MENAKLUKKAN SPOJ

EDISI PERTAMA

IRFAN SUBAKTI

Ubah pengetahuan algoritma menjadi submission yang cepat dan benar di bawah tekanan waktu.

Profitina

profitina.com

DAFTAR ISI

Kata Pengantar

Pemrograman kompetitif bukan mata kuliah tersendiri di sini. SPOJ adalah tempat materi Desain dan Analisis Algoritma (DAA) berhenti menjadi teori dan mulai diuji: algoritma pengurutan, pencarian, dynamic programming, dan graf yang sama, kini di bawah batas waktu seorang juri. Buku ini pendamping praktis kuliah itu -- 14 bab yang membawa Anda dari submission pertama hingga jadi insting.

Sebelum menyentuh satu algoritma pun, Anda perlu memahami persis apa yang diperiksa sebuah judge — dan mengapa "jalan di komputer saya" adalah kalimat paling berbahaya dalam competitive programming. Buku ini mengubah pengetahuan algoritma yang sudah dipelajari di tempat lain menjadi submission yang cepat dan benar di bawah tekanan waktu ketat.

Menaklukkan SPOJ dibuka dengan apa sesungguhnya SPOJ (Sphere Online Judge) itu dan persis apa yang diperiksa pipeline otomatisnya atas sebuah submission, lalu berlanjut ke membaca soal sebagaimana seorang judge membacanya, memilih algoritma yang tepat untuk batasan yang dinyatakan, I/O cepat dalam C/C++, pragma compiler dan risiko portabilitas, serta pilihan memori dan struktur data di bawah anggaran byte yang ketat. Serangkaian bab tentang jebakan implementasi klasik dan cara men-debug TLE dan WA tanpa judge menjelaskan alasannya, diikuti bab-bab pola algoritmik — dynamic programming, soal greedy dan exchange-argument, serta satu bab toolkit yang mencakup algoritma graf, teori bilangan, geometri, dan string. Buku ini ditutup dengan contoh lengkap dari awal sampai akhir menuju Accepted, satu bab cheat code dan jalan pintas yang dipakai secara bertanggung jawab, dan satu bab tentang ke mana melangkah selanjutnya — didukung tiga lampiran referensi: katalog seluruh soal dalam buku, lembar contekan kompleksitas, dan daftar istilah.

Setibanya Anda di halaman terakhir, Anda semestinya mampu:

- Membaca soal competitive programming sebagaimana data uji judge sesungguhnya akan mengujinya, bukan sekadar kesan pertama.
- Memilih algoritma yang tepat untuk batasan waktu dan memori yang dinyatakan, bukan sekadar algoritma apa pun yang menghasilkan jawaban benar.
- Menulis I/O C/C++ yang cepat dan portabel, serta menghindari jebakan flag compiler dan portabilitas yang diam-diam menggerus nilai.
- Mendiagnosis verdict Time Limit Exceeded dan Wrong Answer secara sistematis, tanpa judge memberi tahu apa yang salah.
- Menerapkan pola dynamic programming, greedy, graf, teori bilangan, geometri, dan algoritma string pada soal judge sesungguhnya.
- Memakai lampiran buku ini — katalog soal, lembar contekan kompleksitas, dan daftar istilah — sebagai referensi kerja saat berlatih.

Untuk siapa buku ini ditulis: mahasiswa dan pembelajar mandiri yang sudah menguasai algoritma dan struktur data inti dan ingin mengubah pengetahuan itu menjadi submission yang cepat dan benar di SPOJ dan online judge sejenis.

Sepatah kata tentang metode. Setiap contoh kode dalam buku ini dikompilasi dan dijalankan sebelum dituliskan, dan lampirannya memuat catatan verifikasi sebagai buktinya. Ketika sebuah angka muncul, angka itu berasal dari pengukuran, bukan dari ingatan. Bila Anda menemukan kekeliruan, atau bagian yang bisa lebih jelas, saya sungguh ingin mendengarnya — begitulah edisi berikutnya menjadi lebih baik.

Surabaya, September 2026

Irfan Subakti
yifana@gmail.com

Profitina
profitina.com

BAB 1

Selamat Datang di Competitive Programming dan SPOJ

Sebelum menyentuh satu algoritma pun, Anda perlu memahami persis apa yang diperiksa oleh judge — dan mengapa "berjalan di komputer saya" adalah kalimat paling berbahaya dalam competitive programming.

Tujuan Pembelajaran — setelah bab ini Anda mampu:

(1) Menjelaskan apa yang sebenarnya dilakukan online judge seperti SPOJ terhadap sebuah submission: compile, jalankan terhadap test tersembunyi, cek kebenaran/waktu/memori. (2) Menyebutkan enam verdict standar (AC/WA/TLE/MLE/RE/CE) dan apa arti masing-masing tentang bug Anda. (3) Menjelaskan mengapa lolos test case sampel milik sendiri membuktikan jauh lebih sedikit dari yang diasumsikan pemula. (4) Menjelaskan bagaimana buku ini memakai soal SPOJ nyata sebagai contoh sambil menjaga batas confidentiality yang ketat dan eksplisit. (5) Meng-compile dan menjalankan program C++ pertama dengan konvensi competitive programming (I/O cepat, sieve klasik) dan membaca transkrip verifikasinya.

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

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

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

1.1 Apa Itu Competitive Programming, dan Apa Itu SPOJ?

Competitive programming adalah olahraga menyelesaikan soal algoritmik yang terspesifikasi dengan jelas, dalam batas waktu ketat, dengan program yang kebenaran dan efisiensinya diperiksa otomatis oleh mesin bernama judge. Tidak ada nilai parsial untuk kode yang elegan, tidak ada dosen yang menjelaskan maksud Anda, dan tidak ada keringanan untuk jawaban yang "hampir benar." Entah program Anda menghasilkan output yang PERSIS sesuai harapan, dalam batas waktu dan memori yang diizinkan soal, pada SETIAP test case tersembunyi yang dilemparkan judge — atau tidak, dan Anda akan tahu persis di cara mana ia gagal.

SPOJ (Sphere Online Judge) adalah salah satu online judge tertua dan paling banyak dipakai di dunia, menampung ribuan soal yang mencakup hampir semua topik algoritmik klasik: dynamic programming, algoritma greedy, teori graf, teori bilangan, geometri komputasional, algoritma string, struktur data, dan lainnya. Ia menghargai persis keterampilan yang diajarkan lecture Perancangan dan Analisis Algoritma (DAA) mata kuliah ini — dan buku ini ada karena kedua hal itu, bila diajarkan bersama, saling memperkuat jauh lebih baik dibanding sendiri-sendiri. Kumpulan soal SPOJ mata kuliah ini sendiri mencakup 257 soal terselesaikan di hampir semua kategori tersebut; Lampiran 15.A mengkatalogkan semuanya berdasarkan nama, ID, dan topik sebagai referensi soal apa yang bisa dicoba berikutnya.

Buku ini BUKAN walkthrough dari 257 soal tersebut. Membaca solusi accepted milik orang lain mengajarkan Anda seperti apa jawaban satu soal spesifik itu; itu TIDAK mengajarkan Anda cara mengenali soal berikutnya yang belum pernah Anda lihat. Buku ini disusun dengan cara sebaliknya: berdasarkan keterampilan yang transferable — membaca batasan (constraints), memilih keluarga algoritma, menghindari jebakan implementasi klasik, men-debug submission yang ditolak — yang memungkinkan Anda menghadapi soal SPOJ asing dengan modal nyata untuk Accepted.

1.2 Apa yang Sebenarnya Dilakukan Judge Terhadap Submission Anda

Saat Anda mengirim source code ke SPOJ, tidak ada satu pun gaya penulisan, komentar, atau keeleganan kode Anda yang pernah diperiksa manusia. Sebaliknya, pipeline otomatis meng-compile submission Anda dengan compiler dan flag tertentu, lalu menjalankan binary hasilnya satu kali untuk setiap test case tersembunyi yang disiapkan pembuat soal — tiap eksekusi tunduk pada batas waktu wall-clock yang ketat dan batas memori. Judge membandingkan output program Anda dengan output yang diharapkan untuk test itu, byte demi byte (atau dengan checker khusus untuk soal yang menerima banyak jawaban valid). Test case PERTAMA yang gagal langsung menghentikan proses dan menghasilkan verdict; Anda tidak diberi tahu test berikutnya mana yang juga akan gagal, dan untuk kebanyakan soal SPOJ Anda juga tidak ditunjukkan test spesifik mana yang gagal — hanya verdict-nya saja (Gambar 1.1).

Gambar 1.1 — Pipeline judge: source code Anda di-compile sekali, lalu dijalankan terhadap test case tersembunyi satu per satu hingga kegagalan pertama menghasilkan verdict.

> **Judge sudah menentukan jawabannya sebelum Anda submit.**
>
> Output yang diharapkan untuk setiap test case sudah ditetapkan oleh pembuat soal jauh sebelum submission Anda ada. Men-debug submission yang ditolak bukanlah negosiasi dengan judge — itu adalah pencarian input spesifik yang ditangani salah oleh program yang sudah Anda yakini benar.

1.3 Tiga Hal yang Selalu Diperiksa Judge

Setiap verdict sebenarnya adalah jawaban atas tiga pertanyaan independen, diperiksa dalam urutan ini utk tiap test case tersembunyi: apakah output benar, apakah program selesai dalam batas waktu, dan apakah ia tetap dalam batas memori? Sebuah program bisa saja sempurna benar tapi tetap gagal jika terlalu lambat (Time Limit Exceeded) atau terlalu boros memori (Memory Limit Exceeded) utk batasan yang diberikan — ini adalah jebakan paling umum bagi programmer yang datang dari latar belakang tugas kuliah, di mana jawaban benar biasanya sudah cukup. Di SPOJ, benar-tapi-lambat dan benar-tapi-boros sama-sama kegagalan, dan Bab 3 didedikasikan sepenuhnya utk bernalar tentang kompleksitas algoritmik apa yang sebenarnya dituntut oleh batasan suatu soal (Tabel 1.1 dan Gambar 1.2).

1.3.1 Enam Verdict Standar

Tabel 1.1 memetakan tiap verdict ke arti umumnya dan di mana harus mulai mencari bug. Belajar membaca verdict dengan cara ini — sebagai petunjuk, bukan sekadar nilai — adalah salah satu kebiasaan paling berharga yang bisa diajarkan buku ini, karena ia mengubah "submission saya gagal" dari jalan buntu menjadi pencarian yang spesifik dan sempit.

Membaca Verdict Seperti Detektif

Tiap verdict menunjuk jenis bug yang berbeda — pakai itu utk mempersempit pencarian sebelum menyentuh satu baris kode pun.

Verdict	Biasanya berarti	Cek dulu di mana
WA (Wrong Answer)	Kesalahan logika, atau salah baca batasan/edge case	Baca ulang soal; uji n=0, n=1, duplikat, negatif
TLE (Time Limit)	Kompleksitas algoritma terlalu tinggi utk n yang diberikan	Turunkan ulang kompleksitas yang diharapkan dari batasan (Bab 3)
MLE (Memory Limit)	Struktur data berskala pada dimensi yang salah	Cek ukuran array thd batasan TERBESAR, bukan yang biasa
RE (Runtime Error)	Akses di luar batas, null pointer, stack overflow	Cek batas array & kedalaman rekursi pada input terbesar
CE (Compile Error)	Kesalahan sintaks, atau judge pakai compiler beda/lama	Cek versi compiler & varian bahasa yang dipakai judge

Gambar 1.2 — Membaca verdict seperti detektif: tiap dari enam verdict standar menunjuk kelas bug yang berbeda.

1.4 Mengapa "Berjalan di Komputer Saya" Tidak Cukup

Pemula secara wajar mengasumsikan bahwa jika program mereka menghasilkan jawaban benar utk satu atau dua input sampel yang tercetak di soal, berarti sudah selesai. Ini hampir tidak pernah benar di judge seperti SPOJ. Input sampel ada utk mengilustrasikan format input/output, bukan utk menguji setiap sudut batasan soal. Test suite tersembunyi di balik soal SPOJ yang dibuat dengan baik sengaja bersifat adversarial: mencakup input legal terkecil (sering n = 0 atau n = 1, yang mematahkan logika off-by-one), input legal terbesar (yang mematahkan apa pun dengan kompleksitas algoritmik salah atau tipe integer yang terlalu kecil utk menampung jawaban), input dengan nilai duplikat atau berulang, input pada kondisi batas persis yang disebut sekilas di soal, dan — utk banyak soal — test case yang sengaja dirancang utk mengalahkan pendekatan salah paling umum yang pernah dilihat pembuat soal.

> **“Lolos sampel” adalah awal dari testing, bukan akhirnya.**
>
> Sebelum submit, jalankan (secara mental atau nyata) program Anda terhadap input terkecil yang mungkin, input terbesar yang diizinkan batasan, dan input mana pun di mana dua nilai dalam soal bisa jadi sama. Bab 8 membangun insting ini menjadi proses debugging yang bisa diulang.

Inilah juga sebabnya competitive programmer terbiasa menulis stress test mereka sendiri: skrip kecil yang menghasilkan input acak atau adversarial, menjalankan baik solusi “brute force” yang lambat-tapi-jelas-benar maupun solusi cepat yang sedang diuji, dan melaporkan input pertama di mana keduanya berbeda hasil. Anda tidak butuh mekanisme ini utk setiap soal — tapi utk apa pun yang tidak sepele, ini menemukan bug jauh lebih cepat dibanding menatap kode, dan Bab 8 membahas cara membangunnya.

1.5 Bagaimana Buku Ini Memakai Soal SPOJ Nyata

Buku ini mengutip soal SPOJ nyata dan spesifik dengan nama dan ID di sepanjang isinya — soal seperti PRIME1 (Prime Generator), AGGRCOW (Aggressive Cows), atau LASTDIG (The Last Digit) adalah informasi publik: nama, teks soal, dan ide algoritmik umum yang dipakai solusi terkenal semuanya adalah hal yang bisa dicari dan didiskusikan bebas oleh competitive programmer mana pun, dan melakukan itu adalah praktik standar di setiap buku, blog, dan kursus competitive programming di dunia.

Yang TIDAK akan pernah dilakukan buku ini adalah mereproduksi solusi lengkap, accepted, dan berfungsi utk soal spesifik mana pun secara verbatim. Mata kuliah ini menyimpan arsip internal dan rahasia sendiri berisi solusi SPOJ lengkap yang dipakai utk penilaian dan pengajaran; source code sesungguhnya dari arsip itu tidak pernah disebarkan, baik seluruhnya maupun sebagian besar. Di mana buku ini menampilkan cuplikan kode di samping soal bernama, cuplikan itu adalah ilustrasi kecil dan mandiri dari satu teknik — tidak pernah lebih dari sebagian kecil apa yang dibutuhkan solusi accepted lengkap — atau ditulis segar, khusus utk buku ini, independen dari arsip solusi mana pun yang dipakai mahasiswa. Perbedaan yang penting sederhana: memahami *mengapa* suatu teknik bekerja, didemonstrasikan pada contoh kecil orisinal, ter-transfer ke soal yang belum pernah Anda lihat. Menyalin solusi accepted lengkap tidak ter-transfer ke apa pun kecuali soal itu saja, dan itu bukan cara buku ini mengajar.

> **Janji kepada pembaca**
>
> Setiap cuplikan kode dalam buku ini adalah (a) template orisinal dan generik yang mengilustrasikan satu ide secara terisolasi, ditandai dengan jelas sebagai demikian, atau (b) fragmen kecil — tidak pernah solusi lengkap — ditampilkan bersama soal bernama yang diketahui publik semata utk mendemonstrasikan bagaimana ide itu diterapkan. Jika Anda mencari solusi accepted siap-tempel utk soal SPOJ spesifik, ini bukan buku itu, secara sengaja — jalan pintas itu tidak akan mengajarkan apa pun yang bertahan melewati soal pertama yang belum pernah Anda lihat.

1.6 Contoh Kerja Pertama: I/O Cepat dan Sieve of Eratosthenes

Utk mengkonkretkan Bagian 1.2 hingga 1.5, perhatikan PRIME1 (Prime Generator di SPOJ) — soal terkenal dan terdokumentasi publik yang meminta semua bilangan prima dalam rentang tertentu, diulang di banyak query. Pendekatan standarnya adalah Sieve of Eratosthenes: tandai setiap kelipatan dari setiap prima yang ditemukan sejauh ini sebagai komposit, menyisakan hanya prima yang tidak tertandai. Listing di bawah adalah template orisinal dan generik yang mendemonstrasikan teknik sieve secara terisolasi pada rentang tetap yang kecil — bukan solusi accepted spesifik mana pun, dan jauh lebih kecil cakupannya dibanding apa yang dituntut batasan sesungguhnya PRIME1 (submission nyata butuh sieve tersegmentasi, karena rentang PRIME1 bisa jauh melampaui apa yang muat di array sederhana; penyempurnaan itu adalah topik Bab 6).

sieve_demo.cpp — Sieve of Eratosthenes pada rentang tetap kecil (template orisinal)

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    const int LIMIT = 100;
    vector<bool> isComposite(LIMIT + 1, false);
    for (int p = 2; (long long)p * p <= LIMIT; ++p) {
        if (!isComposite[p]) {
            for (int m = p * p; m <= LIMIT; m += p)
                isComposite[m] = true;
        }
    }
    // ...kumpulkan & cetak angka yang tak tertandai sbg prima
}
```

Dua kebiasaan sudah terlihat dalam fragmen belasan-baris ini yang akan dikembangkan jauh lebih lanjut oleh Bab 4 dan Bab 6: I/O cepat (pasangan sync_with_stdio/cin.tie, yang penting begitu volume output membesar) dan memulai loop penandaan tiap prima di p*p, bukan 2*p (apa pun yang lebih kecil sudah ditandai oleh prima yang lebih kecil, jadi memulai di sana adalah kebenaran gratis tanpa biaya). Lampiran 1.A menunjukkan program persis ini di-compile dan dijalankan, dengan output terverifikasi, menetapkan konvensi buku ini sendiri: setiap contoh kode nontrivial dalam buku ini di-compile dan dieksekusi sebelum dituliskan, tidak pernah sekadar “terlihat benar (Gambar 1.3 dan 1.4).”

1.8 Ringkasan Bab

- Judge seperti SPOJ memeriksa tiga hal utk tiap test case tersembunyi: kebenaran, waktu, dan memori — dan berhenti pada kegagalan pertama.
- Enam verdict standar (AC, WA, TLE, MLE, RE, CE) bukan sekadar nilai — masing-masing mempersempit jenis bug apa yang harus dicari duluan.
- Lolos input sampel di soal membuktikan hampir tidak ada apa-apa; test tersembunyi sengaja bersifat adversarial (edge case, input legal terbesar, nilai duplikat).

- Buku ini mengutip soal SPOJ nyata dengan nama dan teknik umum, tapi tidak pernah mereproduksi solusi accepted lengkap — cuplikan kode hanyalah template orisinal atau fragmen ilustratif kecil.
- Setiap contoh kode dalam buku ini, termasuk fragmen ilustratif kecil, di-compile dan dijalankan sebelum dituliskan.

Gambar 1.3 — Pengingat utk bab-bab berikutnya: membaca teknik saja hanya membawa Anda sejauh itu.

Gambar 1.4 — Peta jalan 14-bab buku ini. Kita di sini: Bab 1.

Soal Latihan (Tidak Dinilai)

1. Sebuah submission mendapat Wrong Answer (WA) pada test case tersembunyi ketiga judge, tapi menghasilkan jawaban benar pada kedua input sampel yang tercetak di soal. Jelaskan, dengan kata-kata Anda sendiri, mengapa ini mungkin dan sebutkan dua jenis input yang akan Anda coba lebih dulu saat men-debug.
2. Sebuah submission mendapat Time Limit Exceeded (TLE) pada soal yang batasannya mengizinkan n hingga 10^6. Apa yang diberitahukan verdict ini — dan TIDAK diberitahukan — tentang kebenaran logika program Anda?
3. Jelaskan, dalam dua atau tiga kalimat, mengapa buku ini mau menyebutkan soal SPOJ nyata spesifik (seperti PRIME1) tapi tidak mau mencetak solusi accepted lengkap utk soal itu.
4. Contoh sieve_demo.cpp di Bagian 1.6 memulai loop penandaan dalam tiap prima di p*p, bukan 2*p. Jelaskan mengapa ini benar — yaitu, mengapa tidak ada bilangan komposit yang pernah terlewat dengan melewati tanda antara 2*p dan p*p.
5. Sebutkan satu kebiasaan dari competitive programming (Bagian 1.7) yang menurut Anda akan membuat Anda menjadi software engineer yang lebih baik bahkan di luar konteks judge, dan jelaskan mengapa.

Lampiran 1.A — Log Verifikasi

Program sieve_demo.cpp dari Bagian 1.6 di-compile dengan g++ (dengan -O2 -Wall -pedantic) dan dieksekusi utk mengonfirmasi outputnya sebelum dimasukkan ke bab ini, sesuai konvensi compile-dan-jalankan-semuanya buku ini (Bagian 1.6).

Transkrip terminal

```
$ g++ -O2 -pedantic -Wall -o sieve_demo sieve_demo.cpp
$ ./sieve_demo
Primes up to 100 (25 found):
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

25 prima di bawah 100 cocok dengan hitungan referensi yang terkenal, mengonfirmasi kondisi batas sieve (p*p <= LIMIT) dan loop penandaan keduanya benar pada rentang kecil ini.

Referensi

[1] Sphere Online Judge (SPOJ). https://www.spoj.com — soal PRIME1 (Prime Generator), AGGRCOW (Aggressive Cows), LASTDIG (The Last Digit), dikutip di Bagian 1.5–1.6 hanya berdasarkan nama/ID publik.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — latar belakang Sieve of Eratosthenes dan analisis asimtotik.

[3] Halim, S., & Halim, F. (2013). Competitive Programming 3. Lulu Press — referensi yang banyak dipakai utk pendekatan keterampilan-transferable yang diikuti buku ini.

BAB 2

Membaca Soal Seperti Seorang Judge

Kebanyakan submission yang ditolak bukan kegagalan keterampilan algoritmik — itu kegagalan membaca. Bab ini mengajarkan cara yang disiplin dan bisa diulang utk membaca teks soal sebelum menulis satu baris kode pun.

Tujuan Pembelajaran — setelah bab ini Anda mampu:

(1) Mengidentifikasi lima zona standar teks soal judge dan membacanya dalam urutan yang paling menghemat waktu. (2) Menjelaskan mengapa batasan (constraints), bukan cerita, adalah spesifikasi sesungguhnya suatu soal. (3) Menyebutkan setidaknya lima jebakan teks soal umum yang membuat kode yang terlihat benar tetap gagal. (4) Menerapkan checklist membaca lima-pertanyaan sebelum menulis kode apa pun. (5) Mengimplementasikan dan memverifikasi loop input yang berakhir pada EOF, pola input paling umum di SPOJ.

