasangan berurutannya dihubungkan sebuah sisi; *siklus* adalah lintasan dengan paling sedikit satu sisi yang kembali ke awalnya tanpa memakai sisi yang sama dua kali. Graf disebut *terhubung* bila ada lintasan di antara setiap pasangan simpul.

Pernyataan soal juri hampir selalu menomori simpul $1, 2, \ldots, n$ lalu mendaftar sisinya, satu per baris, sebagai dua nomor simpul (ditambah bobot bila grafnya berbobot). PT07Y, misalnya, memberi n dan m pada baris pertama lalu m baris "u v". Hal pertama yang harus diputuskan adalah cara menyimpan daftar itu di memori agar algoritma dapat menjawab pertanyaan yang paling sering diajukannya: *simpul mana saja yang bertetangga dengan simpul ini?*

12.2.2 Menyimpan graf: adjacency list

Ada tiga cara umum menyimpan graf, dirangkum dalam **Tabel 12.1**. **Daftar sisi** (*edge list*) menyimpan m pasangan persis seperti dibaca; hanya itu yang dibutuhkan algoritma Kruskal, karena Kruskal cukup mengurutkan sisi lalu memeriksanya satu per satu. **Matriks ketetanggaan** (*adjacency matrix*) adalah tabel $n \times n$ yang selnya (u, v) menyatakan ada tidaknya sisi (atau menyimpan bobotnya); ia menjawab "apakah u bertetangga dengan v?" dalam satu langkah, tetapi membutuhkan n^2 sel, yang untuk $n = 10{,}000$ berarti 10^8 sel atau 400 MB `int`, jauh melebihi batas memori SPOJ mana pun yang akan Anda temui. **Daftar ketetanggaan** (*adjacency list*) menyimpan, untuk setiap simpul, daftar tetangganya; memorinya sebanding dengan $n + m$, dan algoritma dapat mengunjungi semua tetangga u dalam waktu sebanding dengan banyaknya. Untuk soal-soal bab ini *adjacency list* adalah pilihan bawaan; dalam C++ ia cukup berupa `vector<vector<int>>` (atau `vector<vector<pair<int,int>>>` untuk bobot), dalam Python berupa *list* berisi *list*.

Tabel 12.1 Tiga cara menyimpan graf, dengan biaya dua pertanyaan yang paling sering diajukan algoritma.

Representasi	Memori	"Tetangga u"	"Apakah (u,v) sisi?"	Dipakai di sini oleh
Daftar sisi	$O(m)$	$O(m)$	$O(m)$	Kruskal (MST)
Matriks ketetanggaan	$O(n^2)$	$O(n)$	$O(1)$	hanya n kecil (*brute force*)
Daftar ketetanggaan	$O(n + m)$	$O(\deg u)$	$O(\deg u)$	BFS, Dijkstra, Kahn, pewarnaan dua warna

Dua hal kecil menyelamatkan banyak *Wrong Answer*. Pertama, pada graf tak berarah setiap sisi harus ditambahkan *dua kali*, sekali ke daftar masing-masing ujungnya; lupa arah kedua termasuk kesalahan klasik Bab 7. Kedua, alokasikan daftar dengan $n + 1$ entri dan pakai indeks $1..n$ secara langsung agar Anda tidak perlu ingat mengurangi satu; entri 0 yang terbuang tidak berarti apa-apa.

12.2.3 Tujuh pertanyaan, tujuh perkakas

Hampir setiap soal graf pada tingkat ini menanyakan salah satu dari segelintir pertanyaan. **Gambar 12.1** adalah peta bab ini: temukan pertanyaannya di kiri, dan perkakas di kanan adalah yang dibangun sisa bab ini, bersama soal SPOJ yang kita pakai untuk berlatih.



Gambar 12.1 Perkakas graf bab ini: pertanyaan yang diajukan soal, teknik yang menjawabnya, dan soal SPOJ yang dipakai untuk berlatih.

Urutan subbab mengikuti tingkat kesulitan pembuktiannya, bukan urutan dalam gambar. Kita mulai dari *union-find*, yang sama sekali tidak memerlukan penelusuran, lalu *breadth-first search* dan dua kerabatnya, kemudian Dijkstra, pengurutan topologis, dan pohon perentang minimum. Ingat anggaran kompleksitas dari Bab 3: dengan n hingga 10^4 dan m hingga 10^6, setiap algoritma di sini berjalan dalam waktu mendekati $O(n+m)$ atau $O((n+m)\log n)$, yaitu beberapa juta hingga beberapa puluh juta langkah dasar, jauh di bawah satu detik dalam C++.

12.3 Pohonkah Ini? Union-Find (PT07Y)

12.3.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ PT07Y, "Is it a tree" [2], memberi graf tak berarah dengan $n \leq 10{,}000$ simpul dan $m \leq 20{,}000$ sisi, lalu meminta satu kata: `YES` bila graf itu pohon, `NO` bila bukan.

> **DEFINISI.** *Pohon* adalah graf tak berarah yang terhubung dan tidak memiliki siklus. Setara dengan itu, pohon adalah graf yang di antara setiap pasangan simpulnya terdapat tepat satu lintasan.

12.3.2 Gagasannya

Sebuah fakta klasik tentang pohon memberi uji yang singkat. Setiap pohon dengan n simpul memiliki tepat $n-1$ sisi, dan sebaliknya, graf dengan n simpul, tepat $n-1$ sisi, dan tanpa siklus pasti terhubung, jadi pohon. Mengapa? Mulailah dari n simpul yang terpisah, yang membentuk n

potongan terpisah (istilahnya *komponen*). Menambahkan sisi yang tidak menutup siklus selalu menggabungkan dua potongan berbeda menjadi satu, sehingga banyaknya potongan berkurang tepat satu. Setelah $n-1$ sisi seperti itu tersisa $n-(n-1)=1$ potongan: grafnya terhubung.

Jadi seluruh ujinya: banyak sisi harus $n-1$, dan tidak boleh ada sisi yang menutup siklus. Sisi $\{u, v\}$ menutup siklus tepat ketika u dan v sudah berada dalam potongan yang sama. Kita membutuhkan struktur yang melacak potongan-potongan itu selama sisi berdatangan dan menjawab "apakah u dan v satu potongan?" dengan cepat. Struktur itu adalah **union-find**, disebut juga *disjoint-set union* (DSU) [3], [4].

12.3.3 Cara kerja union-find

Setiap potongan disimpan sebagai pohon kecil berisi penunjuk induk, dan akar pohon itu menjadi wakil potongannya. `find(x)` mengikuti penunjuk induk dari x hingga akar. `unite(a, b)` mencari kedua akar; bila sama, a dan b sudah terhubung dan fungsi melaporkannya; bila tidak, satu akar digantungkan di bawah akar lainnya. Dua trik menjaga pohon tetap pendek. *Union by size* menggantungkan pohon yang lebih kecil di bawah yang lebih besar, sehingga kedalaman sebuah simpul hanya bertambah bila potongannya paling sedikit berlipat dua, yang membatasi kedalaman hingga $\log_2 n$. *Path halving* membuat setiap simpul di jalan menuju akar menunjuk ke kakeknya, sehingga `find` berikutnya lebih pendek. Dengan kedua trik itu, barisan m operasi pada n elemen berbiaya $O(m\,\alpha(n))$, dengan α fungsi invers Ackermann, fungsi yang tumbuh begitu lambat sehingga $\alpha(n) \le 4$ untuk n berapa pun yang mungkin Anda simpan [3], [5]. Dalam praktik: waktu konstan per operasi.

Kode 12.1 adalah programnya secara lengkap. Struktur `DSU` sepanjang empat belas baris ini akan Anda pakai lagi di Subbab 12.9 dan di banyak soal setelah buku ini.

Kode 12.1 `pt07y.cpp`: graf adalah pohon bila memiliki $n-1$ sisi dan tidak satu pun menutup siklus.

```cpp
// PT07Y - Pohonkah? Union-find: N-1 sisi dan tak ada yang menutup siklus.
#include <cstdio>
#include <numeric>
#include <utility>
#include <vector>
using namespace std;

struct DSU {
    vector<int> parent, size;
    explicit DSU(int n) : parent(n + 1), size(n + 1, 1) {
        iota(parent.begin(), parent.end(), 0);
    }
    int find(int x) {
        while (parent[x] != x) {
            parent[x] = parent[parent[x]];  // pemendekan lintasan (path halving)
            x = parent[x];
        }
        return x;
    }
    bool unite(int a, int b) {  // false jika a dan b sudah terhubung
        a = find(a);
        b = find(b);
        if (a == b) return false;
        if (size[a] < size[b]) swap(a, b);
        parent[b] = a;
        size[a] += size[b];
        return true;
    }
};

int main() {
    int n, m;
    if (scanf("%d %d", &n, &m) != 2) return 0;
    DSU dsu(n);
    bool ok = (m == n - 1);
    for (int i = 0; i < m; i++) {
        int u, v;
        if (scanf("%d %d", &u, &v) != 2) return 0;
        if (!dsu.unite(u, v)) ok = false;  // siklus, atau loop ke diri sendiri
    }
    puts(ok ? "YES" : "NO");
    return 0;
}
```

Perhatikan bahwa program tetap membaca semua m sisi meskipun sudah menemukan siklus. Pada soal dengan satu kasus uji ia bisa berhenti lebih awal, tetapi pada soal dengan beberapa kasus uji, berhenti di tengah satu kasus akan membuat program membaca sisa kasus itu sebagai awal kasus berikutnya; membaca setiap baris masukan adalah kebiasaan yang layak dipertahankan. *Self-loop* "u u" tertangani dengan sendirinya: `unite(u, u)` menemukan akar yang sama dua kali dan melaporkan siklus.

Kode 12.2 adalah gagasan yang sama dalam Python, dengan *union-find* ditulis langsung sebagai *list* `parent` dan fungsi bersarang `find`. Python membaca seluruh masukan sekaligus dengan `sys.stdin.buffer.read().split()`, kebiasaan masukan cepat dari Bab 4.

Kode 12.2 `pt07y.py`: uji yang sama dalam Python.

```python
# PT07Y - Pohonkah? Union-find: N-1 sisi dan tak ada yang menutup siklus.
import sys


def main():
    data = sys.stdin.buffer.read().split()
    n, m = int(data[0]), int(data[1])
    parent = list(range(n + 1))

    def find(x):
        while parent[x] != x:
            parent[x] = parent[parent[x]]  # pemendekan lintasan (path halving)
            x = parent[x]
        return x

    ok = m == n - 1
    for i in range(m):
        a = find(int(data[2 + 2 * i]))
        b = find(int(data[3 + 2 * i]))
        if a == b:
            ok = False  # siklus, atau loop ke diri sendiri
        else:
            parent[a] = b
    print("YES" if ok else "NO")


main()
```

12.3.4 Menjalankannya

Contoh pertama kita adalah pohon kecil: simpul 2 di tengah, terhubung ke 1, 3, dan 4.

```
4 3
1 2
2 3
2 4
```

```
YES
```

Contoh kedua memiliki tiga sisi pada empat simpul, jumlah yang tepat untuk sebuah pohon, tetapi sisi-sisinya membentuk segitiga $1 - 2 - 3$ dan simpul 4 menggantung sendirian. Hitungan sisi saja akan tertipu; pemeriksaan siklus tidak.

```
4 3
1 2
2 3
3 1
```

```
NO
```

12.4 Breadth-First Search: Langkah Kuda (NAKANJ)

12.4.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ NAKANJ, “Minimum Knight moves !!!” [6], memberi hingga 4.096 pasangan petak pada papan catur biasa 8×8, ditulis dalam notasi catur seperti `a1` dan `h8`, lalu menanyakan banyaknya langkah kuda paling sedikit dari petak pertama ke petak kedua. Kuda melangkah dua petak ke satu arah lalu satu petak tegak lurus, sehingga dari petak di tengah papan ia punya delapan kemungkinan langkah.

12.4.2 Gagasannya: jelajahi per cincin

Bayangkan papan sebagai graf dengan 64 simpul, satu per petak, dan sisi di antara dua petak bila kuda dapat melompat dari satu ke yang lain. Setiap sisi berbiaya satu langkah, jadi pertanyaannya adalah panjang lintasan terpendek pada graf *tak berbobot*. **Breadth-first search** (BFS, penelusuran melebar) menjawab tepat pertanyaan itu [3]. Ia menjelajahi graf per cincin: mula-mula petak awal (jarak 0), lalu setiap petak yang berjarak satu langkah (jarak 1), lalu setiap petak yang berjarak satu langkah dari petak-petak itu dan belum pernah terlihat (jarak 2), dan seterusnya. Sebuah antrean masuk-pertama-keluar-pertama menjaga urutan cincin: petak masuk antrean menurut urutan jarak, sehingga keluar pun menurut urutan jarak.

> **WAWASAN KUNCI:** Ketika BFS memberi jarak kepada sebuah simpul, jarak itu sudah pasti. Simpul itu dicapai pertama kali dari simpul pada cincin sebelumnya, dan tidak mungkin ada rute yang lebih pendek, karena setiap simpul pada setiap cincin yang lebih awal sudah dikeluarkan dari antrean dan diperiksa. Dalam lambang, bila $d(v)$ adalah jarak yang diberikan BFS, maka untuk setiap sisi (u, v) berlaku $d(v) \leq d(u) + 1$, dan pemberian pertama mencapai nilai terkecil yang mungkin.

Setiap simpul masuk antrean paling banyak sekali dan setiap sisi diperiksa paling banyak dua kali, sehingga BFS berjalan dalam waktu $O(n + m)$. Pada papan catur itu berarti paling banyak 64 petak dan 64×8 sisi per kueri, beberapa ratus langkah; 4.096 kueri menghabiskan sekitar dua juta langkah, praktis tidak ada.

Kode 12.3 mengubah huruf `a`–`h` menjadi kolom 0..7 dan angka `1`–`8` menjadi baris 0..7, lalu menjalankan satu BFS per kueri. Larik `DR` dan `DC` menyimpan delapan langkah kuda; tabel jarak sekaligus berfungsi sebagai penanda "sudah terlihat", dengan -1 berarti "belum dicapai". Penelusuran berhenti begitu petak tujuan keluar dari antrean.

Kode 12.3 `nakanj.cpp`: satu *breadth-first search* per kueri pada 64 petak papan.

```cpp
// NAKANJ - Langkah kuda minimum: BFS pada papan 8 x 8.
#include <cstdio>
#include <queue>
#include <utility>
using namespace std;

const int DR[8] = {1, 2, 2, 1, -1, -2, -2, -1};
const int DC[8] = {2, 1, -1, -2, -2, -1, 1, 2};

int knightDistance(int r0, int c0, int r1, int c1) {
    int dist[8][8];
    for (auto& row : dist)
        for (int& d : row) d = -1;  // -1 berarti "belum dicapai"
    queue<pair<int, int>> q;
    dist[r0][c0] = 0;
    q.push({r0, c0});
    while (!q.empty()) {
        auto [r, c] = q.front();
        q.pop();
        if (r == r1 && c == c1) return dist[r][c];
        for (int k = 0; k < 8; k++) {
            int nr = r + DR[k], nc = c + DC[k];
            if (nr < 0 || nr >= 8 || nc < 0 || nc >= 8) continue;
            if (dist[nr][nc] != -1) continue;
            dist[nr][nc] = dist[r][c] + 1;
            q.push({nr, nc});
        }
    }
    return -1;  // tak pernah terjadi: kuda mencapai semua petak
}

int main() {
    int t;
    if (scanf("%d", &t) != 1) return 0;
    while (t--) {
        char a[3], b[3];
        if (scanf("%2s %2s", a, b) != 2) break;
        int d = knightDistance(a[1] - '1', a[0] - 'a', b[1] - '1', b[0] - 'a');
        printf("%d\n", d);
    }
    return 0;
}
```

Kode 12.4 `nakanj.py`: penelusuran yang sama dalam Python, dengan `collections.deque` sebagai antrean.

```python
# NAKANJ - Langkah kuda minimum: BFS pada papan 8 x 8.
import sys
from collections import deque

MOVES = [(1, 2), (2, 1), (2, -1), (1, -2), (-1, -2), (-2, -1), (-2, 1), (-1, 2)]


def knight_distance(r0, c0, r1, c1):
    dist = [[-1] * 8 for _ in range(8)]  # -1 berarti "belum dicapai"
    dist[r0][c0] = 0
    q = deque([(r0, c0)])
    while q:
        r, c = q.popleft()
        if (r, c) == (r1, c1):
            return dist[r][c]
        for dr, dc in MOVES:
            nr, nc = r + dr, c + dc
            if 0 <= nr < 8 and 0 <= nc < 8 and dist[nr][nc] == -1:
                dist[nr][nc] = dist[r][c] + 1
                q.append((nr, nc))
    return -1  # tak pernah terjadi: kuda mencapai semua petak


def main():
    data = sys.stdin.read().split()
    t = int(data[0])
    out = []
    for i in range(t):
        a, b = data[1 + 2 * i], data[2 + 2 * i]
        out.append(knight_distance(ord(a[1]) - 49, ord(a[0]) - 97,
                                   ord(b[1]) - 49, ord(b[0]) - 97))
    print("\n".join(map(str, out)))


main()
```

Dalam Python (**Kode 12.4**), pakailah `collections.deque` dan jangan pernah `list.pop(0)`: menghapus elemen pertama *list* Python menggeser semua elemen lainnya dan mengubah penelusuran $O(n+m)$ menjadi $O(n^2)$. Program C++ memakai `std::queue`, yang di baliknya adalah *deque*.

12.4.3 Menjalankannya

```
3
a1 h8
b1 c3
d4 d4
```

```
6
1
0
```

Dari sudut ke sudut seberang dibutuhkan enam langkah; `b1` ke `c3` cukup satu lompatan; dan sebuah petak ke dirinya sendiri membutuhkan nol langkah, kasus tepi yang akan salah dijawab program yang hanya melaporkan jarak *setelah* melangkah. Karena hanya ada 64 petak, alternatifnya adalah menjalankan 64 penelusuran sekali di awal lalu menjawab setiap kueri dari tabel 64×64; itulah jenis prakomputasi yang dianjurkan Bab 3 setiap kali banyaknya pertanyaan berbeda kecil.

12.5 Dua Kelompok Tanpa Konflik: Graf Bipartit (BUGLIFE)

12.5.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ BUGLIFE, "A Bug's Life" [7], bercerita tentang seorang ahli biologi yang mengamati pasangan serangga yang berinteraksi dan menganggap setiap interaksi terjadi antara dua serangga berlainan jenis kelamin. Setiap skenario memberi hingga 2.000 serangga dan hingga 1.000.000 interaksi; program harus mencetak `Scenario #i:` diikuti `Suspicious bugs found!` bila pengamatan bertentangan dengan anggapan itu, dan `No suspicious bugs found!` bila tidak. Buang ceritanya, dan pertanyaannya menjadi: dapatkah simpul-simpul graf diwarnai dengan dua warna sehingga setiap sisi menghubungkan dua warna yang berbeda?

> **DEFINISI.** Sebuah graf disebut *bipartit* (dapat diwarnai dua warna) bila simpul-simpulnya dapat dibagi menjadi dua kelompok sehingga setiap sisi menghubungkan simpul dari satu kelompok dengan simpul dari kelompok lainnya.

12.5.2 Gagasannya

Warnai simpul awal dengan warna 0. Tetangganya kemudian terpaksa berwarna 1, tetangga mereka berwarna 0, dan seterusnya: pada graf yang terhubung, warna simpul pertama menentukan semua warna lainnya. BFS adalah cara yang nyaman untuk menyebarkan warna, cincin demi cincin. Bila penelusuran pernah menemukan sisi yang kedua ujungnya sudah berwarna sama, graf itu tidak bipartit; bila ia selesai tanpa sisi seperti itu, pewarnaan yang dibangunnya sah. Sebuah graf bipartit tepat ketika ia tidak memiliki siklus dengan panjang ganjil [5]: mengelilingi siklus ganjil, warna harus berganti sebanyak bilangan ganjil kali dan kembali berbeda dari warna awalnya.

Grafnya tidak harus terhubung. Sebagian serangga mungkin tak pernah berinteraksi dengan yang lain, dan interaksi dapat membentuk beberapa kelompok terpisah. Karena itu gelung luar dalam **Kode 12.5** memulai penelusuran baru dari setiap simpul yang masih belum berwarna; lupa gelung itu dan hanya menelusuri dari simpul 1 adalah *Wrong Answer* yang paling sering terjadi pada soal ini.

Kode 12.5 `buglife.cpp`: pewarnaan dua warna dengan BFS pada setiap komponen terhubung, dengan pembaca *buffer* byte untuk masukan sejuta baris.

```cpp
// BUGLIFE - A bug's life: dapatkah graf interaksi diwarnai dua warna (bipartit)?
#include <cstdio>
#include <queue>
#include <vector>
using namespace std;

static char buf[1 << 16];
static size_t bufLen = 0, bufPos = 0;

int readChar() {
    if (bufPos == bufLen) {
        bufLen = fread(buf, 1, sizeof(buf), stdin);
        bufPos = 0;
        if (bufLen == 0) return -1;
    }
    return buf[bufPos++];
}

bool readInt(int& x) {
    int c = readChar();
    while (c != -1 && (c < '0' || c > '9')) c = readChar();
    if (c == -1) return false;
    x = 0;
    for (; c >= '0' && c <= '9'; c = readChar()) x = x * 10 + (c - '0');
    return true;
}

bool bipartite(const vector<vector<int>>& adj) {
    int n = (int)adj.size() - 1;
    vector<int> color(n + 1, -1);  // -1 = belum diwarnai, selain itu 0 atau 1
    queue<int> q;
    for (int s = 1; s <= n; s++) {
        if (color[s] != -1) continue;  // komponen terhubung baru dimulai di s
        color[s] = 0;
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v : adj[u]) {
                if (color[v] == -1) {
                    color[v] = 1 - color[u];
                    q.push(v);
                } else if (color[v] == color[u]) {
                    return false;  // ada sisi di dalam satu kelas warna
                }
            }
```

Kode 12.5 (Lanjutan) *buglife.cpp: pewarnaan dua warna dengan BFS pada setiap komponen terhubung, dengan pembaca buffer byte untuk masukan sejuta baris.*

```
        }
    }
    return true;
}

int main() {
    int scenarios;
    if (!readInt(scenarios)) return 0;
    for (int sc = 1; sc <= scenarios; sc++) {
        int n, m;
        if (!readInt(n) || !readInt(m)) return 0;
        vector<vector<int>> adj(n + 1);
        for (int i = 0; i < m; i++) {
            int a, b;
            if (!readInt(a) || !readInt(b)) return 0;
            adj[a].push_back(b);
            adj[b].push_back(a);
        }
        printf("Scenario #%d:\n", sc);
        bool ok = bipartite(adj);
        puts(ok ? "No suspicious bugs found!" : "Suspicious bugs found!");
    }
    return 0;
}
```

Dengan hingga sejuta baris interaksi per skenario, kecepatan masukan penting, dan program memakai pembaca *buffer* `fread` dari Bab 4 alih-alih `scanf`. Kerja per skenario adalah $O(n + m)$. **Kode 12.6** melakukan hal yang sama dalam Python.

Kode 12.6 `buglife.py`: pewarnaan dua warna yang sama dalam Python.

```python
# BUGLIFE - A bug's life: dapatkah graf interaksi diwarnai dua warna (bipartit)?
import sys
from collections import deque


def bipartite(n, adj):
    color = [-1] * (n + 1)  # -1 = belum diwarnai, selain itu 0 atau 1
    for s in range(1, n + 1):
        if color[s] != -1:
            continue  # komponen terhubung baru dimulai di s
        color[s] = 0
        q = deque([s])
        while q:
            u = q.popleft()
            for v in adj[u]:
                if color[v] == -1:
                    color[v] = 1 - color[u]
                    q.append(v)
                elif color[v] == color[u]:
                    return False  # ada sisi di dalam satu kelas warna
    return True


def main():
    data = sys.stdin.buffer.read().split()
    pos = 1
    out = []
    for sc in range(1, int(data[0]) + 1):
        n, m = int(data[pos]), int(data[pos + 1])
        pos += 2
        adj = [[] for _ in range(n + 1)]
        for _ in range(m):
            a, b = int(data[pos]), int(data[pos + 1])
            pos += 2
            adj[a].append(b)
            adj[b].append(a)
        out.append("Scenario #%d:" % sc)
        out.append("No suspicious bugs found!" if bipartite(n, adj)
                   else "Suspicious bugs found!")
    print("\n".join(out))


main()
```

12.5.3 Menjalankannya

Contoh kita memiliki dua skenario: empat serangga yang interaksinya membentuk siklus panjang 4 (genap, jadi dapat diwarnai dua warna) dan tiga serangga yang membentuk segitiga (ganjil, jadi tidak).

```
2
4 4
1 2
2 3
3 4
4 1
3 3
1 2
2 3
3 1
```

```
Scenario #1:
No suspicious bugs found!
Scenario #2:
Suspicious bugs found!
```

12.6 Lintasan Terpanjang di Pohon (PT07Z)

12.6.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ PT07Z, "Longest path in a tree" [8], memberi pohon dengan $n \leq 10{,}000$ simpul sebagai $n - 1$ sisi, lalu menanyakan banyaknya sisi pada lintasan terpanjang di antara dua simpul mana pun. Bilangan itu disebut *diameter* pohon.

12.6.2 Gagasannya: dua kali penelusuran

Brute force akan menjalankan BFS dari setiap simpul dan mengambil jarak terbesar yang pernah ditemukan: n penelusuran masing-masing $O(n)$, total $O(n^2)$, yaitu 10^8 langkah untuk $n = 10^4$. Itu mungkin lolos dalam C++, tetapi ada cara yang jauh lebih baik yang hanya membutuhkan dua penelusuran:

1. Jalankan BFS dari simpul sembarang s dan misalkan a simpul yang terjauh dari s.
2. Jalankan BFS dari a dan misalkan b simpul yang terjauh dari a. Jarak dari a ke b adalah diameternya.

Mengapa penelusuran pertama selalu berakhir di salah satu ujung lintasan terpanjang? Metode ini sudah terkenal [5], dan inti pembuktiannya adalah argumen penyambungan. Ambil lintasan terpanjang $x - \cdots - y$ dengan panjang D, dan andaikan simpul terjauh a dari s *bukan* ujung lintasan terpanjang mana pun. Ikuti lintasan dari s ke a. Di titik mana pun ia meninggalkan lintasan terpanjang itu (atau, bila tak pernah menyentuhnya, di titik ia meninggalkan jalan dari s menuju lintasan itu), sisa jalan menuju a paling sedikit sama panjang dengan sisa jalan menuju y, karena a paling sedikit sejauh y dari s. Menyambung potongan dari x ke titik tinggal itu dengan potongan dari titik tinggal itu ke a menghasilkan lintasan dengan panjang paling sedikit D yang berakhir di a, dan pada kasus kedua bahkan lebih panjang dari D: kontradiksi dalam kedua kasus. Jadi ada lintasan terpanjang yang berakhir di a, dan penelusuran kedua, yang mencari simpul terjauh dari a, menemukan ujung lainnya. Soal Latihan 12.4 meminta Anda menuliskan kedua kasus itu secara lengkap, dan penyelesaiannya di Subbab 12.15 melakukannya.

Pada BFS, simpul keluar dari antrean menurut urutan jarak, sehingga simpul *terakhir* yang keluar adalah salah satu yang terjauh. **Kode 12.7** memakai pengamatan itu alih-alih memindai larik jarak sekali lagi, dan **Kode 12.8** melakukan hal yang sama dalam Python.