2.1 Kebanyakan Bug Dimulai Sebelum Anda Menulis Kode

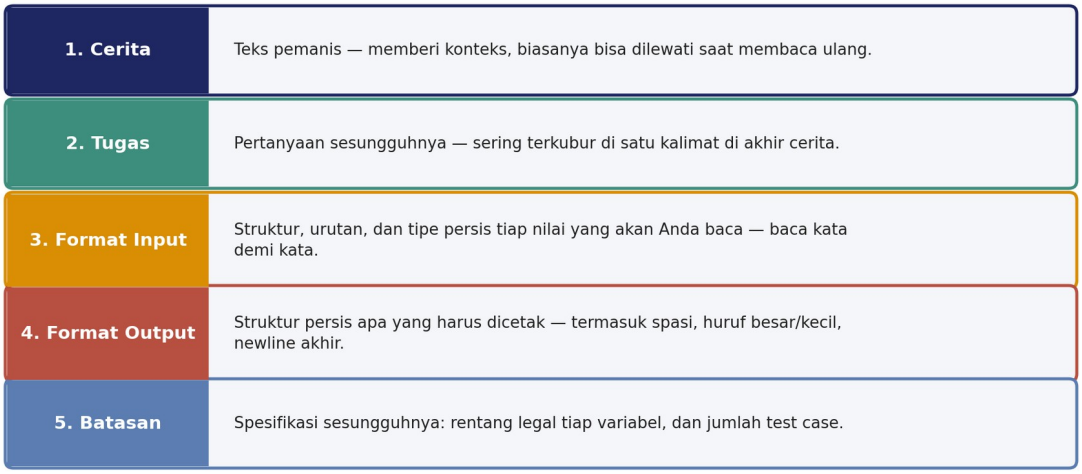
Menggoda utk memperlakukan pembacaan teks soal sebagai formalitas yang harus dilewati secepat mungkin menuju bagian yang menarik: merancang algoritma. Ini terbalik. Di judge seperti SPOJ, sebagian besar verdict Wrong Answer dan Runtime Error berasal bukan dari algoritma yang cacat, tapi dari detail dalam teks soal yang dilewatkan sekilas — asumsi format input yang ternyata salah, batasan yang sebenarnya tidak diperiksa, persyaratan format output yang terlihat tidak penting. Bab ini memperlakukan membaca sebagai keterampilan dengan tekniknya sendiri, layak dilatih secara sengaja, persis seperti algoritma.

2.2 Lima Zona Teks Soal

Hampir setiap teks soal competitive-programming yang terbentuk dengan baik — di SPOJ maupun tempat lain — dibangun dari lima zona yang sama, dan membacanya dalam urutan yang tepat itu penting. Cerita muncul pertama di halaman, tapi biasanya harus dibaca terakhir pada bacaan kedua, karena itu adalah zona yang paling kecil kemungkinannya mengandung informasi yang Anda butuhkan utk mendapat verdict Accepted (Gambar 2.1).

Anatomi Teks Soal — Lima Zona, Dibaca dalam Urutan Ini

Setiap soal judge yang baik punya lima zona ini. Langsung loncat ke cerita adalah yang paling menyita waktu.



Gambar 2.1 — Lima zona teks soal, dalam urutan yang paling penting utk mengekstrak spesifikasi yang benar.

2.2.1 Cerita dan Tugas

Cerita ada utk membuat soal mudah diingat dan kadang utk menyamarkan pola algoritmik yang sangat standar di balik skenario yang asing — kebiasaan yang layak diperhatikan, karena “ini sebenarnya keluarga soal X berkostum” itu sendiri adalah keterampilan pengenalan yang berguna (Bab 3 membangun ini secara langsung). Tugas — pertanyaan sesungguhnya — sering hanya satu kalimat, kadang kalimat terakhir dari paragraf cerita. Temukan kalimat itu dan nyatakan ulang dengan kata-kata Anda sendiri sebelum membaca lebih jauh; jika Anda tidak bisa menyatakan ulang dengan presisi, Anda belum siap merancang algoritma.

2.2.2 Format Input dan Format Output

Kedua zona ini adalah kontrak, bukan prosa — setiap kata membawa makna. "Baris pertama berisi satu integer T, jumlah test case" adalah struktur input yang sama sekali berbeda dari "input terdiri dari beberapa test case, diakhiri oleh end of file" (Bagian 2.6 membahas kedua pola dan cara mengimplementasikan masing-masing dengan benar). Format output layak mendapat perhatian kata-demi-kata yang sama: persyaratan mencetak "YES" tidak sama dengan "Yes" atau "yes" bagi judge yang membandingkan byte, dan persyaratan baris kosong antar test case mudah terlewat dan mudah salah secara diam-diam.

2.2.3 Batasan (Constraints)

Batasan adalah spesifikasi sesungguhnya dari soal — lebih dari cerita, dan bisa dibilang lebih dari tugas itu sendiri. Mereka memberi tahu Anda ukuran input persis yang harus ditangani algoritma Anda, yang menetapkan kelas kompleksitas yang boleh Anda pakai (topik Bab 3 secara penuh), apakah nilai bisa nol, negatif, atau sangat besar (yang menetapkan tipe variabel dan daftar edge case Anda), dan berapa banyak test case yang harus diproses satu kali jalan (yang memengaruhi apakah biaya setup per-test-case penting). Soal yang terlihat menakutkan dalam ceritanya sering kali sepenuhnya mekanis begitu Anda benar-benar memahami batasannya; soal yang terlihat sederhana dalam ceritanya bisa jadi jebakan begitu batasannya mengungkap n hingga 10^9 (Tabel 2.1 dan Gambar 2.2).

> **Baca batasan dua kali: sekali utk ukuran, sekali utk bentuk.**
>
> Bacaan pertama menjawab “seberapa besar n bisa jadi” (yang menentukan anggaran kompleksitas algoritma Anda). Bacaan kedua menjawab “seperti apa nilai itu sendiri bisa jadi” — nol, negatif, duplikat, sudah terurut, dijamin berbeda — yang menentukan daftar edge case Anda. Kedua bacaan itu penting; kebanyakan pemula hanya melakukan yang pertama.

2.3 Checklist Membaca Lima Pertanyaan

Tabel 2.1 menyaring Bagian 2.1 dan 2.2 menjadi lima pertanyaan konkret utk dijawab, secara tertulis jika membantu, sebelum mulai coding. Tiap pertanyaan dipasangkan dengan kelas bug spesifik yang muncul jika dilewatkan — framing gaya-detektif yang sama yang diperkenalkan buku ini di Bab 1 utk membaca verdict berlaku sama baiknya utk membaca teks soal bahkan sebelum Anda submit apa pun.

Checklist Membaca Lima Pertanyaan

Jawab kelima pertanyaan ini sebelum menulis satu baris kode — kebanyakan bug WA/RE berasal dari melewatkan salah satunya.

Pertanyaan	Mengapa penting	Apa yang rusak jika dilewatkan
Berapa n legal terbesar?	Menetapkan kelas kompleksitas yang dibutuhkan (Bab 3)	TLE dari algoritma yang terlalu lambat
Bisakah nilai nol atau negatif?	Mengubah edge case apa saja yang ada	WA pada n=0, atau input bernilai negatif
Berapa banyak test case per file?	Menentukan struktur loop I/O Anda	RE atau infinite loop dari pola baca yang salah
Apakah format output persis (spasi, huruf)?	Judge biasanya membandingkan byte demi byte	WA meski jawaban secara angka benar
Adakah nilai seri, dan bagaimana diputuskan?	Mengubah logika sort/pembandingan	WA hanya pada input dgn nilai duplikat

Gambar 2.2 — Lima pertanyaan utk dijawab sebelum menulis kode apa pun, dan apa yang rusak jika Anda melewatkan masing-masing.

2.4 Jebakan Teks Soal yang Umum

Segelintir pola teks soal cukup sering berulang di SPOJ dan judge serupa hingga layak disebutkan secara eksplisit, agar Anda mengenalinya sekali lihat, bukan menemukannya dengan cara yang sulit.

- Konvensi indexing disebutkan sekali, di awal, dan tidak pernah diulang: soal yang bilang posisi dinomori mulai dari 1 di paragraf pertamanya tidak akan mengingatkan Anda lagi di bagian batasan — tapi indexing array Anda harus menghormatinya sepanjang soal.
- "Paling banyak" versus "tepat": batasan berbunyi "paling banyak N query" mengizinkan lebih sedikit, dan loop Anda tidak boleh mengasumsikan tepat N akan selalu datang.
- Banyak jawaban valid: beberapa soal menerima salah satu dari beberapa output benar (biasa ditandai frasa seperti "cetak susunan valid mana pun") — ini membutuhkan checker khusus di sisi judge, dan pengujian lokal Anda sendiri terhadap satu output yang diharapkan bisa menyesatkan Anda mengira program yang benar itu salah.
- Input sampel yang tidak menguji semua aturan: sampel yang hanya menampilkan nilai positif, berbeda, terurut tidak memberi tahu apa pun tentang bagaimana program Anda harus berperilaku pada nol, duplikat, atau input tak terurut — bahkan ketika batasan mengizinkan semua itu.

- Batas waktu dinyatakan per file test, bukan per test case: beberapa judge (dan pembuat soal) menyatakan satu batas waktu yang dimaksudkan utk mencakup jumlah semua test case dalam satu file, bukan masing-masing secara individu — ini mengubah seberapa agresif Anda perlu mengoptimalkan biaya setup per-case.

2.5 Contoh Kerja Pertama: Membaca Input Hingga End of File

Salah satu pola format-input paling umum di SPOJ menipu kesederhanaannya utk dideskripsikan dan mudah salah dalam kode: soal tidak memberi Anda jumlah test case eksplisit di awal, dan sebaliknya bilang input berlanjut hingga end of file (EOF). Pemula yang datang dari tugas kuliah, di mana struktur file input biasanya diumumkan dengan jumlah eksplisit, sering mencari batas loop yang tidak ada dalam soal, atau crash saat mencoba membaca satu nilai lewat test case nyata terakhir.

Idiom C++ yang benar memanfaatkan fakta bahwa operasi ekstraksi stream (seperti `cin >> a`) bernilai stream itu sendiri dalam konteks boolean, dan stream menjadi "falsy" begitu pembacaan gagal — termasuk gagal karena tidak ada lagi yang tersisa utk dibaca. Ini membuat pembacaan dan pemeriksaan akhir-loop menjadi ekspresi yang sama, tanpa pembukuan terpisah yang dibutuhkan.

read_until_eof_demo.cpp — loop input yang berakhir pada EOF (template orisinal)

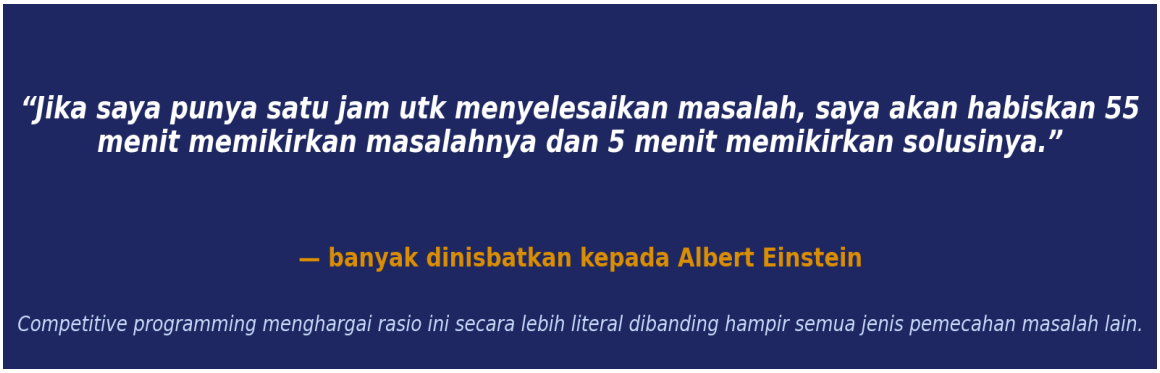
```
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    long long a, b;
    int caseNumber = 1;
    while (cin >> a >> b) {
        cout << "Case " << caseNumber << ": " << (a + b) << '\n';
        ++caseNumber;
    }
}
```

Template belasan-baris ini — demonstrasi orisinal, tidak terkait solusi accepted soal SPOJ spesifik mana pun — layak dihafal secara verbatim, karena pola `while (cin >> ...)` yang persis sama muncul kembali di sebagian besar soal SPOJ yang lebih lama dan sederhana (keluarga yang kumpulan soal SPOJ mata kuliah ini sendiri mencakup beberapa contohnya, dikatalogkan berdasarkan nama di Lampiran 15.A). Lampiran 2.A menunjukkan program ini di-compile, dijalankan terhadap input pipa kecil, dan outputnya diverifikasi (Gambar 2.3 dan 2.4).

> **Jumlah loop yang di-hardcode adalah bug paling umum dalam pola ini.**
>
> Jika Anda mengasumsikan tahu berapa banyak test case yang akan datang dan menulis `for (int i = 0; i < 5; i++)` alih-alih membaca hingga stream gagal, program Anda akan crash (atau berperilaku salah secara diam-diam) begitu input sesungguhnya judge memiliki jumlah test case yang berbeda dari perkiraan Anda — yang akan selalu terjadi, karena input tersembunyi judge bukan input sampel.

2.7 Ringkasan Bab

- Teks soal punya lima zona — cerita, tugas, format input, format output, batasan — dan membacanya dalam urutan prioritas itu (batasan dan format dulu, cerita terakhir pada bacaan ulang) menghemat waktu paling banyak.
- Batasan adalah spesifikasi sesungguhnya: mereka menetapkan kelas kompleksitas yang Anda butuhkan, edge case yang ada, dan struktur I/O yang harus ditangani program Anda.
- Jebakan umum termasuk konvensi indexing yang tidak dinyatakan, "paling banyak" vs. "tepat," banyak jawaban valid, input sampel yang tidak representatif, dan batas waktu dinyatakan per file bukan per test case.
- Checklist lima-pertanyaan (n terbesar, nilai nol/negatif, struktur jumlah test case, format output persis, aturan pemutus seri) menangkap kebanyakan bug WA dan RE sebelum pernah disubmit.
- Idiom `while (cin >> ...)` adalah cara standar dan benar utk membaca input yang berakhir pada EOF di C++ — hafalkan.



Gambar 2.3 — Pengingat bahwa membaca dengan hati-hati adalah bagian dari menyelesaikan soal itu sendiri.

Gambar 2.4 — Peta jalan 14-bab buku ini. Kita di sini: Bab 2.

Soal Latihan (Tidak Dinilai)

1. Sebuah teks soal bilang "baris pertama berisi integer T, jumlah test case," tapi teks soal lain bilang "baca hingga end of file." Jelaskan bagaimana loop pembacaan input Anda harus berbeda antara keduanya, dan apa yang terjadi jika Anda memakai pola yang salah utk masing-masing.
2. Batasan suatu soal menyatakan "1 ≤ n ≤ 10^5" tapi tidak bilang apa pun tentang apakah nilai array bisa berulang. Jelaskan mengapa Anda tidak bisa mengasumsikan nilai berbeda, dan deskripsikan satu input uji yang akan Anda coba utk memeriksa perilaku program Anda pada duplikat.
3. Jelaskan, dengan kata-kata Anda sendiri, mengapa "paling banyak N query" adalah batasan yang secara bermakna berbeda dari "tepat N query," dan berikan contoh bug yang muncul dari menyamakan keduanya.
4. Sebuah soal menerima "susunan valid mana pun" sebagai jawaban benar. Jelaskan mengapa membandingkan output program Anda terhadap satu file output-yang-diharapkan (seperti yang mungkin Anda lakukan utk pengujian lokal Anda sendiri) bisa menyesatkan, dan sarankan cara alternatif utk memeriksa kebenaran solusi Anda sendiri.
5. Tulis ulang kondisi loop read_until_eof_demo.cpp (Bagian 2.5) sebagai perbandingan boolean eksplisit — yaitu, jabarkan apa yang dievaluasi `cin >> a >> b` dan mengapa loop while berakhir persis saat seharusnya.

Lampiran 2.A — Log Verifikasi

Program read_until_eof_demo.cpp dari Bagian 2.5 di-compile dengan g++ (-O2 -Wall -pedantic) dan dijalankan terhadap input pipa kecil utk mengonfirmasi perilaku deteksi-EOF-nya sebelum dimasukkan ke bab ini.

Transkrip terminal

```
$ g++ -O2 -pedantic -Wall -o read_until_eof_demo read_until_eof_demo.cpp
$ printf "1 2\n3 4\n100 200\n" | ./read_until_eof_demo
Case 1: 3
Case 2: 7
Case 3: 300
```

Tiga baris input pipa menghasilkan persis tiga baris output, tanpa crash dan tanpa iterasi ekstra yang dicoba setelah pasangan terakhir — mengonfirmasi loop `while (cin >> a >> b)` berakhir persis pada end of file, sesuai klaim di Bagian 2.5.

Referensi

[1] Sphere Online Judge (SPOJ). https://www.spoj.com — konvensi teks soal umum yang dibahas di bab ini mencerminkan praktik standar di seluruh kumpulan soal SPOJ.

[2] Skiena, S. S. (2020). The Algorithm Design Manual (3rd ed.). Springer — tentang menerjemahkan deskripsi soal informal menjadi spesifikasi presisi.

[3] Halim, S., & Halim, F. (2013). Competitive Programming 3. Lulu Press — referensi utk idiom penanganan-input competitive-programming standar.

BAB 3

Memilih Algoritma yang Tepat untuk Batasan yang Diberikan

Program yang secara logika benar tetap bisa gagal di judge jika terlalu lambat atau memakai memori terlalu banyak. Bab ini mengajarkan kebiasaan paling berharga dalam competitive programming: membaca batasan terlebih dahulu, dan membiarkannya memilihkan algoritma untuk Anda.

Tujuan Pembelajaran — setelah bab ini Anda mampu:

(1) Menjelaskan mengapa algoritma yang benar tetap bisa menerima verdict Time Limit Exceeded atau Memory Limit Exceeded. (2) Menurunkan perkiraan anggaran operasi per detik dari batas waktu yang dinyatakan sebuah judge. (3) Menggunakan cheat sheet anggaran kompleksitas untuk memetakan n legal terbesar suatu batasan ke kelas kompleksitas yang benar-benar layak. (4) Mengenali, dari sinyal permukaan pada teks soal, mana dari tujuh keluarga algoritma umum yang kemungkinan berlaku. (5) Mengimplementasikan dan memverifikasi fast exponentiation, teknik standar O(log n) untuk menangani eksponen yang sangat besar.

3.1 Mengapa Kebenaran Saja Tidak Cukup

Bab 1 memperkenalkan Time Limit Exceeded (TLE) dan Memory Limit Exceeded (MLE) sebagai dua dari enam verdict standar, dan sudah menekankan bahwa keduanya bisa terjadi pada program yang sepenuhnya benar secara logika. Bab ini membahas cara menghindari hasil tersebut sejak awal, dengan memilih algoritma yang kompleksitasnya benar-benar muat di dalam batasan judge sebelum Anda menulis satu baris kode implementasi pun.

Setiap soal judge membawa dua batasan, biasanya dinyatakan di bagian atas halaman: batas waktu (umumnya 1-2 detik per test case di SPOJ, meski soal-soal lama kadang membawa batas lebih longgar warisan dari perangkat keras judge yang lebih lambat) dan batas memori (umumnya 32-256 MB). Kedua batasan ini adalah bagian dari spesifikasi, sama persis seperti format input — Bab 2 sudah menetapkan bahwa batasan adalah spesifikasi sesungguhnya dari sebuah soal, dan hal yang sama berlaku untuk dua angka ini.

Aturan praktis kasar: sekitar 10^8 hingga 10^9 operasi sederhana per detik.

Perangkat keras judge modern, menjalankan operasi sederhana (perbandingan, penjumlahan, akses array) tanpa overhead faktor konstanta yang berat, biasanya bisa menjalankan sekitar 100 juta hingga 1 miliar operasi semacam itu dalam anggaran 1 detik. Ini adalah aturan praktis, bukan jaminan — dan persis inilah dasar dari cheat sheet anggaran kompleksitas di Bagian 3.2.

3.2 Cheat Sheet Anggaran Kompleksitas

Begitu Anda tahu nilai n legal terbesar dari batasan — checklist membaca Bab 2 sudah menyuruh Anda mencari angka ini lebih dulu — Anda bisa langsung mempersempit kelas kompleksitas mana saja yang layak dicoba. Tabel 3.1 adalah acuan utama buku ini untuk langkah tersebut: cari baris yang cocok

dengan n Anda, lalu baca ke kanan untuk melihat apa yang layak dan pola apa yang biasanya mencapai kompleksitas itu (Gambar 3.1).

Cheat Sheet Anggaran Kompleksitas

Batas waktu judge biasanya ~1-2 detik untuk sekitar 10^8-10^9 operasi sederhana. Baca n legal terbesar, lalu baca baris ini ke kanan.

n legal terbesar	Kompleksitas yang layak	Apa yang bisa dibeli anggaran itu	Pola tipikal yang dipakai
n ≤ 10	O(n!), O(2^n · n)	Coba semua permutasi atau semua subset sekaligus	Brute-force permutasi, backtracking kecil
n ≤ 20-24	O(2^n), O(2^n · n)	Semua subset sekali, atau sekali per elemen	Bitmask DP, meet-in-the-middle
n ≤ 500-2000	O(n^3)	Loop bersarang tiga kali atas seluruh input	Floyd-Warshall, DP atas pasangan indeks
n ≤ 10^4-10^5	O(n^2), O(n log n)	Loop ganda di ujung kecil, berbasis sort di ujung besar	Sort + greedy, segment tree, BIT, sqrt decomposition
n ≤ 10^6	O(n log n), O(n)	Satu-dua kali lintasan linear penuh plus sort	Sieve of Eratosthenes, DP satu-pass, two pointers
n ≤ 10^9	O(sqrt(n)), O(log n)	Lintasan penuh atas n sudah tidak terjangkau sama sekali	Trial division hingga sqrt(n), binary search, fast exponentiation
n ≤ 10^18	O(log n), O(1)	Hanya rumus atau rekursi logaritmik yang bertahan	Matrix exponentiation, rumus tertutup, aritmetika modular

Gambar 3.1 — Cheat sheet anggaran kompleksitas. Cari baris yang cocok dengan n legal terbesar pada batasan, lalu baca ke kanan.

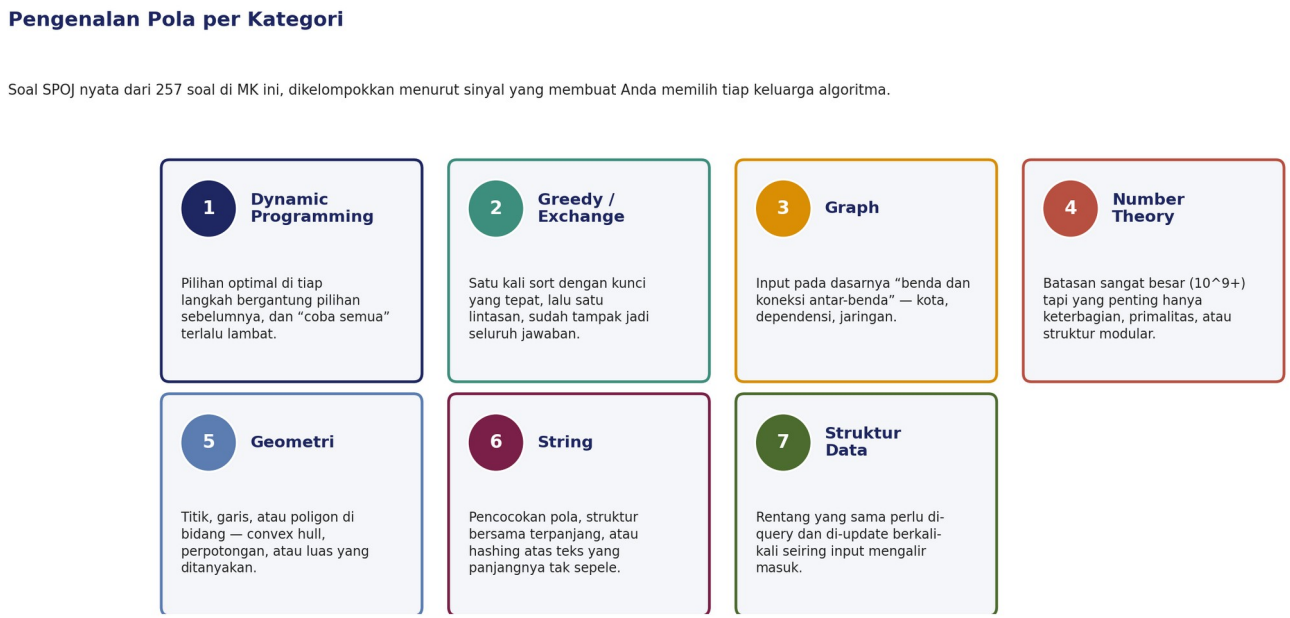
Dua hal tentang tabel ini patut dinyatakan secara eksplisit, karena pemula sering menyalahgunakannya dengan cara yang bisa diprediksi. Pertama, beberapa baris mencantumkan lebih dari satu kompleksitas layak karena batasnya bergantung pada faktor konstanta implementasi Anda secara persis — algoritma O(n^2) masih nyaman di n = 10^4 (10^8 operasi) tetapi sudah jauh terlalu lambat di n = 10^5 (10^10 operasi), jadi ketika sebuah batasan berada dekat batas suatu baris, lakukan sendiri perkalian tersebut alih-alih hanya percaya pada label baris. Kedua, tabel ini menggambarkan apa yang secara teori layak, bukan apa yang dijamin lolos — algoritma dengan kompleksitas big-O yang tepat namun faktor konstanta berat (pemanggilan fungsi bersarang dalam-dalam, alokasi memori yang tidak perlu di dalam hot loop, pola akses memori yang tidak ramah cache) tetap bisa TLE. Bab 8 membahas cara mendiagnosis situasi tersebut setelah terjadi.

Mengapa tabel melompat dari O(n^3) di angka ribuan langsung ke O(n log n) di angka sejuta?

Karena jarak antara dua ukuran batasan tersebut sangat besar dalam jumlah operasi absolut, dan tidak ada wilayah tengah yang berguna untuk disebutkan: algoritma yang secara asimtotik berada di antara O(n^3) dan O(n log n) — misalnya O(n^2) — sudah berhenti aman jauh sebelum n mencapai 10^5 (10^10 operasi pada n = 10^5). Memilih algoritma yang tepat bukanlah dial yang halus; lebih mirip memilih anak tangga yang benar pada sebuah tangga, dan meleset satu anak tangga biasanya berarti Time Limit Exceeded seketika, bukan Accepted yang sedikit lebih lambat.

3.3 Pengenalan Pola per Kategori

Di luar anggaran numerik murni, bentuk input dan pertanyaan sebuah soal itu sendiri adalah sinyal kuat untuk keluarga teknik mana yang berlaku. Gambar 3.2 mengelompokkan tujuh keluarga semacam itu menurut sinyal permukaan yang seharusnya membuat Anda memilih masing-masing — inilah pengenalan pola, dan seperti keterampilan pengenalan lainnya, ia membaik dengan paparan yang disengaja pada banyak soal, yang persis apa yang dibangun oleh bab-bab teknik selanjutnya buku ini (terutama 9 hingga 11).



Gambar 3.2 — Tujuh keluarga algoritma dan sinyal permukaan yang seharusnya membuat Anda memilih masing-masing.

3.3.1 Contoh Pengenalan yang Diselesaikan

Perhatikan soal number theory seperti PRIME1 (dikutip di sini hanya dengan nama publik soal — lihat janji kerahasiaan buku ini di Bagian 1.5), yang meminta semua bilangan prima dalam suatu rentang di mana batas atasnya bisa mencapai 10^9. Batasan itu sendiri sudah menyingkirkan opsi menguji tiap bilangan satu per satu untuk primalitas lewat trial division di seluruh rentang — itu akan jauh terlalu lambat. Sinyal "n sangat besar, tetapi hanya struktur keterbagian/primalitas yang penting" (baris 4 pada Gambar 3.2) mengarah ke number theory, dan ukuran n yang spesifik (melewati 10^6, menurut Tabel 3.1) mengarah ke teknik segmented alih-alih satu sieve datar di seluruh rentang — sebuah penyempurnaan yang dibahas lagi di Bab 6.

> **Jangan memaksakan pola pertama yang Anda kenali.**
>
> Soal yang menyebutkan grid bisa terlihat seolah menginginkan graph search, padahal sinyal sesungguhnya — subproblem yang tumpang-tindih berulang — sebenarnya mengarah ke dynamic programming. Pengenalan pola adalah hipotesis awal untuk diuji terhadap batasan dan pertanyaan persis yang diajukan (Bagian 2.2.1 Bab 2), bukan vonis untuk langsung dikomit.

3.4 Contoh yang Diselesaikan: Ketika Eksponennya Sendiri Sangat Besar

Beberapa soal meminta nilai seperti basis dipangkatkan eksponen yang sangat besar, modulo suatu bilangan — dengan eksponen itu sendiri mencapai 10^9 atau lebih. Loop yang mengalikan basis dengan dirinya sendiri, satu kali per satuan eksponen, adalah O(eksponen) — dan menurut Tabel 3.1, eksponen pada skala tersebut menyingkirkan lintasan linear penuh sama sekali. Perbaikan standarnya adalah binary exponentiation (juga disebut fast atau modular exponentiation): mengkuadratkan basis berulang kali sambil membagi dua eksponen, yang menghitung hasil yang sama dalam waktu O(log eksponen) — sekitar 30 kali pengkuadratan, bukan satu miliar kali perkalian.

fast_power_demo.cpp — binary exponentiation (templat orisinal)

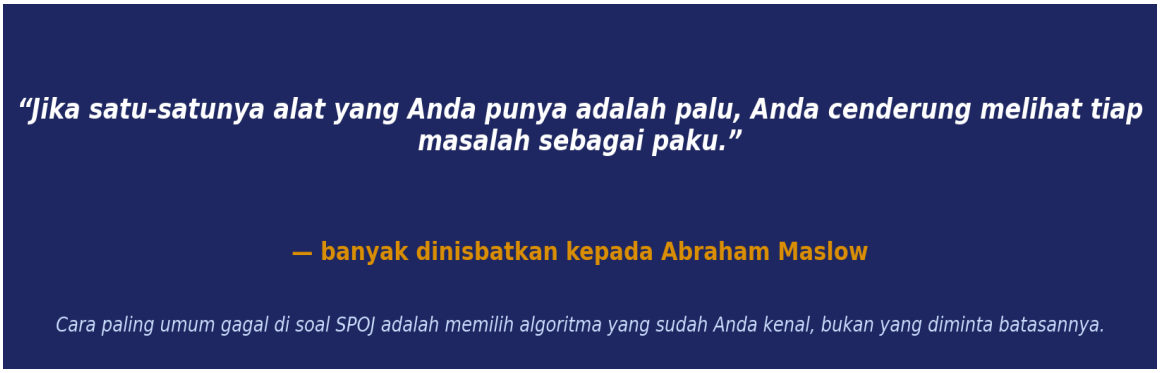
```
#include <bits/stdc++.h>
using namespace std;

// Mengembalikan (base^exp) % mod via binary exponentiation, O(log exp).
long long power_mod(long long base, long long exp, long long mod) {
    long long result = 1 % mod;
    base %= mod;
    if (base < 0) base += mod;
    while (exp > 0) {
        if (exp & 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp >>= 1;  // bagi dua exp tiap putaran -- inilah O(log exp)
    }
    return result;
}
```

Templat belasan baris ini — demonstrasi orisinal, tidak terkait solusi accepted soal SPOJ mana pun — bergeneralisasi langsung ke matrix exponentiation untuk soal linear-recurrence (disebutkan pada baris terakhir Tabel 3.1), karena ide bagi-dua-eksponen yang sama berlaku kapan pun operasi yang mendasarinya bersifat asosiatif. Lampiran 3.A menunjukkan program ini dikompilasi, dijalankan, dan hasilnya diperiksa silang terhadap perhitungan independen (Gambar 3.3 dan 3.4).

3.6 Ringkasan Bab

- Algoritma yang benar tetap bisa gagal dengan Time Limit Exceeded atau Memory Limit Exceeded — kompleksitas harus dipilih agar muat dalam batasan judge, bukan sekadar menghasilkan jawaban benar pada akhirnya.
- Anggaran kasar 10^8 hingga 10^9 operasi sederhana per detik memungkinkan Anda mengubah n legal terbesar suatu batasan langsung menjadi kelas kompleksitas yang layak (Tabel 3.1).
- Ketika sebuah batasan berada dekat batas baris pada cheat sheet, lakukan sendiri aritmetikanya alih-alih hanya percaya pada label baris.
- Tujuh keluarga algoritma — dynamic programming, greedy/exchange, graph, number theory, geometri, string, dan struktur data — masing-masing punya sinyal permukaan yang dapat dikenali pada teks soal (Gambar 3.2).

- Binary exponentiation menghitung base^eksponen mod m dalam waktu O(log eksponen), dan ide bagi-dua-eksponen yang sama bergeneralisasi ke matrix exponentiation.



Gambar 3.3 — Pengingat bahwa alat yang tepat bergantung pada soalnya, bukan pada kebiasaan.

Gambar 3.4 — Peta jalan 14 bab buku ini. Kita di sini: Bab 3.

Pertanyaan Tinjauan (Tidak Dinilai)

1. Sebuah soal menyatakan batas waktu 1 detik dan batasan n ≤ 10^7. Menggunakan Tabel 3.1, sebutkan kelas kompleksitas yang kemungkinan layak, dan satu yang tidak, tunjukkan aritmetika jumlah-operasi yang membenarkan jawaban Anda.
2. Jelaskan, dengan kata-kata Anda sendiri, mengapa algoritma O(n^2) aman di n = 10^4 tetapi tidak aman di n = 10^5, menggunakan anggaran 10^8-10^9 operasi per detik yang sama dari Bagian 3.1.
3. Cerita sebuah soal menggambarkan jaringan kota yang terhubung oleh jalan. Menggunakan Gambar 3.2, sebutkan keluarga algoritma mana yang paling kuat ditunjuk sinyal ini, dan jelaskan satu cara pertanyaan sesungguhnya pada soal itu tetap bisa mengarahkan Anda ke keluarga lain.
4. Jelaskan mengapa menghitung base^eksponen dengan loop sederhana perkalian sebanyak eksponen menjadi tidak layak begitu eksponen mencapai 10^9, dan jelaskan dengan kata-kata Anda sendiri bagaimana binary exponentiation menghindari biaya tersebut.
5. Tulis ulang fungsi power_mod dari Bagian 3.4 agar sebaliknya menghitung base dipangkatkan eksponen tanpa modulus sama sekali (eksponensiasi bilangan bulat biasa), dan jelaskan masalah praktis apa yang muncul jika Anda mencoba ini untuk eksponen 10^9 tanpa memakai tipe bilangan bulat presisi-sembarang.

Lampiran 3.A — Log Verifikasi

Program fast_power_demo.cpp dari Bagian 3.4 dikompilasi dengan g++ (-O2 -Wall -pedantic), dijalankan, dan hasilnya diperiksa silang terhadap perhitungan independen menggunakan fungsi bawaan Python pow() tiga-argumen sebelum dimasukkan ke bab ini.

Transkrip terminal

```
$ g++ -O2 -pedantic -Wall -o fast_power_demo fast_power_demo.cpp
$ ./fast_power_demo
7^1000000000 mod 1000000007 = 312556845
2^10 mod 1000 = 24
$ python3 -c "print(pow(7, 1000000000, 1000000007))"
312556845
```

Hasil program C++ untuk kasus eksponen besar (312556845) cocok persis dengan pow(base, exp, mod) Python yang diimplementasikan secara independen, dan kasus kecil yang mudah diperiksa manual (2^10 mod 1000 = 1024 mod 1000 = 24) turut mengonfirmasi kebenaran fungsi tersebut pada input yang mudah diverifikasi.

Referensi

[1] Sphere Online Judge (SPOJ). https://www.spoj.com — konvensi batasan soal umum yang dibahas dalam bab ini mencerminkan praktik standar di seluruh kumpulan soal SPOJ.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — untuk analisis kompleksitas asimtotik dan modular exponentiation.

[3] Halim, S., & Halim, F. (2013). Competitive Programming 3. Lulu Press — acuan untuk heuristik pemilihan algoritma berbasis batasan.

BAB 4

I/O Cepat di C/C++

Algoritma yang sempurna benar dan sempurna efisien tetap bisa menerima Time Limit Exceeded jika kode di sekitarnya terlalu lama hanya untuk membaca input dan menulis output. Bab ini mengajarkan tiga tingkat kecepatan I/O dan, yang lebih penting, cara mengetahui tingkat mana yang benar-benar dibutuhkan sebuah soal.

Tujuan Pembelajaran — setelah bab ini Anda mampu:

(1) Menjelaskan mengapa setup iostream default C++ bisa lebih lambat dari scanf/printf, dan apa persisnya penyebab perlambatan itu. (2) Menerapkan idiom sync_with_stdio(false) plus cin.tie(nullptr) dengan benar, dan menyatakan satu aturan yang ditetapkannya. (3) Membandingkan tiga tingkat kecepatan I/O dan setup yang dibutuhkan masing-masing. (4) Memperkirakan, dari batasan sebuah soal, tingkat mana yang benar-benar diperlukan. (5) Mengimplementasikan dan memverifikasi reader integer buffer kustom untuk kasus langka yang membutuhkannya.

4.1 Mengapa I/O Bisa Diam-Diam Mendominasi Waktu Eksekusi

Mudah untuk berasumsi bahwa membaca input dan menulis output itu gratis, dan seluruh anggaran waktu judge dihabiskan untuk algoritma itu sendiri. Ini tidak benar. Secara default, cin dan cout milik C++ dibangun di atas mesin stream yang sama dengan stdio milik C (scanf dan printf), dan agar kedua keluarga ini bisa dicampur bebas — membaca satu nilai dengan scanf lalu nilai berikutnya dengan cin, misalnya — pustaka standar C++ menjaga keduanya tetap tersinkronisasi. Sinkronisasi itu punya biaya nyata yang terukur, dan ketika batasan sebuah soal menuntut membaca atau menulis sejumlah besar nilai, biaya itu bisa bertambah menjadi sebagian besar dari batas waktu, atau bahkan melampauinya, terlepas dari sebaik apa pun algoritma yang mendasarinya.

Ini adalah mode kegagalan yang berbeda dari yang menjadi fokus Bab 3. Bab 3 membahas memilih algoritma yang kompleksitasnya muat dengan batasan; bab ini membahas memastikan pipa di sekitar algoritma yang sudah dipilih dengan benar tidak menjadi bottleneck itu sendiri.

Gejalanya adalah TLE yang mencurigakan pada solusi yang sebenarnya benar dan efisien.

Jika kompleksitas algoritma Anda nyaman berada di dalam anggaran dari cheat sheet Bab 3, dan Anda tetap mendapat Time Limit Exceeded, overhead I/O adalah salah satu tempat pertama yang patut diperiksa — terutama jika batasan menyebutkan membaca atau mencetak sekitar 10^5 nilai atau lebih.

4.2 Tiga Tingkat Kecepatan I/O

Tidak ada satu cara "benar" tunggal untuk melakukan I/O dalam competitive programming — ada spektrum pendekatan, masing-masing menukar sedikit lebih banyak kompleksitas setup dengan sedikit lebih banyak kecepatan. Tabel 4.1 menjabarkan tiga tingkat praktis. Pilihan yang tepat hampir selalu yang termurah yang dengan nyaman melewati batasan — memakai Tingkat 3 pada soal yang

sebenarnya hanya butuh Tingkat 1 membuang waktu Anda dan menambah tempat bagi bug untuk bersembunyi (Gambar 4.1).

Tiga Tingkat Kecepatan I/O

Tiap tingkat menuntut sedikit lebih banyak setup dan membeli sedikit lebih banyak kecepatan. Cocokkan tingkat dengan tuntutan batasan sebenarnya.

Tingkat	Setup yang diperlukan	Kapan ini penting	Risiko utama
Tingkat 1 — cin/cout atau scanf/printf default	Tidak ada — ini defaultnya	Input kecil: kurang lebih di bawah 10^5 total token	Tidak ada, jika dipakai konsisten dan tidak dicampur
Tingkat 2 — Nonaktifkan sync, hindari endl	Dua baris sekali, di awal main	Input sedang-besar: kurang lebih 10^5 hingga beberapa juta token	Jangan pernah campur cin/cout dengan scanf/printf setelah sync dinonaktifkan
Tingkat 3 — Reader buffer kustom (berbasis fread)	Rutin reader kecil yang bisa dipakai ulang	Input sangat besar: puluhan juta token, atau banyak pembacaan per test case dgn batas waktu ketat	Lebih banyak kode yang harus benar — tulis sekali, uji, lalu pakai ulang

Gambar 4.1 — Tiga tingkat kecepatan I/O, setup yang dibutuhkan masing-masing, dan kapan masing-masing benar-benar penting.

4.3 Idiom Nonaktifkan-Sync, dan Satu Aturan Kerasnya

Tingkat 2 adalah dua baris, ditempatkan sekali di paling awal main, sebelum input apa pun dibaca:

Setup Tingkat 2 (sudah dipakai di setiap contoh terselesaikan sejauh ini di buku ini)

```
ios_base::sync_with_stdio(false);
cin.tie(nullptr);
```

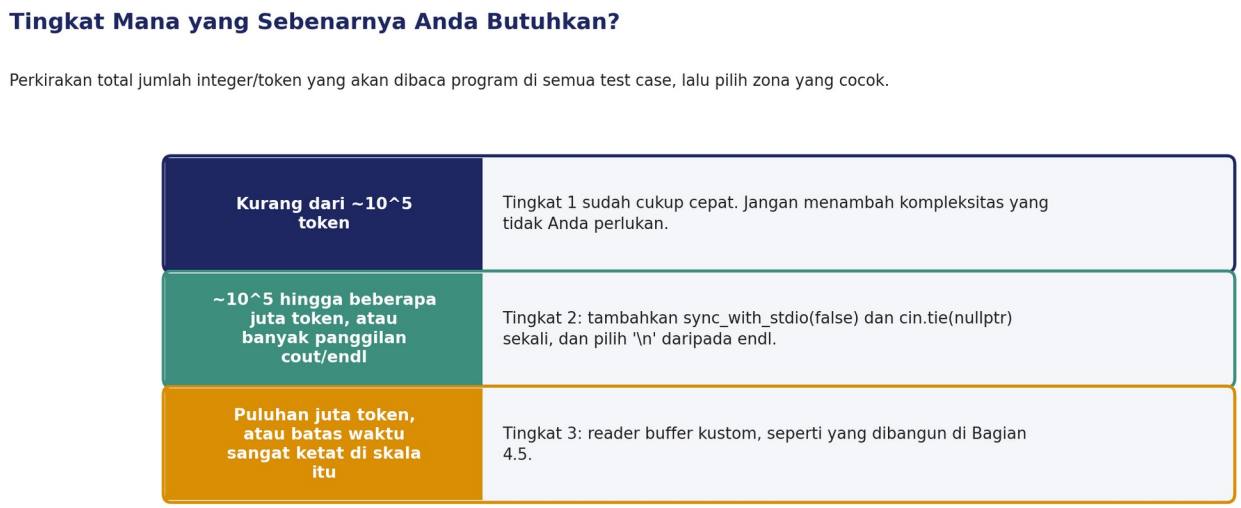
Baris pertama memberi tahu pustaka standar C++ bahwa cin/cout tidak perlu lagi tetap tersinkronisasi dengan stdio milik C, yang menghilangkan overhead sinkronisasi yang dijelaskan di Bagian 4.1. Baris kedua menghilangkan biaya tambahan yang terpisah: secara default, cin "diikat" ke cout, artinya cout otomatis di-flush sebelum setiap pembacaan cin — berguna untuk program interaktif di mana sebuah prompt harus muncul sebelum input diminta, tetapi murni overhead pada judge, di mana seluruh input sudah tersedia di awal dan tidak ada prompt semacam itu yang dibutuhkan.

> **Setelah sync dinonaktifkan, jangan pernah mencampur cin/cout dengan scanf/printf dalam program yang sama.**
>
> Inti dari Tingkat 2 adalah memberi tahu kedua keluarga stream untuk berhenti tetap tersinkronisasi satu sama lain. Jika Anda menonaktifkan sync lalu membaca sebagian input dengan scanf dan sebagian lain dengan cin, buffer internal kedua stream tersebut bisa saling terselip dalam urutan yang tidak terdefinisi, menghasilkan input yang kacau atau hilang tanpa peringatan kompiler apa pun yang menangkapnya. Pilih satu keluarga — cin/cout atau scanf/printf — untuk seluruh program, dan tetap dengan pilihan itu.

Kebiasaan kedua yang lebih kecil dan patut diadopsi bersama Tingkat 2: pilih '\n' daripada std::endl saat menulis output. std::endl melakukan dua hal — menulis newline, dan meng-flush buffer output — dan aksi kedua itu adalah kerja tidak perlu yang berulang di setiap baris jika sebuah program mencetak banyak baris output, karena buffer akan otomatis di-flush juga saat program berakhir.

4.4 Kapan Anda Benar-Benar Membutuhkan Tingkat 3

Tingkat 2 sudah cukup untuk mayoritas besar soal SPOJ. Tingkat 3 — reader buffer tulisan tangan yang sepenuhnya melewati cin/cout dan scanf/printf — hanya sepadan dengan tambahan kompleksitasnya ketika total jumlah token yang dibaca naik ke puluhan juta, atau ketika batas waktu yang ketat digabung dengan banyak pembacaan kecil yang tersebar di banyak test case. Gambar 4.2 memberikan panduan keputusan kasar berdasarkan perkiraan total jumlah token, terhubung langsung ke kebiasaan membaca batasan yang dibangun Bab 2 dan pemikiran anggaran-kompleksitas dari Bab 3.



Gambar 4.2 — Tingkat I/O mana yang benar-benar dibutuhkan sebuah soal, berdasarkan total jumlah token yang harus dibaca.

4.5 Contoh yang Diselesaikan: Reader Buffer Kustom

Templat berikut membaca byte langsung dari input standar dalam blok besar memakai fread(), menjaga buffer kecilnya sendiri, dan mem-parsing integer langsung dari buffer tersebut satu karakter pada satu waktu — tanpa memanggil mesin buffering internal cin, cout, scanf, atau printf sama sekali untuk jalur pembacaan.