Kode 12.7 `pt07z.cpp`: diameter pohon dari dua kali *breadth-first search*.

```cpp
// PT07Z - Lintasan terpanjang di pohon: dua kali BFS menemukan diameter.
#include <cstdio>
#include <queue>
#include <utility>
#include <vector>
using namespace std;

// Mengembalikan simpul terjauh dari s dan jaraknya (dalam sisi).
pair<int, int> farthest(const vector<vector<int>>& adj, int s) {
    vector<int> dist(adj.size(), -1);
    queue<int> q;
    dist[s] = 0;
    q.push(s);
    int last = s;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        last = u;  // BFS mengeluarkan simpul urut jarak: yang terakhir terjauh
        for (int v : adj[u])
            if (dist[v] == -1) {
                dist[v] = dist[u] + 1;
                q.push(v);
            }
    }
    return {last, dist[last]};
}

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    vector<vector<int>> adj(n + 1);
    for (int i = 0; i < n - 1; i++) {
        int u, v;
        if (scanf("%d %d", &u, &v) != 2) return 0;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    int a = farthest(adj, 1).first;  // salah satu ujung lintasan terpanjang
    printf("%d\n", farthest(adj, a).second);
    return 0;
}
```

Kode 12.8 `pt07z.py`: dua penelusuran yang sama dalam Python.

```python
# PT07Z - Lintasan terpanjang di pohon: dua kali BFS menemukan diameter.
import sys
from collections import deque


def farthest(adj, s):
    """Kembalikan simpul terjauh dari s dan jaraknya (dalam sisi)."""
    dist = [-1] * len(adj)
    dist[s] = 0
    q = deque([s])
    last = s
    while q:
        u = q.popleft()
        last = u  # BFS mengeluarkan simpul urut jarak: yang terakhir terjauh
        for v in adj[u]:
            if dist[v] == -1:
                dist[v] = dist[u] + 1
                q.append(v)
    return last, dist[last]


def main():
    data = sys.stdin.buffer.read().split()
    n = int(data[0])
    adj = [[] for _ in range(n + 1)]
    for i in range(n - 1):
        u, v = int(data[1 + 2 * i]), int(data[2 + 2 * i])
        adj[u].append(v)
        adj[v].append(u)
    a, _ = farthest(adj, 1)  # salah satu ujung lintasan terpanjang
    print(farthest(adj, a)[1])


main()
```

12.6.3 Menjalankannya

Pohon pada contoh kita memiliki simpul pusat 2 dengan cabang ke 1, 3, dan cabang yang lebih panjang $4 - 5 - 6$. Lintasan terpanjang, $1 - 2 - 4 - 5 - 6$ (atau $3 - 2 - 4 - 5 - 6$), memiliki 4 sisi.

```
6
1 2
2 3
2 4
4 5
5 6
```

```
4
```

Depth-first search yang ditulis dengan rekursi juga akan berhasil, tetapi pohon dengan 10.000 simpul bisa berupa satu rantai panjang, dan rekursi sedalam 10.000 tingkat mendekati batas *stack* bawaan pada sebagian sistem serta jauh melampaui batas rekursi bawaan Python yang 1.000. BFS dengan antrean eksplisit tidak menanggung risiko itu, dan itulah alasan bab ini memilihnya di mana pun ia mencukupi.

12.7 Lintasan Terpendek Berbobot: Dijkstra (SHPATH)

12.7.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ SHPATH, "The Shortest Path" [9], menggambarkan sebuah negeri dengan hingga 10.000 kota. Untuk setiap kota, masukan memberi namanya (hingga sepuluh huruf kecil), banyaknya tetangga, dan untuk setiap tetangga indeksnya serta biaya perjalanan berupa bilangan bulat positif. Kemudian datang hingga 100 kueri, masing-masing sepasang nama kota, dan untuk setiap kueri program mencetak biaya total minimum. Ada hingga 10 kasus uji seperti itu dalam satu masukan, dan pernyataan soal menjamin tidak ada biaya minimum yang melebihi 200.000.

Ini soal Pak Darto dari Subbab 12.1: lintasan terpendek, tetapi kini sisi-sisinya berbiaya berbeda, sehingga BFS tidak lagi benar. Rute dengan tiga jalan murah bisa mengalahkan rute dengan satu jalan mahal.

12.7.2 Gagasannya: tetapkan kota terdekat lebih dulu

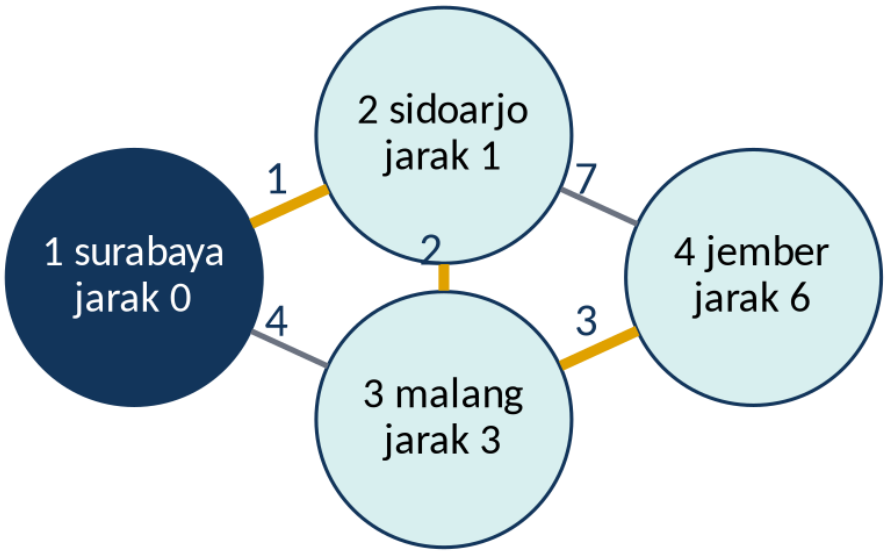
Algoritma Dijkstra [10] menyimpan, untuk setiap simpul v, jarak sementara $d(v)$: biaya rute terbaik ke v yang ditemukan sejauh ini. Pada awalnya $d(s) = 0$ untuk sumber s dan $d(v) = \infty$ untuk yang lain. Algoritma mengulang satu langkah: ambil simpul belum-selesai u dengan jarak sementara terkecil, nyatakan selesai, lalu coba perbaiki setiap tetangga v melalui u:

$$d(v) \leftarrow \min(d(v),\ d(u) + w(u, v)) \tag{12.1}$$

dengan $w(u, v)$ biaya sisinya. Pembaruan ini disebut *relaksasi* sisi (u, v). Persamaan (12.1) adalah ketaksamaan yang sama yang dijaga BFS, dengan bobot sisi menggantikan angka 1.

> **WAWASAN KUNCI:** Bila semua bobot tidak negatif, simpul belum-selesai dengan jarak sementara terkecil tidak mungkin diperbaiki lagi kemudian. Rute lain mana pun menuju simpul itu harus meninggalkan daerah yang sudah selesai melalui suatu simpul belum-selesai x dengan $d(x) \geq d(u)$, lalu menambahkan sisi-sisi berbiaya tak negatif, sehingga biayanya paling sedikit $d(u)$. Di sinilah tepatnya sisi negatif akan merusak algoritma, dan itulah sebabnya algoritma Dijkstra tidak boleh dipakai bila ada bobot yang negatif.

Gambar 12.2 memperlihatkan jarak akhir pada peta kecil yang kita pakai sebagai uji di bawah. Dari Surabaya, Malang lebih murah dicapai lewat Sidoarjo ($1 + 2 = 3$) daripada langsung (4), dan Jember lewat Sidoarjo dan Malang ($1 + 2 + 3 = 6$) daripada lewat Sidoarjo saja ($1 + 7 = 8$).



Gambar 12.2 Peta uji empat kota kita dengan jarak Dijkstra yang dihasilkan dari Surabaya; sisi yang disorot membentuk pohon lintasan terpendek.

12.7.3 Heap dengan penghapusan malas

"Ambil simpul belum-selesai dengan jarak terkecil" adalah tugas antrean prioritas. `std::priority_queue` pada C++ dan `heapq` pada Python tidak dapat menurunkan kunci elemen yang sudah ada di dalamnya, jadi kedua program memakai trik baku yang disebut *penghapusan malas* (*lazy deletion*): setiap kali sebuah jarak membaik, dorong pasangan baru $(d(v), v)$; bila yang dikeluarkan adalah pasangan yang jaraknya tidak lagi sama dengan $d(v)$ saat ini, pasangan itu basi, dan program cukup melewatinya. *Heap* dapat menampung hingga m pasangan, sehingga setiap *push* dan *pop* berbiaya $O(\log m) = O(\log n)$, dan satu kali Dijkstra berbiaya

$$T_{\text{Dijkstra}} = O((n + m)\log n). \tag{12.2}$$

Dua hal lagi penting untuk SHPATH. Tujuan sudah pasti begitu ia dikeluarkan dari *heap*, jadi penelusuran dapat berhenti di situ alih-alih menuntaskan seluruh graf. Dan nama kota harus diubah menjadi indeks; `unordered_map<string,int>` (atau `dict` pada Python) melakukannya dalam waktu harapan konstan per nama.

Kode 12.9 `shpath.cpp`: Dijkstra dengan *heap* biner dan penghapusan malas, satu penelusuran per kueri, berhenti di tujuan.

```cpp
// SHPATH - Lintasan terpendek: Dijkstra dengan heap biner, sekali per kueri.
#include <cstdio>
#include <functional>
#include <queue>
#include <string>
#include <unordered_map>
#include <utility>
#include <vector>
using namespace std;

typedef pair<int, int> PII;  // heap: (jarak, simpul); adj: (simpul, biaya)

int dijkstra(const vector<vector<PII>>& adj, int s, int t) {
    const int INF = 1 << 30;
    vector<int> dist(adj.size(), INF);
    priority_queue<PII, vector<PII>, greater<PII>> pq;  // jarak terkecil di puncak
    dist[s] = 0;
    pq.push({0, s});
    while (!pq.empty()) {
        auto [d, u] = pq.top();
        pq.pop();
        if (d != dist[u]) continue;  // entri basi: u membaik setelah push
        if (u == t) return d;        // tujuan sudah pasti begitu dikeluarkan
        for (auto [v, w] : adj[u]) {
            if (d + w < dist[v]) {
                dist[v] = d + w;
                pq.push({dist[v], v});
            }
        }
    }
    return dist[t];
}

int main() {
    int s;
    if (scanf("%d", &s) != 1) return 0;
    while (s--) {
        int n;
        if (scanf("%d", &n) != 1) return 0;
        vector<vector<PII>> adj(n + 1);
        unordered_map<string, int> id;
        char name[16];
        for (int i = 1; i <= n; i++) {
            int p;
            if (scanf("%15s %d", name, &p) != 2) return 0;
            id[name] = i;
```

Kode 12.9 (Lanjutan) *shpath.cpp: Dijkstra dengan heap biner dan penghapusan malas, satu penelusuran per kueri, berhenti di tujuan.*

```cpp
            while (p--) {
                int nr, cost;
                if (scanf("%d %d", &nr, &cost) != 2) return 0;
                adj[i].push_back({nr, cost});
            }
        }
        int r;
        if (scanf("%d", &r) != 1) return 0;
        while (r--) {
            char a[16], b[16];
            if (scanf("%15s %15s", a, b) != 2) return 0;
            printf("%d\n", dijkstra(adj, id[a], id[b]));
        }
    }
    return 0;
}
```

Kode 12.10 `shpath.py`: algoritma yang sama dengan `heapq`.

```
# SHPATH - Lintasan terpendek: Dijkstra dengan heap biner, sekali per kueri.
import sys
from heapq import heappop, heappush


def dijkstra(adj, s, t):
    dist = [1 << 30] * len(adj)
    dist[s] = 0
    pq = [(0, s)]
    while pq:
        d, u = heappop(pq)
        if d != dist[u]:
            continue  # entri basi: u membaik setelah push
        if u == t:
            return d  # tujuan sudah pasti begitu dikeluarkan
        for v, w in adj[u]:
            if d + w < dist[v]:
                dist[v] = d + w
                heappush(pq, (d + w, v))
    return dist[t]


def main():
    data = sys.stdin.buffer.read().split()
    pos = 0
    s = int(data[pos]); pos += 1
    out = []
    for _ in range(s):
        n = int(data[pos]); pos += 1
        adj = [[] for _ in range(n + 1)]
        ident = {}
        for i in range(1, n + 1):
            ident[data[pos]] = i
            p = int(data[pos + 1]); pos += 2
            for _ in range(p):
                adj[i].append((int(data[pos]), int(data[pos + 1])))
                pos += 2
        r = int(data[pos]); pos += 1
        for _ in range(r):
            a, b = data[pos], data[pos + 1]; pos += 2
            out.append(dijkstra(adj, ident[a], ident[b]))
    print("\n".join(map(str, out)))


main()
```

Kode 12.10 adalah terjemahan baris demi baris dengan `heapq`, yang selalu berupa *min-heap*. Dalam C++, `greater<PII>` mengubah *max-heap* bawaan menjadi *min-heap*; bila lupa, program selalu memperluas simpul *terjauh* lebih dulu, tetap berhenti dan tetap mencetak bilangan, tetapi salah pada sebagian besar masukan. Jaraknya muat dalam `int`, karena pernyataan soal membatasi setiap jawaban hingga 200.000; pada soal tanpa jaminan seperti itu, pakailah `long long`.

12.7.4 Menjalankannya, dan biaya 1.000 kueri

Masukan untuk **Gambar 12.2** mendaftar setiap kota beserta tetangganya; perhatikan bahwa jalan tak berarah muncul dua kali, sekali dalam daftar setiap ujungnya, persis seperti format SHPATH.

```
1
4
surabaya
2
2 1
3 4
sidoarjo
3
1 1
3 2
4 7
malang
3
1 4
2 2
4 3
jember
2
2 7
3 3
3
surabaya jember
sidoarjo malang
jember surabaya
```

```
6
2
6
```

Sekarang anggarannya. Dengan batas pernyataan soal bisa ada $10 \times 100 = 1{,}000$ kueri, masing-masing satu kali Dijkstra berbiaya Persamaan (12.2) pada hingga $n = 10^4$ simpul. Generator ukuran maksimum kita (Subbab 12.10) membangun 10 kasus uji dengan 10.000 kota dan sekitar 30.000 jalan masing-masing, dan padanya program C++ membutuhkan sekitar 5 detik dan program Python sekitar 27 detik. Seberapa padat data juri yang sebenarnya, pernyataan soal tidak mengatakannya. Dua perbaikan membuat program aman dalam kedua keadaan. Pertama, banyak kueri sering berbagi sumber: hitung seluruh larik jarak sekali per sumber yang berbeda (tanpa berhenti di tujuan) dan jawab semua kueri bersumber sama dari larik itu. Kedua, bila grafnya jarang dan kuerinya banyak, berhenti dini di tujuan sudah memangkas kerja rata-rata secara berarti. Soal Latihan 12.6 meminta Anda menambahkan perbaikan pertama.

12.8 Mengerjakan Sesuatu Secara Berurutan: Pengurutan Topologis (TOPOSORT)

12.8.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ TOPOSORT, "Topological Sorting" [11], memberi $n \leq 10{,}000$ pekerjaan dan $m \leq 1{,}000{,}000$ aturan "x harus dikerjakan sebelum y". Program harus mencetak urutan semua pekerjaan yang mematuhi setiap aturan; bila ada beberapa urutan, ia harus mencetak yang *terkecil secara leksikografis*, yaitu yang terkecil bila dibandingkan bilangan demi bilangan dari kiri. Bila tidak ada urutan, karena aturannya mengandung siklus seperti "1 sebelum 2, 2 sebelum 1", ia harus mencetak `Sandro fails.`. Batas waktunya ketat, setengah detik.

> **DEFINISI.** *Urutan topologis* sebuah graf berarah adalah penyusunan simpul-simpulnya dalam satu baris sehingga setiap sisi (x, y) mengarah dari kiri ke kanan. Graf berarah memiliki urutan topologis jika dan hanya jika ia tidak memiliki siklus berarah; graf seperti itu disebut DAG (*directed acyclic graph*).

12.8.2 Algoritma Kahn

Derajat masuk (*in-degree*) sebuah simpul adalah banyaknya sisi yang mengarah kepadanya: banyaknya pekerjaan yang harus mendahuluinya. Pekerjaan dengan derajat masuk 0 *siap*: tidak ada yang menghalangi kita mengerjakannya sekarang. **Algoritma Kahn** [12] mengulang: ambil satu pekerjaan yang siap, catat, lalu hapus sisi-sisi keluarnya, yang menurunkan derajat masuk setiap penerusnya sebesar satu; penerus yang derajat masuknya mencapai 0 menjadi siap. Bila setiap pekerjaan akhirnya tercatat, daftar itu adalah urutan topologis, karena sebuah pekerjaan baru dicatat setelah semua pendahulunya. Bila himpunan pekerjaan siap kosong padahal masih ada pekerjaan tersisa, setiap pekerjaan tersisa masih menunggu pekerjaan tersisa lainnya, dan menelusuri hubungan "menunggu" itu mundur pada akhirnya harus mengulang sebuah pekerjaan: ada siklus.

Untuk mendapatkan urutan terkecil secara leksikografis, simpan pekerjaan yang siap dalam *min-heap* dan selalu ambil yang terkecil. Pilihan *greedy* ini benar berdasarkan argumen pertukaran ala Bab 10: bilangan pertama jawaban pasti suatu pekerjaan yang siap, dan memilih pekerjaan siap yang terkecil tidak pernah merugikan, karena urutan sah mana pun yang dimulai dengan pekerjaan lebih besar dapat dibandingkan pada posisi itu dan lebih besar. Argumen yang sama berulang pada setiap langkah. Dengan *heap*, algoritmanya berbiaya $O(n \log n + m)$.

Kode 12.11 memakai pembaca `fread`, karena sejuta baris sisi dengan batas 0,5 detik adalah persis keadaan yang dipersiapkan Bab 4.

Kode 12.11 `toposort.cpp`: algoritma Kahn dengan *min-heap* berisi pekerjaan siap, dan pendeteksian siklus dengan menghitung.

```cpp
// TOPOSORT - Urutan topologis terkecil leksikografis: Kahn dengan min-heap.
#include <cstdio>
#include <functional>
#include <queue>
#include <vector>
using namespace std;

static char buf[1 << 16];
static size_t bufLen = 0, bufPos = 0;

int readChar() {
    if (bufPos == bufLen) {
        bufLen = fread(buf, 1, sizeof(buf), stdin);
        bufPos = 0;
        if (bufLen == 0) return -1;
    }
    return buf[bufPos++];
}

bool readInt(int& x) {  // melewati apa pun selain digit atau tanda minus
    int c = readChar();
    while (c != -1 && c != '-' && (c < '0' || c > '9')) c = readChar();
    if (c == -1) return false;
    bool neg = (c == '-');
    if (neg) c = readChar();
    x = 0;
    for (; c >= '0' && c <= '9'; c = readChar()) x = x * 10 + (c - '0');
    if (neg) x = -x;
    return true;
}

int main() {
    int n, m;
    if (!readInt(n) || !readInt(m)) return 0;
    vector<vector<int>> adj(n + 1);
    vector<int> indeg(n + 1, 0);
    for (int i = 0; i < m; i++) {
        int x, y;
        if (!readInt(x) || !readInt(y)) return 0;
        adj[x].push_back(y);
        indeg[y]++;
    }
    priority_queue<int, vector<int>, greater<int>> ready;  // terkecil di puncak
    for (int v = 1; v <= n; v++)
        if (indeg[v] == 0) ready.push(v);
    vector<int> order;
```

Kode 12.11 (Lanjutan) *toposort.cpp: algoritma Kahn dengan min-heap berisi pekerjaan siap, dan pendeteksian siklus dengan menghitung.*

```cpp
    while (!ready.empty()) {
        int u = ready.top();
        ready.pop();
        order.push_back(u);
        for (int v : adj[u])
            if (--indeg[v] == 0) ready.push(v);
    }
    if ((int)order.size() < n) {  // ada pekerjaan yang tak pernah siap: siklus
        puts("Sandro fails.");
        return 0;
    }
    for (int i = 0; i < n; i++) printf("%d%c", order[i], i + 1 < n ? ' ' : '\n');
    return 0;
}
```

Kode 12.12 `toposort.py`: algoritma yang sama dengan `heapq`.

```python
# TOPOSORT - Urutan topologis terkecil leksikografis: Kahn dengan min-heap.
import sys
from heapq import heapify, heappop, heappush


def main():
    data = sys.stdin.buffer.read().split()
    n, m = int(data[0]), int(data[1])
    adj = [[] for _ in range(n + 1)]
    indeg = [0] * (n + 1)
    for i in range(m):
        x, y = int(data[2 + 2 * i]), int(data[3 + 2 * i])
        adj[x].append(y)
        indeg[y] += 1
    ready = [v for v in range(1, n + 1) if indeg[v] == 0]
    heapify(ready)  # terkecil di puncak
    order = []
    while ready:
        u = heappop(ready)
        order.append(u)
        for v in adj[u]:
            indeg[v] -= 1
            if indeg[v] == 0:
                heappush(ready, v)
    if len(order) < n:  # ada pekerjaan yang tak pernah siap: siklus
        print("Sandro fails.")
    else:
        print(" ".join(map(str, order)))


main()
```

Kode 12.12 adalah versi Python-nya; pada masukan terbesar ia terlalu lambat untuk batas 0,5 detik, seperti diukur di Subbab 12.10. Antrean biasa sebagai ganti *heap* tetap menghasilkan *sebuah* urutan topologis, hanya bukan yang terkecil; pada soal ini itu berarti *Wrong Answer*, dan pada soal yang menerima urutan sah apa pun (banyak soal begitu, dengan pemeriksa khusus) itu akan diterima. Carilah kata “lexicographically smallest” atau “any” di bagian keluaran pernyataan soal.

12.8.3 Menjalankannya

Contoh pertama kita memiliki enam pekerjaan. Pekerjaan 1 dan 2 siap sejak awal; 3 membutuhkan 1 dan 2; 5 membutuhkan 2; 4 membutuhkan 3 dan 5; 6 membutuhkan 4.

```
6 6
1 3
2 3
3 4
5 4
4 6
2 5
```

```
1 2 3 5 4 6
```

Telusuri dengan tangan: *heap* mula-mula {1,2}; mengambil 1 membuat pekerjaan 3 masih menunggu 2; mengambil 2 membebaskan 3 dan 5; mengambil 3 membuat 4 menunggu 5; mengambil 5 membebaskan 4; lalu 4 membebaskan 6. Contoh kedua adalah siklus tiga aturan.

```
3 3
1 2
2 3
3 1
```

```
Sandro fails.
```

12.9 Menghubungkan Semuanya dengan Murah: Kruskal (MST)

12.9.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ MST, "Minimum Spanning Tree" [13], memberi graf tak berarah yang terhubung dengan $n \leq 10{,}000$ simpul dan $m \leq 100{,}000$ sisi berbobot, dengan nilai mutlak bobot paling besar 10^6, lalu menanyakan total bobot pohon perentang minimum.

> **DEFINISI.** *Pohon perentang* (*spanning tree*) sebuah graf terhubung adalah himpunan bagian sisinya yang membentuk pohon yang memuat setiap simpul; ia memiliki tepat $n-1$ sisi. *Pohon perentang minimum* (*minimum spanning tree*, MST) adalah pohon perentang yang total bobotnya sekecil mungkin.

12.9.2 Algoritma Kruskal dan mengapa ia benar

Algoritma Kruskal [14] bersifat *greedy*: urutkan sisi dari yang termurah hingga termahal lalu periksa dalam urutan itu, ambil sebuah sisi bila ia menggabungkan dua potongan berbeda dan lewati bila ia akan menutup siklus. Potongan-potongan dilacak dengan *union-find* dari Subbab 12.3, itulah sebabnya kedua subbab berbagi kode `DSU`. Pengurutan berbiaya $O(m \log m)$ dan kerja *union-find* hampir linear, sehingga seluruh algoritma berbiaya $O(m \log m)$.

Mengapa pilihan *greedy* itu aman? Alasannya adalah **sifat potongan** (*cut property*) [3]: ambil cara apa pun membagi simpul menjadi dua sisi; di antara sisi-sisi graf yang menyeberangi pembagian itu, yang termurah termasuk dalam suatu pohon perentang minimum. Bukti dengan pertukaran: ambil MST yang tidak memuat sisi penyeberang termurah e itu. Menambahkan e menciptakan siklus, dan siklus itu harus menyeberangi pembagian sekali lagi melalui suatu sisi lain f. Membuang f menghasilkan pohon perentang lain yang bobotnya berubah sebesar $w(e) - w(f) \leq 0$, jadi ia juga minimum dan memuat e. Ketika Kruskal mengambil sisi yang menggabungkan potongan A dan B, sisi itu adalah sisi termurah yang menyeberang antara A dan sisanya (setiap sisi yang lebih murah sudah diperiksa, dan entah diambil entah ternyata berada di dalam satu potongan), sehingga sifat potongan menyatakan pilihan itu aman.

Total bobot pohon dapat mencapai $(n-1) \times 10^6 \approx 10^{10}$, yang tidak muat dalam `int` 32 bit. Akumulatornya harus `long long`; dalam **Kode 12.13** itulah satu-satunya baris yang diberi komentar alasannya, sedangkan **Kode 12.14** tidak perlu berhati-hati karena bilangan bulat Python tidak pernah meluap. Bobot negatif tidak menjadi masalah bagi Kruskal; bobot itu cukup terurut paling depan.

Kode 12.13 `mst.cpp`: algoritma Kruskal dengan *union-find* dan total 64 bit.

```cpp
// MST - Pohon perentang minimum: algoritma Kruskal dengan union-find.
#include <algorithm>
#include <cstdio>
#include <numeric>
#include <utility>
#include <vector>
using namespace std;

struct Edge {
    int w, u, v;
    bool operator<(const Edge& o) const { return w < o.w; }
};

struct DSU {
    vector<int> parent, size;
    explicit DSU(int n) : parent(n + 1), size(n + 1, 1) {
        iota(parent.begin(), parent.end(), 0);
    }
    int find(int x) {
        while (parent[x] != x) {
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }
    bool unite(int a, int b) {
        a = find(a);
        b = find(b);
        if (a == b) return false;
        if (size[a] < size[b]) swap(a, b);
        parent[b] = a;
        size[a] += size[b];
        return true;
    }
};
```

Kode 12.13 (Lanjutan) *mst.cpp: algoritma Kruskal dengan union-find dan total 64 bit.*

```cpp
int main() {
    int n, m;
    if (scanf("%d %d", &n, &m) != 2) return 0;
    vector<Edge> edges(m);
    for (auto& e : edges)
        if (scanf("%d %d %d", &e.u, &e.v, &e.w) != 3) return 0;
    sort(edges.begin(), edges.end());  // sisi termurah lebih dulu
    DSU dsu(n);
    long long total = 0;  // hingga (N-1) * 10^6: melebihi kapasitas int
    int taken = 0;
    for (const auto& e : edges) {
        if (taken == n - 1) break;
        if (dsu.unite(e.u, e.v)) {  // lewati sisi yang akan menutup siklus
            total += e.w;
            taken++;
        }
    }
    printf("%lld\n", total);
    return 0;
}
```

Kode 12.14 `mst.py`: algoritma yang sama dalam Python; bilangan bulat Python tidak pernah meluap.

```python
# MST - Pohon perentang minimum: algoritma Kruskal dengan union-find.
import sys


def main():
    data = sys.stdin.buffer.read().split()
    n, m = int(data[0]), int(data[1])
    edges = []
    for i in range(m):
        u, v, w = data[2 + 3 * i: 5 + 3 * i]
        edges.append((int(w), int(u), int(v)))
    edges.sort()  # sisi termurah lebih dulu
    parent = list(range(n + 1))

    def find(x):
        while parent[x] != x:
            parent[x] = parent[parent[x]]
            x = parent[x]
        return x

    total = taken = 0
    for w, u, v in edges:
        if taken == n - 1:
            break
        a, b = find(u), find(v)
        if a != b:  # lewati sisi yang akan menutup siklus
            parent[a] = b
            total += w
            taken += 1
    print(total)


main()
```

12.9.3 Menjalankannya

Contoh kita memiliki lima simpul dan tujuh sisi. Diurutkan menurut bobot, sisinya adalah $1-3$ (1), $2-3$ (2), $4-5$ (3), $1-2$ (4), $2-4$ (5), $3-4$ (8), $3-5$ (9). Kruskal mengambil tiga yang pertama, melewati $1-2$ karena 1 dan 2 sudah terhubung lewat 3, mengambil $2-4$, lalu berhenti dengan empat sisi: $1+2+3+5=11$.

```
5 7
1 2 4
1 3 1
2 3 2
2 4 5
3 4 8
4 5 3
3 5 9
```

```
11
```

12.10 Menguji Perkakasnya

Setiap program dalam bab ini diperiksa seperti yang diajarkan Bab 8: dengan membandingkannya terhadap program lambat yang jelas benar, pada ratusan masukan acak yang kecil. Folder `Chapter12/tests` berisi tiga berkas. `gen.py` menghasilkan masukan acak untuk setiap soal, entah kecil (segelintir simpul) entah berukuran maksimum yang diizinkan pernyataan soal. `brute.py` menghitung jawaban dengan cara yang terlalu lambat untuk juri tetapi sulit keliru: *depth-first search* ditambah hitungan sisi untuk PT07Y, lintasan terpendek semua pasangan Floyd–Warshall [3] untuk NAKANJ, SHPATH, dan PT07Z, semua permutasi dalam urutan

leksikografis untuk TOPOSORT, semua himpunan bagian berisi $n-1$ sisi untuk MST, dan semua 2^n pewarnaan untuk BUGLIFE. `stress.py` mengompilasi program C++ lalu menjalankan C++, Python, dan *brute force* pada masukan yang sama, dan berhenti pada ketidakcocokan pertama.