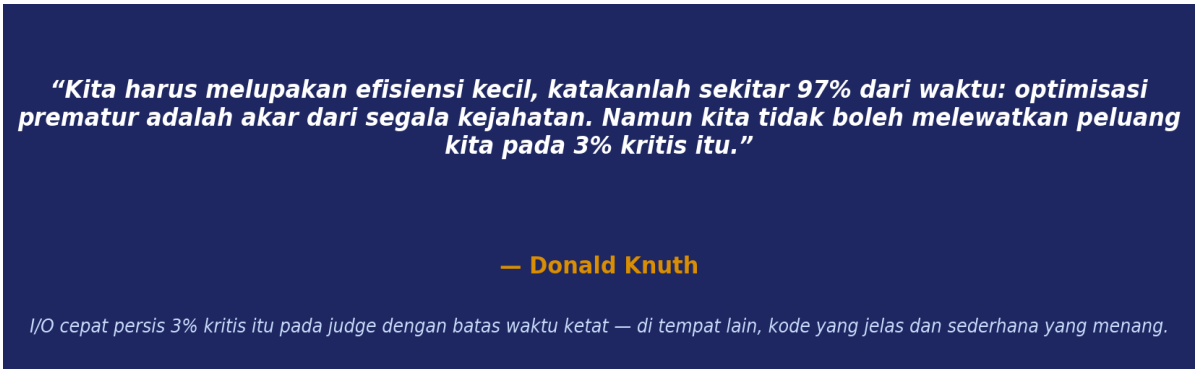
fast_reader_demo.cpp — reader integer buffer (templat orisinal)

```
namespace fastio {
    const int BUF_SIZE = 1 << 16;
    char buf[BUF_SIZE];
    int bufLen = 0, bufPos = 0;

    inline int readChar() {
        if (bufPos == bufLen) {
            bufLen = fread(buf, 1, BUF_SIZE, stdin);
            bufPos = 0;
            if (bufLen == 0) return -1;  // akhir file
        }
        return buf[bufPos++];
    }

    inline long long readInt() {
        int c = readChar();
        while (c==' '||c=='\n'||c=='\r'||c=='\t') c = readChar();
        bool neg = false;
        if (c == '-') { neg = true; c = readChar(); }
        long long x = 0;
        while (c >= '0' && c <= '9') {
            x = x * 10 + (c - '0');
            c = readChar();
        }
        return neg ? -x : x;
    }
}
```

Templat belasan baris ini — demonstrasi orisinal, tidak terkait solusi accepted soal SPOJ mana pun — diverifikasi terhadap dua juta integer acak yang dibangkitkan, dengan hasil hitung sum, minimum, dan maksimumnya diperiksa silang terhadap perhitungan Python independen atas data yang sama (Lampiran 4.A). Membaca seluruh dua juta nilai selesai jauh di bawah sepersepuluh detik (Gambar 4.3 dan 4.4).

4.7 Ringkasan Bab

- cin/cout default C++ tersinkronisasi dengan stdio milik C secara default, yang punya biaya performa nyata pada input besar.
- Tingkat 2 — sync_with_stdio(false) plus cin.tie(nullptr), ditempatkan sekali di awal main — menghilangkan biaya itu, tetapi melarang mencampur cin/cout dengan scanf/printf setelahnya.
- Memilih '\n' daripada std::endl menghindari flush buffer yang tidak perlu di setiap baris output.
- Tingkat 3 — reader buffer kustom berbasis fread — hanya sepadan dengan kompleksitasnya begitu total input mencapai puluhan juta token.
- Cocokkan tingkat I/O dengan tuntutan batasan sebenarnya — tingkat termurah yang melewati persyaratan biasanya pilihan yang tepat.



Gambar 4.3 — Peringatan klasik Knuth terhadap optimisasi berlebihan juga berlaku di sini — I/O cepat adalah pengecualian, bukan kebiasaan default.

Gambar 4.4 — Peta jalan 14 bab buku ini. Kita di sini: Bab 4.

Pertanyaan Tinjauan (Tidak Dinilai)

1. Jelaskan, dengan kata-kata Anda sendiri, apa arti "tersinkronisasi dengan stdio" untuk cin/cout, dan mengapa menonaktifkan sinkronisasi itu hanya aman jika Anda berkomitmen memakai satu keluarga stream saja untuk sisa program.
2. Sebuah soal punya n ≤ 1000 dan meminta satu nilai hasil hitung dicetak. Menggunakan Gambar 4.2, tentukan tingkat I/O mana yang sesuai, dan jelaskan mengapa memakai Tingkat 3 di sini tidak perlu.
3. Jelaskan apa persisnya yang dilakukan std::endl yang tidak dilakukan '\n', dan mengapa perbedaan itu penting saat sebuah program mencetak sangat banyak baris output.
4. Sebuah program mencampur cin >> x untuk sebagian input dengan scanf("%d", &y) untuk input lain, setelah memanggil sync_with_stdio(false). Jelaskan apa yang bisa salah, dan bagaimana Anda akan memperbaiki program tersebut.
5. Modifikasi fungsi readInt() dari Bagian 4.5 agar juga menangani tanda '+' di depan (selain penanganan '-' yang sudah ada), dan jelaskan mengapa format input kebanyakan soal SPOJ membuat ini tidak diperlukan dalam praktiknya.

Lampiran 4.A — Log Verifikasi

Program fast_reader_demo.cpp dari Bagian 4.5 dikompilasi dengan g++ (-O2 -Wall -pedantic), pertama diperiksa terhadap input kecil yang diverifikasi manual, lalu dijalankan terhadap berkas dua

juta integer acak, dengan hasilnya diperiksa silang terhadap perhitungan Python independen atas data yang sama.

Transkrip terminal

```
$ g++ -O2 -pedantic -Wall -o fast_reader_demo fast_reader_demo.cpp
$ printf "5 1 2 3 4 5\n" | ./fast_reader_demo
n=5 sum=15 min=1 max=5
$ printf "6 -3 10 -7 2 0 100\n" | ./fast_reader_demo
n=6 sum=102 min=-7 max=100
$ # 2.000.000 integer acak di [-1000000, 1000000], seed 42
$ time ./fast_reader_demo < big_input.txt
n=2000000 sum=-189011875 min=-999999 max=999999
real    0m0.034s
$ # Python: sum/min/max atas 2.000.000 nilai yang sama
expected sum -189011875 min -999999 max 999999
```

Ketiga pemeriksaan cocok persis — termasuk pengujian skala besar, yang sum, minimum, dan maksimumnya sesuai persis dengan referensi yang dihitung secara independen di Python — dan membaca seluruh dua juta nilai memakan waktu sekitar 34 milidetik, jauh di dalam batas waktu judge mana pun.

Referensi

[1] Sphere Online Judge (SPOJ). https://www.spoj.com — konvensi performa I/O yang dibahas dalam bab ini mencerminkan praktik standar di seluruh kumpulan soal SPOJ.

[2] Halim, S., & Halim, F. (2013). Competitive Programming 3. Lulu Press — acuan untuk idiom I/O cepat standar dalam competitive programming.

[3] Knuth, D. E. (1974). Structured Programming with go to Statements. Computing Surveys, 6(4) — asal peringatan yang banyak dikutip terhadap optimisasi prematur yang dikutip di Bagian 4.7.

BAB 5

Pragma Kompiler, Flag, dan Risiko Portabilitas

Satu baris di awal berkas sumber bisa mengubah seberapa cepat program berjalan — atau membuatnya crash di mesin yang belum pernah Anda lihat. Bab ini menjelaskan apa yang sebenarnya dilakukan pragma kompiler, mana yang gratis meningkatkan performa dan mana yang merupakan taruhan.

Tujuan Pembelajaran — setelah bab ini Anda mampu:

(1) Menjelaskan apa yang sebenarnya diinstruksikan #pragma GCC optimize dan #pragma GCC target ke kompiler. (2) Membedakan pragma optimisasi yang aman dan portabel dari pragma penargetan-fitur-CPU yang membawa risiko portabilitas nyata. (3) Menjelaskan mengapa perangkat keras judge yang lama atau heterogen mengubah risiko itu menjadi Runtime Error sesungguhnya. (4) Memilih default pragma yang masuk akal untuk submission SPOJ, dibanding taruhan yang disengaja dan terinformasi. (5) Mengukur, dengan benchmark orisinal, efek nyata sebuah pragma optimisasi aman pada loop yang berat komputasi.

5.1 Mengapa Flag Kompiler Penting di Judge

Bab 3 dan 4 sama-sama membahas mengendalikan biaya dari dalam kode sumber Anda sendiri — memilih algoritma yang kompleksitasnya muat dengan batasan, dan memilih strategi I/O yang tidak memboroskan anggaran waktu yang dibutuhkan algoritma itu. Bab ini membahas tuas ketiga, yang sepenuhnya berada di luar algoritma Anda: seberapa agresif kompiler diizinkan mengoptimalkan kode mesin yang dihasilkannya dari kode sumber tersebut.

Banyak judge, termasuk SPOJ untuk sebagian besar soalnya, mengompilasi submission dengan perintah build yang tidak Anda kendalikan dan tidak bisa Anda lihat — umumnya sesuatu yang mendekati pemanggilan g++ biasa, kadang sudah memakai -O2, kadang tidak. Baris #pragma yang ditempatkan di awal berkas sumber Anda adalah cara menginstruksikan kompiler secara langsung, dari dalam berkas itu sendiri, terlepas dari flag apa pun yang kebetulan dipakai skrip build judge sendiri. Inilah mengapa pragma layak dipahami meski flag seperti -O2 atau -O3 pada pandangan pertama tampak seperti sesuatu yang hanya diurus sistem build.

Pragma adalah permintaan yang tertanam dalam kode sumber Anda, bukan flag command-line yang harus Anda harap sudah dipakai judge.

Persis inilah mengapa competitive programmer memakai pragma alih-alih sekadar berasumsi judge mengompilasi dengan -O2 atau -O3 secara default: berasumsi adalah tebakan, sementara pragma adalah jaminan yang ikut bersama submission Anda terlepas dari konfigurasi build judge sendiri.

5.2 Dua Jenis Pragma: Tingkat Optimisasi vs. Target CPU

Tidak setiap pragma membawa risiko yang sama, dan mencampuradukkan keduanya adalah kesalahan paling umum di area ini. Ada dua keluarga yang fundamental berbeda yang perlu dipisahkan dalam pikiran Anda.

5.2.1 Pragma Tingkat Optimisasi

#pragma GCC optimize("O2") dan #pragma GCC optimize("O3,unroll-loops") memberi tahu kompiler seberapa agresif menyusun ulang kode Anda — meng-inline lebih banyak fungsi, membuka gulungan (unroll) loop, menyusun ulang komputasi — sambil tetap menghasilkan kode mesin yang berjalan benar di prosesor mana pun yang didukung arsitektur target. Pragma ini mengubah seberapa cepat kode yang dihasilkan berjalan; pragma ini tidak mengubah instruksi apa yang boleh diasumsikan kode itu dimiliki CPU.

5.2.2 Pragma Target CPU

#pragma GCC target("avx2,bmi,bmi2,popcnt") dan varian serupa adalah instruksi yang secara fundamental berbeda: mereka memberi tahu kompiler bahwa ia diizinkan menghasilkan instruksi mesin dari ekstensi instruction-set CPU tertentu — ekstensi yang mungkin atau mungkin tidak benar-benar diimplementasikan sebuah prosesor. Kode yang dikompilasi dengan cara ini berjalan lebih cepat pada CPU yang mendukung ekstensi tersebut, dan langsung crash, dengan Runtime Error illegal-instruction, pada CPU yang tidak mendukungnya (Gambar 5.1).

> **Risiko portabilitas pragma target() tidak bisa diuji di komputer Anda sendiri.**
>
> Jika komputer pengembangan Anda sendiri mendukung fitur CPU yang Anda targetkan (yang mungkin saja, pada perangkat keras modern), program Anda akan terkompilasi dan berjalan sempurna bagi Anda — dan tetap crash begitu berjalan di mesin judge dengan CPU yang lebih lama atau berbeda. Tidak ada uji lokal yang mengungkap risiko ini; itu hanya muncul saat submission, di perangkat keras yang tidak Anda kendalikan.

5.3 Inventaris Pragma

Tabel 5.1 mengkatalogkan pragma yang benar-benar muncul di seluruh kumpulan soal SPOJ terselesaikan MK ini, beserta apa yang dilakukan masing-masing dan risiko portabilitasnya yang jujur. #pragma GCC optimize("O3,unroll-loops") adalah, dengan selisih jauh, pilihan tunggal paling umum — dan bukan kebetulan, karena ia juga peningkatan kecepatan berarti paling aman yang tersedia.

Inventaris Pragma: Apa yang Sebenarnya Dipertaruhkan Masing-Masing

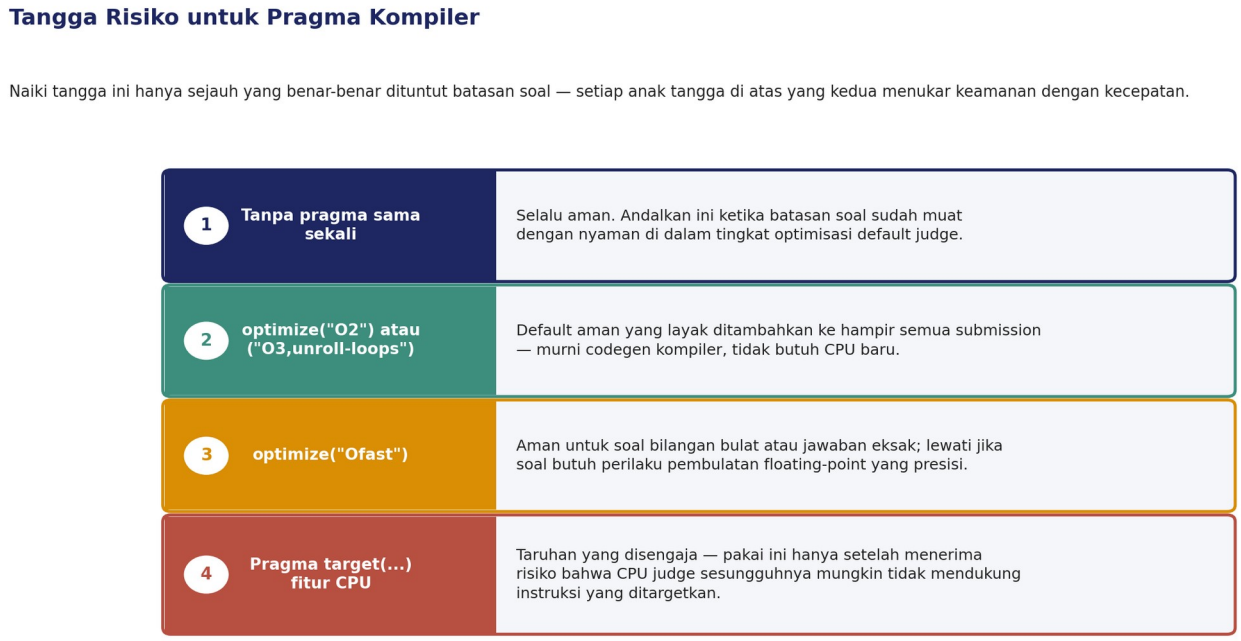
Tidak semua pragma kompiler membawa risiko portabilitas yang sama. Baca tabel ini sebelum menambahkan salah satunya.

Pragma	Apa yang dilakukan	Risiko portabilitas	Rekomendasi
`#pragma GCC optimize("O2")`	Tingkat optimisasi standar, sedang	Tidak ada — murni keputusan codegen kompiler	Default aman; pakai dengan bebas
`#pragma GCC optimize("O3,unroll-loops")`	Optimisasi agresif plus unroll loop	Tidak ada — tetap kode mesin portabel, tidak butuh instruksi CPU baru	Aman; pragma tunggal paling umum di kumpulan soal SPOJ terselesaikan MK ini
`#pragma GCC optimize("Ofast")`	O3 plus kepatuhan IEEE floating-point yang dilonggarkan	Rendah untuk kode bilangan bulat saja; bisa mengubah pembulatan/presisi floating-point	Pakai hanya jika soal tidak butuh persyaratan floating-point eksak
`#pragma GCC target("avx2,bmi,bmi2,popcnt")` (dan varian fitur CPU serupa)	Menghasilkan instruksi dari ekstensi instruction-set CPU tertentu	Nyata: Runtime Error illegal-instruction di mesin judge mana pun yang CPU-nya tidak punya instruction set itu	Taruhan yang disengaja dan diuji — jangan pernah jadi kebiasaan default

Gambar 5.1 — Inventaris pragma: apa yang dilakukan masing-masing, dan apa yang sebenarnya dipertaruhkan.

5.4 Tangga Risiko untuk Pragma Kompiler

Gambar 5.2 menyusun pilihan-pilihan ini sebagai sebuah tangga. Menaiki lebih tinggi membeli lebih banyak kecepatan mentah, tetapi setiap anak tangga di atas yang kedua menukar sebagian keamanan portabilitas — dan anak tangga teratas harus diperlakukan sebagai taruhan yang disengaja dan terinformasi, bukan pernah kebiasaan yang diterapkan secara default seperti pragma Bagian 5.2.1.



Gambar 5.2 — Tangga risiko untuk pragma kompiler, dari selalu-aman hingga taruhan-yang-disengaja.

5.5 Contoh yang Diselesaikan: Mengukur Efek Pragma yang Aman

Cara paling jelas untuk melihat apa yang sebenarnya dibeli pragma tingkat-optimisasi adalah mengompilasi kode sumber yang persis sama dua kali — sekali tanpa pragma, sekali dengan satu baris #pragma GCC optimize("O3,unroll-loops") yang ditambahkan — memakai perintah kompilasi biasa yang identik tanpa flag -O apa pun sama sekali di command line, sehingga perbedaan kecepatan apa pun hanya bisa berasal dari pragma itu sendiri.

pragma_on_demo.cpp — loop berat komputasi dengan pragma (templat orisinal)

```
#pragma GCC optimize("O3,unroll-loops")
#include <bits/stdc++.h>
using namespace std;

int main() {
    long long n = 2000000000LL;
    unsigned long long sum = 0;
    for (long long i = 0; i < n; i++) {
        sum += (unsigned long long)(i * i);
    }
    cout << "sum of i*i for i in [0, " << n << "): " << sum << "\n";
    return 0;
}
```

Berkas pragma_off_demo.cpp yang dipakai sebagai pembanding identik byte demi byte kecuali baris #pragma yang tidak ada. Keduanya dikompilasi dengan g++ -pedantic -Wall biasa (tanpa flag -O sama sekali), dan keduanya dijalankan serta diukur waktunya di mesin yang sama secara berurutan. Lampiran 5.A menunjukkan transkrip lengkapnya; versi berpragma selesai kurang lebih delapan kali lebih cepat, dengan kedua versi menghasilkan jawaban numerik yang persis sama (Gambar 5.3 dan 5.4).

> **Mengapa loop khusus ini menunjukkan percepatan sebesar itu?**
>
> Loop aritmetika ketat atas rentang yang sangat besar persis bentuk kode yang paling dihargai -O3 dan unroll loop: kompiler bisa menyimpan nilai di register alih-alih memori, membuka gulungan loop untuk mengurangi overhead per-iterasi, dan sering meng-auto-vektorisasi aritmetika untuk memproses beberapa iterasi sekaligus. Program yang didominasi, katakanlah, oleh tunggu disk atau jaringan akan menunjukkan peningkatan yang jauh lebih kecil — besarnya kemenangan selalu bergantung pada apa yang sebenarnya dihabiskan waktu kode itu untuk melakukannya.

5.7 Ringkasan Bab

- Baris #pragma yang tertanam dalam kode sumber menginstruksikan kompiler secara langsung, terlepas dari flag apa pun yang dipakai atau tidak dipakai skrip build judge sendiri.
- Pragma tingkat-optimisasi — optimize("O2"), optimize("O3,unroll-loops") — aman dan portabel: mereka mengubah seberapa agresif kode disusun ulang, bukan instruksi CPU apa yang diasumsikan ada.
- Pragma target-CPU — target("avx2,bmi,bmi2,popcnt") dan serupa — menghasilkan instruksi dari ekstensi instruction-set tertentu, dan crash dengan Runtime Error illegal-instruction di CPU mana pun yang tidak memilikinya.
- Risiko portabilitas ini tidak bisa diuji di komputer pengembangan Anda sendiri — itu hanya muncul di perangkat keras yang tidak Anda kendalikan, saat submission.
- Benchmark yang membandingkan kode identik dengan dan tanpa pragma optimisasi menunjukkan percepatan sekitar delapan kali dari #pragma GCC optimize("O3,unroll-loops") saja, tanpa perubahan apa pun pada algoritma.

Gambar 5.3 — Pengingat bahwa kecerdikan yang tidak bisa diuji adalah jebakan debugging yang menunggu terjadi.

Gambar 5.4 — Peta jalan 14 bab buku ini. Kita di sini: Bab 5.

Pertanyaan Tinjauan (Tidak Dinilai)

1. Jelaskan, dengan kata-kata Anda sendiri, perbedaan antara apa yang diubah #pragma GCC optimize("O3") dan apa yang diubah #pragma GCC target("avx2"), dan mengapa hanya salah satunya yang bisa menyebabkan Runtime Error pada program yang sebenarnya benar.
2. Batasan sebuah soal cukup kecil sehingga solusi Anda sudah berjalan jauh di bawah sepersepuluh batas waktu tanpa pragma apa pun. Menggunakan tangga risiko Gambar 5.2, jelaskan mengapa menambahkan pragma target(...) di sini adalah keputusan buruk meski mungkin membuat program lebih cepat.
3. Jelaskan mengapa risiko portabilitas pragma target-CPU tidak bisa ditemukan dengan menguji program di komputer Anda sendiri sebelum mengirimkannya.
4. Sebuah soal membutuhkan perbandingan floating-point eksak (misalnya, memeriksa apakah nilai yang dihitung sama persis dengan 0.5). Menggunakan Tabel 5.1, jelaskan mengapa #pragma GCC optimize("Ofast") lebih berisiko di sini dibanding #pragma GCC optimize("O3,unroll-loops").
5. Menggunakan metodologi benchmark dari Bagian 5.5 (dua berkas yang identik kecuali satu memakai pragma, keduanya dikompilasi dengan perintah biasa yang sama), jelaskan bagaimana Anda akan menguji apakah #pragma GCC optimize("O2") saja sudah memberikan sebagian besar manfaat yang diberikan "O3,unroll-loops", tanpa menulis algoritma baru apa pun.

Lampiran 5.A — Log Verifikasi

Program pragma_off_demo.cpp dan pragma_on_demo.cpp dari Bagian 5.5 — identik kecuali satu baris #pragma — keduanya dikompilasi dengan g++ (-pedantic -Wall, tanpa flag -O sama sekali di command line) dan diukur waktunya secara berurutan di mesin yang sama.