Generatornya sengaja memasukkan kasus-kasus yang canggung: satu simpul saja, graf dengan simpul terisolasi, *self-loop* dan sisi berulang pada PT07Y dan MST, bobot negatif pada MST, beberapa skenario dalam satu masukan BUGLIFE, dan kueri yang sumbernya sama dengan tujuannya pada NAKANJ dan SHPATH. Jalankan di bawah ini memakai 200 masukan acak per soal, dengan program C++ dibangun memakai AddressSanitizer dan UndefinedBehaviorSanitizer sehingga indeks di luar batas akan menghentikan jalannya uji meskipun kebetulan tidak mengubah jawaban.

```
buglife    200 random tests: C++ and Python agree with brute force
mst        200 random tests: C++ and Python agree with brute force
nakanj     200 random tests: C++ and Python agree with brute force
pt07y      200 random tests: C++ and Python agree with brute force
pt07z      200 random tests: C++ and Python agree with brute force
shpath     200 random tests: C++ and Python agree with brute force
toposort   200 random tests: C++ and Python agree with brute force
```

Jalankan kedua mengukur waktu satu masukan berukuran maksimum per soal. Angka-angkanya berasal dari mesin kami; angka Anda akan berbeda, tetapi perbandingannya tidak.

```
buglife   max input 8684.9 KB   C++  0.081 s   Python  2.930 s
mst       max input 1673.9 KB   C++  0.041 s   Python  0.665 s
nakanj    max input   24.0 KB   C++  0.046 s   Python  0.587 s
pt07y     max input   92.7 KB   C++  0.013 s   Python  0.041 s
pt07z     max input   95.5 KB   C++  0.021 s   Python  0.060 s
shpath    max input 5256.4 KB   C++  5.326 s   Python 26.867 s
toposort  max input 9549.6 KB   C++  0.100 s   Python  1.347 s
```

Pengukuran ini menjawab pertanyaan "bisakah Python lolos?" untuk bab ini, dengan **Tabel 12.2** sebagai ringkasannya. Bila masukannya kecil (PT07Y, PT07Z, NAKANJ), Python sama baiknya dengan C++. Bila masukannya satu megabyte bilangan atau lebih, Python menghabiskan sebagian besar waktunya untuk mengubah teks menjadi bilangan bulat dan menyusuri *list*, dan ia 10 hingga 40 kali lebih lambat daripada C++. Pada TOPOSORT, yang batasnya 0,5 detik, program C++ membutuhkan 0,10 detik pada masukan terbesar yang dapat kita bangkitkan dan Python membutuhkan 1,35 detik: kirimkan C++.

Tabel 12.2 Waktu pada masukan terbesar yang kita bangkitkan per soal, dibulatkan dari jalankan di atas, dan apakah pengiriman Python realistis.

Soal	Masukan maksimum	C++	Python	Python kemungkinan lolos?
PT07Y, PT07Z	10^4 simpul	0,01–0,02 s	0,04–0,06 s	ya
NAKANJ	4.096 kueri	0,05 s	0,6 s	ya, bila batasnya 1 s atau lebih
MST	10^5 sisi	0,04 s	0,7 s	di ambang batas
BUGLIFE, TOPOSORT	10^6 sisi	0,08–0,10 s	1,3–2,9 s	tidak
SHPATH	1.000 kali Dijkstra	5,3 s	26,9 s	tidak; perbaiki algoritmanya dulu

12.11 Menggunakannya Secara Nyata

Ketujuh program itu adalah templat. Sebagian besar soal graf yang akan Anda temui berikutnya adalah salah satunya dengan cara membaca masukan yang berbeda atau sedikit pelintiran, dan

mengenali pelintiran itu adalah keterampilannya. Beberapa contoh dari SPOJ, untuk dicoba setelah bab ini:

- **Grid adalah graf.** Labirin, peta, atau papan mana pun dengan "pindah ke sel tetangga" adalah BFS atas sel, persis seperti NAKANJ dengan empat atau delapan langkah alih-alih delapan langkah kuda. Bila penelusuran harus dimulai dari banyak sel sekaligus (setiap piksel putih pada *bitmap*, setiap titik api di hutan), masukkan semuanya ke antrean dengan jarak 0 sebelum gelung dimulai: itulah *multi-source BFS*, dan soal SPOJ BITMAP [15] adalah latihan klasiknya.
- **Lintasan terpendek berbiaya** adalah SHPATH dengan format masukan lain. Soal SPOJ HIGHWAYS [16] menanyakan rute tercepat antara dua kota dengan waktu tempuh pada jalan-jalannya, penerapan langsung **Kode 12.9**.
- **Jaringan termurah** adalah MST. Soal SPOJ CSTREET [17] mengalikan total panjang pohon perentang minimum dengan harga per satuan panjang; satu-satunya jebakan baru adalah hasil kalinya membutuhkan 64 bit.
- **Urutan tugas** adalah pengurutan topologis. Bila soal hanya membutuhkan *suatu* urutan, antrean biasa menggantikan *heap* dan algoritmanya menjadi $O(n + m)$.

Di luar juri, program yang sama bekerja pada data nyata. Mobil *laundry* pada Subbab 12.1 dapat diberi tabel kecil berisi tempat dan menit perjalanan, diketik sekali, lalu program SHPATH menjawab "berapa lama sampai sana?" untuk setiap pasangan. Jadwal sekolah menjadi masukan TOPOSORT dengan mata pelajaran dinomori sesuai urutan yang disukai sekolah, sehingga urutan sah terkecil secara leksikografis juga yang paling wajar. Rencana kabel jaringan adalah masukan MST dengan panjang kabel terukur antara setiap pasangan ruangan yang dapat dihubungkan. Dalam setiap kasus, satu-satunya pekerjaan adalah menomori benda-bendanya, menuliskan pasangannya, dan membaca kembali jawabannya dengan nama-namanya.

12.12 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Menambahkan sisi tak berarah hanya dalam satu arah. Setiap baris sisi graf tak berarah harus menghasilkan dua entri di *adjacency list*. Penelusuran dari simpul 1 pada versi satu arah menjangkau lebih sedikit simpul dan memberi jawaban yang tampak masuk akal. *Stress test* di Subbab 12.10 langsung menangkapnya.

Menelusuri hanya dari simpul 1. BUGLIFE dan setiap pertanyaan "untuk semua komponen" membutuhkan gelung luar atas semua simpul yang belum diwarnai atau belum dikunjungi. Contoh kecil sering terhubung, sehingga *bug* ini lolos dari uji contoh.

Memakai BFS pada graf berbobot. BFS menghitung sisi, bukan biaya. Bila sisi-sisinya berbobot berbeda, pakailah Dijkstra; bila bobotnya hanya 0 dan 1, "0-1 BFS" berbasis *deque* juga bekerja (Soal Latihan 12.7).

Memakai Dijkstra dengan bobot negatif. Wawasan kunci Subbab 12.7.2 membutuhkan bobot tak negatif. Dengan sisi negatif program dapat menyelesaikan simpul terlalu cepat dan mencetak jarak yang terlalu besar. Bobot negatif membutuhkan algoritma Bellman–Ford [3].

Lupa pemeriksaan entri basi pada Dijkstra malas. Tanpa `if (d != dist[u]) continue;` program tetap benar tetapi dapat memperluas sebuah simpul berkali-kali, dan pada masukan padat bisa jauh lebih lambat.

Rekursi terlalu dalam. *Depth-first search* rekursif pada lintasan berisi 10^5 simpul dapat meluapkan *stack* dalam C++ dan pasti melampaui batas rekursi bawaan Python. Pilih BFS, atau tulis DFS dengan *stack* eksplisit.

Luapan pada total. Total MST dan jumlah lintasan dapat melampaui $2^{31}-1$. Perkirakan jawaban terbesar yang mungkin dari batasan, seperti diajarkan Bab 3, dan pilih `long long` bila melebihi sekitar 2×10^9.

Max-heap di tempat yang membutuhkan min-heap. `priority_queue<int>` mengembalikan elemen *terbesar*. Untuk Dijkstra dan untuk pekerjaan siap terkecil pada TOPOSORT, berikan `greater<>`, atau dorong nilai yang dinegasikan.

Mereset data per kasus uji. Pada masukan berkasus uji jamak, *adjacency list*, larik jarak, dan *union-find* harus dibuat baru untuk setiap kasus. Mendeklarasikannya di dalam gelung kasus uji, seperti pada kode-kode di atas, membuat hal ini otomatis.

12.13 Ringkasan Bab

- Graf adalah himpunan simpul dan himpunan sisi. Untuk soal juri simpanlah sebagai *adjacency list* dengan simpul $1..n$; tambahkan sisi tak berarah dalam kedua arah.
- **Union-find** melacak potongan-potongan terhubung selama penyisipan sisi dengan waktu hampir konstan per operasi. Graf adalah pohon tepat ketika ia memiliki $n-1$ sisi dan tidak satu pun menutup siklus (PT07Y).
- **Breadth-first search** menemukan lintasan terpendek bila setiap sisi berbiaya sama, dalam waktu $O(n+m)$ (NAKANJ). Variannya menguji pewarnaan dua warna (BUGLIFE) dan menemukan diameter pohon dengan dua kali penelusuran (PT07Z).
- **Algoritma Dijkstra** menemukan lintasan terpendek dengan bobot tak negatif dengan selalu menetapkan simpul belum-selesai yang terdekat; dengan *heap* biner dan penghapusan malas ia berjalan dalam $O((n+m)\log n)$, Persamaan (12.2) (SHPATH).
- **Algoritma Kahn** menghasilkan urutan topologis atau mendeteksi siklus; *min-heap* berisi simpul siap memberi urutan terkecil secara leksikografis (TOPOSORT).
- **Algoritma Kruskal** membangun pohon perentang minimum dengan mengambil sisi dari termurah hingga termahal selama sisi itu menggabungkan dua potongan; sifat potongan membuktikan kebenarannya (MST).
- Setiap program diuji dengan *stress test* terhadap *brute force* dan diukur waktunya pada masukan maksimum. C++ lolos di mana-mana; Python baik untuk masukan kecil dan terlalu lambat bila sejuta sisi harus dibaca.

12.14 Soal Latihan

1. Sebuah graf memiliki $n=6$ simpul dan $m=5$ sisi $\{1,2\},\{2,3\},\{3,1\},\{4,5\},\{5,6\}$. Apa yang dicetak program PT07Y, dan uji mana dari kedua ujinya yang gagal?
2. Tanpa komputer, berikan jarak kuda dari `a1` ke `b2`, dan jelaskan mengapa jaraknya bukan 2.
3. Ubahlah gagasan BUGLIFE untuk graf yang sebagian sisinya berkata “kelompok sama” dan sebagian lagi “kelompok berbeda”. Apa yang disimpan penelusuran pada setiap sisi, dan apa uji kontradiksinya?
4. Buktikan dengan cermat bahwa simpul a yang ditemukan penelusuran pertama di Subbab 12.6 adalah ujung suatu lintasan terpanjang. Ambil lintasan terpanjang $x-\cdots-y$ dan tangani secara terpisah kasus ketika lintasan dari s ke a berbagi simpul dengannya dan kasus ketika tidak.
5. Jalankan Dijkstra dengan tangan dari `jember` pada peta **Gambar 12.2**. Daftarlah urutan kota-kota diselesaikan dan jarak yang dihasilkan.

6. Ubahlah **Kode 12.9** sehingga ia menghitung satu larik jarak penuh per sumber berbeda dalam satu kasus uji dan menjawab setiap kueri bersumber itu dari larik tersebut. Berapa biaya kasus terburuk yang baru dalam banyaknya sumber berbeda k?
7. Pada graf yang bobot sisinya hanya 0 dan 1, jelaskan mengapa *deque*, dengan sisi-0 didorong ke depan dan sisi-1 ke belakang, memberi jarak terpendek yang benar dalam waktu $O(n+m)$.
8. TOPOSORT dengan antrean FIFO biasa sebagai ganti *heap*: berikan urutan yang dicetaknya pada contoh pertama Subbab 12.8.3 bila pekerjaan siap diambil dalam urutan menjadi siap, dan katakan apakah itu yang terkecil secara leksikografis.
9. Pada contoh MST di Subbab 12.9.3, ubah bobot sisi $3-4$ dari 8 menjadi -2. Berapa bobot MST yang baru, dan sisi mana yang dipakainya?
10. Sebuah soal memberi $n = 2 \times 10^5$ kota dan $m = 4 \times 10^5$ jalan dan menanyakan satu lintasan terpendek. Perkirakan banyaknya langkah dasar Dijkstra dengan *heap*, dan versi $O(n^2)$ yang memindai semua simpul untuk mencari yang terdekat. Mana yang muat dalam satu detik?

12.15 Penyelesaian Soal Latihan

1. Program mencetak `NO`. Hitungan sisinya benar, $m = 5 = n - 1$, tetapi sisi $\{3,1\}$ menghubungkan 3 dan 1 yang sudah terhubung lewat 2, sehingga `unite` mengembalikan false: segitiga itu adalah siklus. (Graf dengan $n-1$ sisi yang memuat siklus selalu tidak terhubung; di sini $\{4,5,6\}$ adalah potongan terpisah.)

2. Jaraknya 4. Kuda selalu berpindah ke petak berwarna lain, sehingga setelah banyak langkah ganjil ia berdiri di warna yang lain; `a1` dan `b2` sama-sama gelap, jadi banyak langkahnya harus genap. Dua langkah tidak cukup (dari `a1` kuda hanya mencapai `b3` dan `c2`, dan tak satu pun dari keduanya memiliki langkah kuda ke `b2`), dan ada rute empat langkah, misalnya `a1`–`c2`–`e1`–`d3`–`b2`.

3. Simpan pada setiap sisi paritas $p \in \{0,1\}$: 0 untuk "kelompok sama", 1 untuk "kelompok berbeda". Ketika penelusuran bergerak dari u ke v yang belum berwarna melalui sisi berparitas p, tetapkan $\text{color}(v) = \text{color}(u) \oplus p$ (*exclusive or*). Ketika ia bertemu v yang sudah berwarna, terjadi kontradiksi tepat ketika $\text{color}(v) \neq \text{color}(u) \oplus p$. Dengan $p = 1$ pada setiap sisi, ini adalah BUGLIFE.

4. Misalkan $D = d(x,y)$ panjang lintasan terpanjang dan a simpul terjauh dari s, jadi $d(s,a) \geq d(s,v)$ untuk setiap v. Tulis P untuk lintasan $x - \cdots - y$.

Kasus 1: lintasan dari s ke a berbagi simpul dengan P. Misalkan p simpul *terakhir* lintasan s-ke-a yang terletak pada P, dan q simpul pertama P yang dicapai dari s. Baik q maupun p terletak pada P; namai ujung-ujung P sehingga p terletak di antara q dan y (atau $p = q$). Maka lintasan dari s ke y melewati p, sehingga $d(s,y) = d(s,p) + d(p,y)$, dan $d(s,a) = d(s,p) + d(p,a)$. Dari $d(s,a) \geq d(s,y)$ kita peroleh $d(p,a) \geq d(p,y)$. Setelah p, jalan menuju a meninggalkan P untuk seterusnya, sehingga berjalan dari x sepanjang P ke p lalu terus ke a adalah sebuah lintasan, dengan panjang $d(x,p) + d(p,a) \geq d(x,p) + d(p,y) = D$. Ia lintasan terpanjang, dan berakhir di a.

Kasus 2: lintasan dari s ke a tidak berbagi simpul dengan P. Misalkan q simpul P yang terdekat ke s dan r simpul terakhir yang dimiliki bersama oleh lintasan dari s ke a dan lintasan dari s ke q; maka $r \neq q$, jadi $d(r,q) \geq 1$. Sekarang $d(s,y) = d(s,r) + d(r,q) + d(q,y)$ dan $d(s,a) = d(s,r) + d(r,a)$, sehingga $d(r,a) \geq d(r,q) + d(q,y)$. Lintasan $x \to q \to r \to a$ memiliki panjang $d(x,q) + d(q,r) + d(r,a) \geq d(x,q) + 2\,d(q,r) + d(q,y) = D + 2\,d(q,r) > D$, yang mustahil bagi lintasan

terpanjang dengan panjang D. Jadi Kasus 2 tidak pernah terjadi, dan pada Kasus 1 simpul a mengakhiri suatu lintasan terpanjang.

5. Dari `jember`: pertama diselesaikan `jember` (0); tetangganya memberi `malang` 3 dan `sidoarjo` 7; `malang` (3) diselesaikan berikutnya dan memperbaiki `sidoarjo` menjadi $3 + 2 = 5$ dan `surabaya` menjadi $3 + 4 = 7$; `sidoarjo` (5) diselesaikan dan memperbaiki `surabaya` menjadi $5 + 1 = 6$; terakhir `surabaya` (6). Urutan: jember, malang, sidoarjo, surabaya; jarak 0, 3, 5, 6. Jarak terakhir cocok dengan kueri ketiga di Subbab 12.7.4.

6. Kelompokkan kueri suatu kasus uji menurut sumbernya (`map<int, vector<int>>` dari sumber ke indeks kueri, dengan tetap menyimpan urutan asli untuk pencetakan). Untuk setiap sumber berbeda jalankan Dijkstra tanpa berhenti dini dan isi larik `dist` penuh, lalu jawab kuerinya dengan pencarian di larik itu. Biayanya menjadi $O(k\,(n+m)\log n)$ dengan $k \le r$ sumber berbeda, alih-alih $O(r\,(n+m)\log n)$; bila banyak kueri berbagi sumber, k jauh lebih kecil daripada r.

7. *Deque* selalu menampung simpul yang jarak sementaranya D atau $D+1$ untuk nilai terkecil saat ini D, dengan semua nilai D di depan. Sisi-0 menghasilkan simpul berjarak D, yang tempatnya di depan; sisi-1 menghasilkan $D+1$, yang tempatnya di belakang. Jadi bagian depan *deque* selalu berisi simpul dengan jarak sementara terkecil, persis simpul yang akan diambil Dijkstra, dan *heap* tidak diperlukan. Setiap simpul hanya dapat didorong sebanyak terbatas kali per perbaikan, dan dengan pemeriksaan entri basi setiap simpul diperluas sekali: $O(n+m)$.

8. Dengan antrean FIFO, pekerjaan siap 1 dan 2 diambil lebih dulu; pekerjaan 1 tidak membebaskan apa pun; pekerjaan 2 membebaskan 3 dan 5 (dalam urutan itu, urutan aturannya); lalu 3 membuat 4 menunggu 5, 5 membebaskan 4, dan 4 membebaskan 6. Urutannya 1 2 3 5 4 6, yang pada masukan ini kebetulan sama dengan urutan *heap*. Perubahan kecil memperlihatkan bedanya: tambahkan pekerjaan 7 tanpa aturan. Antrean dimulai sebagai [1,2,7] dan mencetak 1 2 7 3 5 4 6; *heap* mencetak 1 2 3 5 4 6 7. Keduanya urutan sah, tetapi hanya urutan *heap* yang terkecil secara leksikografis, karena pada posisi ketiga $3 < 7$.

9. Terurut: $3-4$ (-2), $1-3$ (1), $2-3$ (2), $4-5$ (3), lalu sisanya. Kruskal mengambil keempatnya, yang sudah menghubungkan kelima simpul: $-2+1+2+3=4$. Pohonnya $\{3-4, 1-3, 2-3, 4-5\}$.

10. Dengan *heap*: $(n+m)\log_2 n \approx 6\times10^5\times18 \approx 1.1\times10^7$ langkah, jauh di bawah satu detik. Versi $O(n^2)$ membutuhkan sekitar 4×10^{10} langkah, ratusan detik. Hanya versi *heap* yang muat.

Daftar Pustaka

[1] Sphere Online Judge (SPOJ), "Problems — classical," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/classical/

[2] SPOJ, "PT07Y — Is it a tree," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/PT07Y/

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[4] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, 1975.

[5] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

[6] SPOJ, "NAKANJ — Minimum Knight moves !!!," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/NAKANJ/

[7] SPOJ, "BUGLIFE — A Bug's Life," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/BUGLIFE/

[8] SPOJ, "PT07Z — Longest path in a tree," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/PT07Z/

[9] SPOJ, "SHPATH — The Shortest Path," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/SHPATH/

[10] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[11] SPOJ, "TOPOSORT — Topological Sorting," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/TOPOSORT/

[12] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[13] SPOJ, "MST — Minimum Spanning Tree," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/MST/

[14] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, 1956.

[15] SPOJ, "BITMAP — Bitmap," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/BITMAP/

[16] SPOJ, "HIGHWAYS — Highways," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/HIGHWAYS/

[17] SPOJ, "CSTREET — Cobbled streets," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/CSTREET/

Bab 13

Perkakas Teori Bilangan, Geometri, dan String

Tiga keluarga soal muncul di setiap juri berulang-ulang: soal tentang bilangan bulat (keterbagian, bilangan prima, sisa bagi), soal tentang titik dan bangun di bidang datar, dan soal tentang teks. Setiap keluarga memiliki sekumpulan kecil teknik yang mengubah tugas yang tampak mustahil menjadi beberapa puluh baris kode, dan setiap keluarga memiliki jebakannya sendiri: luapan (*overflow*) dalam teori bilangan, pembulatan dalam geometri, dan waktu kuadratik yang tersembunyi dalam kode *string* yang tampak polos. Bab ini membangun ketiga perkakas itu di sekitar tujuh soal SPOJ yang nyata. Di akhir bab Anda akan mampu menghitung pangkat raksasa modulo suatu bilangan dengan pengkuadratan berulang (LASTDIG); menabulasi totient Euler untuk sejuta bilangan dengan saringan (ETF); memutuskan apakah bilangan 63 bit adalah prima dengan uji Miller–Rabin deterministik (PON); memakai perkalian silang untuk orientasi, selimut konveks, dan luas poligon dengan bilangan bulat yang eksak; menemukan pasangan terdekat di antara lima puluh ribu titik dengan bagi dan kuasai (CLOPPAIR); menemukan setiap kemunculan pola dalam teks raksasa dengan algoritma Knuth–Morris–Pratt (NHAY); mendeteksi pengulangan dengan fungsi prefiks (PERIOD); dan menghitung substring yang berbeda dengan *suffix array* (DISUBSTR). Seperti di Bab 12, setiap teknik disertai pembuktian dengan kata-kata sederhana, program lengkap dalam C++17 dan Python, keluaran nyata, dan *stress test*.

13.1 Sebidang Kebun, Selembar Struk, dan Sebuah Obrolan

Keluarga Bu Ratna menanam buah dan durian di sebidang lahan di lereng bukit di Jember. Lahan itu bukan persegi panjang: batasnya mengikuti sungai kecil di satu sisi dan jalan lama di sisi lain, dan kantor pertanahan mencatatnya sebagai posisi GPS tujuh patok sudutnya. Ketika keluarga ingin tahu berapa hektare lahan mereka, berapa banyak pupuk yang harus dipesan, atau seberapa panjang pagar baru yang mengelilingi seluruh kebun, mereka membutuhkan geometri: luas poligon yang diberikan titik-titik sudutnya, dan panjang pagar terpendek yang mengelilingi sekumpulan pohon. Keduanya hanya beberapa baris kode begitu Anda mengenal perkalian silang.

Keponakannya bekerja di toko yang mencetak struk dengan digit pemeriksa, dan sistem toko itu juga harus menyebar kode barang secara merata ke sejumlah rak; kedua pekerjaan itu adalah aritmetika sisa bagi. Anak perempuannya, yang mencatat penjualan kebun dalam grup obrolan panjang bersama para pembeli, ingin menemukan setiap pesan yang menyebut kode pesanan seorang pembeli di antara setengah juta baris obrolan. Mencari teks dengan cepat dan benar, bahkan ketika polanya bertumpang-tindih dengan dirinya sendiri, adalah soal *string*.

Pemrograman kompetitif mengubah kebutuhan sehari-hari itu menjadi teka-teki yang tepat dengan batas yang besar. Batas itulah yang membuat tekniknya diperlukan: pencarian naif cukup untuk satu pesan pendek dan sia-sia untuk sejuta karakter; pembagian percobaan cukup untuk bilangan empat digit dan sia-sia untuk bilangan sembilan belas digit. Ketiga perkakas dalam bab ini adalah jawaban baku yang dihimpun dalam buku-buku seperti panduan Laaksonen [1] dan *Competitive Programming 4* karya Halim dkk. [2], dan itulah jawaban yang diharapkan soal-soal SPOJ di bawah.

13.2 Gagasannya dengan Kata-Kata Sederhana

13.2.1 Sisa bagi menjaga bilangan tetap kecil

Sebagian besar soal teori bilangan menanyakan sesuatu tentang bilangan yang terlalu besar untuk dituliskan: digit terakhir $7^{2147483000}$, hasil "modulo $10^9 + 7$", apakah bilangan sembilan belas digit itu prima. Alat utamanya adalah **aritmetika modular**. Untuk bilangan bulat positif m, $a \bmod m$ adalah sisa ketika a dibagi m, sebuah bilangan antara 0 dan $m - 1$. Sisa bagi berperilaku baik terhadap penjumlahan dan perkalian:

$$(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m \tag{13.1}$$

dan hal yang sama berlaku dengan $+$ sebagai ganti $\cdot$. Jadi kita boleh mereduksi setelah setiap langkah dan tidak pernah memegang bilangan yang lebih besar dari sekitar m^2. Dengan m di bawah 2^{31} hasil kalinya muat dalam `long long` 64 bit; dengan m hingga 2^{63} tidak muat, dan kita membutuhkan hasil kali 128 bit, seperti diperlihatkan Subbab 13.5.

13.2.2 Bilangan bulat eksak untuk geometri

Soal geometri memberi koordinat, dan cara yang menggoda adalah menghitung sudut dan kemiringan dengan `double`. Hampir selalu ada cara yang lebih baik: hampir setiap pertanyaan dasar (apakah lintasan berbelok ke kiri atau ke kanan, apakah dua ruas garis berpotongan, apakah sebuah titik berada di dalam segitiga, berapa luas sebuah poligon) dapat dijawab dengan satu besaran, yaitu **perkalian silang** (*cross product*), yang hanya memakai perkalian dan pengurangan. Dengan koordinat bulat ia dihitung secara eksak, dan program tidak pernah perlu memutuskan apakah 10^{-12} itu "nol". Bilangan pecahan baru muncul di akhir sekali, ketika jawabannya sendiri berupa jarak dengan enam angka desimal.

13.2.3 Jangan pernah melihat karakter yang sama dua kali

Cara naif mencari pola sepanjang m dalam teks sepanjang n membandingkan pola di setiap posisi: hingga $n \cdot m$ perbandingan karakter, 10^{12} untuk teks dan pola sepanjang sejuta karakter. Algoritma *string* dalam bab ini berbagi satu gagasan: ketika sebuah perbandingan gagal, manfaatkan apa yang sudah Anda ketahui tentang teks alih-alih memulai lagi dari awal. Fungsi prefiks (Subbab 13.8) mencatat, untuk setiap prefiks pola, berapa banyak yang dapat dipakai ulang, dan itu menurunkan pencarian menjadi $O(n + m)$.

Tabel 13.1 mendaftar soal-soal bab ini beserta batasan yang memaksa setiap teknik.

Tabel 13.1 Soal-soal bab ini, batas yang menyingkirkan cara naif, dan teknik yang sesuai.

Subbab	Soal SPOJ	Batasan yang menentukan	Teknik	Waktu
13.3	LASTDIG	$b \le 2.1 \times 10^9$	pemangkatan cepat	$O(\log b)$
13.4	ETF	20.000 kueri, $n \le 10^6$	saringan totient	$O(N \log\log N)$ sekali
13.5	PON	$n < 2^{63}$	Miller–Rabin, 12 basis	$O(\log n)$ perkalian
13.6	—	jawaban eksak	perkalian silang, selimut, *shoelace*	$O(n \log n)$
13.7	CLOPPAIR	$n \le 50{,}000$	bagi dan kuasai	$O(n \log n)$
13.8	NHAY	teks tanpa batas panjang	Knuth–Morris–Pratt	$O(n + m)$
13.9	PERIOD	$n \le 10^6$	fungsi prefiks	$O(n)$
13.10	DISUBSTR	$n \le 1{,}000$	*suffix array* + LCP	$O(n \log^2 n)$

13.3 Pangkat Raksasa: Pemangkatan Cepat (LASTDIG)

13.3.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ LASTDIG, "The last digit" [3], memberi hingga 30 pasangan bilangan bulat a dan b dengan $0 \le a \le 20$ dan $0 \le b \le 2{,}147{,}483{,}000$, tidak keduanya nol, lalu menanyakan digit terakhir a^b. Bilangan a^b sendiri dapat memiliki miliaran digit, sehingga menghitungnya lalu mengambil digit terakhirnya tidak mungkin, bahkan dengan bilangan bulat Python yang tak terbatas.