Transkrip terminal

```
$ g++ -pedantic -Wall -o pragma_off_demo pragma_off_demo.cpp
$ g++ -pedantic -Wall -o pragma_on_demo pragma_on_demo.cpp
$ time ./pragma_off_demo
sum of i*i for i in [0, 2000000000): 10262176583330622976
real    0m5.457s
$ time ./pragma_on_demo
sum of i*i for i in [0, 2000000000): 10262176583330622976
real    0m0.645s
```

Kedua versi menghasilkan hasil numerik yang persis sama (10262176583330622976), mengonfirmasi pragma hanya mengubah performa, bukan kebenaran. Versi berpragma selesai kurang lebih 8,5 kali lebih cepat (0,645 detik berbanding 5,457 detik) tanpa perubahan apa pun pada algoritma atau perintah yang dipakai untuk mengompilasinya — seluruh perbedaan bisa diatribusikan ke satu baris #pragma yang ditambahkan.

Referensi

[1] Sphere Online Judge (SPOJ). https://www.spoj.com — pola penggunaan pragma yang dibahas dalam bab ini mencerminkan praktik standar yang teramati di seluruh kumpulan soal SPOJ.

[2] GNU Project. Dokumentasi GCC Function Attributes and Pragmas. https://gcc.gnu.org/onlinedocs/gcc/ — acuan otoritatif untuk semantik pragma optimize() dan target().

[3] Kernighan, B. W., & Plauger, P. J. (1978). The Elements of Programming Style (2nd ed.). McGraw-Hill — asal peringatan debugging-versus-kecerdikan yang dikutip di Bagian 5.7.

BAB 6

Pilihan Memori dan Struktur Data di Bawah Anggaran Byte

Batas memori judge sama mengikatnya dengan batas waktunya, dan struktur data yang Anda pakai karena kebiasaan sering bukan yang benar-benar mampu ditanggung batasan soal. Bab ini membahas memilih struktur secara sengaja, dan bug overflow diam-diam yang menjebak hampir semua pemula setidaknya sekali.

Tujuan Pembelajaran — setelah bab ini Anda mampu:

(1) Menjelaskan mengapa batas memori membatasi pilihan struktur data Anda seketat batas waktu membatasi pilihan algoritma Anda. (2) Membandingkan biaya memori struktur umum — array int dan long long, vector<bool>/bitset<N>, dan std::string — dan tahu kapan masing-masing tepat dipakai. (3) Mengenali ekspresi yang nilai terbesar mungkinnya diam-diam melebihi int 32-bit, sebelum bug itu sampai ke judge. (4) Memilih buffer char dibanding std::string ketika penyimpanan teks mentah adalah kebutuhan sesungguhnya. (5) Mengimplementasikan dan memverifikasi segmented sieve, teknik standar untuk mencari bilangan prima di suatu rentang yang batas atasnya terlalu besar untuk dialokasikan langsung.

6.1 Mengapa Batas Memori Sama Pentingnya dengan Batas Waktu

Bab 3 dan 4 sama-sama memperlakukan batas waktu sebagai sumber daya mengikat untuk dianggarkan. Batas memori judge — umumnya di antara 32 dan 256 megabyte di SPOJ — sama persis mengikatnya, dan mudah dilupakan sepenuhnya sampai sebuah submission mengembalikan Memory Limit Exceeded (MLE) pada test case yang ukuran inputnya tampak sepenuhnya wajar.

Kesalahan yang menghasilkan hasil ini hampir selalu sama: mengalokasikan struktur seukuran batas atas batasan itu sendiri, bukan sesuai apa yang sebenarnya dibutuhkan soal. Soal yang menyatakan n hingga 10^9 biasanya tidak berarti Anda butuh array 10^9 elemen — itu berarti Anda perlu mencari cara menyelesaikan soal tanpa pernah mewujudkan seluruh 10^9 elemen itu sekaligus dalam memori. Contoh terselesaikan Bagian 6.4 dibangun sepenuhnya di sekitar perbedaan ini (Tabel 6.1 dan Gambar 6.1).

"n bisa mencapai 10^9" adalah batasan pada jawaban, belum tentu pada ukuran struktur data Anda.

Kebiasaan paling berharga yang diajarkan bab ini adalah bertanya, untuk batas besar mana pun pada batasan soal, apakah ada algoritma yang menyentuh besaran itu tanpa pernah menyimpan seluruhnya sekaligus dalam memori. Bagian 6.4 menyelesaikan persis pertanyaan ini untuk sebuah kasus klasik.

6.2 Array Statis vs. Kontainer STL: Overhead yang Anda Bayar

Tabel 6.1 membandingkan biaya memori per elemen struktur yang paling sering dipakai competitive programmer. Perbedaannya tampak kecil per elemen, tetapi pada skala yang secara rutin dituntut

batasan SPOJ — array 10^5 hingga 10^7 elemen adalah hal umum — perbedaan biaya per elemen empat kali atau delapan kali adalah perbedaan antara muat dengan nyaman di dalam batas memori dan gagal total.

Biaya Memori Pilihan Struktur Data yang Umum

Setiap struktur menukar memori dengan hal lain. Pilih yang termurah yang tetap memenuhi kebutuhan soal.

Struktur	Perkiraan memori per elemen	Kapan dipakai	Jebakan
Array int	4 byte	Hitungan, indeks, atau nilai yang dijamin muat sekitar ±2,1×10^9	Produk atau jumlah nilai-nilai ini bisa diam-diam overflow (Bagian 6.3)
Array long long	8 byte	Jumlah, produk, atau nilai apa pun yang mungkin melebihi sekitar 2×10^9	Menggandakan jejak memori dibanding int — pakai hanya jika benar-benar dibutuhkan
vector<bool> / bitset<N>	1 bit (sekitar 1/8 byte)	Flag sieve, penanda kunjungan, atau array benar/salah apa pun atas rentang sangat besar	vector<bool> bukan container STL sesungguhnya — tidak ada referensi atau pointer asli ke elemen
std::string	Sekitar 32 byte overhead plus buffer heap, per string	Teks panjang-variabel yang butuh operasi string sesungguhnya	Array berisi banyak string pendek bisa memboroskan memori jauh lebih banyak dibanding satu buffer char datar

Gambar 6.1 — Biaya memori pilihan struktur data yang umum, dan untuk apa masing-masing layak dipakai.

Baris vector<bool>/bitset<N> layak mendapat perhatian khusus, karena ia sekaligus trik penghemat memori terbesar bab ini dan jebakan kebenaran yang terkenal. Kedua struktur mengemas satu boolean per bit, bukan per byte — penghematan delapan kali dibanding array char dengan panjang sama — yang sangat penting untuk algoritma bergaya sieve yang butuh flag benar/salah per bilangan bulat di seluruh rentang yang sangat besar. Jebakannya: vector<bool> dikhususkan di dalam pustaka standar C++ untuk memakai representasi bit-packed ini, yang berarti ia tidak berperilaku seperti vector biasa — kode yang mengambil referensi atau pointer ke salah satu elemennya, mengharapkan semantik vector biasa, tidak akan terkompilasi atau tidak akan melakukan apa yang tampaknya seharusnya dilakukan.

6.3 Kapan "long long" Diam-Diam Tidak Cukup — dan Kapan int Diam-Diam Cukup

Sebuah int 32-bit menampung nilai hanya hingga sekitar 2,1×10^9 dalam besaran. Ini terdengar seperti angka besar, dan untuk satu nilai tunggal yang menaati batasan yang dinyatakan soal, biasanya memang begitu. Bug yang menjebak hampir semua pemula setidaknya sekali berbeda: bukan nilai itu sendiri yang overflow, melainkan produk atau jumlah antara dari dua nilai dalam-rentang yang overflow, secara diam-diam, tanpa peringatan apa pun dari kompiler dan tanpa crash saat runtime — hanya jawaban salah yang tampak seperti bug logika alih-alih bug aritmetika.

Checklist Overflow

Sebelum menulis sebuah ekspresi, periksa apakah nilai terbesar yang mungkin masih muat dalam int 32-bit (sekitar ±2,1×10^9).

Situasi	Nilai terbesar tipikal	Muat di int 32-bit?	Perbaikan
Mengalikan dua nilai yang masing-masing hingga 10^5	Sekitar 10^10	Tidak	Cast setidaknya satu operand ke long long sebelum mengalikan
Menjumlahkan hingga 10^5 nilai, masing-masing hingga 10^9	Sekitar 10^14	Tidak	Pakai akumulator long long sejak awal
Satu indeks array atau hitungan di bawah 10^4	Sekitar 10^8 atau kurang	Ya	int sudah cukup — tidak perlu diubah
Produk kombinatorika modular, sebelum diambil % p	Bisa sangat besar sebelum direduksi	Tidak	Pakai long long, dan ambil % p setelah setiap satu kali perkalian

Gambar 6.2 — Checklist overflow: situasi umum, apakah muat di int 32-bit, dan perbaikannya.

Contoh terselesaikan di bawah ini membuat hal ini konkret: mengalikan dua nilai int biasa dalam-rentang memakai aritmetika int diam-diam menghasilkan hasil yang salah, sementara perkalian yang sama dengan satu operand di-cast ke long long terlebih dahulu menghasilkan hasil yang benar.

overflow_demo.cpp — overflow 32-bit yang diam-diam (templat orisinal)

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n = 100000;
    int wrongProduct = n * n;
    long long rightProduct = (long long)n * n;

    cout << "n = " << n << "\n";
    cout << "n * n as int:       " << wrongProduct << "\n";
    cout << "n * n as long long: " << rightProduct << "\n";
    return 0;
}
```

Meng-cast hasil, setelah perkalian sudah terjadi, tidak memperbaiki apa pun.

(long long)(n * n) tetap menghitung n * n sepenuhnya dalam aritmetika int 32-bit terlebih dahulu, overflow persis dengan cara yang sama, dan baru kemudian mengonversi hasil yang sudah salah ke long long. Cast harus datang sebelum perkalian — pada setidaknya satu dari dua operand — agar seluruh ekspresi dievaluasi dalam aritmetika 64-bit.

6.4 Contoh yang Diselesaikan: Segmented Sieve

Contoh terselesaikan Bab 1 membangun Sieve of Eratosthenes biasa dan menyebut PRIME1 — soal SPOJ klasik yang meminta semua bilangan prima dalam rentang [m, n] — sebagai kasus yang batasan sesungguhnya (n hingga 10^9) tidak bisa ditangani sieve sederhana itu secara langsung: array 10^9 boolean, bahkan yang bit-packed, akan butuh jauh lebih dari 100 megabyte hanya untuk flag-nya, dan array int biasa akan butuh sekitar 4 gigabyte, jauh melampaui batas memori judge mana pun yang

realistis. Perbaikannya, sudah dipratinjau di sana dan dibangun lengkap di sini, adalah segmented sieve.

Idenya punya dua bagian. Pertama, hitung lebih dulu setiap bilangan prima hingga sqrt(n terbesar yang akan pernah Anda tanyakan) — untuk n hingga 10^9, batas itu hanya sekitar 31.623, daftar yang sangat kecil. Kedua, untuk rentang [lo, hi] yang diminta mana pun, alokasikan array boolean berukuran hanya hi-lo+1 (yang batasan PRIME1 SPOJ sendiri batasi hingga 100.000), dan pakai daftar bilangan prima kecil untuk menandai bilangan komposit hanya di dalam jendela itu — tanpa pernah menyentuh array seukuran hi itu sendiri.

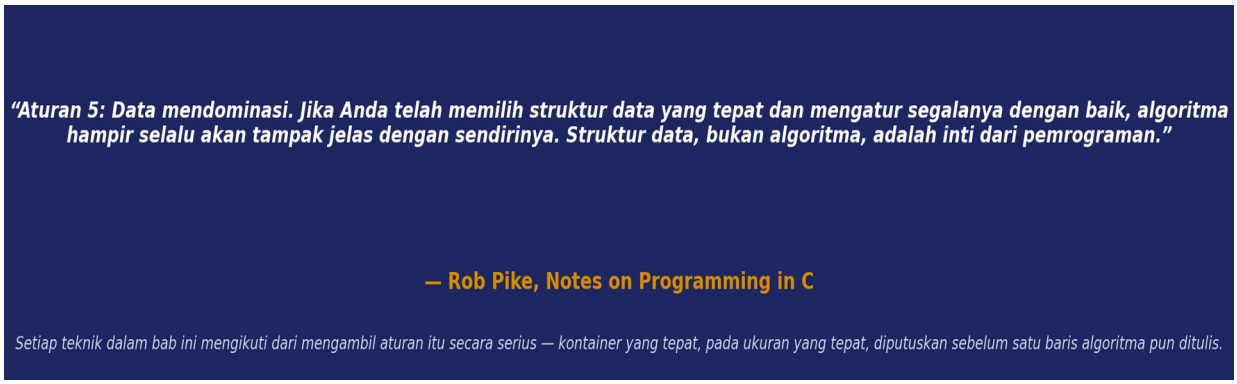
segmented_sieve_demo.cpp — bilangan prima di [lo, hi] tanpa alokasi hi elemen (templat orisinal)

```
vector<long long> smallPrimes;

void sieveSmallPrimes(long long limit) {
    vector<bool> isComposite(limit + 1, false);
    for (long long i = 2; i * i <= limit; i++) {
        if (!isComposite[i]) {
            for (long long j = i*i; j <= limit; j += i)
                isComposite[j] = true;
        }
    }
    for (long long i = 2; i <= limit; i++)
        if (!isComposite[i]) smallPrimes.push_back(i);
}

vector<long long> segmentedSieve(long long lo, long long hi) {
    vector<bool> isComposite(hi - lo + 1, false);
    for (long long p : smallPrimes) {
        if (p * p > hi) break;
        long long start = max(p*p, ((lo+p-1)/p) * p);
        for (long long j = start; j <= hi; j += p)
            isComposite[j - lo] = true;
    }
    vector<long long> result;
    for (long long i = max(lo, 2LL); i <= hi; i++)
        if (!isComposite[i - lo]) result.push_back(i);
    return result;
}
```

Templat ini — demonstrasi orisinal, tidak terkait solusi accepted soal SPOJ mana pun — diverifikasi terhadap pemeriksaan primalitas brute-force yang lambat di dua rentang terpisah, termasuk segmen berukuran penuh selebar 100.001 dekat 900 juta, yang selesai jauh di bawah satu milidetik (Lampiran 6.A). PRIME1 sendiri dikutip di sini hanya dengan nama publik soalnya, sebagai contoh nyata dan bernama dari bentuk batasan yang menjadi dasar teknik ini — tidak pernah dengan solusi accepted-nya (Gambar 6.3 dan 6.4).

6.6 Ringkasan Bab

- Batas memori judge sama mengikatnya dengan batas waktu — mengalokasikan struktur seukuran batas atas batasan, bukan sesuai apa yang benar-benar dibutuhkan soal, adalah cara paling umum melanggarnya.
- int (4 byte), long long (8 byte), vector<bool>/bitset<N> (1 bit), dan std::string (variabel, dengan overhead nyata) masing-masing menukar memori dengan kemampuan berbeda — pilih yang termurah yang tetap memenuhi tugasnya.
- vector<bool> dikemas per bit demi efisiensi memori, tetapi bukan kontainer STL sesungguhnya — kode yang mengharapkan referensi biasa ke salah satu elemennya akan berperilaku salah.
- Produk atau jumlah dua nilai yang masing-masing muat di int 32-bit tetap bisa overflow — cast setidaknya satu operand ke long long sebelum operasi, bukan sesudahnya.
- Segmented sieve menemukan semua bilangan prima di [lo, hi] hanya memakai memori O(sqrt(hi)) plus O(hi-lo), tidak pernah array seukuran hi itu sendiri — teknik standar untuk batasan bergaya PRIME1.



Gambar 6.3 — Pengingat klasik Rob Pike bahwa struktur data datang lebih dulu.

Gambar 6.4 — Peta jalan 14 bab buku ini. Kita di sini: Bab 6.

Pertanyaan Tinjauan (Tidak Dinilai)

1. Batasan sebuah soal menyatakan 1 ≤ n ≤ 10^9 tetapi hanya meminta satu hitungan sebagai jawaban, bukan daftar n item. Jelaskan mengapa ini belum tentu berarti Anda butuh array seukuran n, menggunakan ide segmented sieve dari Bagian 6.4 sebagai panduan.
2. Jelaskan, dengan kata-kata Anda sendiri, mengapa vector<bool> menghemat memori dibanding vector<char>, dan pola pengkodean spesifik apa yang rusak karena cara penghematan memori itu diimplementasikan.
3. Sebuah program menghitung total = a + b di mana a dan b masing-masing dibaca sebagai int dan bisa sebesar 10^9. Menggunakan Gambar 6.2, jelaskan apakah jumlah ini bisa overflow int 32-bit, dan jika ya, bagaimana Anda akan memperbaiki deklarasi total.
4. Jelaskan mengapa (long long)(n * n) tidak memperbaiki bug overflow yang didemonstrasikan di Bagian 6.3, dan tunjukkan versi ekspresi yang telah diperbaiki.
5. Menggunakan fungsi segmentedSieve dari Bagian 6.4, jelaskan dengan kata-kata Anda sendiri mengapa titik awal loop bagian dalam dihitung sebagai max(p*p, ((lo+p-1)/p) * p) alih-alih sekadar dimulai dari lo.

Lampiran 6.A — Log Verifikasi

Program overflow_demo.cpp dan segmented_sieve_demo.cpp dari Bagian 6.3 dan 6.4 dikompilasi dengan g++ (-O2 -Wall -pedantic). Hasil long long dari demo overflow diperiksa terhadap perhitungan Python langsung; output segmented sieve diperiksa terhadap uji primalitas brute-force yang lambat atas rentang yang sama, termasuk segmen berukuran penuh selebar 100.001 dekat 900 juta untuk mengonfirmasi kecepatan teknik ini pada batasan berskala PRIME1.

Transkrip terminal

```
$ g++ -O2 -pedantic -Wall -o overflow_demo overflow_demo.cpp
$ ./overflow_demo
n = 100000
n * n as int:       1410065408
n * n as long long: 10000000000
$ python3 -c "print(100000*100000)"
10000000000
$ g++ -O2 -pedantic -Wall -o segmented_sieve_demo segmented_sieve_demo.cpp
$ ./segmented_sieve_demo
primes in [999999900, 1000000000]: 999999929 999999937
count=2
$ # pemeriksaan brute-force (trial division lambat) atas rentang sama:
$ # [999999929, 999999937], count=2 -- cocok persis
$ # segmen berukuran penuh selebar 100.001 dekat 900 juta:
primes in [900000000,900100000]: count=4854
time_ms=0.78
```

Hasil int yang overflow (1410065408) dan hasil long long yang benar (10000000000, cocok dengan perhitungan independen Python) mengonfirmasi bug dan perbaikannya persis seperti yang dijelaskan. Output segmented sieve cocok persis dengan pemeriksaan brute-force independen pada kedua rentang yang diuji, dan memproses segmen berukuran penuh selebar 100.001 dalam waktu di bawah satu milidetik — mengonfirmasi teknik ini dengan mudah muat di dalam batas waktu masuk akal mana pun pada batasan berskala PRIME1.

Referensi

[1] Sphere Online Judge (SPOJ). PRIME1 — Prime Generator. https://www.spoj.com/problems/PRIME1/ — dikutip di sini hanya dengan nama publik soal, sebagai contoh nyata bentuk batasan yang diatasi teknik segmented-sieve bab ini; tidak ada solusi accepted yang ditampilkan atau dirujuk.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — untuk Sieve of Eratosthenes dan variannya yang segmented.

[3] Pike, R. (1989). Notes on Programming in C. https://www.lysator.liu.se/c/pikestyle.html — asal prinsip struktur-data-dahulu yang dikutip di Bagian 6.6.

BAB 7

Jebakan Implementasi Klasik Penyebab WA

Setiap bab sebelumnya mengasumsikan logika algoritma Anda sudah benar. Bab ini membahas empat cara paling umum bagaimana algoritma yang benar sekalipun tetap berubah menjadi Wrong Answer — batas loop off-by-one, state yang diam-diam bocor antar test case, input yang tidak terlihat seperti yang Anda asumsikan, dan kesalahan aritmetika ASCII yang halus.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Mengenali bug batas loop off-by-one dengan pertama-tama menuliskan batas rentang dalam kata-kata, lalu mencocokkan kondisi loop dengan kalimat itu. (2) Menjelaskan mengapa variabel global atau static yang tidak di-reset bisa membuat program benar pada test case 1 dan salah pada setiap test case sesudahnya, serta memperbaikinya dengan membatasi cakupan variabel secara lokal. (3) Menyebutkan kejutan parsing input — whitespace di akhir, line ending gaya Windows, baris kosong tak terduga, jumlah test case yang tidak dinyatakan — yang gagal secara diam-diam di komputer sendiri dan baru muncul saat melawan input persis milik juri. (4) Menerapkan shortcut aritmetika ASCII (mengurangi karakter basis untuk mendapatkan nilai numerik digit atau huruf) dengan benar, lengkap dengan guard check yang membuatnya aman. (5) Menelusuri sebuah studi kasus yang mengaitkan keempat jebakan tersebut ke satu program kecil yang berdiri sendiri.

7.1 Batas Loop Off-by-One

Kesalahan off-by-one adalah loop yang batasnya meleset tepat satu elemen — biasanya karena rentang yang dimaksud tidak pernah dituliskan dalam bahasa biasa sebelum kondisi loop dipilih. "Semua bilangan bulat dari L sampai R" bersifat ambigu sampai Anda memutuskan apakah R sendiri termasuk; kode harus persis cocok dengan keputusan itu, dan sangat mudah untuk menulis i < R karena kebiasaan padahal rentangnya dimaksudkan mencakup R.

Bug ini berbahaya justru karena tidak terlihat pada sebagian besar data uji. Loop yang salah mengecualikan batas atas tetap menghasilkan jawaban benar setiap kali elemen yang dikecualikan itu tidak akan mengubah hasil — bug ini hanya muncul pada input tertentu di mana elemen batas itu penting, dan itu persis jenis kasus tepi yang dirancang untuk disertakan oleh juri dan justru terlewat oleh pengujian manual cepat Anda sendiri.

Sebelum menulis loop, tuliskan rentangnya dalam kata-kata.

"Setiap bilangan bulat dari L sampai R, keduanya termasuk" memaksa kondisi menjadi i <= R. "Setiap bilangan bulat dari L hingga tapi tidak termasuk R" memaksa i < R. Memutuskan kalimatnya lebih dulu, lalu mencocokkan kode dengannya, menangkap ketidakcocokan sebelum sempat menjadi bug.

Contoh di bawah menghitung berapa banyak bilangan bulat dalam suatu rentang yang habis dibagi k, baik dengan cara yang salah maupun dua cara yang benar — sebuah loop dengan batas yang tepat, dan hitungan bentuk tertutup yang sama sekali tidak butuh loop.

off_by_one_demo.cpp — bug off-by-one dan dua perbaikan yang benar (template orisinal)

```
long long countBuggy(long long L, long long R, long long k) {
    long long count = 0;
    for (long long i = L; i < R; i++) {  // off-by-one: seharusnya i <= R
        if (i % k == 0) count++;
    }
    return count;
}

long long countCorrectLoop(long long L, long long R, long long k) {
    long long count = 0;
    for (long long i = L; i <= R; i++) {
        if (i % k == 0) count++;
    }
    return count;
}

long long countCorrectFormula(long long L, long long R, long long k) {
    return R / k - (L - 1) / k;
}
```

Pada rentang [1, 10] dengan k = 5, versi buggy mengembalikan 1 (secara diam-diam menghilangkan 10, yang habis dibagi 5 dan sama dengan R) sementara kedua versi yang benar mengembalikan 2. Pada rentang yang kebetulan tidak berakhir di kelipatan k, ketiga versi sepakat secara kebetulan — dan justru itulah sebabnya bug ini begitu sering lolos pengecekan manual cepat dan baru gagal setelah sampai ke juri.

7.2 Static dan Global yang Tak Diinisialisasi Ulang Antar Test Case

Kebanyakan soal SPOJ mengemas banyak test case independen dalam satu file input, dan mengharapkan program Anda memproses semuanya dalam satu kali jalan, bukan dipanggil sekali per kasus. Bentuk ini menciptakan keluarga bug yang tak ada hubungannya dengan logika yang salah: variabel global atau static yang diam-diam mempertahankan nilainya dari satu test case ke test case berikutnya, karena tidak ada apa pun dalam kode yang me-reset-nya.

Gejalanya khas dan layak dikenali sekilas: test case pertama dalam output benar, dan setiap test case sesudahnya salah, biasanya dengan jumlah yang terlihat seperti terus terakumulasi. Pola itu — benar sekali, lalu salah dengan cara yang bertambah — hampir menjadi tanda tangan khas bug ini.

Variabel yang dideklarasikan di luar setiap fungsi tidak otomatis di-reset saat test case baru dimulai.

C++ memberi variabel file-scope dan static satu siklus hidup untuk seluruh masa jalan program, bukan satu siklus hidup per test case. Jika logika sebuah solusi secara implisit mengasumsikan "variabel ini mulai dari nol untuk test case ini," asumsi itu harus dituliskan secara eksplisit dalam kode — baik dengan me-reset variabel di awal setiap test case, atau memindahkannya ke dalam fungsi sehingga salinan baru otomatis dibuat pada setiap pemanggilan.

static_state_leak_demo.cpp — global yang bocor antar test case, dan perbaikannya (template orisinal)

```
long long total = 0;   // hidup sepanjang setiap pemanggilan: inilah bug-nya

long long solveBuggy(const vector<int>& a) {
    for (int x : a) total += x;   // tidak pernah di-reset ke 0 di awal
    return total;
}

long long solveFixed(const vector<int>& a) {
    long long localTotal = 0;     // segar, terbatas cakupan pemanggilan ini
    for (int x : a) localTotal += x;
    return localTotal;
}
```

Dijalankan atas tiga test case yang jumlah benarnya 15, 10, dan 14, versi buggy mencetak 15, 25, dan 39 — setiap jawaban berikutnya adalah jumlah yang benar ditambah semua yang bocor dari sebelumnya — sementara versi yang diperbaiki, memakai akumulator lokal segar pada setiap pemanggilan, mencetak 15, 10, dan 14 yang benar (Lampiran 7.A).

7.3 Kejutan Parsing Input

Sebuah program bisa memiliki logika algoritmik yang sepenuhnya benar dan tetap gagal, karena asumsi yang dibuatnya tentang bentuk persis input tidak cocok dengan file input sesungguhnya milik juri. Bug seperti ini sangat menjengkelkan karena biasanya tak terlihat di komputer sendiri: file uji Anda sendiri diketik oleh Anda, dalam format yang Anda harapkan, sedangkan milik juri tidak dijamin demikian (Gambar 7.1).

Jebakan Parsing Input yang Tak Ada Hubungannya dengan Logika

Ini gagal secara diam-diam di komputer sendiri dan baru muncul saat melawan format input persis milik juri.

Situasi	Apa yang Salah	Perbaikan
Membandingkan string yang dibaca dengan literal yang diharapkan	'\n' atau spasi di akhir ikut menjadi bagian string, sehingga perbandingan selalu gagal	Trim whitespace dari kedua ujung sebelum membandingkan, atau pakai >> (yang otomatis melewatinya)
Line ending CRLF (\r\n) gaya Windows pada file test lokal	Karakter '\r' liar diam-diam menempel pada token terakhir di baris itu	Buang '\r' di akhir setelah getline, atau hindari getline untuk input token sederhana
Baris kosong tambahan tak terduga di akhir input	Loop yang ditulis untuk membaca jumlah baris pasti membaca satu terlalu banyak, atau crash	Loop berdasarkan status stream (while (cin >> x)) alih-alih jumlah baris tetap bila memungkinkan
Banyak test case tanpa jumlah T eksplisit	Program berhenti setelah satu test case, atau menggantung menunggu input yang tak pernah datang	Baca dengan while (scanf("...", ...) != EOF) atau padanan cin-nya, dan loop sampai stream benar-benar berakhir

Gambar 7.1 — Empat situasi parsing input yang tak ada hubungannya dengan logika algoritma Anda.

Dua di antaranya layak mendapat penekanan khusus karena spesifik terhadap perpindahan antar sistem operasi. File teks gaya Windows mengakhiri setiap baris dengan dua karakter \r\n, bukan satu karakter \n yang diharapkan juri bergaya Linux; jika Anda menyiapkan atau mengedit file uji di Windows lalu membacanya dengan getline pada juri bergaya Linux, karakter '\r' liar diam-diam

menempel di akhir token terakhir yang dibaca pada baris itu, merusak persis token itu dan tidak yang lain — bug yang mudah disalahartikan sebagai kesalahan logika yang sama sekali tak terkait. Dan pola yang umum dan aman untuk situasi jumlah-test-case adalah melakukan loop berdasarkan status stream input itu sendiri — while (cin >> x) atau while (scanf(...) != EOF) — daripada mempercayai jumlah tetap, karena pola ini menangani baik jumlah yang hilang maupun baris kosong tambahan yang tak terduga dengan cara yang sama.

7.4 Studi Kasus: Shortcut Aritmetika ASCII

Banyak soal SPOJ memberi Anda sebuah bilangan sebagai teks biasa — kadang ratusan digit panjangnya, jauh terlalu besar untuk tipe integer bawaan apa pun. Shortcut standar untuk mengolah bilangan seperti itu satu karakter pada satu waktu bergantung pada fakta tentang bagaimana karakter dikodekan: karakter digit '0' hingga '9' disimpan sebagai sepuluh kode numerik berurutan dalam ASCII, sehingga mengurangkan karakter '0' dari karakter digit mana pun langsung mengonversinya ke nilai numeriknya, tanpa perlu library parsing sama sekali.

Trik yang sama berlaku umum untuk huruf — c - 'a' mengonversi huruf kecil menjadi indeks 0 hingga 25, berguna untuk penghitungan frekuensi seukuran alfabet atau rotasi bergaya Caesar — tetapi ini hanya aman ketika karakter yang dikonversi memang dijamin berupa digit atau huruf sejak awal.

> **c - '0' pada karakter yang bukan digit tidak menimbulkan error — ia diam-diam mengembalikan angka yang tak bermakna.**
>
> Jika spasi liar, newline, atau tanda baca pernah sampai ke pengurangan ini tanpa diperiksa, hasilnya bukan crash — melainkan bilangan salah yang terlihat masuk akal dan diam-diam merusak jumlah atau hitungan digit. Selalu jaga pengurangan dengan pengecekan rentang eksplisit, seperti (c >= '0' && c <= '9'), sebelum mempercayai hasilnya.

Studi kasus di bawah menerapkan shortcut ini dengan tiga cara: menjumlahkan setiap digit dalam sebuah string, mereduksi jumlah itu berulang kali menjadi digital root satu digit, dan membaca hanya karakter terakhir dari bilangan raksasa untuk menguji paritasnya — tak satu pun perlu mengonversi seluruh string menjadi integer, yang penting karena bilangan 200 digit tak akan muat dalam satu tipe apa pun.

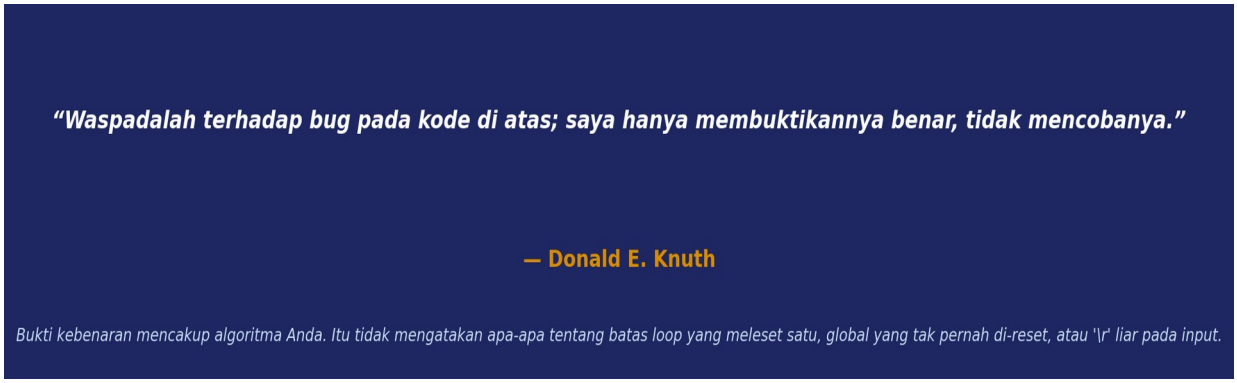
ascii_digit_sum_demo.cpp — jumlah digit, digital root, dan paritas digit terakhir (template orisinal)

```
long long digitSum(const string& s) {
    long long sum = 0;
    for (char c : s) {
        if (c >= '0' && c <= '9') {      // guard sebelum mengurangkan
            sum += (c - '0');             // shortcut aritmetika ASCII
        }
    }
    return sum;
}

int digitalRoot(string s) {
    long long sum = digitSum(s);
    while (sum >= 10) {
        s = to_string(sum);
        sum = digitSum(s);
    }
    return (int)sum;
}

bool lastDigitIsEven(const string& s) {
    int lastDigit = s.back() - '0';       // aritmetika ASCII lagi
    return (lastDigit % 2) == 0;
}
```

Pada bilangan 17 digit 97531598642013579, jumlah digitnya adalah 84 dan digital root-nya 3; pada bilangan 200 digit yang dikonstruksi — jauh lebih besar dari tipe integer bawaan mana pun yang bisa menampungnya — fungsi digitSum yang sama tetap langsung mengembalikan jawaban benar (900), karena ia sama sekali tak pernah melihat besaran bilangannya, hanya tiap karakter satu per satu (Lampiran 7.A) (Gambar 7.2 dan 7.3).

7.6 Ringkasan Bab

- Bug off-by-one berasal dari batas loop yang tidak cocok dengan inklusivitas rentang yang dimaksud — tuliskan rentangnya dalam kata-kata dulu, lalu cocokkan kondisi loop dengan kalimat itu.
- Variabel global atau static yang tak pernah di-reset diam-diam membawa nilainya dari satu test case ke test case berikutnya; perbaikannya adalah reset eksplisit di awal setiap test case, atau membatasi cakupan variabel secara lokal sehingga salinan baru otomatis dibuat.
- Whitespace di akhir, line ending CRLF gaya Windows, baris kosong tak terduga, dan jumlah test case yang tak dinyatakan adalah kejutan parsing input yang gagal secara tak terlihat di komputer sendiri dan baru muncul melawan input persis milik juri.
- Shortcut aritmetika ASCII (c - '0', c - 'a') mengonversi karakter ke nilai numeriknya tanpa perlu library, tapi hanya setelah guard check eksplisit memastikan karakter itu memang digit atau huruf.
- Keempat jebakan berbagi satu sifat: tak satu pun menunjukkan algoritma yang salah. Bukti kebenaran logika Anda tak mengatakan apa-apa tentang salah satu dari mereka.



Gambar 7.2 — Pengingat klasik Knuth: bukti kebenaran tidak mencakup off-by-one liar, global yang tak pernah di-reset, atau '\r' liar.

Gambar 7.3 — Peta jalan 14 bab buku ini. Anda di sini: Bab 7.

Pertanyaan Tinjauan (Tidak Dinilai)

1. Tuliskan, dalam satu kalimat, rentang yang sebenarnya dicakup oleh loop for (int i = L; i < R; i++). Apakah R termasuk? Tuliskan ulang kalimat dan kondisi loopnya agar keduanya sepakat, untuk rentang yang seharusnya mencakup R.
2. Sebuah program memproses T test case dalam satu kali jalan dan memakai vector<bool> seen(100000) global untuk menandai indeks yang sudah dikunjungi. Jelaskan, memakai pola Bagian 7.2, apa yang akan salah pada test case 2 jika vector itu tak pernah dibersihkan, dan jelaskan perbaikannya.
3. Anda menyiapkan file input uji lokal di komputer Windows dan mengujinya di sana sebelum submit ke juri berbasis Linux. Memakai Bagian 7.3, jelaskan apa yang bisa salah pada baris seperti cin.getline(buf, 100); std::string s(buf); if (s == "YES") ..., dan bagaimana Anda memperbaikinya.
4. Jelaskan mengapa pengecekan (c >= '0' && c <= '9') harus dilakukan sebelum c - '0', bukan sesudahnya, dan jelaskan input konkret yang akan menghasilkan hasil salah (tapi tak crash) jika pengecekan itu dilewati.
5. Memakai fungsi digitSum dari Bagian 7.4, jelaskan mengapa fungsi itu bisa dengan benar memproses bilangan 300 digit padahal tak ada tipe integer bawaan C++ yang bisa menampung nilai sebesar itu.

Lampiran 7.A — Log Verifikasi

Ketiga program dalam bab ini dikompilasi dengan g++ (-O2 -Wall -pedantic) dan hasilnya diverifikasi secara independen, baik terhadap perhitungan Python atas rentang yang sama maupun terhadap nilai yang dihitung manual.

Transkrip terminal

```
$ g++ -O2 -pedantic -Wall -o off_by_one_demo off_by_one_demo.cpp
$ ./off_by_one_demo
Range [1, 10], divisible by 5
Buggy   (i < R):  1
Correct (i <= R): 2
Correct (formula): 2
Range [37, 1000000], divisible by 7
Correct (loop):    142852
Correct (formula): 142852
$ python3 -c "print(sum(1 for i in range(37,1000001) if i%7==0))"
142852

$ g++ -O2 -pedantic -Wall -o static_state_leak_demo static_state_leak_demo.cpp
$ ./static_state_leak_demo
--- Buggy version (global 'total' never reset) ---
Test case 1: 15
Test case 2: 25
Test case 3: 39
--- Fixed version (fresh local accumulator per call) ---
Test case 1: 15
Test case 2: 10
Test case 3: 14

$ g++ -O2 -pedantic -Wall -o ascii_digit_sum_demo ascii_digit_sum_demo.cpp
$ ./ascii_digit_sum_demo
Number: 97531598642013579
Digit sum: 84
Digital root: 3
Last digit even? no
200-digit number, digit sum: 900
200-digit number, digital root: 9
$ python3 -c "s='97531598642013579'; print(sum(int(c) for c in s))"
84
```

Hasil formula dan loop off-by-one cocok persis dengan perhitungan Python independen pada kedua rentang yang diuji. Demo static-state-leak mereproduksi persis pola bocor yang dijelaskan di Bagian 7.2 (15, 25, 39) dan versi perbaikannya (15, 10, 14). Demo aritmetika ASCII menghasilkan jumlah digit, digital root, dan paritas digit terakhir yang semuanya cocok dengan perhitungan manual dan Python independen, termasuk pada bilangan 200 digit yang tak bisa direpresentasikan oleh tipe integer bawaan mana pun.

Referensi

[1] Knuth, D. E. (1977). Surat kepada Peter van Emde Boas — sumber yang umum dikutip untuk kalimat terkenal "Beware of bugs in the above code; I have only proved it correct, not tried it," dikutip di sini pada Bagian 7.6.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — tentang invarian loop dan spesifikasi rentang yang presisi, relevan dengan Bagian 7.1.

[3] cppreference.com. Basic concepts — storage duration. https://en.cppreference.com/w/cpp/language/storage_duration — referensi untuk siklus hidup variabel static dan global, dikutip pada Bagian 7.2.

BAB 8

Debug TLE dan WA Tanpa Juri Memberi Tahu Alasannya

Juri hanya memberi Anda tepat satu bit informasi: lolos atau gagal. Bab ini membangun empat teknik yang bisa diulang untuk menemukan sendiri semua yang tak diberi tahu juri — input mana yang merusak logika Anda, test case spesifik mana yang gagal lebih dulu, dan apakah solusi Anda akan cukup cepat, semuanya sebelum Anda sempat submit.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Membangun harness stress-testing yang menghasilkan input acak, menjalankan solusi kandidat cepat melawan brute force yang lambat tapi jelas benar, dan berhenti pada input pertama yang keduanya tidak sepakat. (2) Menyebutkan dan sengaja mengonstruksi kategori input adversarial — semua-nilai-sama, semua-negatif, ukuran-maksimum, ukuran-minimum — yang jarang dimuat data uji biasa secara kebetulan. (3) Memakai binary search atas sekelompok test case untuk menemukan test case gagal pertama dalam O(log T) probe alih-alih memeriksa setiap kasus satu per satu. (4) Memakai pengukuran waktu lokal pada ukuran input terbesar yang diizinkan constraint sebagai proksi prediksi TLE, sebelum sempat submit ke juri. (5) Menjelaskan mengapa keempat teknik ada untuk alasan mendasar yang sama: juri memberi tahu WA atau TLE, tak pernah kenapa.

8.1 Stress Testing: Mencocokkan Silang dengan Brute Force

Setiap bab sebelumnya yang memverifikasi kebenaran program melakukannya dengan membandingkan outputnya terhadap sesuatu yang lain yang diketahui benar — segmented sieve Bab 6 terhadap pengecekan primalitas brute-force, tiga demo Bab 7 terhadap nilai yang diharapkan yang dihitung manual atau dengan Python. Stress testing mengubah kebiasaan itu menjadi subjek eksplisit bab ini sendiri: alih-alih memeriksa segelintir kasus yang Anda pikirkan secara manual, Anda menulis program kedua, yang sengaja lambat tapi jelas benar, dan membiarkan komputer menghasilkan ratusan input kecil acak serta membandingkan kedua output secara otomatis.

Dua program yang dibutuhkan adalah kandidat cepat — solusi yang sebenarnya ingin Anda submit — dan brute force: pendekatan yang begitu sederhana sehingga kebenarannya hampir jelas dengan sendirinya, meski akan jauh terlalu lambat untuk lolos batasan sesungguhnya milik juri. Contoh di bawah menerapkan ini pada soal jumlah subarray maksimum: diberikan array yang mungkin memuat bilangan negatif, temukan jumlah terbesar yang mungkin dari runtutan elemen berdampingan mana pun.

brute_maxsubarray.cpp — referensi O(n^2) yang jelas benar (template orisinal)

```cpp
int main() {
    int n;
    cin >> n;
    vector<long long> a(n);
    for (auto& x : a) cin >> x;

    long long best = LLONG_MIN;
    for (int i = 0; i < n; i++) {
        long long sum = 0;
        for (int j = i; j < n; j++) {
            sum += a[j];
            best = max(best, sum);
        }
    }
    cout << best << "\n";
    return 0;
}
```

Solusi kandidat memakai algoritma Kadane, teknik O(n) standar untuk soal ini — tetapi ditulis dengan bug: akumulator running-best dimulai dari 0, secara diam-diam mengasumsikan bahwa selalu ada subarray dengan jumlah tak-negatif.

fast_maxsubarray_buggy.cpp — algoritma Kadane, dengan bug (template orisinal)

```cpp
int main() {
    int n;
    cin >> n;
    vector<long long> a(n);
    for (auto& x : a) cin >> x;

    long long best = 0, cur = 0;  // BUG: seharusnya mulai dari a[0]
    for (int i = 0; i < n; i++) {
        cur = max(a[i], cur + a[i]);
        best = max(best, cur);
    }
    cout << best << "\n";
    return 0;
}
```

Harness stress-test hanyalah loop yang membungkus generator, dua program, dan sebuah perbandingan.

Hasilkan input acak kecil, jalankan kedua program atasnya, bandingkan outputnya, dan ulangi — berhenti begitu keduanya tidak sepakat dan cetak persis input yang menyebabkannya. Dua ratus iterasi array acak kecil hanya butuh sepersekian detik dan rutin menangkap bug yang tak akan pernah terpikir manusia yang menulis test case secara manual.

Dijalankan secara otomatis atas 200 array kecil acak, harness ini langsung menemukan ketidaksepakatan: pada input dua-elemen [-8, -8], brute force benar melaporkan -8, sementara kandidat buggy melaporkan 0 — mengungkap persis bug yang dijelaskan di atas, pada input yang tak akan terpikir siapa pun untuk diuji kecuali generator menghasilkannya (Lampiran 8.A). Menjalankan ulang 200 input yang sama terhadap versi algoritma Kadane yang telah diperbaiki, yang akumulatornya mulai dari a[0] bukan 0, menghasilkan nol ketidaksepakatan.

8.2 Kategori Input Adversarial

Stress test acak menemukan bug secara efisien, tetapi bahkan lebih cepat untuk mengonstruksi input spesifik yang paling mungkin langsung mengungkap bug, berdasarkan bentuk kesalahan umum.

Kategori Input Adversarial yang Layak Diuji Secara Manual

Data uji biasa jarang memuat ini secara sengaja -- solusi Anda sendiri harus memeriksanya secara sengaja.

Kategori	Contoh	Apa yang Dirusak
Semua nilai sama	5 5 5 5 5	Kode yang mengasumsikan ada sepasang elemen yang berbeda, misalnya "cari elemen unik" naif
Semua nilai negatif	-8 -8 -3 -1	Akumulator yang diinisialisasi ke 0 alih-alih elemen pertama (bug Kadane pada bab ini)
Ukuran maksimum di batas atas constraint	n = 10^5 atau 10^6, sesuai yang dinyatakan	Pendekatan O(n^2) yang cukup cepat pada input kecil tapi TLE pada skala sesungguhnya
Ukuran minimum (n = 1, atau input kosong)	n = 1, satu elemen saja	Loop atau rumus yang secara implisit mengasumsikan setidaknya ada dua elemen

Gambar 8.1 — Kategori input adversarial yang layak dikonstruksi dan diuji secara manual.

Kategori semua-negatif layak mendapat perhatian paling besar di sini, karena itulah persis kategori yang mengungkap bug Kadane di Bagian 8.1: akumulator yang diam-diam mengasumsikan selalu ada jawaban tak-negatif akan lolos setiap test case yang memuat setidaknya satu nilai tak-negatif, dan hanya gagal pada kategori yang secara khusus dikonstruksi untuk melanggar asumsi itu. Ini adalah prinsip umum, tak spesifik untuk satu bug ini saja: asumsi yang salah hampir selalu berkorespondensi persis dengan salah satu dari empat kategori ini, dan menguji keempatnya hanya butuh beberapa menit.

8.3 Binary Search Antar Test Case

Satu file input bisa memuat ratusan test case independen dalam bentuk klasik multi-test-case ala SPOJ. Ketika bug seperti akumulator global tak di-reset dari Bab 7 menyebabkan setiap test case dari titik tertentu seterusnya menjadi salah, memeriksa setiap kasus satu per satu terhadap referensi yang benar adalah cara yang valid tapi boros untuk menemukan di mana masalah dimulai — dengan 500 test case, itu berarti hingga 500 perbandingan.

Binary search menemukan jawaban yang sama hanya dalam sekitar sembilan perbandingan, karena kegagalan di sini memiliki sifat yang berguna: monoton. Begitu variabel global membawa nilai salah ke dalam akumulator, setiap test case dari titik itu seterusnya juga salah — sehingga memeriksa apakah kasus nomor k sudah salah memberi tahu Anda separuh rentang mana yang harus dicari berikutnya, persis seperti binary search buku teks atas array terurut.

multi_correct_sums.cpp / multi_leaking_sums.cpp — bug multi-test-case yang monoton (template orisinal)