13.3.2 Gagasannya: kuadratkan dan kalikan

Digit terakhir sebuah bilangan adalah sisanya modulo 10, dan menurut Persamaan (13.1) kita dapat mereduksi modulo 10 setelah setiap perkalian. Itu saja belum cukup: mengalikan b kali membutuhkan dua miliar langkah. **Pemangkatan cepat** (*binary exponentiation*) memakai digit biner b. Tulis $b = \sum_k \beta_k \, 2^k$ dengan setiap bit $\beta_k \in \{0,1\}$. Maka

$$a^b = \prod_{k \,:\, \beta_k = 1} a^{2^k}, \qquad a^{2^{k+1}} = (a^{2^k})^2 \tag{13.2}$$

sehingga program menyimpan kuadrat berjalan $a, a^2, a^4, a^8, \ldots$, satu pengkuadratan per bit, dan mengalikan hasil dengan kuadrat saat itu setiap kali bitnya 1. Banyaknya langkah sama dengan banyaknya bit b, paling banyak 31 di sini; **Kode 13.1** dan **Kode 13.2** mengimplementasikannya. Rutin yang sama, dengan modulus lain, adalah kuda beban Subbab 13.5 dan soal-soal tak terhitung yang berkata "cetak jawabannya modulo $10^9 + 7$".

Kode 13.1 `lastdig.cpp`: pemangkatan cepat modulo 10.

```cpp
// LASTDIG - Digit terakhir a^b: pemangkatan cepat modulo 10.
#include <cstdio>

long long powMod(long long base, long long exp, long long mod) {
    long long result = 1 % mod;
    base %= mod;
    while (exp > 0) {
        if (exp & 1) result = result * base % mod;  // bit exp ini bernilai 1
        base = base * base % mod;                   // base^(2^k) utk bit berikutnya
        exp >>= 1;
    }
    return result;
}

int main() {
    int t;
    if (scanf("%d", &t) != 1) return 0;
    while (t--) {
        long long a, b;
        if (scanf("%lld %lld", &a, &b) != 2) return 0;
        printf("%lld\n", powMod(a, b, 10));
    }
    return 0;
}
```

Kode 13.2 `lastdig.py`: gelung yang sama dalam Python; fungsi bawaan `pow(a, b, 10)` melakukan hal yang sama dalam satu panggilan.

```python
# LASTDIG - Digit terakhir a^b: pemangkatan cepat modulo 10.
import sys


def pow_mod(base, exp, mod):
    result = 1 % mod
    base %= mod
    while exp > 0:
        if exp & 1:
            result = result * base % mod  # bit exp ini bernilai 1
        base = base * base % mod  # base^(2^k) utk bit berikutnya
        exp >>= 1
    return result


def main():
    data = sys.stdin.read().split()
    t = int(data[0])
    out = [pow_mod(int(data[1 + 2 * i]), int(data[2 + 2 * i]), 10)
           for i in range(t)]
    print("\n".join(map(str, out)))  # pow(a, b, 10) adalah padanan bawaannya


main()
```

Dua kasus tepi menentukan soal ini. Bila $b = 0$ jawabannya 1 (untuk $a \neq 0$), dan program harus mencetaknya; gelung yang dimulai dengan `result = base` alih-alih `result = 1` salah di sini. Dan bila $a = 0$ dan $b > 0$ jawabannya 0. Ungkapan `1 % mod` di awal membuat rutin itu benar bahkan untuk $m = 1$, ketika setiap sisa bagi adalah 0, kasus yang tidak berbiaya apa-apa untuk ditangani dan muncul di soal-soal lain.

13.3.3 Menjalankannya

```
4
2 10
3 0
0 7
7 2147483000
```

```
4
1
0
1
```

$2^{10} = 1024$ berakhir dengan 4; $3^0 = 1$; $0^7 = 0$; dan pangkat-pangkat 7 berakhir dalam siklus berulang 7,9,3,1 sepanjang 4, dan 2,147,483,000 habis dibagi 4, sehingga $7^{2147483000}$ berakhir dengan 1. Siklus itu memberi penyelesaian kedua: setiap digit terakhir berulang dengan periode 1, 2, atau 4, sehingga a^b mod 10 hanya bergantung pada a mod 10 dan b mod 4 (dengan $b = 0$ ditangani tersendiri). Pemangkatan cepat tidak membutuhkan wawasan semacam itu dan bekerja untuk setiap modulus, dan itulah sebabnya ia yang perlu dipelajari.

13.4 Menghitung Bilangan Koprima untuk Sejuta Bilangan: Totient Euler (ETF)

13.4.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ ETF, "Euler Totient Function" [4], memberi hingga 20.000 bilangan n dengan $1 \le n \le 10^6$ dan menanyakan, untuk masing-masing, berapa banyak bilangan bulat k antara 1 dan n yang tidak memiliki pembagi bersama dengan n selain 1.

> **DEFINISI.** Dua bilangan bulat disebut *koprima* (relatif prima) bila faktor persekutuan terbesarnya 1. *Totient Euler* $\varphi(n)$ adalah banyaknya bilangan bulat k dengan $1 \le k \le n$ yang koprima dengan n. Sebagai contoh $\varphi(9) = 6$: bilangan 1, 2, 4, 5, 7, 8.

13.4.2 Rumus hasil kali

Menghitung dengan `gcd` untuk setiap k berbiaya $O(n \log n)$ per kueri, $2 \times 10^4 \times 10^6$ langkah pada kasus terburuk: jauh terlalu lambat. Rumus yang membuat φ murah adalah

$$\varphi(n) = n \prod_{p \mid n} \left(1 - \frac{1}{p}\right) \tag{13.3}$$

dengan hasil kali atas bilangan prima berbeda p yang membagi n [5]. Alasannya, dengan kata-kata sederhana: di antara bilangan $1..n$, tepat sebagian $1/p$ adalah kelipatan p, dan untuk bilangan prima yang berbeda bagian-bagian itu bergabung secara bebas (di baliknya ada prinsip inklusi–eksklusi, atau teorema sisa Tiongkok). Untuk $n = 36 = 2^2 \cdot 3^2$: $\varphi(36) = 36 \cdot \frac{1}{2} \cdot \frac{2}{3} = 12$.

Persamaan (13.3) menyarankan dua program. Dengan banyak kueri dan batas atas yang kecil, hitunglah φ untuk *setiap* bilangan hingga 10^6 sekaligus dengan saringan: mulai dengan $\varphi(i) = i$, dan untuk setiap bilangan prima p kalikan entri semua kelipatannya dengan $1 - 1/p$, yang dalam bilangan bulat berarti `phi[j] -= phi[j] / p`. Bilangan p adalah prima tepat ketika belum disentuh bilangan prima yang lebih kecil, yaitu ketika `phi[p]` masih sama dengan p. Saringan itu berbiaya $\sum_p N/p = O(N \log\log N)$ langkah, sekitar tiga juta untuk $N = 10^6$, lalu setiap kueri cukup dicari di tabel. Itulah **Kode 13.3**.

Kode 13.3 `etf.cpp`: saringan totient; setiap kueri dijawab dari tabel.

```cpp
// ETF - Totient Euler untuk banyak n <= 10^6: satu saringan, lalu O(1) per kueri.
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    const int LIMIT = 1000000;
    vector<int> phi(LIMIT + 1);
    for (int i = 0; i <= LIMIT; i++) phi[i] = i;
    for (int p = 2; p <= LIMIT; p++) {
        if (phi[p] != p) continue;  // p tersentuh prima lebih kecil: komposit
        for (int j = p; j <= LIMIT; j += p)
            phi[j] -= phi[j] / p;  // kalikan dengan (1 - 1/p)
    }
    int t;
    if (scanf("%d", &t) != 1) return 0;
    while (t--) {
        int n;
        if (scanf("%d", &n) != 1) return 0;
        printf("%d\n", phi[n]);
    }
    return 0;
}
```

Dalam Python, saringan atas sejuta entri dengan gelung dalam di tingkat Python memakan beberapa detik, sehingga **Kode 13.4** menempuh jalan lain: ia memfaktorkan setiap kueri dengan pembagian percobaan memakai 168 bilangan prima di bawah 1.000. Setiap $n \le 10^6$ memiliki paling banyak satu faktor prima di atas $\sqrt{n} \le 1000$, dan apa pun yang tersisa setelah membagi habis dengan prima-prima kecil adalah faktor itu. Biayanya paling banyak 168 pembagian per kueri, sekitar 3,4 juta secara total, yang ditangani Python dalam sepersekian detik.

Kode 13.4 `etf.py`: $\varphi(n)$ lewat pemfaktoran dengan bilangan prima di bawah 1.000.

```python
# ETF - Totient Euler untuk banyak n <= 10^6: faktorkan n dgn prima di bawah 1000.
import sys

PRIMES = [p for p in range(2, 1001)
          if all(p % d for d in range(2, int(p ** 0.5) + 1))]


def phi(n):
    result = n
    for p in PRIMES:
        if p * p > n:
            break
        if n % p == 0:
            while n % p == 0:
                n //= p
            result -= result // p  # kalikan dengan (1 - 1/p)
    if n > 1:  # sisanya satu faktor prima di atas sqrt(n semula)
        result -= result // n
    return result


def main():
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    print("\n".join(str(phi(int(x))) for x in data[1:1 + t]))


main()
```

13.4.3 Menjalankannya

```
5
1
9
10
36
999983
```

```
1
6
4
12
999982
```

$\varphi(1) = 1$ menurut definisi (1 koprima dengan dirinya sendiri); $\varphi(9) = 9 \cdot \frac{2}{3} = 6$; $\varphi(10) = 10 \cdot \frac{1}{2} \cdot \frac{4}{5} = 4$; $\varphi(36) = 12$ seperti dihitung di atas; dan 999,983 adalah prima, sehingga $\varphi = 999{,}982$.

13.5 Primakah Bilangan 19 Digit? Miller–Rabin (PON)

13.5.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ PON, "Prime or Not" [6], memberi hingga 500 bilangan bulat n dengan $2 \le n \le 2^{63} - 1$ dan meminta `YES` atau `NO`: apakah n prima? Pembagian percobaan hingga $\sqrt{n} \approx 3 \times 10^9$ sia-sia, apalagi saringan.

13.5.2 Dari Fermat ke Miller–Rabin

Teorema kecil Fermat menyatakan bahwa untuk bilangan prima p dan setiap a yang tidak habis dibagi p,

$$a^{p-1} \equiv 1 \pmod{p}. \tag{13.4}$$

Jadi bila $a^{n-1} \bmod n \neq 1$ untuk suatu a, maka n pasti komposit, dan pemangkatan cepat menghitung pangkat itu dalam sekitar 63 pengkuadratan. Sayangnya kebalikannya tidak berlaku: ada bilangan komposit, yaitu *bilangan Carmichael*, yang memenuhi $a^{n-1} \equiv 1$ untuk setiap a yang koprima dengan n. Yang terkecil adalah $561 = 3 \cdot 11 \cdot 17$.

Uji **Miller–Rabin** [7] memperkuat uji Fermat dengan satu fakta lagi: modulo bilangan prima, akar kuadrat dari 1 hanyalah 1 dan -1 (yaitu $n - 1$). Tulis $n - 1 = d \cdot 2^s$ dengan d ganjil. Untuk n prima, barisan

$$a^d,\ a^{2d},\ a^{4d},\ \ldots,\ a^{2^s d} = a^{n-1} \pmod{n} \tag{13.5}$$

berakhir dengan 1 menurut Persamaan (13.4), dan setiap sukunya adalah kuadrat suku sebelumnya, sehingga angka 1 pertama dalam barisan harus didahului $n - 1$, kecuali barisannya dimulai dengan 1. Basis a yang membuat hal ini gagal disebut *saksi* (*witness*) bahwa n komposit. Untuk n komposit paling sedikit tiga perempat dari semua basis adalah saksi, sehingga uji acak sangat andal; tetapi juri menginginkan kepastian, dan untuk bilangan 64 bit kepastian itu tersedia. Telah diverifikasi dengan komputasi menyeluruh bahwa dua belas bilangan prima pertama 2,3,5, … ,37 sebagai basis tidak memberi jawaban keliru untuk n mana pun di bawah 3.18×10^{23} [8], jauh di atas $2^{63} \approx 9.2 \times 10^{18}$. Jadi uji dengan kedua belas basis itu adalah uji keprimaan *deterministik* untuk setiap masukan PON.

13.5.3 Hasil kali 128 bit

Mengkuadratkan sisa bagi yang mendekati 2^{63} menghasilkan hasil kali mendekati 2^{126}, yang meluapkan 64 bit dan diam-diam memberi sampah. GCC dan Clang menyediakan tipe `unsigned __int128`, yang hasil kalinya eksak; **Kode 13.5** memakainya dalam `mulMod`. Ini ekstensi kompiler, bukan C++ standar, dan kompiler Microsoft tidak memilikinya (Bab 5 membahas soal portabilitas semacam ini); di SPOJ, yang mengompilasi C++ dengan GCC, tipe itu tersedia. Bilangan bulat Python tidak memiliki batas ukuran, sehingga **Kode 13.6** memakai `pow` bawaan dengan tiga argumen.

Kode 13.5 `pon.cpp`: Miller–Rabin deterministik dengan dua belas basis prima dan hasil kali 128 bit.

```
// PON - Prima atau bukan untuk n < 2^63: Miller-Rabin deterministik.
#include <cstdint>
#include <cstdio>

typedef uint64_t u64;
typedef unsigned __int128 u128;  // ekstensi GCC/Clang, bukan C++ standar

u64 mulMod(u64 a, u64 b, u64 m) { return (u64)((u128)a * b % m); }

u64 powMod(u64 a, u64 e, u64 m) {
    u64 r = 1;
    a %= m;
    for (; e > 0; e >>= 1) {
        if (e & 1) r = mulMod(r, a, m);
        a = mulMod(a, a, m);
    }
    return r;
}

bool isPrime(u64 n) {
    if (n < 2) return false;
    const u64 bases[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
    for (u64 p : bases)
        if (n % p == 0) return n == p;  // prima kecil dan kelipatannya
    u64 d = n - 1;
    int s = 0;
    while ((d & 1) == 0) {  // n - 1 = d * 2^s dengan d ganjil
        d >>= 1;
        s++;
    }
    for (u64 a : bases) {
        u64 x = powMod(a, d, n);
        if (x == 1 || x == n - 1) continue;
        bool witness = true;  // a bukti n komposit, kecuali tercapai n - 1
        for (int r = 1; r < s && witness; r++) {
            x = mulMod(x, x, n);
            if (x == n - 1) witness = false;
        }
        if (witness) return false;
    }
    return true;  // 12 basis ini cukup untuk setiap n di bawah 3.18 * 10^23
}
```

Kode 13.5 (Lanjutan) *pon.cpp: Miller–Rabin deterministik dengan dua belas basis prima dan hasil kali 128 bit.*

```cpp
int main() {
    int t;
    if (scanf("%d", &t) != 1) return 0;
    while (t--) {
        unsigned long long n;
        if (scanf("%llu", &n) != 1) return 0;
        puts(isPrime(n) ? "YES" : "NO");
    }
    return 0;
}
```

Kode 13.6 `pon.py`: uji yang sama dalam Python.

```python
# PON - Prima atau bukan untuk n < 2^63: Miller-Rabin deterministik.
import sys

BASES = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37)


def is_prime(n):
    if n < 2:
        return False
    for p in BASES:
        if n % p == 0:
            return n == p  # prima kecil dan kelipatannya
    d, s = n - 1, 0
    while d % 2 == 0:  # n - 1 = d * 2^s dengan d ganjil
        d //= 2
        s += 1
    for a in BASES:
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(s - 1):
            x = x * x % n
            if x == n - 1:
                break
        else:
            return False  # a adalah saksi: n komposit
    return True  # 12 basis ini cukup untuk setiap n di bawah 3.18 * 10^23


def main():
    data = sys.stdin.read().split()
    t = int(data[0])
    print("\n".join("YES" if is_prime(int(x)) else "NO" for x in data[1:1 + t]))


main()
```

Gelung prima kecil di awal `isPrime` melakukan dua tugas: ia menjawab langsung untuk kedua belas bilangan prima basis itu sendiri (basis tidak boleh kelipatan n), dan ia membuang kelipatan bilangan-bilangan prima itu, yang merupakan sebagian besar bilangan komposit, sebelum pemangkatan apa pun. Konstruksi `for ... else` pada Python menjalankan bagian `else` hanya bila gelung tidak ditinggalkan lewat `break`, yang persis berarti "tidak ada langkah kuadrat yang mencapai $n - 1$".

13.5.4 Menjalankannya

```
6
2
561
1000000007
3825123056546413051
9223372036854775783
9223372036854775807
```

```
YES
NO
YES
NO
YES
NO
```

Masukannya sengaja dipilih yang sulit: 561 adalah bilangan Carmichael terkecil; $10^9 + 7$ adalah bilangan prima yang muncul di separuh pernyataan soal "modulo" di dunia; 3,825,123,056,546,413,051 adalah bilangan komposit yang lolos uji Miller–Rabin untuk setiap basis prima hingga 31 dan baru tertangkap oleh basis kedua belas, 37; $2^{63} - 25$ adalah bilangan prima terbesar di bawah 2^{63}; dan $2^{63} - 1 = 7^2 \cdot 73 \cdot 127 \cdot 337 \cdot 92737 \cdot 649657$ adalah komposit.

13.6 Perkakas Geometri: Orientasi, Selimut Konveks, dan Luas

13.6.1 Perkalian silang

Untuk tiga titik a, b, c di bidang, **perkalian silang** vektor $b - a$ dan $c - a$ adalah

$$\text{cross}(a, b, c) = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x). \tag{13.6}$$

Tandanya menyatakan *orientasi*: positif bila lintasan $a \to b \to c$ berbelok ke kiri (berlawanan arah jarum jam), negatif bila berbelok ke kanan, nol bila ketiga titik segaris. Nilai mutlaknya sama dengan dua kali luas segitiga abc [9]. Dengan koordinat bulat yang nilai mutlaknya hingga 10^6, setiap hasil kali paling besar sekitar 4×10^{12}, jauh di dalam jangkauan `long long`. Hampir setiap predikat geometri dibangun dari Persamaan (13.6): dua ruas garis berpotongan bila ujung-ujung masing-masing terletak pada sisi yang berbeda dari ruas lainnya, sebuah titik berada di dalam poligon konveks bila ia berada di kiri setiap sisinya, dan seterusnya.

13.6.2 Selimut konveks

Selimut konveks (*convex hull*) sekumpulan titik adalah poligon konveks terkecil yang memuat semuanya, bentuk yang diambil karet gelang ketika direntangkan mengelilingi paku-paku di titik-titik itu. Untuk kebun di Subbab 13.1, itulah pagar terpendek yang mengelilingi setiap pohon. Algoritma **monotone chain Andrew** [10] membangunnya dalam $O(n \log n)$: urutkan titik menurut x (lalu y), sapu dari kiri ke kanan sambil menyimpan selimut *bawah* sebagai tumpukan, dan keluarkan titik terakhir selama ia dan titik baru membentuk belokan yang bukan ke kiri; lalu sapu dari kanan ke kiri untuk selimut *atas*. Setiap titik didorong dan dikeluarkan paling banyak sekali per sapuan, sehingga setelah pengurutan kerjanya linear. Dengan uji `cross(...) <= 0`, titik yang terletak pada sisi selimut dibuang; dengan `< 0` titik itu akan dipertahankan, dan sebagian soal meminta yang satu, sebagian yang lain.

13.6.3 Luas poligon

Luas poligon sederhana dengan titik sudut $p_0, p_1, \ldots, p_{k-1}$ secara berurutan diberikan oleh **rumus shoelace** (rumus tali sepatu) [9]:

$$A = \frac{1}{2}\left|\sum_{i=0}^{k-1}(x_i\, y_{i+1} - y_i\, x_{i+1})\right|, \qquad p_k = p_0. \tag{13.7}$$

Setiap suku adalah perkalian silang dua titik sudut berurutan dilihat dari titik asal, dua kali luas bertanda sebuah segitiga; segitiga-segitiga di luar poligon saling menghapus. Jumlah di dalam tanda mutlak adalah bilangan bulat untuk titik sudut bulat, sehingga program menyimpan *dua kali* luasnya sebagai bilangan bulat dan mencetak separuhnya di akhir, dengan `.0` atau `.5`: tidak ada pembulatan di mana pun.

Kode 13.7 (**Kode 13.8** dalam Python) menyatukan ketiga alat itu dan menjalankannya pada tiga belas titik; **Gambar 13.1** memperlihatkan hasilnya. Titik (0,2) terletak pada sisi antara (0,0) dan (0,4) sehingga dibuang, dan lima titik di dalam selimut diabaikan.

Kode 13.7 `geometry.cpp`: orientasi dengan perkalian silang, selimut konveks *monotone chain*, dan luas *shoelace*.

```cpp
// Perkakas geometri: orientasi, selimut konveks (monotone chain), luas poligon.
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

struct Pt {
    long long x, y;
    bool operator<(const Pt& o) const { return x < o.x || (x == o.x && y < o.y); }
};

// Perkalian silang (b - a) dan (c - a): > 0 belok kiri, < 0 kanan, 0 segaris.
long long cross(const Pt& a, const Pt& b, const Pt& c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

// Monotone chain Andrew: selimut berlawanan jarum jam, titik segaris dibuang.
vector<Pt> convexHull(vector<Pt> p) {
    sort(p.begin(), p.end());
    p.erase(unique(p.begin(), p.end(), [](const Pt& a, const Pt& b) {
        return a.x == b.x && a.y == b.y;
    }), p.end());
    if (p.size() < 3) return p;
    vector<Pt> h(2 * p.size());
    size_t k = 0;
    for (size_t i = 0; i < p.size(); i++) {  // selimut bawah, kiri ke kanan
        while (k >= 2 && cross(h[k - 2], h[k - 1], p[i]) <= 0) k--;
        h[k++] = p[i];
    }
    for (size_t i = p.size() - 1, t = k + 1; i-- > 0;) {  // selimut atas, kembali
        while (k >= t && cross(h[k - 2], h[k - 1], p[i]) <= 0) k--;
        h[k++] = p[i];
    }
    h.resize(k - 1);  // titik terakhir mengulang titik pertama
    return h;
}
```

Kode 13.7 (Lanjutan) *geometry.cpp: orientasi dengan perkalian silang, selimut konveks monotone chain, dan luas shoelace.*

```cpp
// Rumus shoelace: dua kali luas bertanda (positif bila berlawanan jarum jam).
long long twiceArea(const vector<Pt>& poly) {
    long long s = 0;
    for (size_t i = 0; i < poly.size(); i++) {
        const Pt& a = poly[i];
        const Pt& b = poly[(i + 1) % poly.size()];
        s += a.x * b.y - a.y * b.x;
    }
    return s;
}

int main() {
    Pt a{0, 0}, b{4, 0}, c{2, 3}, d{6, 0};
    printf("cross(a,b,c) = %lld (left turn)\n", cross(a, b, c));
    printf("cross(a,c,b) = %lld (right turn)\n", cross(a, c, b));
    printf("cross(a,b,d) = %lld (collinear)\n", cross(a, b, d));
    vector<Pt> pts = {{0, 0}, {4, -1}, {8, 1}, {8, 3}, {7, 5}, {3, 6}, {0, 4},
                      {0, 2}, {4, 2}, {2, 3}, {5, 3}, {6, 1}, {1, 1}};
    vector<Pt> hull = convexHull(pts);
    printf("hull (%zu vertices):", hull.size());
    for (const Pt& p : hull) printf(" (%lld,%lld)", p.x, p.y);
    long long a2 = twiceArea(hull);
    printf("\ntwice the area = %lld, area = %lld.%s\n", a2, a2 / 2,
           a2 % 2 ? "5" : "0");
    return 0;
}
```

Kode 13.8 `geometry.py`: perkakas yang sama dalam Python.

```python
# Perkakas geometri: orientasi, selimut konveks (monotone chain), luas poligon.


def cross(a, b, c):
    """Perkalian silang (b - a) dan (c - a): > 0 kiri, < 0 kanan, 0 segaris."""
    return (b[0] - a[0]) * (c[1] - a[1]) - (b[1] - a[1]) * (c[0] - a[0])


def convex_hull(points):
    """Monotone chain Andrew: berlawanan jarum jam, titik segaris dibuang."""
    p = sorted(set(points))
    if len(p) < 3:
        return p
    lower, upper = [], []
    for q in p:  # selimut bawah, kiri ke kanan
        while len(lower) >= 2 and cross(lower[-2], lower[-1], q) <= 0:
            lower.pop()
        lower.append(q)
    for q in reversed(p):  # selimut atas, kanan ke kiri
        while len(upper) >= 2 and cross(upper[-2], upper[-1], q) <= 0:
            upper.pop()
        upper.append(q)
    return lower[:-1] + upper[:-1]  # tiap rantai berakhir di awal rantai lainnya

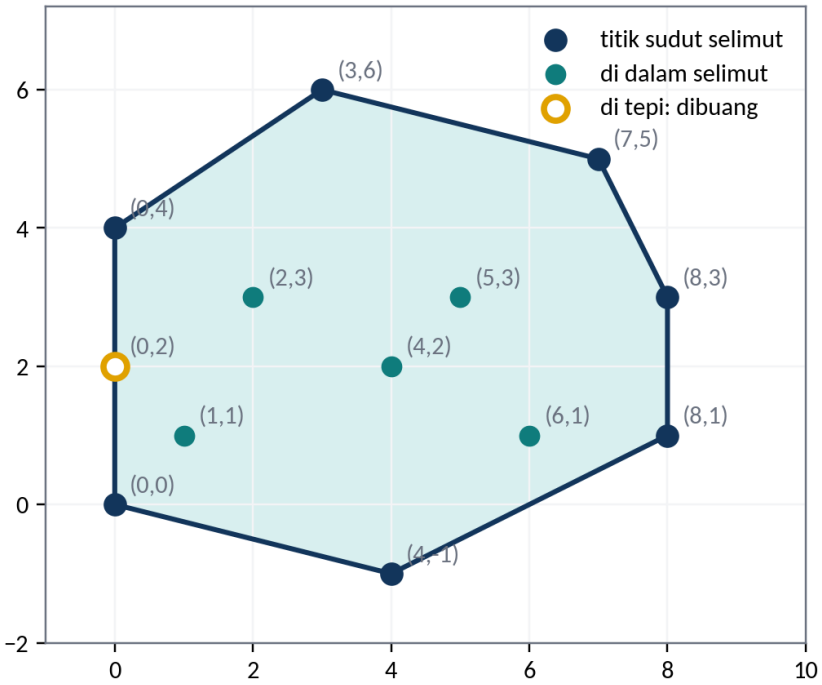

def twice_area(poly):
    """Rumus shoelace: 2 kali luas bertanda (positif bila berlawanan jarum jam)."""
    n = len(poly)
    return sum(poly[i][0] * poly[(i + 1) % n][1] - poly[i][1] * poly[(i + 1) % n][0]
               for i in range(n))


def main():
    a, b, c, d = (0, 0), (4, 0), (2, 3), (6, 0)
    print("cross(a,b,c) = %d (left turn)" % cross(a, b, c))
    print("cross(a,c,b) = %d (right turn)" % cross(a, c, b))
    print("cross(a,b,d) = %d (collinear)" % cross(a, b, d))
    pts = [(0, 0), (4, -1), (8, 1), (8, 3), (7, 5), (3, 6), (0, 4),
           (0, 2), (4, 2), (2, 3), (5, 3), (6, 1), (1, 1)]
    hull = convex_hull(pts)
    print("hull (%d vertices):" % len(hull), " ".join("(%d,%d)" % p for p in hull))
    a2 = twice_area(hull)
    half = "5" if a2 % 2 else "0"
    print("twice the area = %d, area = %d.%s" % (a2, a2 // 2, half))
```

Kode 13.8 (Lanjutan) *geometry.py: perkakas yang sama dalam Python.*

```python
main()
```

```
cross(a,b,c) = 12 (left turn)
cross(a,c,b) = -12 (right turn)
cross(a,b,d) = 0 (collinear)
hull (7 vertices): (0,0) (4,-1) (8,1) (8,3) (7,5) (3,6) (0,4)
twice the area = 86, area = 43.0
```



Gambar 13.1 Tiga belas titik pada **Kode 13.7**: tujuh titik sudut selimut, lima titik di dalam, dan satu titik pada sisi selimut yang dibuang oleh *monotone chain*.

Selimutnya memiliki tujuh titik sudut, didaftar berlawanan arah jarum jam mulai dari titik paling bawah-kiri, dan luasnya 43 satuan persegi. Untuk lahan Bu Ratna, tujuh patok sudut dalam meter memberi luas dalam meter persegi dengan rumus yang sama; dibagi 10.000 menjadi hektare. Pekerjaan sesungguhnya dalam penerapan seperti itu adalah mengubah lintang dan bujur GPS terlebih dahulu menjadi meter pada grid datar setempat, yang untuk lahan selebar beberapa ratus meter dapat dilakukan dengan faktor skala sederhana per derajat.