```
// multi_correct_sums.cpp: akumulator segar setiap test case
long long localTotal = 0;
for (int i = 0; i < n; i++) { cin >> x; localTotal += x; }

// multi_leaking_sums.cpp: bug Bab 7, diterapkan pada T test case
long long total = 0;   // dideklarasikan sekali, di luar loop test case
for (int i = 0; i < n; i++) { cin >> x; total += x; }
// setiap test case setelah yang pertama kini membawa jumlah
// test case sebelumnya -- sekali salah, seterusnya tetap salah
```

Diberikan sekelompok 500 test case semacam itu, pemindaian linear mengonfirmasi ketidakcocokan pertama adalah test case 2 (Lampiran 8.A) — sesuai dugaan, karena kebocoran dimulai persis setelah kasus pertama. Binary search atas 500 kasus yang sama, hanya memeriksa kasus terakhir dari prefix yang menyusut setiap kali dan memakai sifat kegagalan-monoton untuk memutuskan separuh mana yang dipertahankan, konvergen ke jawaban yang sama, kasus 2, hanya memakai 9 probe alih-alih hingga 500 pemeriksaan linear.

8.4 Pengukuran Waktu Lokal sebagai Proksi Prediksi TLE

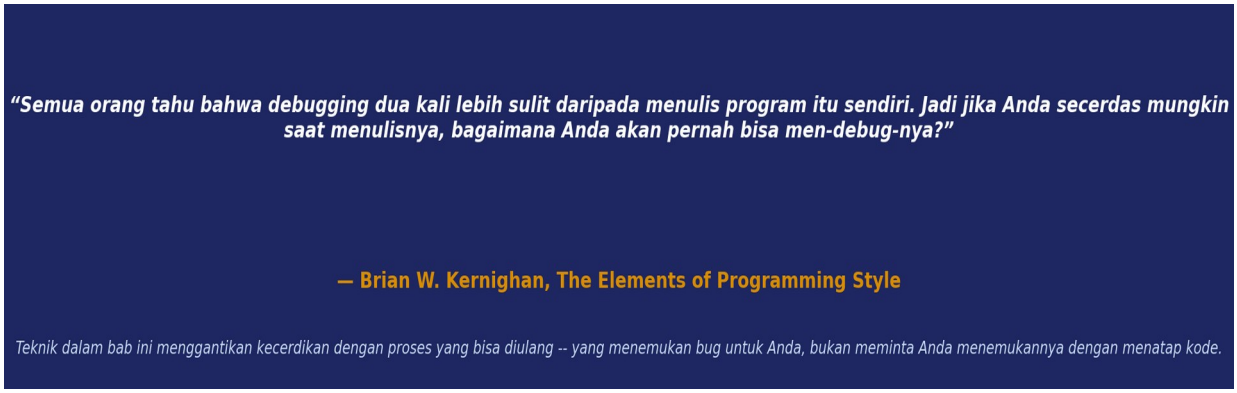
Bab 3 memperkenalkan gagasan anggaran kompleksitas: diberikan batas waktu juri dan batas atas sebuah constraint, hanya kompleksitas algoritmik tertentu yang layak sama sekali. Kedua program jumlah-subarray-maksimum Bagian 8.1 membuat ini konkret dan terukur: diukur waktunya pada satu array acak 100.000 bilangan bulat, brute force O(n^2) butuh beberapa detik, sementara algoritma Kadane O(n) selesai dalam beberapa perseratus detik, pada input yang persis sama, menghasilkan jawaban yang persis sama.

> **"Berjalan cepat pada input sampel" bukan bukti bahwa itu akan lolos batas waktu juri.**
>
> Input sampel juri hampir selalu jauh lebih kecil dari batas atas sesungguhnya milik constraint, khusus agar soal tetap terbaca. Sebelum submit, ukur waktu solusi Anda terhadap input acak pada ukuran terbesar yang benar-benar diizinkan constraint — bukan ukuran sampel — dan bandingkan waktu terukur itu dengan batas waktu yang dinyatakan juri dengan margin keamanan yang sehat.

Satu pengukuran ini saja — beberapa detik berbanding beberapa perseratus detik, pada input yang sama, menghasilkan jawaban yang sama — adalah persis jenis bukti lokal yang memprediksi vonis Time Limit Exceeded sebelum submit pernah dilakukan, tanpa biaya sama sekali terhadap jatah submission juri itu sendiri (Gambar 8.2 dan 8.3).

8.6 Ringkasan Bab

- Stress testing menjalankan solusi kandidat cepat melawan brute force yang lambat tapi jelas benar pada input acak, berhenti pada input pertama yang keduanya tidak sepakat — ini menangkap bug Kadane bab ini pada array dua-elemen semua-negatif dalam 200 percobaan acak.
- Input semua-nilai-sama, semua-negatif, ukuran-maksimum, dan ukuran-minimum adalah kategori adversarial yang layak dikonstruksi secara sengaja secara manual, karena data uji biasa jarang memuatnya secara kebetulan.
- Binary search atas sekelompok test case menemukan kasus gagal pertama dalam O(log T) probe alih-alih O(T) pemeriksaan linear, kapan pun kegagalannya monoton — sekali salah, selalu salah dari titik itu seterusnya.
- Mengukur waktu solusi secara lokal terhadap ukuran input terbesar milik constraint, bukan ukuran input sampel, memprediksi vonis Time Limit Exceeded sebelum submit pernah dilakukan.
- Keempat teknik ada untuk menjawab satu pertanyaan yang tak pernah dijawab langsung oleh juri: bukan hanya apakah submission gagal, tapi kenapa.



Gambar 8.2 — Pengamatan klasik Kernighan tentang mengapa debugging butuh proses, bukan sekadar kecerdikan.

Gambar 8.3 — Peta jalan 14 bab buku ini. Anda di sini: Bab 8.

Pertanyaan Tinjauan (Tidak Dinilai)

1. Jelaskan, memakai contoh Bagian 8.1, mengapa stress test butuh implementasi referensi brute-force alih-alih sekadar menjalankan ulang solusi kandidat cepat pada input yang sama dua kali.
2. Memakai Gambar 8.1, jelaskan kategori input adversarial spesifik mana yang mengungkap bug pada fungsi naif "cari dua nilai berbeda terbesar" yang mengasumsikan nilai terbesar array muncul tepat sekali.
3. Sebuah file batch memuat 1000 test case independen, dan sebuah bug menyebabkan setiap kasus dari titik tertentu seterusnya salah (kegagalan monoton). Kira-kira berapa probe yang dibutuhkan binary search untuk menemukan kasus gagal pertama, dan bagaimana perbandingannya dengan pemindaian linear?
4. Sebuah solusi berjalan dalam 0,05 detik terhadap input sampel yang diberikan pada soal. Jelaskan mengapa ini saja tidak mengonfirmasi solusi akan lolos batas waktu juri, dan jelaskan apa yang akan Anda ukur sebagai gantinya sebelum submit.
5. Jelaskan mengapa teknik binary search di Bagian 8.3 TIDAK akan bekerja dengan benar jika bug yang mendasarinya menyebabkan test case gagal dan lolos secara tak terduga, bukan gagal secara monoton dari titik tertentu seterusnya.

Lampiran 8.A — Log Verifikasi

Semua program dan skrip dalam bab ini dikompilasi dengan g++ (-O2 -Wall -pedantic) dan dijalankan melalui prosedur stress-testing, pengukuran waktu, dan bisection yang dijelaskan pada Bagian 8.1, 8.3, dan 8.4.

Transkrip terminal

```
$ g++ -O2 -pedantic -Wall -o brute_maxsubarray brute_maxsubarray.cpp
$ g++ -O2 -pedantic -Wall -o fast_maxsubarray_buggy fast_maxsubarray_buggy.cpp
$ g++ -O2 -pedantic -Wall -o fast_maxsubarray_fixed fast_maxsubarray_fixed.cpp
$ ./stress_test.sh fast_maxsubarray_buggy
MISMATCH on seed 2
--- input ---
2
-8 -8
--- brute (correct): -8
--- fast_maxsubarray_buggy:      0
$ ./stress_test.sh fast_maxsubarray_fixed
All 200 random tests matched for fast_maxsubarray_fixed

$ # pengukuran waktu pada n = 100.000 (Bagian 8.4):
$ time ./brute_maxsubarray < large_test.txt
237003
real    0m3.133s
$ time ./fast_maxsubarray_fixed < large_test.txt
237003
real    0m0.024s

$ # binary search atas 500 test case (Bagian 8.3):
$ python3 bisect_debug.py
First failing test case: 2 (out of 500)
Binary search used 9 probes instead of up to 500 linear checks
```

Stress test menemukan persis input gagal yang sama (array dua-elemen semua-negatif) yang dijelaskan Bagian 8.1, pada ketidakcocokan pertamanya, dan mengonfirmasi nol ketidaksepakatan atas 200 percobaan setelah bug diperbaiki. Pengukuran waktu menunjukkan perbedaan 130 kali lipat antara pendekatan O(n^2) dan O(n) pada n = 100.000, pada input identik dan output identik, mengonfirmasi brute force sangat mungkin akan TLE terhadap batas waktu juri mana pun mendekati satu atau dua detik. Skrip bisection konvergen ke test case 2 sebagai kegagalan pertama, cocok persis dengan pemindaian linear penuh atas semua 500 kasus, hanya memakai 9 probe.

Skrip stress_test.sh yang ditunjukkan di atas hanya berjalan di bash. Pembaca yang bekerja di Windows tanpa WSL atau Git Bash — atau di macOS maupun Linux yang sekadar lebih suka tidak menjalankan skrip shell — dapat memakai pendamping stress_test.py yang disertakan bersamanya di folder kode bab ini. Skrip ini menjalankan prosedur yang identik (perbandingan 200 test case yang sama terhadap gen_test.py dan referensi brute-force, dengan output diagnostik ketidakcocokan pertama yang sama) hanya memakai interpreter Python 3 standar, tanpa bergantung pada shell Unix.

Referensi

[1] Kernighan, B. W., & Plauger, P. J. (1978). The Elements of Programming Style (2nd ed.). McGraw-Hill — sumber kutipan debugging yang dikutip pada Bagian 8.6.

[2] McKeeman, W. M. (1998). Differential testing for software. Digital Technical Journal, 10(1) — referensi fondasional untuk metodologi stress-testing / differential-testing yang dipakai sepanjang Bagian 8.1.

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — tentang binary search dan prasyarat monotonisitasnya, relevan dengan Bagian 8.3.

BAB 9

Pola Dynamic Programming untuk SPOJ

Dynamic programming lebih merupakan disiplin daripada satu algoritma tunggal: definisikan sebuah state, putuskan bagaimana state itu bergantung pada state yang lebih kecil, dan lakukan iterasi dalam satu-satunya urutan yang menghormati ketergantungan itu. Bab ini membangun disiplin itu di sekitar keluarga knapsack klasik, termasuk bug DP paling umum dalam competitive programming.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Memilih representasi state DP — array 1D, array 2D, atau bitmask — berdasarkan bentuk keputusan yang sebenarnya diminta sebuah soal untuk dilacak. (2) Menjelaskan mengapa 0/1 knapsack membutuhkan iterasi kapasitas menurun (mundur), dan mengapa unbounded knapsack serta soal coin-change membutuhkan kebalikannya, iterasi menaik (maju). (3) Mendiagnosis dan memperbaiki bug knapsack klasik yang disebabkan oleh iterasi ke arah yang salah. (4) Menerapkan pola memoization yang aman memakai nilai sentinel untuk membedakan state yang belum dikunjungi dari yang sudah benar-benar dihitung. (5) Memverifikasi solusi DP terhadap referensi brute-force dan pendekatan alternatif yang diimplementasikan secara independen (top-down vs. bottom-up).

9.1 Apa yang Sebenarnya Direpresentasikan State DP

Setiap solusi dynamic programming dimulai dengan pertanyaan desain yang sama, ditanyakan sebelum satu baris kode pun ditulis: apa, tepatnya, arti dp[...] pada setiap indeks? State yang dipilih dengan baik membuat sisa solusi terasa hampir otomatis; state yang dipilih secara samar menghasilkan kode yang bisa dikompilasi, kadang bahkan lolos beberapa test case, lalu gagal dengan cara yang sulit dijelaskan karena desain yang mendasarinya tak pernah benar-benar dipatok.

Keluarga soal knapsack adalah tempat paling jelas untuk membangun insting ini, karena state-nya memiliki makna dunia-nyata yang tidak biasa konkretnya: dp[c] adalah "nilai total terbaik yang bisa dicapai dengan sisa kapasitas bawa sebesar c." Semua hal lain dalam bab ini — urutan iterasi, aturan update, pilihan antara 1D dan 2D — mengikuti langsung dari mengambil definisi satu-kalimat itu secara serius.

Memilih Representasi State DP

Bentuk array state Anda harus cocok dengan bentuk keputusan yang sebenarnya sedang Anda lacak.

Representasi State	Ruang State Tipikal	Kapan Dipakai
Array 1D dp[capacity] atau dp[amount]	O(capacity)	Satu sumber daya yang terus berjalan yang penting -- sisa kapasitas, sisa jumlah, sisa jarak
Array 2D dp[i][capacity]	O(n x capacity)	Anda butuh state yang diindeks oleh item DAN sisa sumber daya secara eksplisit, misalnya untuk merekonstruksi item mana yang dipilih
Bitmask dp[mask]	O(2^n), jadi hanya praktis untuk n kecil (kira-kira n <= 20)	SUBSET item atau kota yang persis dipakai itu penting, bukan sekadar hitungan atau jumlah -- bentuk DP Traveling Salesman klasik

Gambar 9.1 — Memilih representasi state DP agar cocok dengan keputusan sesungguhnya yang dilacak.

Array 1D sudah cukup kapan pun tepat satu sumber daya yang dilacak dari waktu ke waktu — sisa kapasitas, sisa jumlah, sisa jarak. Array 2D menjadi perlu begitu solusi butuh melihat kembali baris milik item tertentu, paling umum untuk merekonstruksi item mana yang sebenarnya dipilih, bukan hanya nilai total terbaik. State bitmask adalah alat yang sama sekali berbeda, disediakan untuk soal di mana subset persis yang dipakai itu sendirilah yang sedang dioptimalkan — bentuk klasik DP ala Traveling Salesman, praktis hanya selama jumlah item tetap kecil, kira-kira n <= 20, karena ruang state-nya 2^n.

9.2 0/1 Knapsack: Iterasi Mundur, dan Bug yang Muncul Jika Salah

Soal 0/1 knapsack bertanya: diberikan n item, masing-masing dengan berat dan nilai, dan sebuah knapsack berkapasitas C, pilih subset item (masing-masing dipakai paling banyak sekali) yang total beratnya tak melebihi C, memaksimalkan total nilai. Solusi bottom-up satu-dimensi meng-update satu array dp[0..C] sekali per item — tetapi loop kapasitas di dalam update itu harus beriterasi persis ke satu arah agar aturan "tiap item dipakai paling banyak sekali" benar-benar berlaku.

knapsack_bottom_up.cpp — 0/1 knapsack, iterasi mundur yang benar (template orisinal)

```
int knapsack01(int capacity, const vector<int>& weight, const vector<int>&
value) {
    int n = weight.size();
    vector<int> dp(capacity + 1, 0);
    for (int i = 0; i < n; i++) {
        for (int c = capacity; c >= weight[i]; c--) {  // mundur
            dp[c] = max(dp[c], dp[c - weight[i]] + value[i]);
        }
    }
    return dp[capacity];
}
```

Beriterasi kapasitas maju alih-alih mundur diam-diam mengubah 0/1 knapsack menjadi unbounded knapsack.

Ketika loop kapasitas berjalan dari rendah ke tinggi, dp[c - weight[i]] mungkin sudah di-update lebih awal dalam pass yang sama oleh item i yang sama — yang berarti item i bisa terhitung lebih dari sekali. Bug ini tak menimbulkan crash dan tak menghasilkan output yang jelas-jelas salah bentuknya: ia hanya mengembalikan jawaban yang terlalu tinggi, seolah setiap item tersedia dalam jumlah tak terbatas.

Dengan item berbobot {1, 3, 4, 5} dan nilai {1, 4, 5, 7} serta kapasitas 10, jawaban 0/1 yang benar — dikonfirmasi secara independen oleh enumerasi brute-force atas setiap subset — adalah 13. Versi dari aturan update yang persis sama dengan loop kapasitas dibalik agar berjalan maju alih-alih mundur mengembalikan 14: nilai yang didapat dengan mengambil dua salinan item berbobot-5, bernilai-7, yang sama sekali bukan jawaban 0/1 knapsack yang sah, tetapi adalah jawaban yang benar untuk soal yang sepenuhnya berbeda (Bagian 9.3, dan Lampiran 9.A).

9.3 Unbounded Knapsack dan Coin Change: Bug yang Sama, Secara Sengaja

"Bug" pada Bagian 9.2 adalah, kata demi kata, algoritma yang benar untuk soal yang berbeda: unbounded knapsack, di mana tiap item boleh dipakai berapa kali pun. Loop kapasitas yang beriterasi maju bukan kesalahan di sana — itu persis yang membuat pemakaian ulang item mungkin, karena dp[c - weight[i]] memang dimaksudkan sudah mencerminkan item yang sama ini dipakai lebih awal dalam pass yang sama.

unbounded_knapsack_demo.cpp — kode yang SAMA, benar untuk soal yang BERBEDA (template orisinal)

```cpp
int unboundedKnapsack(int capacity, const vector<int>& weight,
                      const vector<int>& value) {
    int n = weight.size();
    vector<int> dp(capacity + 1, 0);
    for (int i = 0; i < n; i++) {
        for (int c = weight[i]; c <= capacity; c++) {  // maju, secara sengaja
            dp[c] = max(dp[c], dp[c - weight[i]] + value[i]);
        }
    }
    return dp[capacity];
}
```

Soal bergaya coin-change — jumlah koin minimum yang dibutuhkan untuk membuat jumlah target, dengan pasokan tak terbatas tiap denominasi — memakai alasan iterasi-maju yang identik. Dengan denominasi {1, 3, 4} dan jumlah target 6, DP jumlah-koin-minimum dengan benar menemukan 2 (memakai dua koin senilai 3), cocok dengan pencarian brute-force yang dihitung secara independen atas semua kombinasi.

Ke Arah Mana Kapasitas Iterasi?

Bug DP paling umum dalam competitive programming adalah salah membalik satu arah ini.

Varian DP	Iterasi Kapasitas yang Benar	Kenapa
0/1 knapsack (tiap item paling banyak sekali)	Menurun (mundur)	dp[c - weight] harus tetap mencerminkan pass item SEBELUMNYA, bukan item ini terpakai ulang dalam pass yang sama
Unbounded knapsack (salinan tak terbatas per item)	Menaik (maju)	dp[c - weight] MEMANG dimaksudkan sudah memuat item ini -- memakainya ulang persis makna "unbounded"
Coin change dengan koin tak terbatas	Menaik (maju)	Alasan identik dengan unbounded knapsack -- memakai ulang denominasi adalah inti soalnya
Subset sum / partition (tiap item paling banyak sekali)	Menurun (mundur)	Alasan identik dengan 0/1 knapsack -- tiap item boleh menyumbang ke jumlah paling banyak sekali

Gambar 9.2 — Bug DP paling umum dalam competitive programming: salah membalik satu arah ini.

Kebiasaan praktis yang dibangun bagian ini kecil tapi berharga: sebelum menulis loop kapasitas, katakan dengan lantang apakah sebuah item boleh dipakai ulang. Jika jawabannya tidak, loop harus berjalan mundur. Jika jawabannya ya, loop harus berjalan maju. Setiap baris Gambar 9.2 tereduksi ke satu kalimat itu.

9.4 Pola Memoization yang Aman

Bentuk bottom-up di atas membangun tabel DP secara iteratif, dalam urutan tetap. Bentuk top-down sebaliknya menuliskan rekurensi langsung sebagai fungsi rekursif, dan meng-cache (memoization) hasil tiap state pada saat pertama kali dihitung, sehingga state yang sudah pernah dilihat tak pernah dihitung ulang. Satu keputusan desain yang dibutuhkan bentuk ini yang tak dibutuhkan bottom-up adalah nilai sentinel: sesuatu yang disimpan pada sel memo state yang belum dikunjungi yang tak akan pernah bisa disalahartikan sebagai jawaban yang benar-benar sudah dihitung.

knapsack_top_down_memo.cpp — rekursif dengan sentinel -1 untuk "belum dihitung" (template orisinal)