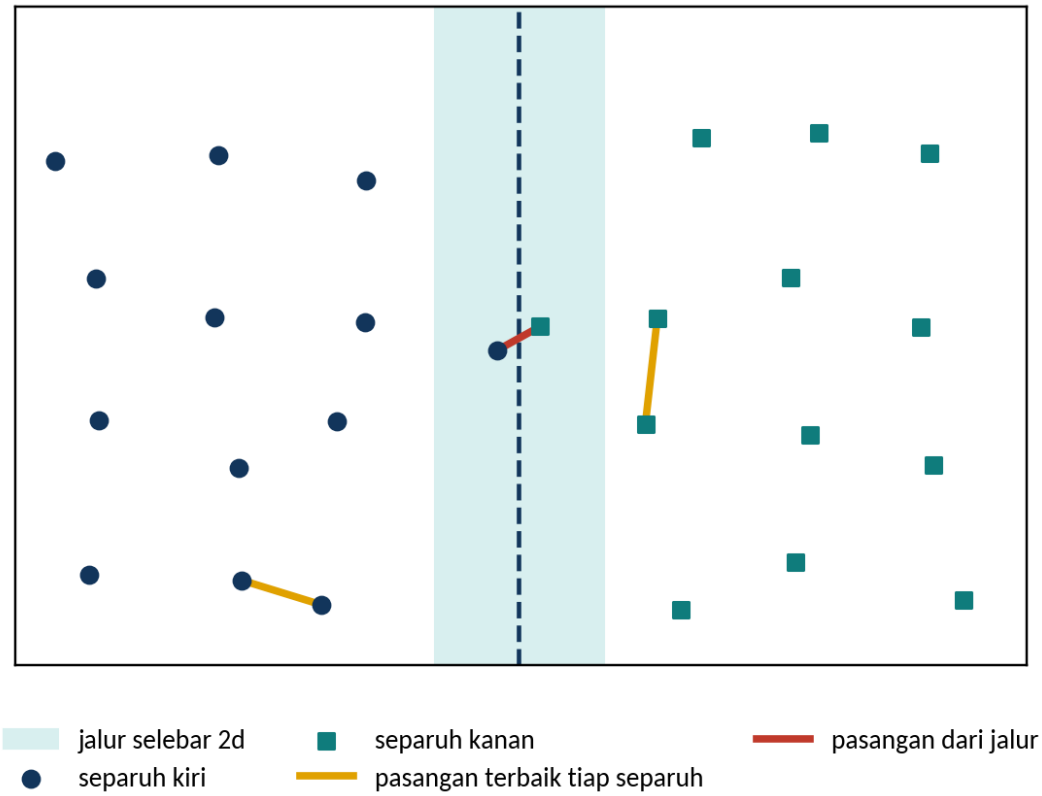
13.7 Pasangan Titik Terdekat (CLOPPAIR)

13.7.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ CLOPPAIR, "Closest Point Pair" [11], memberi $n \leq 50{,}000$ titik berbeda dengan koordinat bulat bernilai mutlak paling besar 10^6, dan menjamin tepat satu pasangan memiliki jarak terkecil. Program harus mencetak indeks pasangan itu (dihitung dari 0, indeks yang lebih kecil lebih dulu) dan jaraknya, dibulatkan hingga enam angka desimal.

13.7.2 Bagi dan kuasai

Membandingkan semua pasangan berbiaya $n(n-1)/2 \approx 1.25 \times 10^9$ perhitungan jarak: terlalu lambat. Algoritma klasik $O(n \log n)$ [12], [13] memakai bagi dan kuasai. Urutkan titik menurut x dan belah di median menjadi separuh kiri dan separuh kanan. Secara rekursif carilah pasangan terdekat di setiap separuh; misalkan d yang lebih kecil dari kedua jarak itu. Pasangan terdekat secara keseluruhan adalah salah satu dari kedua pasangan itu, atau pasangan dengan satu titik di setiap sisi garis pembagi. Pasangan seperti itu hanya dapat lebih dekat dari d bila kedua titiknya terletak di *jalur* selebar $2d$ di sekitar garis itu, pita yang diarsir pada **Gambar 13.2**.



Gambar 13.2 Langkah penggabungan algoritma pasangan terdekat: pasangan terbaik tiap separuh memberi d, dan hanya titik di jalur selebar $2d$ di sekitar garis pembagi yang perlu dibandingkan menyeberanginya.

Fakta pentingnya adalah bahwa di dalam jalur setiap titik hanya perlu dibandingkan dengan sejumlah konstan titik lain. Urutkan titik-titik jalur menurut y. Untuk sebuah titik p, titik q mana pun di jalur yang lebih dekat dari d juga harus berjarak kurang dari d secara vertikal. Persegi panjang berukuran $2d \times d$ di atas p paling banyak memuat 8 titik, karena masing-masing dari kedua perseginya yang berukuran $d \times d$ terletak dalam satu separuh dan titik-titik dalam separuh yang sama berjarak paling sedikit d satu sama lain, sehingga paling banyak 4 muat di setiap persegi. Karena itu gelung dalam, yang berjalan mundur dari p menyusuri jalur menurut urutan y dan berhenti begitu selisih vertikal mencapai d, memeriksa sejumlah konstan titik. Langkah jalur itu linear, dan waktu jalannya memenuhi $T(n) = 2T(n/2) + O(n)$, yang penyelesaiannya $O(n\log n)$ (rekurens jenis yang dianggarkan Bab 3).

Dua detail implementasi membuat **Kode 13.9** dan **Kode 13.10** cepat dan sederhana. Rekursinya mengembalikan titik-titiknya terurut menurut y, dengan menggabungkan kedua separuh yang sudah terurut, persis seperti *merge sort*; jadi tidak ada pengurutan di dalam rekursi. Dan semua perbandingan memakai jarak *kuadrat*, bilangan bulat eksak hingga 8×10^{12}; akar kuadrat diambil sekali saja, di akhir, untuk pencetakan.

Kode 13.9 `cloppair.cpp`: bagi dan kuasai atas jarak kuadrat bulat; setiap panggilan meninggalkan bagian lariknya terurut menurut y.

```cpp
// CLOPPAIR - Pasangan titik terdekat: bagi dan kuasai dalam O(n log n).
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <vector>
using namespace std;

struct Pt {
    long long x, y;
    int id;
};

long long bestD2 = -1;  // kuadrat jarak pasangan terbaik sejauh ini (-1: belum ada)
int bestA = -1, bestB = -1;

void consider(const Pt& p, const Pt& q) {
    long long dx = p.x - q.x, dy = p.y - q.y, d2 = dx * dx + dy * dy;
    if (bestD2 < 0 || d2 < bestD2) {
        bestD2 = d2;
        bestA = min(p.id, q.id);
        bestB = max(p.id, q.id);
    }
}

bool byY(const Pt& a, const Pt& b) { return a.y < b.y; }

// Saat masuk pts[lo, hi) terurut menurut x; saat keluar terurut menurut y.
void solve(vector<Pt>& pts, int lo, int hi, vector<Pt>& buf) {
    if (hi - lo <= 3) {
        for (int i = lo; i < hi; i++)
            for (int j = i + 1; j < hi; j++) consider(pts[i], pts[j]);
        sort(pts.begin() + lo, pts.begin() + hi, byY);
        return;
    }
    int mid = (lo + hi) / 2;
    long long midX = pts[mid].x;
    solve(pts, lo, mid, buf);
    solve(pts, mid, hi, buf);
    merge(pts.begin() + lo, pts.begin() + mid, pts.begin() + mid,
          pts.begin() + hi, buf.begin(), byY);
    copy(buf.begin(), buf.begin() + (hi - lo), pts.begin() + lo);
    int stripLen = 0;  // jalur disimpan di buf, terurut menurut y
    for (int i = lo; i < hi; i++) {
        long long dx = pts[i].x - midX;
        if (dx * dx >= bestD2) continue;  // terlalu jauh dari garis pembagi
        for (int j = stripLen - 1; j >= 0; j--) {
```

Kode 13.9 (Lanjutan) *cloppair.cpp: bagi dan kuasai atas jarak kuadrat bulat; setiap panggilan meninggalkan bagian lariknya terurut menurut y.*

```
            long long dy = pts[i].y - buf[j].y;
            if (dy * dy >= bestD2) break;  // hanya sedikit j yang lolos uji ini
            consider(pts[i], buf[j]);
        }
        buf[stripLen++] = pts[i];
    }
}

int main() {
    int n;
    if (scanf("%d", &n) != 1) return 0;
    vector<Pt> pts(n), buf(n);
    for (int i = 0; i < n; i++) {
        if (scanf("%lld %lld", &pts[i].x, &pts[i].y) != 2) return 0;
        pts[i].id = i;
    }
    sort(pts.begin(), pts.end(), [](const Pt& a, const Pt& b) {
        return a.x < b.x || (a.x == b.x && a.y < b.y);
    });
    solve(pts, 0, n, buf);
    printf("%d %d %.6f\n", bestA, bestB, sqrt((double)bestD2));
    return 0;
}
```

Kode 13.10 `cloppair.py`: algoritma yang sama dalam Python, setiap panggilan mengembalikan *list* baru terurut menurut *y*.

```python
# CLOPPAIR - Pasangan titik terdekat: bagi dan kuasai dalam O(n log n).
import sys
from heapq import merge
from math import sqrt


def closest(pts):
    """pts terurut menurut x; mengembalikan ((d2, a, b), pts terurut menurut y)."""
    n = len(pts)
    best = (float("inf"), -1, -1)
    if n <= 3:
        for i in range(n):
            for j in range(i + 1, n):
                (x1, y1, a), (x2, y2, b) = pts[i], pts[j]
                d2 = (x1 - x2) ** 2 + (y1 - y2) ** 2
                best = min(best, (d2, min(a, b), max(a, b)))
        return best, sorted(pts, key=lambda p: p[1])
    mid = n // 2
    mid_x = pts[mid][0]
    left, ly = closest(pts[:mid])
    right, ry = closest(pts[mid:])
    best = min(best, left, right)
    by_y = list(merge(ly, ry, key=lambda p: p[1]))
    strip = []  # titik dekat garis pembagi, terurut menurut y
    for p in by_y:
        if (p[0] - mid_x) ** 2 >= best[0]:
            continue  # terlalu jauh dari garis pembagi
        for q in reversed(strip):
            if (p[1] - q[1]) ** 2 >= best[0]:
                break  # hanya sedikit q yang lolos uji ini
            d2 = (p[0] - q[0]) ** 2 + (p[1] - q[1]) ** 2
            best = min(best, (d2, min(p[2], q[2]), max(p[2], q[2])))
        strip.append(p)
    return best, by_y


def main():
    data = sys.stdin.buffer.read().split()
    n = int(data[0])
    pts = sorted((int(data[1 + 2 * i]), int(data[2 + 2 * i]), i) for i in range(n))
    (d2, a, b), _ = closest(pts)
    print("%d %d %.6f" % (a, b, sqrt(d2)))


main()
```

Titik-titik membawa indeks aslinya melewati pengurutan, dan pasangannya disimpan sebagai (indeks lebih kecil, indeks lebih besar), karena keluaran menuntut $a < b$ dan rekursi dapat menjumpai kedua titik dalam urutan mana pun.

13.7.3 Menjalankannya

```
6
0 0
5 5
-3 4
6 7
10 -2
1 9
```

```
1 3 2.236068
```

Titik 1 dan 3 adalah (5,5) dan (6,7), berjarak $\sqrt{1+4} = \sqrt{5} \approx 2.236068$. *Format string* `%.6f` mencetak tepat enam angka desimal, sesuai permintaan soal; `cout << d` tanpa `fixed` dan

`setprecision(6)` akan mencetak `2.23607`, sebuah *Wrong Answer* pada juri yang membandingkan teks.

13.8 Mencari Jarum dalam Tumpukan Jerami: Knuth–Morris–Pratt (NHAY)

13.8.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ NHAY, "A Needle in the Haystack" [14], memberi beberapa kasus uji hingga akhir masukan. Setiap kasus memberi panjang sebuah *needle* (jarum, yaitu pola), *needle*-nya, dan sebuah *haystack* (tumpukan jerami, yaitu teks) yang panjangnya sengaja tidak dibatasi, dengan catatan bahwa program sebaiknya tidak menyimpan seluruh *haystack*. Untuk setiap kasus program mencetak setiap posisi (dihitung dari 0) tempat *needle* muncul dalam *haystack*, satu per baris. Pemeriksa bawaan SPOJ membandingkan keluaran token demi token dan mengabaikan baris kosong tambahan, sehingga baris kosong yang dicetak program kita setelah setiap kasus uji tidak berbahaya.

13.8.2 Border dan fungsi prefiks

> **DEFINISI.** *Border* sebuah *string* s adalah *string* yang sekaligus merupakan prefiks sejati dan sufiks dari s. Sebagai contoh, `aba` memiliki border `a` dan *string* kosong; `abab` memiliki border `ab`. *Fungsi prefiks* sebuah *string* s sepanjang m adalah larik $\pi[0..m-1]$ dengan $\pi[i]$ panjang border terpanjang dari prefiks $s[0..i]$.

Untuk *needle* `aabaab`, fungsi prefiksnya adalah $\pi = [0,1,0,1,2,3]$: prefiks `aabaab` diakhiri `aab`, yang juga merupakan awalnya. **Tabel 13.2** menghitungnya langkah demi langkah.

Tabel 13.2 Fungsi prefiks `aabaab`, satu prefiks per baris.

i	Prefiks $s[0..i]$	Border terpanjang	$\pi[i]$
0	`a`	(tidak ada)	0
1	`aa`	`a`	1
2	`aab`	(tidak ada)	0
3	`aaba`	`a`	1
4	`aabaa`	`aa`	2
5	`aabaab`	`aab`	3

Fungsi prefiks dapat dihitung dalam waktu $O(m)$ dengan memakai ulang nilai-nilai sebelumnya [12], [15]. Untuk memperpanjang border $s[0..i-1]$ sepanjang $k = \pi[i-1]$, bandingkan $s[i]$ dengan $s[k]$, karakter yang mengikuti border itu. Bila cocok, border bertambah menjadi $k+1$. Bila tidak, cobalah border berikutnya yang lebih pendek, yaitu border terpanjang dari border itu sendiri, sepanjang $\pi[k-1]$, dan ulangi; ketika k mencapai 0 dan tetap tidak cocok, $\pi[i] = 0$.

$$\pi[i] = \max\{\, k+1 : \; k \in \{\pi[i-1],\ \pi[\pi[i-1]-1],\ \ldots,\ 0\},\ s[k] = s[i] \,\} \cup \{0\} \tag{13.8}$$

Mengapa ini linear dan bukan kuadratik, padahal ada gelung `while` di dalamnya? Nilai k naik paling banyak satu per karakter dan setiap langkah gelung `while` menurunkannya paling sedikit satu, sehingga total langkah `while` atas seluruh *string* paling banyak sama dengan total kenaikan: paling banyak m.

13.8.3 Mencari dengan fungsi prefiks

Pencarian **Knuth–Morris–Pratt** (KMP) [15] menjalankan gelung yang sama atas *haystack*. Ia menyimpan k, panjang prefiks *needle* terpanjang yang berakhir di posisi *haystack* saat ini. Setiap karakter *haystack* yang baru entah memperpanjang kecocokan itu, entah membuat pencarian mundur menyusuri border-border *needle* persis seperti pada Persamaan (13.8); ia tidak pernah bergerak mundur di *haystack*. Ketika k mencapai m, *needle* berakhir di sini, di posisi i, jadi dimulai di $i - m + 1$; pencarian lalu mundur ke $\pi[m-1]$ untuk menemukan kemunculan yang bertumpang-tindih. Seluruh pencarian berbiaya $O(n+m)$ dan membaca *haystack* tepat sekali, dari kiri ke kanan. Itulah sebabnya **Kode 13.11** dapat membaca *haystack* satu karakter demi satu karakter dari *buffer* masukan tanpa pernah menyimpannya: hanya *needle* dan fungsi prefiksnya yang disimpan.

Kode 13.11 `nhay.cpp`: KMP yang mengalirkan *haystack* karakter demi karakter, hanya menyimpan *needle* dan fungsi prefiksnya.

```cpp
// NHAY - Jarum dalam tumpukan jerami: Knuth-Morris-Pratt, membaca haystack
// satu karakter demi satu karakter sehingga tak perlu disimpan.
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

static char buf[1 << 16];
static size_t bufLen = 0, bufPos = 0;

int readChar() {
    if (bufPos == bufLen) {
        bufLen = fread(buf, 1, sizeof(buf), stdin);
        bufPos = 0;
        if (bufLen == 0) return -1;
    }
    return buf[bufPos++];
}

bool isSpace(int c) { return c == ' ' || c == '\n' || c == '\r' || c == '\t'; }

// pi[i] = panjang border sejati terpanjang dari pattern[0..i].
vector<int> prefixFunction(const string& s) {
    vector<int> pi(s.size(), 0);
    for (size_t i = 1; i < s.size(); i++) {
        int k = pi[i - 1];
        while (k > 0 && s[i] != s[k]) k = pi[k - 1];  // coba border lebih pendek
        if (s[i] == s[k]) k++;
        pi[i] = k;
    }
    return pi;
}

int main() {
    while (true) {
        int c = readChar();
        while (c != -1 && (c < '0' || c > '9')) c = readChar();
        if (c == -1) break;
        long long len = 0;
        for (; c >= '0' && c <= '9'; c = readChar()) len = len * 10 + (c - '0');
        string needle;
        c = readChar();
        while (c != -1 && isSpace(c)) c = readChar();
        for (; c != -1 && !isSpace(c); c = readChar()) needle += (char)c;
        vector<int> pi = prefixFunction(needle);
        int m = (int)needle.size(), k = 0;
```

Kode 13.11 (Lanjutan) *nhay.cpp: KMP yang mengalirkan haystack karakter demi karakter, hanya menyimpan needle dan fungsi prefiksnya.*

```
        c = readChar();
        while (c != -1 && isSpace(c)) c = readChar();
        for (long long i = 0; c != -1 && !isSpace(c); c = readChar(), i++) {
            while (k > 0 && (k == m || c != needle[k])) k = pi[k - 1];
            if (c == needle[k]) k++;
            if (k == m) printf("%lld\n", i - m + 1);  // cocok, berakhir di posisi i
        }
        putchar('\n');  // baris kosong setelah tiap kasus uji
        (void)len;      // baris panjang hanya mengumumkan ukuran needle
    }
    return 0;
}
```

Kode 13.12 `nhay.py`: algoritma yang sama dalam Python, yang membaca setiap *haystack* sebagai satu *string*.

```
# NHAY - Jarum dalam tumpukan jerami: Knuth-Morris-Pratt.
import sys


def prefix_function(s):
    """pi[i] = panjang border sejati terpanjang dari s[0..i]."""
    pi = [0] * len(s)
    for i in range(1, len(s)):
        k = pi[i - 1]
        while k > 0 and s[i] != s[k]:
            k = pi[k - 1]  # mundur ke border yang lebih pendek
        if s[i] == s[k]:
            k += 1
        pi[i] = k
    return pi


def kmp_positions(needle, hay):
    pi, m, k, found = prefix_function(needle), len(needle), 0, []
    for i, ch in enumerate(hay):
        while k > 0 and (k == m or ch != needle[k]):
            k = pi[k - 1]
        if ch == needle[k]:
            k += 1
        if k == m:
            found.append(i - m + 1)  # cocok, berakhir di posisi i
    return found


def main():
    data = sys.stdin.buffer.read().split()
    out = []
    for i in range(0, len(data) - 2, 3):
        needle, hay = data[i + 1], data[i + 2]
        out.extend(map(str, kmp_positions(needle, hay)))
        out.append("")  # baris kosong setelah tiap kasus uji
    sys.stdout.write("\n".join(out) + ("\n" if out else ""))


main()
```

Versi Python, **Kode 13.12**, memang menyimpan setiap *haystack*, karena membaca satu karakter demi satu karakter dalam Python itu lambat; pada *haystack* yang sangat besar ia memakai memori sebanding dengan panjangnya. *Byte string* dari `sys.stdin.buffer` diindeks sebagai bilangan bulat, dan perbandingannya tetap bekerja tanpa perubahan.

13.8.4 Menjalankannya

```
2
na
bananana
3
xyz
abc
4
aaaa
aaaaaa
```

```
2
4
6


0
1
2
```

`na` muncul dalam `bananana` di posisi 2, 4, dan 6; `xyz` tidak muncul dalam `abc`, sehingga kasus itu hanya mencetak baris kosongnya; dan `aaaa` muncul dalam `aaaaaa` di 0, 1, dan 2, tiga kemunculan bertumpang-tindih yang akan terlewat oleh pencarian yang melompati setiap kecocokan.

13.9 Prefiks yang Berulang: Periode Sebuah String (PERIOD)

13.9.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ PERIOD, "Period" [16], memberi beberapa *string* dengan panjang hingga 10^6. Untuk setiap prefiks sepanjang $i \geq 2$ yang berupa pengulangan sebuah blok yang lebih pendek, yaitu sama dengan suatu blok yang ditulis $K > 1$ kali berturut-turut, program mencetak i dan K terbesar yang mungkin, dalam urutan i menaik. Setiap kasus uji dimulai dengan baris `Test case #k`.

13.9.2 Gagasannya

Fungsi prefiks menjawab ini untuk semua prefiks sekaligus. Untuk prefiks sepanjang L dengan border terpanjang sepanjang $\pi[L-1]$, bilangan

$$p = L - \pi[L-1] \tag{13.9}$$

adalah panjang *periode* terpendeknya: menggeser prefiks itu sejauh p posisi membuatnya berimpit dengan dirinya sendiri, karena prefiks dan sufiks sepanjang $L-p$ sama. Prefiks itu merupakan pengulangan sebuah blok tepat ketika p membagi L (dan $p < L$, yaitu bordernya tidak kosong). Dalam hal itu bloknya sepanjang p dan diulang $K = L/p$ kali, dan tidak mungkin ada K yang lebih besar, karena blok yang lebih pendek akan memberi border yang lebih panjang. Jadi seluruh soalnya adalah: hitung π, lalu untuk setiap L dari 2 hingga n uji apakah $\pi[L-1] > 0$ dan $L \bmod (L - \pi[L-1]) = 0$. Waktu linear, dan *string* sepanjang 10^6 karakter tidak menjadi masalah; **Kode 13.13** dan **Kode 13.14** melakukan tepat hal itu.

Kode 13.13 `period.cpp`: fungsi prefiks dan Persamaan (13.9) untuk setiap prefiks.

```cpp
// PERIOD - Setiap prefiks yang berupa pengulangan: fungsi prefiks menjawab semua.
#include <cstdio>
#include <vector>
using namespace std;

int main() {
    int t;
    if (scanf("%d", &t) != 1) return 0;
    static char s[1000005];
    for (int tc = 1; tc <= t; tc++) {
        int n;
        if (scanf("%d %1000000s", &n, s) != 2) return 0;
        vector<int> pi(n, 0);
        for (int i = 1; i < n; i++) {
            int k = pi[i - 1];
            while (k > 0 && s[i] != s[k]) k = pi[k - 1];
            if (s[i] == s[k]) k++;
            pi[i] = k;
        }
        printf("Test case #%d\n", tc);
        for (int len = 2; len <= n; len++) {
            int p = len - pi[len - 1];  // periode terpendek prefiks ini
            if (pi[len - 1] > 0 && len % p == 0) printf("%d %d\n", len, len / p);
        }
        putchar('\n');
    }
    return 0;
}
```

Kode 13.14 `period.py`: hal yang sama dalam Python.

```python
# PERIOD - Setiap prefiks yang berupa pengulangan: fungsi prefiks menjawab semua.
import sys


def main():
    data = sys.stdin.buffer.read().split()
    t = int(data[0])
    out = []
    for tc in range(1, t + 1):
        n, s = int(data[2 * tc - 1]), data[2 * tc]
        pi = [0] * n
        for i in range(1, n):
            k = pi[i - 1]
            while k > 0 and s[i] != s[k]:
                k = pi[k - 1]
            if s[i] == s[k]:
                k += 1
            pi[i] = k
        out.append("Test case #%d" % tc)
        for length in range(2, n + 1):
            p = length - pi[length - 1]  # periode terpendek prefiks ini
            if pi[length - 1] > 0 and length % p == 0:
                out.append("%d %d" % (length, length // p))
        out.append("")
    sys.stdout.write("\n".join(out) + "\n")


main()
```

13.9.3 Menjalankannya

```
2
6
ababab
7
aaaaaab
```

```
Test case #1
4 2
6 3

Test case #2
2 2
3 3
4 4
5 5
6 6
```

Untuk `ababab`, prefiks `abab` adalah `ab` dua kali dan `ababab` adalah `ab` tiga kali; `aba` dan `ababa` berperiode 2, yang tidak membagi 3 maupun 5. Untuk `aaaaaab` setiap prefiks berisi `a` saja adalah `a` yang diulang, dan `b` terakhir memutus polanya, sehingga panjang 7 tidak didaftar.

13.10 Menghitung Substring Berbeda: Suffix Array (DISUBSTR)

13.10.1 Soalnya dengan kata-kata kita sendiri

Soal SPOJ DISUBSTR, "Distinct Substrings" [17], memberi hingga 20 *string* dengan panjang paling banyak 1.000 dan menanyakan, untuk masing-masing, berapa banyak substring tak kosong yang berbeda. Substring adalah potongan yang bersambung, dan dua kemunculan potongan yang sama dihitung sekali.

13.10.2 Gagasannya: setiap substring adalah prefiks sebuah sufiks

Setiap substring s adalah prefiks sebuah sufiks: substring yang dimulai di posisi i adalah prefiks sufiks $s[i..n-1]$. Secara total ada $n(n+1)/2$ pasangan (awal, panjang), dan pertanyaannya adalah berapa banyak di antaranya yang mengulang pasangan sebelumnya. Urutkan semua n sufiks secara alfabetis; daftar posisi awal yang terurut itu adalah **suffix array** [1], [18]. Dua sufiks yang berbagi prefiks bersama yang panjang akan bersebelahan dalam urutan, dan prefiks yang dimiliki bersama sebuah sufiks dengan sufiks *mana pun* sebelumnya tepat sama dengan prefiks yang dimilikinya bersama tetangga persis sebelumnya. Bila $\text{lcp}[i]$ adalah panjang prefiks bersama terpanjang sufiks ke-i dan ke-$(i-1)$ dalam urutan, maka sufiks ke-i menyumbang $\text{lcp}[i]$ prefiks yang sudah terhitung, dan

$$\#\,\text{substring berbeda} = \frac{n(n+1)}{2} - \sum_{i=1}^{n-1} \text{lcp}\,[i]. \tag{13.10}$$

13.10.3 Membangun lariknya

Kode 13.15 membangun *suffix array* dengan *prefix doubling*: mula-mula beri peringkat sufiks menurut karakter pertamanya, lalu menurut 2, 4, 8, … karakter pertamanya, setiap kali mengurutkan menurut pasangan (peringkat separuh pertama, peringkat separuh kedua). Setelah $\lceil \log_2 n \rceil$ putaran semua peringkat berbeda. Dengan `std::sort` di setiap putaran biayanya $O(n \log^2 n)$, beberapa ratus ribu perbandingan untuk $n = 1000$. Larik LCP lalu diperoleh dari

algoritma Kasai [19] dalam $O(n)$: proseslah sufiks menurut urutan posisi awalnya, dan manfaatkan fakta bahwa bila sufiks yang dimulai di i berbagi h karakter dengan pendahulunya dalam urutan, sufiks yang dimulai di $i+1$ berbagi paling sedikit $h-1$ karakter dengan pendahulunya sendiri.

Kode 13.15 `disubstr.cpp`: *suffix array* dengan *prefix doubling*, LCP dengan Kasai, dan Persamaan (13.10).

```cpp
// DISUBSTR - Substring berbeda: suffix array (prefix doubling) plus LCP.
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

vector<int> suffixArray(const string& s) {
    int n = (int)s.size();
    vector<int> sa(n), rank(n), tmp(n);
    for (int i = 0; i < n; i++) {
        sa[i] = i;
        rank[i] = (unsigned char)s[i];
    }
    for (int k = 1;; k <<= 1) {  // urutkan menurut 2k karakter pertama
        auto key = [&](int i) {
            return make_pair(rank[i], i + k < n ? rank[i + k] : -1);
        };
        sort(sa.begin(), sa.end(), [&](int a, int b) { return key(a) < key(b); });
        tmp[sa[0]] = 0;
        for (int i = 1; i < n; i++)
            tmp[sa[i]] = tmp[sa[i - 1]] + (key(sa[i - 1]) < key(sa[i]) ? 1 : 0);
        rank = tmp;
        if (rank[sa[n - 1]] == n - 1) break;  // semua peringkat berbeda: selesai
    }
    return sa;
}

// Algoritma Kasai: lcp[i] = prefiks bersama sufiks sa[i-1] dan sa[i].
vector<int> lcpArray(const string& s, const vector<int>& sa) {
    int n = (int)s.size();
    vector<int> rank(n), lcp(n, 0);
    for (int i = 0; i < n; i++) rank[sa[i]] = i;
    for (int i = 0, h = 0; i < n; i++) {
        if (rank[i] == 0) {
            h = 0;
            continue;
        }
        int j = sa[rank[i] - 1];
        while (i + h < n && j + h < n && s[i + h] == s[j + h]) h++;
        lcp[rank[i]] = h;
        if (h > 0) h--;  // sufiks berikutnya menyisakan minimal h - 1 karakter
    }
    return lcp;
}
```

Kode 13.15 (Lanjutan) *disubstr.cpp: suffix array dengan prefix doubling, LCP dengan Kasai, dan Persamaan (13.10).*

```cpp
int main() {
    int t;
    string s;
    if (!(cin >> t)) return 0;
    getline(cin, s);  // sisa baris pertama
    while (t-- && getline(cin, s)) {
        if (!s.empty() && s.back() == '\r') s.pop_back();
        long long n = (long long)s.size();
        if (n == 0) {
            cout << 0 << '\n';
            continue;
        }
        vector<int> sa = suffixArray(s), lcp = lcpArray(s, sa);
        long long total = n * (n + 1) / 2;  // semua substring, termasuk ulangan
        for (int x : lcp) total -= x;       // dimiliki bersama sufiks sebelumnya
        cout << total << '\n';
    }
    return 0;
}
```

Untuk $n \leq 1000$ Python mampu memakai konstruksi yang lebih sederhana, **Kode 13.16**: `sorted(range(n), key=lambda i: s[i:])` mengurutkan sufiks sebagai *string*, yang berbiaya $O(n^2 \log n)$ perbandingan karakter pada kasus terburuk tetapi berjalan dalam C di dalam interpreter, dan LCP antartetangga diukur secara langsung. Batasnya menentukan metodenya, seperti biasa: untuk $n = 10^5$ (soal SPOJ SUBST1 [20]) hanya konstruksi $O(n \log^2 n)$ dan Kasai yang memadai.

Kode 13.16 `disubstr.py`: sufiks terurut dan LCP antartetangga, cukup untuk $n \leq 1000$.

```python
# DISUBSTR - Substring berbeda: sufiks terurut plus prefiks bersama terpanjang.
import sys


def distinct_substrings(s):
    n = len(s)
    order = sorted(range(n), key=lambda i: s[i:])  # cukup untuk n <= 1000
    total = n * (n + 1) // 2  # setiap substring, termasuk yang berulang
    for a, b in zip(order, order[1:]):
        h = 0
        while a + h < n and b + h < n and s[a + h] == s[b + h]:
            h += 1
        total -= h  # prefiks yang sudah terlihat di sufiks sebelumnya
    return total


def main():
    lines = sys.stdin.read().split("\n")
    t = int(lines[0])
    out = [distinct_substrings(lines[i].rstrip("\r")) for i in range(1, t + 1)]
    print("\n".join(map(str, out)))


main()
```

Kedua program membaca *string baris demi baris* dan memakai karakternya apa adanya, karena pernyataan soal mengizinkan karakter selain huruf; membaca dengan `cin >> s` atau `split()` akan memotong *string* yang memuat spasi.

13.10.4 Menjalankannya

```
3
CACAO
AAAA
sate
```

```
12
4
10
```

Untuk `CACAO`: 3 huruf berbeda, 3 pasangan berbeda (`CA` muncul dua kali), 3 rangkaian tiga huruf, 2 rangkaian empat huruf, dan *string* utuhnya, total 12. `AAAA` hanya memiliki 4 (satu per panjang), dan `sate`, tanpa huruf berulang, memiliki semua $4 \cdot 5/2 = 10$.

13.11 Menguji Perkakasnya

Stress test di `Chapter13/tests` mengikuti rencana yang sama dengan Bab 12, dengan *brute force* yang tidak memercayai kecerdikan apa pun: `a ** b % 10` dihitung dengan pangkat utuhnya untuk LASTDIG, penghitungan dengan `gcd` untuk ETF, pembagian percobaan (dan, untuk bilangan di atas 10^{12}, daftar jawaban yang diketahui ditambah pemeriksaan Fermat dengan banyak basis) untuk PON, semua pasangan untuk CLOPPAIR, pemotongan di setiap posisi untuk NHAY, mencoba setiap panjang blok untuk PERIOD, dan `set` Python berisi semua substring untuk DISUBSTR. Generatornya sengaja memasukkan kasus-kasus sulit: bilangan Carmichael dan *strong pseudoprime* untuk PON, titik-titik pada grid mungil 7×7 untuk CLOPPAIR (dengan jaminan pasangan terdekat tunggal tetap ditegakkan), *needle* atas alfabet dua huruf sehingga kecocokan bertumpang-tindih sering terjadi, dan *string* berisi angka dan simbol untuk DISUBSTR. Hasil jalankannya, 200 masukan acak per soal dengan *sanitizer* aktif:

```
cloppair   200 random tests: C++ and Python agree with brute force
disubstr   200 random tests: C++ and Python agree with brute force
etf        200 random tests: C++ and Python agree with brute force
lastdig    200 random tests: C++ and Python agree with brute force
nhay       200 random tests: C++ and Python agree with brute force
pon        200 random tests: C++ and Python agree with brute force
period     200 random tests: C++ and Python agree with brute force
```

Dan satu masukan berukuran maksimum per soal, diukur waktunya:

```
cloppair  max input  721.5 KB   C++  0.051 s   Python  1.065 s
disubstr  max input   19.6 KB   C++  0.020 s   Python  0.075 s
etf       max input  134.5 KB   C++  0.043 s   Python  0.164 s
lastdig   max input    0.4 KB   C++  0.002 s   Python  0.030 s
nhay      max input  977.5 KB   C++  0.019 s   Python  0.283 s
period    max input  976.6 KB   C++  0.091 s   Python  0.717 s
pon       max input    8.6 KB   C++  0.004 s   Python  0.049 s
```

Setiap program C++ berjalan dalam sepersepuluh detik atau kurang pada masukan terbesarnya. Python cukup cepat untuk LASTDIG, PON, ETF (berkat pendekatan pemfaktoran), NHAY, dan DISUBSTR di mesin kami, dan sekitar satu detik untuk PERIOD dan CLOPPAIR, tempat juri dengan batas ketat mungkin menolaknya. Pengukuran NHAY memakai *haystack* berisi sejuta `a` dan *needle* `aaa…ab`, masukan yang membuat pencarian naif paling lambat: KMP tidak peduli.

13.12 Menggunakannya Secara Nyata

Luas dan pagar sebidang lahan. Ukurlah patok sudut (atau pohon) dengan GPS ponsel, ubah setiap posisi menjadi meter ke timur dan ke utara dari satu sudut, lalu masukkan titik-titik sudut

secara berurutan ke rumus *shoelace* untuk luasnya, dan semua pohon ke selimut konveks untuk pagarnya; panjang pagar adalah jumlah panjang sisi-sisi selimut. Bulatkan koordinat ke meter atau desimeter utuh dan kalikan 10 agar tetap dalam bilangan bulat.

Digit pemeriksa dan sisa bagi. Banyak nomor identitas diakhiri digit pemeriksa yang dihitung sebagai jumlah berbobot modulo 10 atau 11; digit yang salah ketik mengubah sisanya. Persamaan (13.1) yang sama yang menjaga LASTDIG tetap kecil memungkinkan program memverifikasi nomor seperti itu tanpa bilangan bulat besar.

Mencari di log dan obrolan. Rutin KMP adalah penyaring aliran: ia membaca teks sekali dan melaporkan kecocokan seketika, sehingga cocok untuk berkas log yang masih terus bertambah. Untuk satu pencarian dalam *string* yang tersimpan, fungsi pustaka (`std::string::find`, `str.find` pada Python) biasanya cukup cepat dalam praktik; mengenal KMP penting ketika masukan juri sengaja dibuat jahat, ketika teksnya tidak dapat disimpan, atau ketika fungsi prefiks itu sendiri yang menjadi jawabannya, seperti pada PERIOD.

Mengenali perkakas ini dalam soal baru. "Cetak jawabannya modulo $10^9 + 7$" berarti aritmetika modular dan pemangkatan cepat (dan sering juga invers modular $a^{p-2} \bmod p$, yang tidak lain Persamaan (13.4) yang disusun ulang). "Hingga 10^6, banyak kueri" berarti saringan. "Hingga 10^{18}, primakah" berarti Miller–Rabin. "Apakah titik ini di kiri garis itu", "luas", "pagar" berarti perkalian silang. "Kemunculan", "border", "periode" berarti fungsi prefiks; "substring berbeda" dan "substring berulang terpanjang" berarti *suffix array* dengan LCP.

13.13 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Luapan sebelum modulus. `a * b % m` dengan operan `int` meluap sebelum `%` diterapkan; ubah ke `long long` lebih dulu, dan untuk modulus di atas sekitar 3×10^9 pakailah hasil kali 128 bit.

Melupakan $b = 0$ atau $a = 0$. Pangkat dengan eksponen 0 bernilai 1; nol dipangkatkan bilangan positif bernilai 0. Gelung yang dimulai dari basis alih-alih dari 1 salah pada kasus pertama.

Sisa bagi negatif. Dalam C++, `-7 % 3` bernilai `-1`, bukan 2. Bila pengurangan dapat membuat nilai negatif, tambahkan modulus sebelum mengambil sisanya: `((x % m) + m) % m`. Operator `%` pada Python selalu mengembalikan hasil tak negatif untuk modulus positif, yang menyembunyikan *bug* ini ketika Anda memindahkan kode Python ke C++.

Fermat alih-alih Miller–Rabin. Uji Fermat meloloskan bilangan Carmichael seperti 561; selalu pakai Miller–Rabin dengan basis deterministik.

Bilangan pecahan padahal bilangan bulat cukup. Membandingkan kemiringan dengan `double` atau menghitung sudut dengan `atan2` mengundang galat pembulatan pada titik-titik segaris. Pakai perkalian silang atas bilangan bulat, dan ambil akar kuadrat hanya untuk pencetakan.

Pembulatan keluaran yang salah. Cetak tepat sebanyak angka desimal yang diminta soal, dengan `printf("%.6f")` atau `cout << fixed << setprecision(6)`.

Titik segaris pada selimut. Putuskan dari pernyataan soal apakah titik pada sisi selimut harus dipertahankan atau dibuang, lalu pilih `<= 0` atau `< 0` dalam uji belokan sesuai keputusan itu.

Kode string kuadratik. Membangun *string* dengan `s = s + c` berulang dalam gelung, memanggil `substr` di dalam gelung, atau memeriksa setiap posisi dengan `compare` masing-masing dapat berbiaya $O(n^2)$. Pakai fungsi prefiks, dan bangun *string* dengan `+=` atau `append`.

Membaca string yang mungkin memuat spasi. Pakai `getline` (C++) atau pecah masukan menjadi baris-baris (Python) bila pernyataan soal tidak menjanjikan satu kata saja.

13.14 Ringkasan Bab

- Aritmetika modular menjaga bilangan tetap kecil, Persamaan (13.1); **pemangkatan cepat** menghitung a^b mod m dalam $O(\log b)$ perkalian, Persamaan (13.2) (LASTDIG).
- **Totient Euler** diperoleh dari faktor-faktor prima, Persamaan (13.3); saringan menabulasinya untuk semua $n \le 10^6$ dalam $O(N \log\log N)$, dan pemfaktoran dengan prima kecil menjawab kueri tunggal (ETF).
- **Miller–Rabin** dengan dua belas bilangan prima pertama sebagai basis adalah uji keprimaan deterministik untuk setiap bilangan 64 bit; C++ membutuhkan hasil kali 128 bit (PON).
- **Perkalian silang** menjawab pertanyaan orientasi secara eksak; bersamanya datang **selimut konveks monotone chain** dalam $O(n \log n)$ dan luas **shoelace**, Persamaan (13.7).
- **Pasangan titik terdekat** ditemukan dengan bagi dan kuasai dalam $O(n \log n)$, memakai jarak kuadrat bulat dan argumen jalur (CLOPPAIR).
- **Fungsi prefiks** memberi pencocokan pola waktu linear (**KMP**, NHAY) dan periode setiap prefiks, Persamaan (13.9) (PERIOD).
- **Suffix array** dengan larik **LCP** menghitung substring berbeda menurut Persamaan (13.10) (DISUBSTR).
- Semua program diuji dengan *stress test* dan diukur waktunya; C++ nyaman di mana-mana, dan Python lolos pada sebagian besar soal ini di mesin kami.

13.15 Soal Latihan

1. Hitunglah digit terakhir $13^{1000003}$ dengan tangan, memakai siklus digit terakhir.
2. Telusuri pemangkatan cepat untuk 3^{13} mod 7: daftarlah bit-bit 13, kuadrat berjalan, dan hasil setelah setiap langkah.
3. Hitung $\varphi(100)$ dan $\varphi(97)$ dengan Persamaan (13.3), dan periksa $\varphi(100)$ dengan menghitung bilangan dalam $1..100$ yang tidak habis dibagi 2 maupun 5.
4. Tunjukkan bahwa 561 adalah bilangan Carmichael dengan memeriksa bahwa $a^{560} \equiv 1 \pmod 3$, $\pmod{11}$, dan $\pmod{17}$ untuk setiap a yang koprima dengan 561. (Petunjuk: 560 habis dibagi 2, 10, dan 16.)
5. Jalankan uji Miller–Rabin dengan basis $a = 2$ pada $n = 561$: tulis $560 = d \cdot 2^s$, hitung barisan Persamaan (13.5), dan tunjukkan bahwa 2 adalah saksi.
6. Pakailah Persamaan (13.6) untuk memutuskan apakah ruas dari (0,0) ke (4,4) dan ruas dari (0,4) ke (4,0) berpotongan, dan apakah ruas dari (0,0) ke (2,2) dan ruas dari (3,3) ke (5,0) berpotongan.
7. Hitung dengan rumus *shoelace* luas segi empat (0,0), (5,0), (6,4), (1,3).
8. Hitung fungsi prefiks `abacaba` lalu pakailah untuk mendaftar semua posisi kemunculan `aba` dalam `abacabacaba`.
9. Untuk *string* `abcabcabcab`, daftarlah prefiks yang dilaporkan PERIOD beserta K-nya.
10. Hitung substring berbeda dari `banana` dengan Persamaan (13.10): tuliskan keenam sufiks dalam urutan terurut, LCP masing-masing dengan pendahulunya, dan hasilnya.

13.16 Penyelesaian Soal Latihan

1. Hanya digit terakhir basis yang berpengaruh, jadi ini sama dengan digit terakhir $3^{1000003}$. Pangkat-pangkat 3 berakhir dengan 3,9,7,1 dan berulang setiap 4. $1{,}000{,}003 = 4 \cdot 250{,}000 + 3$, sehingga digit terakhirnya adalah yang ketiga dalam siklus: 7.

2. $13 = 1101_2$. Mulai dengan hasil $=1$, kuadrat $=3$. Bit 0 bernilai 1: hasil $=1 \cdot 3 = 3$; kuadrat $=9 \bmod 7 = 2$. Bit 1 bernilai 0: hasil tetap 3; kuadrat $=4$. Bit 2 bernilai 1: hasil $=3 \cdot 4 = 12 \bmod 7 = 5$; kuadrat $=16 \bmod 7 = 2$. Bit 3 bernilai 1: hasil $=5 \cdot 2 = 10 \bmod 7 = 3$. Jadi $3^{13} \bmod 7 = 3$. (Pemeriksaan: $3^6 \equiv 1$ menurut Fermat, sehingga $3^{13} = 3^{12} \cdot 3 \equiv 3$.)

3. $100 = 2^2 \cdot 5^2$: $\varphi(100) = 100 \cdot \frac{1}{2} \cdot \frac{4}{5} = 40$. Penghitungan: 50 bilangan genap, 20 kelipatan 5, 10 kelipatan 10 yang terhitung dua kali, sehingga $50 + 20 - 10 = 60$ bilangan berbagi faktor dengan 100, dan $100 - 60 = 40$ tidak. 97 adalah prima, sehingga $\varphi(97) = 96$.

4. Untuk a yang koprima dengan 561, a koprima dengan 3, 11, dan 17. Menurut Fermat, $a^2 \equiv 1 \pmod 3$, $a^{10} \equiv 1 \pmod{11}$, $a^{16} \equiv 1 \pmod{17}$. Karena 560 adalah kelipatan 2, 10, dan 16, $a^{560} \equiv 1$ modulo ketiga bilangan prima itu, jadi juga modulo hasil kalinya 561.

5. $560 = 35 \cdot 2^4$, jadi $d = 35$, $s = 4$. $2^{35} \bmod 561 = 263$; dikuadratkan: $263^2 \bmod 561 = 166$; $166^2 \bmod 561 = 67$; $67^2 \bmod 561 = 1$, dan kuadrat terakhir kembali bernilai 1. Barisannya 263,166,67,1,1: angka 1 pertama didahului 67, yang bukan 1 maupun 560. Jadi 67 adalah akar kuadrat 1 modulo 561 selain ± 1, yang mustahil modulo bilangan prima: 2 adalah saksi, dan 561 komposit. (Memang $67^2 - 1 = 66 \cdot 68$, dan $\gcd(66,561) = 33$ menyingkap faktornya.)

6. Pasangan pertama: untuk ruas $A = (0,0)$, $B = (4,4)$, $\text{cross}(A, B, (0,4)) = 4 \cdot 4 - 4 \cdot 0 = 16 > 0$ dan $\text{cross}(A, B, (4,0)) = 4 \cdot 0 - 4 \cdot 4 = -16 < 0$: ujung-ujung ruas lainnya berada di sisi yang berbeda. Secara simetris, untuk $C = (0,4)$, $D = (4,0)$: $\text{cross}(C, D, A) = 4 \cdot (-4) - (-4) \cdot 0 = -16$ dan $\text{cross}(C, D, B) = 4 \cdot 0 - (-4) \cdot 4 = 16$. Kedua uji berganti tanda, jadi ruas-ruasnya berpotongan (di (2,2)). Pasangan kedua: $\text{cross}((0,0), (2,2), (3,3)) = 0$ dan $\text{cross}((0,0), (2,2), (5,0)) = 2 \cdot 0 - 2 \cdot 5 = -10$; satu ujung terletak pada garisnya tetapi di luar ujung ruas (2,2), dan ujung-ujung (0,0) dan (2,2) terhadap ruas kedua memberi $\text{cross}((3,3), (5,0), (0,0)) = 2 \cdot (-3) - (-3)(-3) = -15$ dan $\text{cross}((3,3), (5,0), (2,2)) = 2 \cdot (-1) - (-3)(-1) = -5$, bertanda sama: tidak berpotongan.

7. Suku-suku $x_i y_{i+1} - y_i x_{i+1}$: $(0 \cdot 0 - 0 \cdot 5) = 0$; $(5 \cdot 4 - 0 \cdot 6) = 20$; $(6 \cdot 3 - 4 \cdot 1) = 14$; $(1 \cdot 0 - 3 \cdot 0) = 0$. Jumlahnya 34, luasnya $34/2 = 17$.

8. $\pi(\text{abacaba}) = [0,0,1,0,1,2,3]$. Untuk pencariannya, fungsi prefiks *needle* `aba` adalah [0,0,1]; menyusuri `abacabacaba`, panjang kecocokan mencapai 3 di posisi 2, 6, dan 10, sehingga `aba` dimulai di 0, 4, dan 8.

9. Periode terpendek setiap prefiks sepanjang paling sedikit 3 adalah 3 (blok `abc`). Ia membagi panjangnya untuk $L = 6$ ($K = 2$) dan $L = 9$ ($K = 3$); prefiks sepanjang 2, 3, 4, 5, 7, 8, 10, 11 bukan pengulangan ($L = 3$ memiliki border kosong). Keluaran: `6 2` dan `9 3`.

10. Sufiks terurut: `a`, `ana`, `anana`, `banana`, `na`, `nana`. LCP dengan pendahulunya: `ana`/`a` 1, `anana`/`ana` 3, `banana`/`anana` 0, `na`/`banana` 0, `nana`/`na` 2; jumlahnya 6. $n(n+1)/2 = 21$, sehingga ada $21 - 6 = 15$ substring berbeda.

Daftar Pustaka

[1] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

[2] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*, Book 2. Singapore: Lulu, 2020.

[3] SPOJ, "LASTDIG — The last digit," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/LASTDIG/

[4] SPOJ, "ETF — Euler Totient Function," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/ETF/

[5] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 6th ed., revised by D. R. Heath-Brown and J. H. Silverman. Oxford, UK: Oxford University Press, 2008.

[6] SPOJ, "PON — Prime or Not," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/PON/

[7] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, no. 1, pp. 128–138, 1980.

[8] J. Sorenson and J. Webster, "Strong pseudoprimes to twelve prime bases," *Mathematics of Computation*, vol. 86, no. 304, pp. 985–1003, 2017.

[9] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Berlin, Germany: Springer, 2008.

[10] A. M. Andrew, "Another efficient algorithm for convex hulls in two dimensions," *Information Processing Letters*, vol. 9, no. 5, pp. 216–219, 1979.

[11] SPOJ, "CLOPPAIR — Closest Point Pair," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/CLOPPAIR/

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[13] M. I. Shamos and D. Hoey, "Closest-point problems," in *Proc. 16th Annual Symposium on Foundations of Computer Science*, 1975, pp. 151–162.

[14] SPOJ, "NHAY — A Needle in the Haystack," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/NHAY/

[15] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[16] SPOJ, "PERIOD — Period," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/PERIOD/

[17] SPOJ, "DISUBSTR — Distinct Substrings," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/DISUBSTR/

[18] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[19] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," in *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2001, pp. 181–192.

[20] SPOJ, "SUBST1 — New Distinct Substrings," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/SUBST1/

Bab 14

Menuju Accepted: Studi Kasus Utuh dan Langkah Berikutnya

Bab-bab sebelumnya membongkar kerajinan ini menjadi bagian-bagiannya: membaca pernyataan soal, menganggarkan waktu dari batasan, masukan dan keluaran cepat, *flag* kompiler, memori, jebakan klasik, *stress test*, serta perkakas algoritmik pemrograman dinamis, pilihan *greedy*, *binary search*, graf, bilangan, geometri, dan *string*. Bab terakhir ini merakitnya kembali. Ia mengikuti tiga soal SPOJ dari pembacaan pertama pernyataan soal hingga program yang akan kita kirim dengan yakin, mengerjakan setiap langkah dalam urutan dan dengan kecermatan yang dituntut sebuah kontes, termasuk langkah yang paling sering dilewati pemula: membuktikan kepada diri sendiri, dengan *brute force* dan *stress test*, bahwa program itu benar sebelum juri mengatakannya. Salah satu dari ketiga studi kasus sengaja ditanami *bug*, supaya Anda dapat menyaksikan *stress test* menemukannya dan menyusutkannya menjadi masukan dua baris. Paruh kedua bab ini memandang ke luar: tempat berlatih setelah SPOJ, kontes apa saja yang berlangsung pada 2026 dan cara mengikutinya dari Indonesia, buku apa yang dibaca berikutnya, dan cara menata latihan berbulan-bulan yang mengubah teknik-teknik buku ini menjadi kebiasaan. Di akhir bab Anda akan mampu menjalankan seluruh alur kerja pada soal baru secara mandiri, dan Anda akan memegang rencana konkret untuk langkah selanjutnya.

14.1 Dari Pengiriman Pertama Menuju Kebiasaan

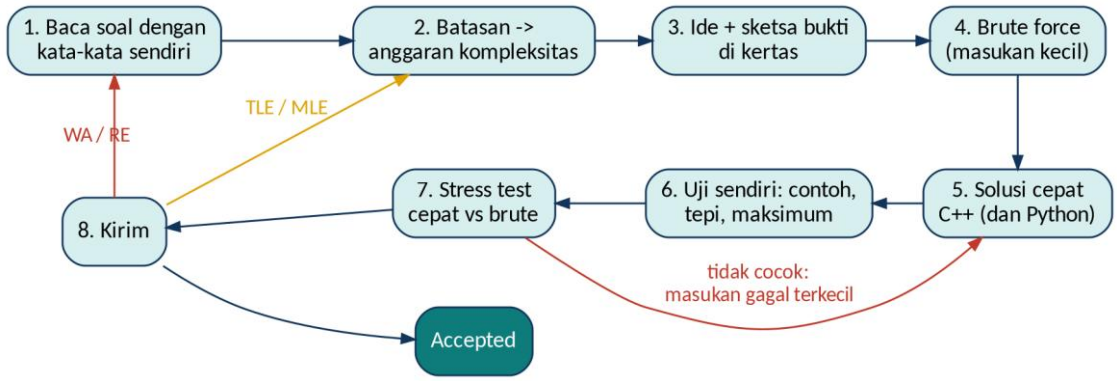
Dimas adalah mahasiswa informatika tahun kedua di Surabaya. Pada bulan pertamanya di SPOJ ia mengirim TEST, soal yang menggemakan bilangan sampai bertemu 42, dan mendapat *Accepted* pada percobaan ketiga, setelah belajar bahwa juri membandingkan keluaran secara persis. Lalu datang rentetan *Wrong Answer* pada soal yang ia yakini sudah terpecahkan, *Time Limit Exceeded* pada soal yang berjalan seketika di laptopnya, dan *Runtime Error* yang hanya muncul di juri. Setiap vonis mengajarinya sesuatu, dan setiap pelajaran itu adalah satu bab buku ini. Yang mengubahnya dari pengumpul vonis menjadi orang yang mengharapkan *Accepted* bukanlah algoritma baru. Melainkan sebuah rutinitas: urutan langkah yang sama, dikerjakan setiap kali, di kertas dan di terminal, sebelum menekan tombol *Submit*.

Rekan-rekan setimnya untuk Indonesia National Contest (INC) bulan Oktober memiliki tujuan yang sama. Mereka akan mendapat lima jam untuk sepuluh soal atau lebih, satu komputer untuk tiga orang, dan tanpa umpan balik apa pun selain vonis. Dalam suasana seperti itu rutinitas bukanlah kemewahan. Satu pengiriman yang salah berharga dua puluh menit penalti dalam penilaian gaya ICPC; *bug* yang ditemukan *stress test* di mesin sendiri tidak berharga apa-apa. Bab ini adalah rutinitas itu, dituliskan dan dipraktikkan tiga kali.

14.2 Alur Kerja

Gambar 14.1 memperlihatkan kedelapan langkahnya. Tiga langkah pertama berlangsung di kertas, empat berikutnya di depan papan ketik, dan langkah terakhir adalah satu-satunya yang melibatkan juri. Panah merah dan kuning adalah jalan kembali: ketidakcocokan dalam *stress test*

mengirim Anda kembali ke solusi cepat dengan sebuah masukan kecil yang gagal di tangan, dan vonis selain *Accepted* mengirim Anda kembali ke pembacaan (untuk *Wrong Answer* atau *Runtime Error*) atau ke anggaran kompleksitas (untuk *Time* atau *Memory Limit Exceeded*).



Gambar 14.1 Alur kerja utuh yang dipakai dalam setiap studi kasus bab ini, dengan jalan kembali setelah *stress test* gagal atau vonis selain *Accepted*.

Tabel 14.1 menjabarkan setiap langkah menjadi pertanyaan yang harus dijawab dan bab buku ini yang mengajarkannya. Tabel ini dimaksudkan untuk disalin ke selembar kartu dan diletakkan di samping papan ketik sampai tidak diperlukan lagi.

Tabel 14.1 Delapan langkah alur kerja, pertanyaan yang dijawab setiap langkah, dan tempat buku ini mengajarkannya.

Langkah	Pertanyaan yang harus dijawab	Bab
1. Baca	Apa persisnya yang ditanyakan? Bagaimana format masukan dan keluaran, termasuk pemisah, baris kasus, dan akhir masukan? Nyatakan ulang dalam dua kalimat.	2
2. Anggaran	n terbesar, banyaknya kasus uji, batas waktu dan memori: kompleksitas mana yang muat? Tipe apa (luapan)?	3, 6
3. Ide	Pola mana (DP, *greedy*, penelusuran, graf, teori bilangan, *string*)? Mengapa benar? Jalankan contoh dengan tangan.	9–13
4. *Brute force*	Program paling sederhana apa yang jelas benar pada masukan mungil?	8
5. Solusi cepat	C++ yang rapi dengan I/O cepat; Python bila anggarannya mengizinkan.	4, 5
6. Uji sendiri	Contoh, kasus terkecil dan terbesar, nilai-nilai yang sama, jebakan yang Anda kenal.	7
7. *Stress test*	Ratusan masukan kecil acak: solusi cepat melawan *brute force*, dengan *sanitizer*.	8
8. Kirim	Berkas yang benar, bahasa yang benar, tanpa keluaran *debug*, format keluaran sekali lagi.	1, 4

Waktu yang dihabiskan untuk langkah 1 sampai 4 terasa terbuang bagi pemula yang ingin segera mengetik kode. Kenyataannya justru sebaliknya. Pernyataan soal yang salah dibaca pada langkah 1 menghabiskan seluruh soal; kompleksitas yang salah ditaksir pada langkah 2 menghabiskan satu kali tulis ulang; dan *brute force* yang ditulis pada langkah 4 hanya memakan lima menit tetapi menghemat satu jam memandangi *Wrong Answer*. Laaksonen [1] dan Halim dkk. [2] memberi nasihat yang sama dengan kata-kata yang hampir sama: berpikirlah dulu, dan ujilah program Anda sendiri sebelum juri melakukannya.

14.3 Studi Kasus 1: Lintasan Prima (PPATH)

14.3.1 Langkah 1: membaca

Soal SPOJ PPATH, “Prime Path” [3], adalah teka-teki tentang bilangan prima empat digit. Kita mulai dari satu bilangan prima empat digit dan ingin mencapai bilangan prima lain dengan

mengubah satu digit setiap kali, dan setiap bilangan di sepanjang jalan harus juga bilangan prima empat digit (jadi digit pertama tidak boleh menjadi 0). Setiap perubahan berbiaya satu satuan; program mencetak biaya minimum, atau kata `Impossible` bila tujuan tidak dapat dicapai. Masukan dimulai dengan banyaknya kasus uji, paling banyak 100, diikuti satu baris per kasus berisi kedua bilangan prima itu.

Dalam dua kalimat kita sendiri: *bilangan prima empat digit adalah simpul-simpul sebuah graf, dan dua bilangan prima dihubungkan bila keduanya berbeda tepat pada satu digit; untuk setiap kasus uji, cetak panjang lintasan terpendek antara kedua bilangan prima yang diberikan, atau* `Impossible`. Pernyataan ulang itu sudah memuat algoritmanya, dan itulah sebabnya langkah 1 layak dikerjakan pelan-pelan.

14.3.2 Langkah 2: anggaran

Ada 1,061 bilangan prima empat digit. Masing-masing memiliki paling banyak $4 \times 9 = 36$ bilangan yang berbeda satu digit darinya, sehingga grafnya memiliki sekitar seribu simpul dan paling banyak $1{,}061 \times 36/2 \approx 19{,}000$ sisi. BFS berbiaya $O(n + m)$, beberapa puluh ribu langkah, dan 100 kasus uji berbiaya beberapa juta: praktis tidak ada. Bahkan Python pun muat dengan leluasa. Tipe: semua bilangan di bawah 10^4, dan jawabannya paling besar sebanyak simpulnya, jadi `int` lebih dari cukup.

14.3.3 Langkah 3: idenya dan mengapa benar

Semua sisi berbiaya satu, jadi ini *breadth-first search* dari Bab 12, pada graf yang tidak pernah disimpan: tetangga sebuah bilangan dibangkitkan seketika dengan mengganti setiap dari keempat digitnya dengan setiap digit lain dan mempertahankan hasil yang prima. Keprimaan diperoleh dari saringan Eratosthenes di bawah 10^4, dihitung sekali. BFS memberi banyak perubahan minimum berdasarkan wawasan kunci Subbab 12.4.2, dan bila tujuan tidak pernah dicapai antreannya kosong dan jawabannya `Impossible`. Awal yang sama dengan tujuan memberi 0 tanpa langkah apa pun.

Dengan tangan, pada kasus uji pertama kita sendiri 1009 → 9001: tetangga 1009 yang prima adalah 1019, 1039, 1049, 1069, 1109, 1409, 1609, 1709, 5009, dan 8009; tidak satu pun adalah tujuannya. BFS menemukan lintasan sepanjang 4,

$$1009 \to 1609 \to 1601 \to 9601 \to 9001, \tag{14.1}$$

dan tidak ada yang lebih pendek. Kedua bilangan berbeda pada digit ribuan dan satuan, jadi paling sedikit dibutuhkan dua perubahan; satu-satunya rute dua perubahan melewati 9009 atau 1001, dan keduanya bukan prima ($9009 = 9 \cdot 1001$ dan $1001 = 7 \cdot 11 \cdot 13$); dan BFS memeriksa secara menyeluruh bahwa rute tiga perubahan pun tidak ada.

14.3.4 Langkah 4: brute force

Brute force harus jelas benar, bukan cepat. Milik kita, di `Chapter14/tests/brute.py`, sama sekali tidak memakai BFS: ia menguji keprimaan dengan pembagian percobaan, menetapkan jarak sumber 0 dan setiap bilangan prima lain "tak hingga", lalu berulang-ulang merelaksasi tetangga setiap bilangan prima, gaya Bellman–Ford, sampai tidak ada jarak yang berubah. Ia benar berdasarkan argumen yang sama dengan Bellman–Ford [4], tidak berbagi kode dengan solusi cepat, dan memakan sepersekian detik yang terasa per sumber, yang tidak masalah untuk sebuah penguji.

14.3.5 Langkah 5: solusi cepat

Kode 14.1 menghitung saringan sekali lalu menjalankan satu BFS per kasus uji. Mengubah digit ke-`pos` (1, 10, 100, atau 1000) dari `u` adalah aritmetika: kurangi digit lama dikali nilai tempatnya, tambahkan digit baru. Gelung atas digit baru dimulai dari 1 untuk tempat ribuan, yang merupakan aturan "tanpa nol di depan" dari pernyataan soal dalam satu ungkapan.

Kode 14.1 `ppath.cpp`: saringan di bawah 10^4 dan satu *breadth-first search* per kasus uji.

```cpp
// PPATH - Lintasan prima: saring prima empat digit, lalu BFS.
#include <cstdio>
#include <queue>
#include <vector>
using namespace std;

int main() {
    vector<bool> composite(10000, false);  // saringan Eratosthenes di bawah 10^4
    composite[0] = composite[1] = true;
    for (int p = 2; p * p < 10000; p++)
        if (!composite[p])
            for (int q = p * p; q < 10000; q += p) composite[q] = true;
    const int POW10[4] = {1, 10, 100, 1000};
    int t;
    if (scanf("%d", &t) != 1) return 0;
    while (t--) {
        int from, to;
        if (scanf("%d %d", &from, &to) != 2) return 0;
        vector<int> dist(10000, -1);
        queue<int> q;
        dist[from] = 0;
        q.push(from);
        while (!q.empty() && dist[to] == -1) {
            int u = q.front();
            q.pop();
            for (int pos = 0; pos < 4; pos++) {  // ubah satu digit setiap kali
                int digit = u / POW10[pos] % 10;
                int base = u - digit * POW10[pos];
                for (int d = (pos == 3 ? 1 : 0); d <= 9; d++) {  // tanpa nol depan
                    int v = base + d * POW10[pos];
                    if (composite[v] || dist[v] != -1) continue;
                    dist[v] = dist[u] + 1;
                    q.push(v);
                }
            }
        }
        if (dist[to] == -1)
            puts("Impossible");
        else
            printf("%d\n", dist[to]);
    }
    return 0;
}
```

Kode 14.2 `ppath.py`: penelusuran yang sama dalam Python, dengan *generator* untuk tetangga dan penugasan irisan dalam saringan.

```python
# PPATH - Lintasan prima: saring prima empat digit, lalu BFS.
import sys
from collections import deque

composite = bytearray(10000)  # saringan Eratosthenes di bawah 10^4
composite[0] = composite[1] = 1
for p in range(2, 100):
    if not composite[p]:
        composite[p * p::p] = b"\x01" * len(range(p * p, 10000, p))


def neighbors(u):
    for pos in (1, 10, 100, 1000):  # ubah satu digit setiap kali
        base = u - (u // pos % 10) * pos
        for d in range(1 if pos == 1000 else 0, 10):  # tanpa nol depan
            v = base + d * pos
            if not composite[v]:
                yield v


def cost(src, dst):
    dist = {src: 0}
    q = deque([src])
    while q and dst not in dist:
        u = q.popleft()
        for v in neighbors(u):
            if v not in dist:
                dist[v] = dist[u] + 1
                q.append(v)
    return dist.get(dst)


def main():
    data = sys.stdin.read().split()
    out = []
    for i in range(int(data[0])):
        c = cost(int(data[1 + 2 * i]), int(data[2 + 2 * i]))
        out.append("Impossible" if c is None else str(c))
    print("\n".join(out))


main()
```

Saringan Python pada **Kode 14.2** memakai penugasan irisan (*slice assignment*), `composite[p * p::p] = ...`, yang menandai semua kelipatan `p` dalam satu panggilan yang diimplementasikan dalam C; inilah cara yang lazim untuk menulis saringan cepat dalam Python murni.

14.3.6 Langkah 6 dan 7: uji kita sendiri, lalu stress test

Contoh kita memuat satu lintasan panjang, satu lintasan sepanjang nol (sumber sama dengan tujuan), dan satu lintasan panjang lainnya:

```
3
1009 9001
2003 2003
4001 8837
```

```
4
0
4
```

Jawaban pertama cocok dengan penelusuran tangan Persamaan (14.1). Apakah `Impossible` pernah dicetak? Di antara bilangan prima empat digit, grafnya ternyata terhubung, sehingga program yang benar tidak pernah mencetaknya; program yang sama sekali lupa kata itu tetap

akan diterima, dan program yang mencetak `impossible` dengan huruf kecil tidak akan pernah ketahuan. Itu bukan alasan untuk ceroboh: pernyataan soal menyebut kata itu mungkin diperlukan, dan kodenya mencetak ejaan persis seperti pada pernyataan soal.

Stress test membandingkan C++, Python, dan *brute force* pada 200 masukan acak berisi hingga lima kasus uji masing-masing, lalu mengukur waktu satu masukan dengan jumlah maksimum 100 kasus uji:

```
ppath      200 random tests: C++ and Python agree with brute force
ppath     max input    1.0 KB   C++  0.016 s   Python  0.185 s
```

14.3.7 Langkah 8: mengirim

Sebelum mengirim, pemeriksaan terakhir: program membaca banyaknya kasus uji lebih dulu, mencetak satu baris per kasus uji, tidak menyisakan keluaran *debug* di `stdout`, dan berkas yang dikirim adalah `ppath.cpp` dengan bahasa diatur ke C++ (GCC). Seandainya ini Python, pengukuran di atas (di bawah 0,2 detik untuk 100 kasus uji) menyatakan bahwa ia pun aman.

14.4 Studi Kasus 2: Lintasan ABC Terpanjang (ABCPATH)

14.4.1 Langkah 1 sampai 3

Soal SPOJ ABCPATH, "ABC Path" [5], memberi grid berisi huruf kapital, masing-masing hingga 50×50, satu demi satu sampai baris `0 0`. Dari sel berisi `A`, lintasan boleh melangkah ke salah satu dari delapan sel tetangga (termasuk diagonal) bila huruf di sana adalah huruf berikutnya dalam alfabet: `A` ke `B`, `B` ke `C`, dan seterusnya. Untuk setiap grid program mencetak `Case C: X`, dengan `C` menghitung grid mulai dari 1 dan `X` panjang lintasan seperti itu yang terpanjang, yang bernilai 0 bila grid tidak memuat `A`.

Anggaran: 2.500 sel, masing-masing delapan tetangga. Penelusuran polos yang mencoba setiap lintasan bisa lambat pada grid yang banyak lintasannya berbagi ekor, jadi kita memakai memo: misalkan $L(r, c)$ panjang lintasan terpanjang yang *dimulai* di sel (r, c). Maka

$$\begin{aligned} L(r,c) \\ = 1 + \max(\{0\} \\ \cup \{\, L(r',c') : (r',c') \text{ bertetangga dengan } (r,c),\ \text{huruf}(r',c') \\ = \text{huruf}(r,c) + 1 \,\}) \end{aligned} \tag{14.2}$$

dan jawabannya adalah maksimum L atas sel-sel `A`. Rekursinya selalu berpindah ke huruf yang lebih besar, sehingga tidak dapat berputar-putar dan kedalamannya paling banyak 26 tingkat; setiap sel dihitung sekali, sehingga satu grid berbiaya $O(8 \cdot HW)$, dua puluh ribu langkah. Inilah pemrograman dinamis pada graf berarah tanpa siklus, persis pola Bab 9: huruf-hurufnya memberi urutan.

14.4.2 Langkah 4 dan 5

Brute force-nya adalah Persamaan (14.2) tanpa memo: rekursi polos yang mencoba setiap lintasan. Kasus terburuknya eksponensial, tetapi pada grid paling besar 5×5 dengan lima huruf berbeda ia selesai seketika. Solusi cepat dalam C++ adalah rekursi bermemo, **Kode 14.3**.

Kode 14.3 `abcpath.cpp`: penelusuran bermemo; tabel memo dikosongkan untuk setiap grid.

```cpp
// ABCPATH - Lintasan huruf berurutan terpanjang dari 'A': pencarian bermemo.
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

int h, w;
char grid[52][52];
int best[52][52];  // lintasan terpanjang mulai (r, c); -1 = belum dihitung

int longest(int r, int c) {
    int& res = best[r][c];
    if (res != -1) return res;
    res = 1;
    for (int dr = -1; dr <= 1; dr++)
        for (int dc = -1; dc <= 1; dc++) {  // 8 tetangga (dan (0,0), dilewati)
            int nr = r + dr, nc = c + dc;
            if (nr < 0 || nr >= h || nc < 0 || nc >= w) continue;
            if (grid[nr][nc] == grid[r][c] + 1) res = max(res, 1 + longest(nr, nc));
        }
    return res;
}

int main() {
    for (int tc = 1; scanf("%d %d", &h, &w) == 2 && (h || w); tc++) {
        for (int r = 0; r < h; r++)
            if (scanf("%51s", grid[r]) != 1) return 0;
        memset(best, -1, sizeof(best));  // lupakan grid sebelumnya
        int answer = 0;
        for (int r = 0; r < h; r++)
            for (int c = 0; c < w; c++)
                if (grid[r][c] == 'A') answer = max(answer, longest(r, c));
        printf("Case %d: %d\n", tc, answer);
    }
    return 0;
}
```

Dalam Python, kita menghindari rekursi sama sekali. Karena setiap langkah berpindah dari satu huruf ke huruf berikutnya, memproses sel dari `Z` turun ke `A` menjamin bahwa ketika sebuah sel diproses, semua sel yang dapat dituju darinya sudah selesai. **Kode 14.4** mengurutkan sel menurut huruf secara menurun dan mengisi tabel dalam urutan itu: rekurens yang sama, dievaluasi dari bawah ke atas.

Kode 14.4 `abcpath.py`: rekurens yang sama dievaluasi dari bawah ke atas, dari `Z` turun ke `A`.

```python
# ABCPATH - Lintasan huruf berurutan terpanjang dari 'A': pencarian bermemo.
import sys


def solve(grid):
    h, w = len(grid), len(grid[0])
    best = [[0] * w for _ in range(h)]  # lintasan terpanjang mulai (r, c)
    order = sorted(((r, c) for r in range(h) for c in range(w)),
                   key=lambda rc: grid[rc[0]][rc[1]], reverse=True)
    for r, c in order:  # dari 'Z' turun ke 'A': semua penerus sudah selesai
        res = 1
        for nr in range(max(0, r - 1), min(h, r + 2)):
            for nc in range(max(0, c - 1), min(w, c + 2)):
                if ord(grid[nr][nc]) == ord(grid[r][c]) + 1:
                    res = max(res, 1 + best[nr][nc])
        best[r][c] = res
    return max([best[r][c] for r in range(h) for c in range(w)
                if grid[r][c] == "A"], default=0)


def main():
    data = sys.stdin.read().split()
    pos, tc, out = 0, 1, []
    while pos + 1 < len(data):
        h, w = int(data[pos]), int(data[pos + 1])
        pos += 2
        if h == 0 and w == 0:
            break
        grid = data[pos:pos + h]
        pos += h
        out.append("Case %d: %d" % (tc, solve(grid)))
        tc += 1
    print("\n".join(out))


main()
```

```
3 4
ABCE
DCDF
AEFG
1 2
BC
0 0
```

```
Case 1: 7
Case 2: 0
```

Pada grid pertama lintasan `A B C D E F G` memakai `A` di kiri atas, salah satu `C`, `D` di kanan baris tengah, `E` di sudut kanan atas, lalu `F` dan `G` turun sepanjang tepi kanan: 7 huruf. Grid kedua tidak memuat `A`, sehingga jawabannya 0, kasus yang layak ada dalam contoh karena program yang menginisialisasi jawabannya dengan 1 akan gagal di situ.

14.4.3 Langkah 7 beraksi: menangkap bug yang ditanam

Sekarang bagian yang paling mendidik. `abcpath_wrong.cpp` adalah program yang sama dengan satu baris dipindahkan: tabel memo dikosongkan sekali di awal `main` alih-alih sekali per grid. Inilah persis *bug* yang ditulis peserta yang kelelahan, dan ia lolos dari contoh pada pernyataan soal, yang hanya memuat satu grid. Contoh kita sendiri di atas pun lolos, karena kebetulan. *Stress test* tidak mengandalkan kebetulan:

```
MISMATCH in abcpath (C++), test 1
3 3
DAC
ACC
DBC
1 3
DDA
4 5
ADEED
DECDE
DEEAE
BBAEA
5 4
BDBA
DDEC
EDCE
EDDD
BCCA
0 0

expected: Case 1: 4 Case 2: 1 Case 3: 2 Case 4: 3
got:      Case 1: 4 Case 2: 1 Case 3: 2 Case 4: 2
```

Grid keempat dari sebuah masukan acak mendapat 2 alih-alih 3. Mengapa? Sel-sel grid baru memakai ulang entri memo yang dihitung untuk grid-grid *sebelumnya* pada koordinat yang sama, dan nilai-nilai itu menggambarkan lintasan di grid yang lain. *Stress test* menemukan *bug* itu pada masukan acak keduanya. Masukan yang gagal itu masih agak besar untuk dinalar, jadi langkah berikutnya adalah *menyusutkannya*: buang grid dan sel selama kedua program masih berbeda pendapat. Beberapa menit mencoba (atau gelung kecil, seperti diperlihatkan Bab 8) menghasilkan masukan gagal berisi dua baris huruf:

```
1 2
AB
1 1
A
0 0
```

Program yang benar dan program yang ber-*bug* pada masukan itu:

```
Case 1: 2
Case 2: 1
Case 1: 2
Case 2: 2
```

Pada grid pertama, sel (0,0) berisi `A` dengan `B` di sebelahnya, sehingga entri memonya menjadi 2. Grid kedua hanya satu `A` di sel yang sama (0,0), dan program ber-*bug* membaca angka 2 yang basi itu. Begitu *bug*-nya sekecil ini, perbaikannya jelas: kosongkan memo untuk setiap grid, yaitu baris `memset(best, -1, sizeof(best));` di dalam gelung **Kode 14.3**. Inilah pelajaran umum Bab 8 dalam praktik: *setiap keadaan yang milik satu kasus uji harus direset untuk kasus berikutnya*, dan masukan acak dengan beberapa kasus uji per berkas adalah cara termurah untuk memeriksanya.

14.5 Studi Kasus 3: Satu dan Nol (ONEZERO)

14.5.1 Langkah 1 dan 2

Soal SPOJ ONEZERO, "Ones and zeros" [6], memberi sekitar 1.000 bilangan n dengan $1 \le n \le 20{,}000$. Untuk masing-masing, program harus mencetak kelipatan positif terkecil dari n yang semua digit desimalnya 0 atau 1. Untuk $n = 2$ jawabannya 10; untuk $n = 7$ jawabannya 1001, karena $1001 = 7 \times 143$.

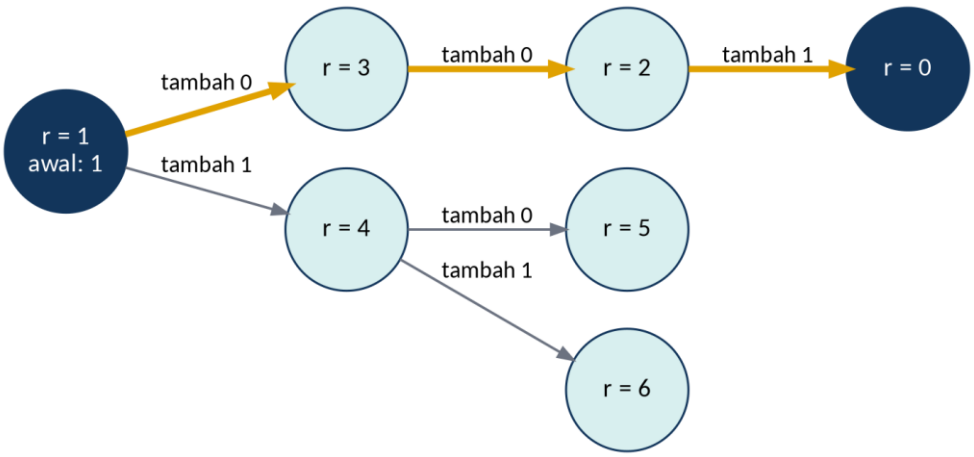
Pertanyaan anggarannya lebih sulit daripada kelihatannya, karena menyangkut *jawaban*, bukan masukan. Seberapa panjang jawabannya bisa? Untuk $n = 9999$ jawabannya 36 angka satu, bilangan 36 digit. Tidak ada tipe bilangan bulat bawaan C++ yang dapat menampungnya, sehingga pendekatan apa pun yang menyimpan bilangan-bilangan calon itu sendiri sebagai bilangan bulat berada dalam bahaya.

14.5.2 Langkah 3: penelusuran atas sisa bagi

Pengamatan kuncinya adalah bahwa untuk keterbagian oleh n hanya sisa bagi modulo n yang penting, dan menambahkan digit d di belakang bilangan x mengubah sisanya dengan cara yang hanya bergantung pada sisa lama:

$$(10x + d) \bmod n = (10\,(x \bmod n) + d) \bmod n. \tag{14.3}$$

Jadi bayangkan n kemungkinan sisa bagi sebagai simpul sebuah graf, dengan sisi "tambah 0" dan sisi "tambah 1" keluar dari setiap simpul. Bilangan 1 bersisa 1 mod n, dan kita ingin mencapai sisa 0 dengan barisan digit tambahan terpendek, karena lebih sedikit digit berarti bilangan yang lebih kecil. Itu lagi-lagi *breadth-first search*, pada graf berisi $n \le 20{,}000$ simpul, tidak pernah atas bilangan-bilangan raksasanya. **Gambar 14.2** memperlihatkan penelusuran untuk $n = 7$.



Gambar 14.2 *Breadth-first search* atas sisa bagi modulo 7: lintasan yang disorot menambahkan 0, 0, 1 pada angka 1 di awal dan mencapai sisa 0, sehingga jawabannya 1001.

Tiga fakta membuatnya benar.

1. *Penelusurannya berakhir.* Di antara $n + 1$ bilangan 1,11,111, ... (dengan hingga $n + 1$ angka satu), dua memiliki sisa yang sama menurut prinsip sarang merpati, dan selisihnya berbentuk 11 ... 100 ... 0, sebuah kelipatan n yang ditulis dengan 0 dan 1. Jadi ada jawaban dengan paling banyak n digit, dan BFS, yang mengunjungi setiap sisa paling banyak sekali, menemukan salah satunya.
2. *BFS menemukan jawaban terpendek, dan di antara yang sama pendeknya, yang terkecil.* BFS mencapai bilangan-bilangan menurut urutan panjangnya; di dalam satu panjang, mencoba "tambah 0" sebelum "tambah 1" mengunjunginya dalam urutan numerik menaik, karena antrean menjaga urutan prefiks-prefiksnya.
3. *Pemangkasan menurut sisa bagi itu aman.* Bila dua bilangan $x < y$ memiliki sisa yang sama, maka untuk digit apa pun yang ditambahkan pada keduanya, hasilnya bersisa sama dan yang dibangun di atas x lebih kecil. Jadi begitu sebuah sisa sudah dicapai, bilangan berikutnya dengan sisa yang sama boleh dilupakan.

Untuk mencetak jawabannya, setiap sisa menyimpan sisa induknya dan digit yang mengantarnya ke sana; menelusuri balik dari sisa 0 lalu membalik urutannya menghasilkan digit-digitnya.

14.5.3 Langkah 5, dan jalan salah yang menggoda

Sebelum program yang benar, program yang salah, karena itulah yang paling sering ditulis orang pertama kali. **Kode 14.5** menjalankan BFS atas bilangan-bilangan calon 1,10,11,100, ... yang disimpan sebagai bilangan bulat tak bertanda 64 bit.

Kode 14.5 `onezero_wrong.cpp`: BFS atas bilangan itu sendiri, yang meluap diam-diam begitu melewati dua puluh digit.

```cpp
// ONEZERO yang menggoda tetapi salah: BFS atas bilangan itu sendiri.
#include <cstdio>
#include <queue>
using namespace std;

int main() {
    int k;
    if (scanf("%d", &k) != 1) return 0;
    while (k--) {
        int n;
        if (scanf("%d", &n) != 1) return 0;
        queue<unsigned long long> q;  // 1, 10, 11, 100, 101, 110, 111, ...
        q.push(1);
        while (true) {
            unsigned long long x = q.front();
            q.pop();
            if (x % n == 0) {
                printf("%llu\n", x);
                break;
            }
            q.push(x * 10);      // meluap diam-diam setelah 20 digit
            q.push(x * 10 + 1);
        }
    }
    return 0;
}
```

Program itu benar untuk setiap n yang jawabannya paling panjang 19 digit. Untuk jawaban yang lebih panjang, `x * 10` berputar balik modulo 2^{64} tanpa peringatan apa pun, "bilangan-bilangan" di antrean tidak lagi tersusun dari 0 dan 1, dan program akhirnya mencetak suatu nilai yang telah berputar dan kebetulan habis dibagi n. Lebih buruk lagi, antreannya berlipat dua pada setiap digit, sehingga bahkan tanpa luapan, jawaban 36 digit akan membutuhkan 2^{36} entri antrean. Inilah program yang salah pada contoh kita, berdampingan dengan yang benar:

```
5
1
2
7
12
9999
```

```
1
10
1001
11100
13506811700559478884
```

```
1
10
1001
11100
11111111111111111111111111111111111111
```

Empat jawaban pertama sama; untuk $n = 9999$ program yang salah mencetak bilangan 20 digit yang penuh digit lain. **Kode 14.6** adalah program yang benar, mengikuti Subbab 14.5.2 baris demi baris, dan **Kode 14.7** adalah kembarannya dalam Python.

Kode 14.6 `onezero.cpp`: BFS atas sisa bagi modulo n dengan tautan induk, lalu digit-digitnya dibaca kembali dari sisa 0.

```cpp
// ONEZERO - Kelipatan n terkecil yang ditulis dengan digit 0 dan 1:
// BFS atas sisa bagi modulo n, tak pernah atas bilangan raksasanya.
#include <algorithm>
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

string smallestZeroOne(int n) {
    vector<int> parent(n, -2);  // -2: sisa belum dicapai, -1: awal
    vector<char> digit(n);
    vector<int> queue;
    queue.reserve(n);
    int start = 1 % n;
    parent[start] = -1;
    digit[start] = '1';
    queue.push_back(start);
    for (size_t head = 0; head < queue.size() && parent[0] == -2; head++) {
        int r = queue[head];
        for (int d = 0; d <= 1; d++) {  // coba tambah 0 sebelum 1: kecil dulu
            int nr = (r * 10 + d) % n;
            if (parent[nr] != -2) continue;
            parent[nr] = r;
            digit[nr] = (char)('0' + d);
            queue.push_back(nr);
        }
    }
    string s;
    for (int r = 0; r != -1; r = parent[r]) s += digit[r];  // telusuri ke awal
    reverse(s.begin(), s.end());
    return s;
}

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

Kode 14.7 `onezero.py`: penelusuran yang sama dalam Python, dengan *list* dan indeks kepala sebagai antrean.

```python
# ONEZERO - Kelipatan n terkecil yang ditulis dengan digit 0 dan 1:
# BFS atas sisa bagi modulo n, tak pernah atas bilangan raksasanya.
import sys


def smallest_zero_one(n):
    parent = [-2] * n  # -2: sisa belum dicapai, -1: awal
    digit = [""] * n
    start = 1 % n
    parent[start], digit[start] = -1, "1"
    queue = [start]
    head = 0
    while head < len(queue) and parent[0] == -2:
        r = queue[head]
        head += 1
        for d in (0, 1):  # coba tambah 0 sebelum 1: bilangan kecil dulu
            nr = (r * 10 + d) % n
            if parent[nr] == -2:
                parent[nr], digit[nr] = r, str(d)
                queue.append(nr)
    s, r = [], 0
    while r != -1:  # telusuri balik dari sisa 0 ke awal
        s.append(digit[r])
        r = parent[r]
    return "".join(reversed(s))


def main():
    data = sys.stdin.read().split()
    print("\n".join(smallest_zero_one(int(x)) for x in data[1:1 + int(data[0])]))


main()
```

14.5.4 Langkah 6 dan 7: pemeriksa alih-alih brute force

Untuk n kecil ada *brute force*: hitunglah $k = 1,2,3,\ldots$ dan bacalah digit biner k sebagai bilangan desimal, yang mendaftar semua bilangan $0/1$ dalam urutan menaik; kelipatan n yang pertama adalah jawabannya. Ia sepakat dengan kedua program pada 200 uji acak dengan $n \leq 300$. Tetapi masukan yang menarik adalah yang besar, tempat *brute force* harus menghitung sampai 2^{36}. Untuk itu kita menulis **pemeriksa** (*checker*): program yang tidak menghitung jawabannya, melainkan memverifikasi jawaban yang diberikan. `check_onezero.py` memeriksa, untuk setiap n dari 1 hingga 20.000, bahwa jawaban yang dicetak hanya berisi 0 dan 1, diawali 1, habis dibagi n (bilangan bulat Python tidak berbatas ukuran), dan bahwa tidak ada bilangan $0/1$ dengan digit lebih sedikit yang habis dibagi n, yang dibuktikannya dengan melacak himpunan sisa yang dapat dicapai dengan setiap banyaknya digit. Pemeriksaan terakhir itu membuktikan minimalitas panjangnya; argumen urutan BFS di Subbab 14.5.2 menangani sisanya. Pada semua 20.000 nilai:

```
checked 20000 values of n, 0 wrong
checked 20000 values of n, 0 wrong
n = 792: wrong answer 420545668853801304
n = 999: wrong answer 4976921130795323847
n = 1188: wrong answer 8877380742563353020
checked 20000 values of n, 162 wrong
```

Baris pertama memeriksa program Python, baris kedua program C++, dan empat baris terakhir program salah yang menggoda itu, yang memberi jawaban keliru untuk 162 dari 20.000 nilai n: tepat 162 nilai yang jawabannya lebih dari 20 digit, panjang ketika `x * 10` pertama kali keluar dari jangkauan bilangan bulat tak bertanda 64 bit. Memeriksa semua 20.000 nilai memakan

sekitar setengah menit dan mencakup setiap masukan yang mungkin diberikan juri; untuk soal dengan ruang masukan kecil, menghabiskannya adalah uji terkuat yang ada.

Waktu pada masukan maksimum berisi 1.000 kueri (baris terakhir *stress test*) adalah potongan informasi yang terakhir:

```
onezero    max input    5.3 KB    C++  0.125 s    Python  2.477 s
```

C++ seketika; Python membutuhkan sekitar 2,5 detik untuk 1.000 kueri berukuran terburuk di mesin kami, karena setiap kueri menjalankan BFS atas hingga 20.000 sisa di dalam interpreter. Bila batas juri ketat, prakomputasi membantu: banyak kueri berulang, dan `dict` berisi jawaban yang sudah dihitung membuat setiap kueri berulang gratis.

14.6 Ketika Vonisnya Bukan Accepted

Bahkan dengan alur kerja itu, sebagian pengiriman gagal. Vonis adalah informasi, dan **Tabel 14.2** adalah peta dari setiap vonis ke penyebabnya yang paling mungkin dan hal berikutnya yang harus dilakukan, menghimpun nasihat bab-bab sebelumnya di satu tempat.

Tabel 14.2 Arti yang biasa dari setiap vonis, apa yang dilakukan berikutnya, dan tempat buku ini membahasnya.

Vonis	Penyebab paling mungkin	Tindakan berikutnya	Bab
Wrong Answer	Salah membaca format; luapan; lupa mereset antarkasus; kasus tepi yang tak ditangani ($n = 1$, kosong, nilai sama); pembulatan keliru	Baca ulang pernyataan soal; daftar kasus tepi; *stress test* dengan beberapa kasus uji per berkas	2, 7, 8
Time Limit Exceeded	Algoritma terlalu lambat untuk maksimum sesungguhnya; I/O lambat; langkah kuadratik tersembunyi	Hitung ulang anggaran; ukur waktu masukan maksimum secara lokal; I/O cepat	3, 4
Runtime Error	Indeks di luar batas; rekursi terlalu dalam; pembagian dengan nol; membaca melewati akhir masukan	*Sanitizer*; *stack* eksplisit; cocokkan ukuran larik dengan batasan	5, 7
Memory Limit Exceeded	Matriks ketetanggaan untuk n besar; `long long` padahal `int` cukup; menyimpan apa yang bisa dialirkan	Hitung ulang memori dalam byte; alirkan masukannya	6
Compilation Error	Ekstensi kompiler; *header* yang tak ada di kompiler juri; bahasa yang salah dipilih	Kompilasi secara lokal dengan *flag* juri	5

Satu aturan layak dinyatakan tersendiri: jangan pernah mengirim ulang program yang sama sambil berharap vonis yang berbeda, dan jangan pernah mengubah dua hal sekaligus. Ubah satu hal, uji ulang secara lokal, lalu kirim lagi.

14.7 Langkah Berikutnya

14.7.1 Juri untuk latihan sehari-hari

SPOJ tetap merupakan arsip soal klasik yang baik, dan semua isi buku ini berlaku langsung di juri-juri lain. Masing-masing memiliki wataknya sendiri, dirangkum dalam **Tabel 14.3**.

- **Codeforces** [7] menyelenggarakan kontes singkat berperingkat beberapa kali sebulan, dibagi menurut tingkat (Div. 4 untuk pemula hingga Div. 1 untuk yang terkuat), dengan *rating* yang mengikuti kemajuan Anda, arsip soal raksasa yang diberi label topik dan tingkat kesulitan, serta pembahasan (*editorial*) setelah setiap ronde. Inilah juri tersibuk di dunia dan langkah alami berikutnya setelah buku ini.

- **AtCoder** [8] menyelenggarakan AtCoder Beginner Contest (ABC) hampir setiap akhir pekan: 100 menit, biasanya tujuh soal A sampai G, berperingkat untuk peserta di bawah 2000 (misalnya ABC 473 pada 29 Agustus 2026 [9]). Soal-soalnya bersih, teruji dengan cermat, dan cocok untuk latihan yang ajek. Pada 2026 AtCoder menata ulang Regular Contest-nya menjadi tiga tingkat, ARC++ (berperingkat 1600–2999), ARC (1200–2799), dan ARC– (800–2399), mulai ARC 215 [10].
- **CSES Problem Set** [11] adalah kumpulan tetap, dikelompokkan menurut topik dari soal pengantar hingga algoritma graf dan *string* tingkat lanjut, yang mencapai 400 soal pada Mei 2025 [12]. Ia berpasangan secara alami dengan panduan Laaksonen [1], karena keduanya berasal dari Antti Laaksonen, dan merupakan cara yang sangat baik untuk memastikan tidak ada teknik baku yang terlewat.
- **TLX** [13], platform Tim Olimpiade Komputer Indonesia (TOKI), memiliki kursus gratis berbahasa Indonesia, mulai dari "Pemrograman Kompetitif Dasar" ke atas, bersama arsip soal olimpiade Indonesia. Bagi pembaca yang bersiap menuju OSN, inilah persinggahan pertama.

Tabel 14.3 Tempat berlatih setelah SPOJ, dan keunggulan masing-masing juri.

Juri	Paling cocok untuk	Bahasa soal	Kontes
SPOJ	soal klasik, arsip panjang	Inggris	sesekali
Codeforces	ronde berperingkat, arsip raksasa berlabel	Inggris, Rusia	beberapa kali sebulan
AtCoder	kontes pemula yang bersih (ABC)	Jepang, Inggris	mingguan
CSES	silabus lengkap teknik baku	Inggris dan lainnya	tidak ada (kumpulan tetap)
TLX (TOKI)	kursus berbahasa Indonesia, persiapan OSN	Indonesia, Inggris	pembinaan nasional

14.7.2 Kontes pada 2026

Bagi siswa sekolah menengah, tangganya adalah olimpiade. **OSN** (Olimpiade Sains Nasional) bidang informatika diselenggarakan Pusat Prestasi Nasional (Puspresnas), dan babak nasionalnya pada 2026 berlangsung di Malang pada 14–20 September [14]. Peserta OSN terbaik mengikuti pelatihan nasional TOKI, tempat tim Indonesia untuk **International Olympiad in Informatics** dipilih. IOI 2026, yang ke-38, diselenggarakan di Tashkent, Uzbekistan, pada 9–16 Agustus 2026 [15].

Bagi mahasiswa, tangganya adalah **ICPC**. Di Indonesia musimnya dimulai dengan **Indonesia National Contest (INC)**, yang diselenggarakan BINUS University sebagai babak penyisihan daring bagi tim-tim Indonesia: pada 2026 INC berlangsung hari Minggu, 25 Oktober, pukul 10.00–15.00 WIB, dengan sepuluh hingga tiga belas soal dalam lima jam, dalam C, C++, Java, atau Python [16]. Tim INC terbaik melaju ke **ICPC Asia Jakarta Regional Contest** yang diselenggarakan luring dan juga terbuka bagi tim dari negara lain [17], dan para juara regional di seluruh Asia melanjutkan ke kejuaraan Asia Pasifik dan **ICPC World Finals**. World Finals ke-50 akan diselenggarakan di Dubai, Uni Emirat Arab, pada 15–20 November 2026 [18], [19].

Format kontes ICPC sangat cocok dengan alur kerja bab ini: tiga mahasiswa berbagi satu komputer, sehingga langkah 1 sampai 4 dikerjakan di kertas selagi seorang rekan mengetik; satu pengiriman yang salah berharga dua puluh menit penalti, sehingga langkah 7 membayar dirinya sendiri; dan tim yang sudah melatih rutinitas bersama unggul jauh atas tiga individu kuat yang belum melakukannya.

14.7.3 Buku untuk selalu di meja

Tiga buku memperluas buku ini ke arah yang berbeda. *Guide to Competitive Programming* karya Laaksonen [1], kini dalam edisi ketiga (2024), ringkas dan membahas teknik-teknik baku beserta pembuktiannya, dan merupakan pendamping CSES Problem Set. *Competitive Programming 4* karya Halim, Halim, dan Effendy [2] bersifat ensiklopedis, disusun menurut jenis soal, dan penuh rujukan ke soal-soal juri. *Introduction to Algorithms* karya Cormen dkk. [4] adalah rujukan untuk pembuktian dan analisis setiap kali sebuah teknik membutuhkan pemahaman yang lebih dalam daripada yang diberikan buku kontes. Dalam bahasa Indonesia, buku dan kursus *Pemrograman Kompetitif Dasar* dari TOKI [13] adalah titik awal yang wajar.

14.7.4 Rencana dua belas minggu

Kemampuan pemrograman kompetitif tumbuh lebih karena latihan yang teratur daripada sesi yang panjang. **Tabel 14.4** adalah rencana untuk dua belas minggu setelah buku ini, sekitar delapan jam per minggu. Setiap blok memiliki satu topik, satu sumber soal, dan satu kebiasaan yang dilatih dengan sengaja.

Tabel 14.4 Rencana latihan dua belas minggu setelah buku ini, dengan satu kebiasaan yang dilatih dengan sengaja di setiap blok.

Minggu	Topik	Di mana	Kebiasaan yang dilatih
1–2	Implementasi, pengurutan, simulasi	AtCoder ABC A–C, CSES pengantar	Langkah 1: nyatakan ulang sebelum mengode
3–4	*Binary search*, dua penunjuk, *greedy*	CSES *sorting and searching*, Codeforces Div. 3/4	Langkah 2: anggaran dari batasan
5–6	Pemrograman dinamis	CSES *dynamic programming*, AtCoder Educational DP	Langkah 3: tulis rekurensnya lebih dulu
7–8	Graf	CSES *graph algorithms*, soal SPOJ Bab 12	Langkah 4: *brute force* untuk setiap soal
9–10	Teori bilangan, *string*, geometri	CSES *mathematics* dan *string algorithms*, soal SPOJ Bab 13	Langkah 7: *stress test* sebelum mengirim
11–12	Kontes campuran di bawah tekanan waktu	Kontes virtual di Codeforces dan AtCoder, set soal INC terdahulu	Seluruh alur kerja di bawah jam

Setelah setiap kontes, lakukan *upsolve*: selesaikan soal-soal yang tidak berhasil Anda selesaikan selama kontes, dengan pembahasan bila perlu. Sebagian besar pembelajaran terjadi di situ. Simpanlah catatan, di kertas atau berkas teks, dengan satu baris per kesalahan yang Anda buat dan cara Anda menemukannya; setelah beberapa bulan catatan itu menjadi versi pribadi Anda dari subbab “Kesalahan yang Sering Terjadi” dalam buku ini.

14.8 Menggunakannya Secara Nyata

Alur kerja ini bukan hanya untuk juri. Setiap kali sebuah program harus *benar*, langkah-langkah yang sama berlaku. Program yang menghitung jadwal antar-jemput *laundry*, jadwal pelajaran sekolah, atau pesanan pupuk kebun dapat diuji persis seperti PPATH: tulis versi lambat yang jelas benar untuk kasus-kasus mungil, bangkitkan kasus acak, lalu bandingkan. Modul yang ruang masukannya kecil, seperti 20.000 nilai n pada ONEZERO, dapat diperiksa secara menyeluruh sebelum dipercaya. Dan ketika sebuah sistem nyata berulah hanya pada sebagian masukan, teknik penyusutan di Subbab 14.4.3 mengubah log yang panjang menjadi reproduksi dua baris yang dapat di-*debug* siapa saja.

Bagi tim yang bersiap menuju INC atau regional Jakarta, hal paling berguna yang dapat dibangun dari buku ini adalah **buku catatan tim** (*team notebook*): beberapa halaman berisi versi teruji milik Anda sendiri dari templat-templat yang paling sering dibutuhkan, seperti *union-find* dan Dijkstra dari Bab 12, pemangkatan modular, Miller–Rabin, perkalian silang, dan fungsi prefiks dari Bab 13, serta skrip *stress test* seperti yang ada di folder `tests` kode buku ini. Periksalah aturan kontes tentang bahan cetak yang diizinkan, ketiklah sendiri setiap templat, dan uji sekali lagi dengan *stress test* sebelum ia masuk ke buku catatan.

14.9 Kesalahan yang Sering Terjadi, dan Cara Menghindarinya

Mengode sebelum memahami. Melewatkan langkah 1 adalah jalan pintas yang paling mahal. Bila Anda tidak dapat menyatakan ulang tugasnya dalam dua kalimat, Anda belum siap mengetik.

Memercayai contoh. Contoh dipilih untuk menjelaskan format, bukan untuk menguji program Anda; contoh ABCPATH yang hanya satu grid menyembunyikan *bug* reset sepenuhnya. Tulis uji Anda sendiri, dan bangkitkan uji acak.

Brute force yang berbagi kode dengan solusi cepat. Bila keduanya memakai fungsi pembantu yang sama, *bug* pada fungsi itu membuat keduanya sepakat pada jawaban yang salah. Tulis *brute force* dengan cara yang paling berbeda dan paling jelas yang Anda bisa.

Stress test dengan satu kasus uji per berkas saja. Keadaan yang bocor dari satu kasus uji ke kasus berikutnya tidak terlihat bila setiap berkas hanya berisi satu kasus. Bangkitkan beberapa kasus per berkas.

Mengabaikan ukuran jawaban. Masukan ONEZERO kecil dan jawabannya raksasa. Perkirakan ukuran keluaran pada langkah 2 secermat ukuran masukan.

Mengubah beberapa hal setelah vonis gagal. Ubah satu hal, uji, dan baru kirim lagi; bila tidak, Anda tidak belajar apa pun dari vonis berikutnya.

Tidak melakukan upsolve. Satu kontes tempat Anda menyelesaikan dua soal lalu membaca pembahasan sisanya mengajarkan lebih banyak daripada tiga kontes tempat Anda hanya menyelesaikan dua soal.

14.10 Ringkasan Bab

- Rutinitas delapan langkah, **Gambar 14.1**, mengubah teknik-teknik buku ini menjadi vonis *Accepted*: baca, anggarkan, ide, *brute force*, solusi cepat, uji sendiri, *stress test*, kirim.
- **PPATH** adalah BFS pada graf implisit berisi bilangan prima empat digit dengan saringan untuk keprimaan; *brute force*-nya adalah relaksasi Bellman–Ford yang berdiri sendiri.
- **ABCPATH** adalah penelusuran bermemo, sebuah DP atas huruf; *bug* yang ditanam (memo yang direset sekali saja) lolos dari contoh, dan *stress test* dengan beberapa grid per berkas menangkapnya lalu menyusutkannya menjadi masukan dua baris.
- **ONEZERO** adalah BFS atas sisa bagi, Persamaan (14.3); BFS menggoda atas bilangan itu sendiri meluap, dan pemeriksa menyeluruh atas semua 20.000 nilai n membuktikan program yang benar itu benar dan yang salah itu salah.
- Setiap vonis selain *Accepted* menunjuk ke sekumpulan kecil penyebab, **Tabel 14.2**.
- Langkah berikutnya: Codeforces, AtCoder, CSES, dan TLX untuk berlatih; OSN dan IOI bagi siswa sekolah; INC, regional ICPC Asia Jakarta, dan ICPC World Finals (Dubai, November 2026) bagi

mahasiswa; Laaksonen, Halim, dan Cormen untuk dibaca; dan rencana dua belas minggu, **Tabel 14.4**, dengan kebiasaan-kebiasaan yang disengaja.

14.11 Soal Latihan

1. Untuk PPATH, jelaskan mengapa gelung digit untuk tempat ribuan harus dimulai dari 1, dan berikan masukan yang membuat program yang memulainya dari 0 mencetak jawaban yang salah.
2. Dengan memakai tetangga 1009 yang didaftar di Subbab 14.3.3, jelaskan mengapa jawaban untuk 1009 → 1019 adalah 1 dan untuk 1009 → 1033 adalah 2.
3. Tulislah *brute force* untuk ONEZERO yang dijelaskan di Subbab 14.5.4 dalam lima baris Python, lalu pakailah untuk mencari jawaban bagi $n = 3$ dan $n = 6$.
4. Untuk ONEZERO dengan $n = 3$, gambarlah BFS atas sisa bagi seperti pada **Gambar 14.2** lalu bacalah jawabannya.
5. Buktikan bahwa untuk setiap n jawaban ONEZERO memiliki paling banyak n digit.
6. Pada ABCPATH, apa yang salah bila rekursinya lupa memeriksa batas untuk tetangga di luar grid? Mengapa program itu mungkin tetap lolos banyak uji?
7. Berikan grid berukuran 1×3 yang membuat program yang hanya menelusuri dari `A` *pertama* yang ditemukannya, alih-alih dari setiap `A`, mencetak jawaban yang salah.
8. Solusi seorang teman mendapat *Time Limit Exceeded*; secara lokal ia berjalan 0,3 detik pada contohnya. Daftarlah tiga hal yang perlu diperiksa, secara berurutan.
9. Rencanakan dua minggu pertama Anda dari **Tabel 14.4**: pilih tiga soal konkret dari AtCoder atau CSES untuk setiap minggu, dan untuk masing-masing tuliskan langkah alur kerja mana yang menurut Anda paling sulit.
10. Tulislah pemeriksa (bukan *brute force*) untuk PPATH yang memverifikasi jawaban k yang diklaim untuk sepasang bilangan prima, dengan lintasannya juga diberikan. Apa saja yang harus diperiksanya?

14.12 Penyelesaian Soal Latihan

1. Pernyataan soal hanya mengizinkan bilangan prima empat digit, dan bilangan yang digit pertamanya 0 bukanlah bilangan empat digit. Program yang gelungnya dimulai dari 0 untuk tempat ribuan juga menerima 0013 = 13, 0017 = 17, dan setiap bilangan prima lain di bawah 1.000 sebagai simpul, karena saringan menyatakannya prima. Lalu muncullah jalan pintas. Untuk pasangan 1013 → 2017 jawaban yang benar adalah 4, misalnya 1013 → 9013 → 9011 → 2011 → 2017, tetapi program yang keliru menemukan 1013 → 0013 → 0017 → 2017 dan mencetak 3. *Stress test* dengan pasangan acak menemukan pasangan seperti itu dengan cepat, karena *brute force* di Subbab 14.3.4 hanya merelaksasi bilangan prima empat digit.

2. 1019 muncul dalam daftar tetangga prima 1009, sehingga dicapai dengan satu perubahan. 1033 berbeda dari 1009 pada dua digit (puluhan dan satuan), sehingga membutuhkan paling sedikit dua perubahan; lintasan 1009 → 1039 → 1033 mengubah digit puluhan lalu digit satuan, dan 1039 maupun 1033 adalah prima, sehingga jawabannya 2.

3.

```
def brute(n):
    k = 1
    while int(bin(k)[2:]) % n:
        k += 1
    return int(bin(k)[2:])
```

$k = 1,2,3, \ldots$ memberi 1,10,11,100,101,110,111, ... Untuk $n = 3$ kelipatan pertamanya 111; untuk $n = 6$ kelipatan pertamanya 1110.

4. Mulai di sisa $1 \bmod 3 = 1$. Tambah 0: $10 \bmod 3 = 1$, sudah terlihat. Tambah 1: $11 \bmod 3 = 2$, baru. Dari 2: tambah 0 memberi $20 \bmod 3 = 2$, sudah terlihat; tambah 1 memberi $21 \bmod 3 = 0$: selesai. Lintasannya awal 1, tambah 1, tambah 1: jawabannya 111.

5. Tinjau bilangan $R_j = 11 \ldots 1$ (j angka satu) untuk $j = 1, \ldots, n + 1$. Ada $n + 1$ bilangan seperti itu dan hanya n kemungkinan sisa modulo n, sehingga dua di antaranya, misalnya R_i dan R_j dengan $i < j$, bersisa sama. Maka $R_j - R_i = 11 \ldots 100 \ldots 0$ ($j - i$ angka satu diikuti i angka nol) habis dibagi n dan ditulis dengan 0 dan 1. Ia memiliki $j \leq n + 1$ digit. Mempertajam hitungannya: bila suatu R_j dengan $j \leq n$ sendiri habis dibagi n, selesai dengan paling banyak n digit; bila tidak, n bilangan $R_1, \ldots, R_n$ memiliki sisa di antara $n - 1$ nilai tak nol, sehingga dua di antaranya sama dengan $j \leq n$, dan selisihnya memiliki $j \leq n$ digit. Jawaban terkecil tidak lebih panjang daripada ini.

6. Tanpa pemeriksaan batas, program membaca `grid[-1][...]` atau `grid[r][w]`, memori di luar baris yang diisinya. Dengan larik global berukuran 52×52, sel di luar jangkauan sering berisi byte nol atau huruf sisa grid sebelumnya yang lebih besar, sehingga program mungkin mencetak jawaban benar hampir sepanjang waktu, dan itulah tepatnya yang membuat *bug* ini berbahaya. AddressSanitizer melaporkan pembacaan sebelum baris pertama, yang berada di luar larik, tetapi tidak melaporkan pembacaan sel di luar lebar atau tinggi grid saat ini yang masih di dalam larik 52×52; hanya pemeriksaan batas yang cermat, atau *stress test* dengan grid berbagai ukuran berturut-turut, yang menyingkapnya.

7. Pakailah `AAB`: `A` pertama di kolom 0 tidak bertetangga dengan `B` di kolom 2, sehingga penelusuran dari `A` pertama saja memberi 1, sedangkan `A` di kolom 1 memberi 2. Jawaban yang benar adalah 2.

8. Pertama, hitung ulang anggaran kompleksitas dengan nilai terbesar yang diizinkan pernyataan soal, termasuk banyaknya kasus uji, dan ukur waktu program secara lokal pada masukan maksimum yang dibangkitkan; contoh bukanlah maksimum. Kedua, periksa kecepatan masukan dan keluaran (Bab 4): `cin` yang tidak dilepas sinkronisasinya, `endl` di dalam gelung, atau `input()` Python untuk sejuta baris. Ketiga, cari kerja kuadratik yang tersembunyi: `erase` dari depan *vector*, penyambungan *string* di dalam gelung, pencarian linear di dalam gelung, atau struktur data yang direset dengan `memset` penuh untuk setiap dari banyak kasus uji kecil.

9. Contoh rencana: minggu 1, soal A, B, dan C AtCoder ABC dari satu kontes terbaru, serta CSES “Weird Algorithm”, “Missing Number”, dan “Repetitions”; langkah tersulit biasanya langkah 1 untuk soal C, yang pernyataan soalnya paling panjang. Minggu 2, CSES “Increasing Array”, “Permutations”, dan “Number Spiral”, serta satu ronde Codeforces Div. 4; untuk “Number Spiral” langkah tersulit adalah langkah 2, karena koordinatnya hingga 10^9 dan simulasi mustahil. Inti latihan ini adalah kolom kedua: menyebut lebih dulu di mana Anda memperkirakan kesulitan membuat Anda melambat di tempat yang tepat.

10. Diberikan dua bilangan prima a, b dan lintasan yang diklaim $a = x_0, x_1, \ldots, x_k = b$, pemeriksa harus memverifikasi bahwa setiap x_i adalah bilangan empat digit dengan digit pertama tak nol dan prima, bahwa x_i yang berurutan berbeda tepat pada satu digit, dan bahwa x_0 dan x_k adalah

bilangan yang diberikan. Itu membuktikan ada lintasan sepanjang k. Untuk membuktikan tidak ada lintasan yang lebih pendek, pemeriksa tetap membutuhkan penelusuran (atau tabel jarak yang tepercaya); inilah beda antara memeriksa kelayakan, yang mudah, dan memeriksa optimalitas, yang biasanya sesulit memecahkan soalnya. Pemeriksa ONEZERO dapat membuktikan optimalitas dengan murah hanya karena himpunan sisa yang dapat dicapai per banyaknya digit itu kecil.

Daftar Pustaka

[1] A. Laaksonen, *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*, 3rd ed. Cham, Switzerland: Springer, 2024.

[2] S. Halim, F. Halim, and S. Effendy, *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*. Singapore: Lulu, 2020.

[3] SPOJ, "PPATH — Prime Path," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/PPATH/

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[5] SPOJ, "ABCPATH — ABC Path," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/ABCPATH/

[6] SPOJ, "ONEZERO — Ones and zeros," accessed Sep. 2026. [Online]. Available: https://www.spoj.com/problems/ONEZERO/

[7] Codeforces, accessed Sep. 2026. [Online]. Available: https://codeforces.com/

[8] AtCoder, "Contests," accessed Sep. 2026. [Online]. Available: https://atcoder.jp/contests/

[9] AtCoder, "AtCoder Beginner Contest 473," Aug. 29, 2026. [Online]. Available: https://atcoder.jp/contests/abc473

[10] AtCoder, "Changes to ARC in 2026," 2026. [Online]. Available: https://atcoder.jp/posts/2026ARC_en

[11] CSES, "CSES Problem Set," accessed Sep. 2026. [Online]. Available: https://cses.fi/problemset/

[12] CSES, "CSES Problem Set May 2025 Update," May 2025. [Online]. Available: https://cses.fi/blog/text/3613

[13] Tim Olimpiade Komputer Indonesia, "TLX — Courses: Pemrograman Kompetitif Dasar," accessed Sep. 2026. [Online]. Available: https://tlx.toki.id/courses/competitive-1

[14] Tim Olimpiade Komputer Indonesia, "OSN 2026 Bidang Informatika," Sep. 2026. [Online]. Available: https://peringkat.toki.id/osn-2026/

[15] IOI 2026 Organizing Committee, "International Olympiad in Informatics 2026, Tashkent," accessed Sep. 2026. [Online]. Available: https://ioi2026.uz/

[16] BINUS University, "ICPC 2026: Indonesia National Contest (INC)," accessed Sep. 2026. [Online]. Available: https://competition.binus.ac.id/icpc2026/index/aboutinc

[17] ICPC Foundation, "The 2026 ICPC Asia Jakarta Onsite Regional Contest," accessed Sep. 2026. [Online]. Available: https://icpc.global/regionals/finder/Jakarta

[18] ICPC Foundation, "The 2026 ICPC World Finals Dubai hosted by Dubai Chamber of Digital Economy," accessed Sep. 2026. [Online]. Available: https://worldfinals.icpc.global/2026/about/

[19] Codeforces, "The 2026 ICPC World Finals: Dubai, U.A.E. November 15–20, 2026," accessed Sep. 2026. [Online]. Available: https://codeforces.com/blog/entry/149884