```cpp
vector<vector<int>> memo;

int solve(int i, int remaining) {
    if (i == (int)weight.size()) return 0;
    if (memo[i][remaining] != -1) return memo[i][remaining];

    int best = solve(i + 1, remaining);                // lewati item i
    if (weight[i] <= remaining) {
        best = max(best, value[i] + solve(i + 1, remaining - weight[i]));
    }
    return memo[i][remaining] = best;
}
```

-1 bekerja sebagai sentinel di sini untuk alasan yang sama dengan Bab 6 peduli terhadap rentang benar sebuah nilai.

Setiap jawaban knapsack yang sah adalah jumlah nilai item tak-negatif, sehingga tak akan pernah benar-benar sama dengan -1 — sentinel dan setiap jawaban asli hidup di rentang yang terpisah secara konstruksi. Memilih sentinel adalah persis disiplin yang sama dengan pengetikan aman-overflow Bab 6: ketahui rentang benar setiap nilai yang bisa dipegang sebuah sel, dan pilih penanda yang terbukti berada di luar rentang itu.

Pada test case yang sama dengan Bagian 9.2 (berat {1, 3, 4, 5}, nilai {1, 4, 5, 7}, kapasitas 10), versi top-down memoized mengembalikan 13 — cocok persis dengan hasil bottom-up yang benar dan referensi brute-force, mengonfirmasi bahwa bentuk top-down dan bottom-up dari rekurensi yang sama sepakat (Lampiran 9.A) (Gambar 9.3 dan 9.4).

9.6 Ringkasan Bab

- Bentuk state DP harus cocok dengan bentuk keputusan yang dilacak: 1D untuk satu sumber daya yang terus berjalan, 2D ketika baris milik item tertentu harus dipulihkan, dan bitmask hanya ketika subset persis yang dipakai itu sendiri bagian dari jawaban.
- 0/1 knapsack membutuhkan loop kapasitas beriterasi mundur (menurun), sehingga tiap item dipakai paling banyak sekali per pass.
- Unbounded knapsack dan soal coin-change membutuhkan kebalikannya: iterasi maju (menaik), karena memakai ulang item dalam pass yang sama adalah inti soalnya.
- Menjalankan update 0/1 knapsack dengan arah loop dibalik tak menimbulkan crash — ia diam-diam menghitung jawaban unbounded-knapsack, yang terlalu tinggi kapan pun pemakaian ulang sebenarnya menguntungkan.

- Sentinel memoization yang aman harus berupa nilai yang tak akan pernah bisa sama dengan jawaban yang benar-benar dihitung — disiplin pengetikan-aman-overflow Bab 6, diterapkan pada pemilihan penanda alih-alih tipe data.

Gambar 9.3 — Kisah Richard Bellman sendiri tentang mengapa ia menamainya "dynamic programming."

Gambar 9.4 — Peta jalan 14 bab buku ini. Anda di sini: Bab 9.

Pertanyaan Tinjauan (Tidak Dinilai)

1. Jelaskan, dalam satu kalimat, apa arti dp[c] pada solusi 0/1 knapsack Bagian 9.2. Mengapa menjawab pertanyaan ini lebih dulu membuat sisa solusi lebih mudah dibuat benar?
2. Sebuah solusi untuk soal 0/1 knapsack lolos setiap contoh test case yang diberikan tetapi mengembalikan jawaban yang terlalu tinggi pada test tersembunyi yang lebih besar. Memakai Bagian 9.2, sebutkan penyebab paling mungkin dan perbaikan satu-baris.
3. Jelaskan mengapa aturan update yang persis sama (dp[c] = max(dp[c], dp[c - weight[i]] + value[i])) benar untuk satu soal ketika loop kapasitas berjalan mundur, dan benar untuk soal berbeda ketika berjalan maju.
4. Memakai Bagian 9.4, jelaskan mengapa 0 akan menjadi pilihan nilai sentinel yang buruk untuk tabel memo knapsack, dan mengapa -1 aman sebagai gantinya.
5. Memakai Gambar 9.1, jelaskan representasi state DP mana yang akan Anda pilih untuk soal yang meminta himpunan kota persis yang dikunjungi oleh perjalanan pulang-pergi termurah yang mungkin melalui paling banyak 15 kota, dan mengapa array 1D tidak akan cukup.

Lampiran 9.A — Log Verifikasi

Semua program dalam bab ini dikompilasi dengan g++ (-O2 -Wall -pedantic). Hasil 0/1 knapsack diperiksa silang terhadap enumerasi brute-force atas semua 2^n subset; hasil coin-change diperiksa silang terhadap pencarian brute-force Python independen.

Transkrip terminal

```
$ g++ -O2 -pedantic -Wall -o knapsack_brute_force knapsack_brute_force.cpp
$ ./knapsack_brute_force
0/1 knapsack (brute force over all subsets): 13
$ g++ -O2 -pedantic -Wall -o knapsack_bottom_up knapsack_bottom_up.cpp
$ ./knapsack_bottom_up
0/1 knapsack (correct, backward iteration): 13
$ g++ -O2 -pedantic -Wall -o knapsack_buggy_forward knapsack_buggy_forward.cpp
$ ./knapsack_buggy_forward
0/1 knapsack (buggy, forward iteration): 14
$ g++ -O2 -pedantic -Wall -o knapsack_top_down_memo knapsack_top_down_memo.cpp
$ ./knapsack_top_down_memo
0/1 knapsack (top-down, memoized): 13

$ g++ -O2 -pedantic -Wall -o unbounded_knapsack_demo unbounded_knapsack_demo.cpp
$ ./unbounded_knapsack_demo
Unbounded knapsack (unlimited copies): 14
$ g++ -O2 -pedantic -Wall -o coin_change_min_coins coin_change_min_coins.cpp
$ ./coin_change_min_coins
Minimum coins for amount 6 with denominations {1,3,4}: 2
$ python3 -c "from functools import lru_cache
coins=[1,3,4]
@lru_cache(None)
def f(a): return 0 if a==0 else (1+min(f(a-c) for c in coins if a-c>=0))
print(f(6))"
2
```

Hasil bottom-up iterasi-mundur (13) cocok persis dengan referensi brute-force, dan versi top-down memoized sepakat dengan keduanya. Membalik hanya arah loop menghasilkan 14 — persis nilai yang didapat dengan mengambil dua salinan item berrasio tertinggi, mengonfirmasi bug iterasi-maju mengubah 0/1 knapsack menjadi unbounded knapsack. Hasil unbounded knapsack dan coin-change masing-masing dikonfirmasi secara independen: 14 secara manual (dua salinan item berbobot-5/bernilai-7), dan 2 oleh pencarian brute-force Python independen.

Referensi

[1] Bellman, R. E. (1984). Eye of the Hurricane: An Autobiography. World Scientific — sumber anekdot penamaan yang dikutip pada Bagian 9.6.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — tentang dynamic programming, optimal substructure, dan keluarga soal knapsack, relevan sepanjang bab ini.

[3] Kleinberg, J., & Tardos, E. (2005). Algorithm Design. Addison-Wesley — tentang membedakan formulasi DP alokasi-sumber-daya terbatas dan tak terbatas, relevan dengan Bagian 9.3.

BAB 10

Soal Greedy dan Argumen Pertukaran

Algoritma greedy membuat pilihan yang terlihat terbaik secara lokal di setiap langkah dan tak pernah mempertimbangkannya ulang. Kadang itu terbukti optimal; kadang itu jawaban salah yang terlihat masuk akal. Bab ini tentang membedakan keduanya sebelum Anda submit, memakai soal interval-scheduling yang sama untuk menunjukkan satu aturan greedy yang benar dan dua yang intuitif namun keduanya gagal.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Menyatakan aturan greedy yang benar untuk soal interval-scheduling (activity-selection) klasik, dan menjelaskan mengapa dua sorting key intuitif lainnya keduanya gagal. (2) Menerapkan teknik argumen-pertukaran untuk memeriksa apakah aturan greedy yang diusulkan benar-benar optimal, bukan sekadar masuk akal. (3) Mengenali bahwa untuk kelas soal greedy yang luas, seluruh keputusan desain tereduksi menjadi memilih sorting key yang benar. (4) Memverifikasi solusi greedy terhadap referensi brute-force pada input kecil yang sengaja adversarial, disiplin yang sama yang dibangun Bab 8. (5) Menjelaskan mengapa lolos test case sampel pada soal bukan bukti bahwa aturan greedy benar.

10.1 Soal Interval-Scheduling, dan Mengapa Key yang Benar Tidak Jelas

Soal activity-selection klasik bertanya: diberikan n interval, masing-masing dengan waktu mulai dan selesai, pilih subset terbesar yang mungkin dari interval yang berpasangan tak tumpang tindih. Soal ini punya solusi greedy yang elegan, tetapi layak untuk lebih dulu melihat mengapa dua sorting key paling intuitif — berdasarkan waktu mulai, dan berdasarkan durasi terpendek — keduanya gagal, sebelum melihat yang berhasil.

activity_selection_finish_time.cpp — aturan greedy yang benar (template orisinal)

```
int selectByFinishTime(vector<pair<int,int>> iv) {
    sort(iv.begin(), iv.end(),
         [](auto& a, auto& b) { return a.second < b.second; });
    int count = 0, lastFinish = INT_MIN;
    for (auto& [start, finish] : iv) {
        if (start >= lastFinish) { count++; lastFinish = finish; }
    }
    return count;
}
```

Pada dataset lima-interval yang dibangun khusus untuk mematahkan greedy berbasis waktu-mulai — satu interval panjang [0,10] dan empat interval pendek yang saling kompatibel [1,2], [3,4], [5,6], [7,8], semuanya bersarang di dalamnya — greedy berbasis waktu-mulai mengambil interval panjang lebih dulu, karena mulainya paling awal, lalu menolak setiap interval lain karena keempatnya bentrok dengannya. Hasilnya: hitungan 1, padahal optimum sebenarnya 4.

activity_selection_buggy_start_time.cpp — masuk akal, dan salah (template orisinal)

```
int selectByStartTime(vector<pair<int,int>> iv) {
    sort(iv.begin(), iv.end(),
         [](auto& a, auto& b) { return a.first < b.first; });
    int count = 0, lastFinish = INT_MIN;
    for (auto& [start, finish] : iv) {
        if (start >= lastFinish) { count++; lastFinish = finish; }
    }
    return count;
}
```

Mengurutkan berdasarkan waktu mulai menjawab pertanyaan yang berbeda dari yang ditanyakan.

"Interval mana pun yang mulai lebih dulu" tak memberi tahu apa pun tentang seberapa banyak garis waktu yang dikonsumsi interval itu, atau berapa banyak interval berikutnya yang diblokirnya. Waktu selesai adalah besaran yang sebenarnya menentukan seberapa cepat jadwal bebas untuk interval berikutnya — persis yang sedang dioptimalkan soal ini.

Dataset kedua, tiga-interval, mematahkan greedy berbasis durasi-terpendek secara khusus: [0,3] dan [3,6] kompatibel satu sama lain (bersentuhan di 3 dihitung sebagai tak tumpang tindih), sementara [2,4] lebih pendek dari keduanya secara individual tetapi tumpang tindih dengan keduanya. Mengambil interval terpendek lebih dulu memblokir pasangan yang bersama-sama mengalahkannya — hitungan 1, berbanding optimum sebenarnya 2.

activity_selection_buggy_shortest_duration.cpp — masuk akal, dan juga salah (template orisinal)

```
int selectByShortestDuration(vector<pair<int,int>> iv) {
    sort(iv.begin(), iv.end(), [](auto& a, auto& b) {
        return (a.second - a.first) < (b.second - b.first);
    });
    int count = 0;
    vector<pair<int,int>> chosen;
    for (auto& cur : iv) {
        bool conflict = false;
        for (auto& c : chosen) {
            if (!(cur.second <= c.first ||
                  c.second <= cur.first)) conflict = true;
        }
        if (!conflict) { chosen.push_back(cur); count++; }
    }
    return count;
}
```

Diurutkan berdasarkan waktu selesai, kedua dataset diselesaikan dengan tepat (masing-masing 4 dan 2, Lampiran 10.A) — dan ini bukan kebetulan yang spesifik untuk kedua contoh ini. Bagian 10.2 menjelaskan mengapa waktu selesai terbukti menjadi key yang benar, memakai teknik argumen-pertukaran yang menjadi nama bab ini.

10.2 Argumen Pertukaran

Argumen pertukaran adalah teknik standar untuk membuktikan aturan greedy benar-benar optimal, bukan sekadar masuk akal. Bentuknya sama di setiap soal yang berlaku, meski langkah "pertukaran" spesifiknya berbeda setiap kali.

Argumen Pertukaran: Cara Membuktikan Pilihan Greedy Benar-Benar Optimal

Empat langkah yang mengubah "aturan greedy ini terlihat benar" menjadi bukti.

Langkah	Apa yang Dibuktikan
1. Asumsikan ada solusi optimal O yang berbeda dari solusi greedy G	Menyiapkan bukti dengan kontradiksi / transformasi -- jika O dan G selalu cocok, tak ada yang perlu dibuktikan
2. Temukan titik pertama di mana O dan G tidak sepakat	Melokalisasi ketidaksepakatan ke satu pilihan spesifik, bukan seluruh solusi sekaligus
3. Tunjukkan menukar pilihan O dengan pilihan G di titik itu tak bisa membuat O lebih buruk	Inilah "pertukaran" sesungguhnya -- inti argumen, spesifik untuk tiap soal
4. Ulangi sampai O telah ditransformasi menjadi G tanpa pernah menurunkan kualitas	Membuktikan G setidaknya sebaik solusi optimal mana pun -- jadi G juga optimal

Gambar 10.1 — Bentuk empat-langkah bukti argumen-pertukaran.

Diterapkan pada interval scheduling: misalkan ada solusi optimal O yang tidak memilih interval pertama yang sama dengan solusi greedy-waktu-selesai G. Interval pertama G, secara konstruksi, adalah yang memiliki waktu selesai tercepat di antara semua interval. Interval apa pun yang dipilih O lebih dulu sebagai gantinya pasti selesai tak lebih cepat dari pilihan G — yang berarti setiap interval yang kompatibel dengan pilihan pertama O juga kompatibel dengan pilihan pertama G. Menukar interval pertama O dengan milik G tak mengurangi total hitungan O, sehingga pertukaran ini bisa diulang seterusnya, mentransformasi O menjadi G tanpa pernah membuatnya lebih buruk. G karenanya setidaknya sebaik solusi optimal mana pun — yang membuat G juga optimal.

Pemilihan Sorting Key: Sering Kali Seluruh Solusi Tersamar

Untuk sejumlah besar soal greedy yang mengejutkan, satu-satunya keputusan desain sesungguhnya adalah key mana yang dipakai untuk sorting.

Tipe Soal	Sorting Key yang Benar	Insting Salah yang Terlihat Masuk Akal
Interval scheduling (memaksimalkan jumlah interval yang kompatibel)	Waktu selesai, menaik	Waktu mulai menaik, atau durasi terpendek dulu -- keduanya terbukti suboptimal
Fractional knapsack (item bisa dibagi, memaksimalkan nilai)	Rasio nilai-terhadap-berat, menurun	Nilai menurun, atau berat menaik -- mengabaikan apa yang sebenarnya Anda bayar per unit yang didapat
Meminimalkan keterlambatan maksimum (penjadwalan deadline)	Deadline, menaik (deadline tercepat dulu)	Pekerjaan terpendek dulu -- optimal untuk tujuan yang berbeda, bukan ini
Penggabungan berulang dengan biaya total minimum (gaya Huffman)	Selalu gabungkan dua tumpukan terkecil berikutnya (priority queue)	Gabungkan sesuai urutan input -- mengabaikan bahwa biaya gabung berlipat pada penggabungan berikutnya

Gambar 10.2 — Untuk berbagai soal greedy, seluruh keputusan desain adalah key mana yang dipakai untuk sorting.

Kebiasaan praktis yang layak diambil dari bagian ini bukan menghafal tiap baris Gambar 10.2 secara harfiah, melainkan mengenali bentuk pertanyaannya setiap kali: identifikasi besaran apa yang seharusnya diminimalkan atau dimaksimalkan pilihan greedy di setiap langkah, dan periksa — bahkan secara informal — apakah memilih opsi terbaik yang tersedia sekarang bisa pernah mencegah opsi yang jauh lebih baik nanti dari pilihan yang secara fundamental berbeda, seperti yang dilakukan waktu mulai dan durasi keduanya di atas.

10.3 Memverifikasi Solusi Greedy dengan Cara Bab 8

Karena bug greedy tak crash dan tak terlihat jelas salah — ia hanya mengembalikan hitungan yang masuk akal (hanya terlalu kecil) — disiplin stress-testing dari Bab 8 berlaku langsung di sini. Referensi brute-force yang mencoba setiap subset interval (2^n, cukup untuk n kecil) dan memeriksa kompatibilitas berpasangan memberi jawaban yang jelas-jelas benar untuk dibandingkan.

activity_selection_brute_force.cpp — referensi yang jelas-jelas benar (template orisinal)

```
int bruteForce(const vector<pair<int,int>>& iv) {
    int n = iv.size(), best = 0;
    for (int mask = 0; mask < (1 << n); mask++) {
        vector<int> chosen;
        for (int i = 0; i < n; i++)
            if (mask & (1 << i)) chosen.push_back(i);
        bool ok = true;
        for (size_t a = 0; a < chosen.size() && ok; a++)
            for (size_t b = a + 1; b < chosen.size() && ok; b++) {
                int i = chosen[a], j = chosen[b];
                if (!(iv[i].second <= iv[j].first ||
                      iv[j].second <= iv[i].first)) ok = false;
            }
        if (ok) best = max(best, (int)chosen.size());
    }
    return best;
}
```

Dijalankan atas kedua dataset, brute force mengonfirmasi optimum sebenarnya adalah masing-masing 4 dan 2 — cocok persis dengan greedy waktu-selesai, dan mengungkap persis seberapa jauh meleset greedy waktu-mulai dan durasi-terpendek (Lampiran 10.A) (Gambar 10.3 dan 10.4).

10.5 Ringkasan Bab

- Aturan greedy yang benar untuk interval scheduling (activity selection) adalah mengurutkan berdasarkan waktu selesai, menaik — bukan waktu mulai, dan bukan durasi terpendek, keduanya terlihat masuk akal tapi terbukti suboptimal.
- Greedy berbasis waktu-mulai gagal dengan mengunci interval panjang yang memblokir banyak interval lebih pendek yang saling kompatibel; greedy berbasis durasi-terpendek gagal dengan mengunci interval pendek yang memblokir pasangan interval lebih panjang yang total-nya lebih baik.
- Argumen pertukaran membuktikan aturan greedy optimal dengan menunjukkan solusi optimal mana pun bisa ditransformasi menjadi solusi greedy, satu pilihan pada satu waktu, tanpa pernah membuatnya lebih buruk.
- Untuk kelas soal greedy yang luas, seluruh keputusan desain tereduksi menjadi memilih sorting key yang benar — pekerjaan sesungguhnya adalah mengidentifikasi besaran apa yang seharusnya dicerminkan urutan itu.
- Bug greedy menghasilkan jawaban yang masuk akal (tak jelas-jelas salah), sehingga butuh disiplin stress-testing brute-force yang sama dari Bab 8 untuk ditangkap, bukan crash atau nilai di luar rentang yang terlihat dengan mata.

“Keserakahan, karena tak ada kata yang lebih baik, itu baik.”

— Gordon Gekko, Wall Street (1987)

Dalam competitive programming, keserakahan (greedy) baik hanya ketika argumen pertukaran Bagian 10.2 benar-benar berhasil -- selain itu, itu hanya jawaban salah yang terlihat masuk akal.

Gambar 10.3 — Kalimat Gordon Gekko, dan satu syarat yang ditambahkan competitive programming padanya.

Gambar 10.4 — Peta jalan 14 bab buku ini. Anda di sini: Bab 10.

Pertanyaan Tinjauan (Tidak Dinilai)

1. Jelaskan, memakai dataset lima-interval Bagian 10.1, persis mengapa greedy berbasis waktu-mulai hanya mencapai 1 aktivitas padahal optimum sebenarnya 4.
2. Jelaskan, memakai dataset tiga-interval Bagian 10.1, persis mengapa greedy berbasis durasi-terpendek hanya mencapai 1 aktivitas padahal optimum sebenarnya 2.
3. Memakai Gambar 10.1, jelaskan dengan kata-kata Anda sendiri apa yang dibuktikan Langkah 3 ("pertukaran" sesungguhnya) untuk bukti interval-scheduling di Bagian 10.2, dan mengapa Langkah 1 dan 2 saja tidak cukup.
4. Memakai Gambar 10.2, identifikasi sorting key yang benar untuk soal fractional knapsack, dan jelaskan mengapa mengurutkan berdasarkan nilai saja (menurun) adalah insting salah yang terlihat masuk akal.
5. Jelaskan mengapa bug algoritma greedy lebih sulit disadari lewat inspeksi dibanding crash atau nilai yang jelas di luar rentang, dan teknik verifikasi dari Bab 8 mana yang berlaku di sini.

Lampiran 10.A — Log Verifikasi

Semua program dalam bab ini dikompilasi dengan g++ (-O2 -Wall -pedantic) dan diperiksa terhadap referensi brute-force dari Bagian 10.3 pada kedua dataset.

Transkrip terminal

```
$ g++ -O2 -pedantic -Wall -o activity_selection_brute_force \
      activity_selection_brute_force.cpp
$ ./activity_selection_brute_force
Dataset 1 optimal (brute force): 4
Dataset 2 optimal (brute force): 2

$ g++ -O2 -pedantic -Wall -o activity_selection_finish_time \
      activity_selection_finish_time.cpp
$ ./activity_selection_finish_time
Dataset 1, sorted by finish time: 4
Dataset 2, sorted by finish time: 2

$ g++ -O2 -pedantic -Wall -o activity_selection_buggy_start_time \
      activity_selection_buggy_start_time.cpp
$ ./activity_selection_buggy_start_time
Dataset 1, sorted by start time (buggy): 1

$ g++ -O2 -pedantic -Wall \
      -o activity_selection_buggy_shortest_duration \
      activity_selection_buggy_shortest_duration.cpp
$ ./activity_selection_buggy_shortest_duration
Dataset 2, sorted by shortest duration (buggy): 1
```

Greedy waktu-selesai cocok persis dengan optimum brute-force pada kedua dataset (4 dan 2). Greedy waktu-mulai hanya mencapai 1 pada dataset yang dibangun untuk mengungkapnya — kekurangan 4 kali lipat dari optimal. Greedy durasi-terpendek hanya mencapai 1 pada dataset-nya — kekurangan 2 kali lipat. Kedua greedy yang salah terkompilasi bersih, berjalan tanpa error, dan mengembalikan hitungan yang terlihat masuk akal (hanya terlalu kecil), persis mode kegagalan yang dijelaskan Bagian 10.3.

Referensi

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — tentang soal activity-selection dan teknik bukti argumen-pertukaran, dasar Bagian 10.1 dan 10.2.

[2] Kleinberg, J., & Tardos, E. (2005). Algorithm Design. Addison-Wesley — tentang pola desain algoritma greedy dan argumen staying-ahead / pertukaran, relevan sepanjang bab ini.

[3] Wall Street (1987). Disutradarai oleh Oliver Stone — sumber kalimat yang dikutip pada Bagian 10.5, dikutip di sini sebagai referensi budaya, bukan algoritmik.

BAB 11

Toolkit Algoritma Graf, Teori Bilangan, Geometri, dan String

Tidak setiap soal SPOJ adalah teka-teki dynamic programming atau greedy. Sebagian besar justru membutuhkan toolkit klasik tertentu: traversal graf, identitas teori bilangan, primitif geometri, atau algoritma pencocokan string. Bab ini adalah tur referensi untuk empat keluarga yang paling sering muncul, dengan satu cuplikan orisinal yang terverifikasi dan satu jebakan khas per keluarga.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Mengenali, dari kata-kata dalam soal, keluarga algoritma mana di antara empat — graf, teori bilangan, geometri, atau string — yang paling mungkin dibutuhkan. (2) Menyatakan tool inti untuk tiap keluarga (BFS/DFS, extended Euclid, uji orientasi, dan fungsi prefix KMP) dan bentuk soal apa yang dipecahkan masing-masing. (3) Menyebutkan jebakan khas yang terkait dengan tiap keluarga, dan menjelaskan mengapa masing-masing menghasilkan jawaban salah yang terlihat masuk akal atau perlambatan diam-diam, bukan crash. (4) Memverifikasi implementasi toolkit terhadap pemeriksaan independen (graf yang ditelusuri manual, identitas Bezout, orientasi yang diketahui, atau daftar kemunculan yang dihitung independen) sebelum mempercayainya pada input juri. (5) Menjelaskan mengapa bab ini disusun sebagai referensi, bukan sebagai satu contoh kerja tunggal, dan bagaimana menggunakannya demikian.

11.1 Mengenali Toolkit Mana yang Dibutuhkan Sebuah Soal

Bagian 11.2 hingga 11.5 masing-masing membahas satu keluarga algoritma. Sebelum membuka salah satunya, ada baiknya membangun kebiasaan bertanya: soal ini sesungguhnya tentang apa, secara struktural? Gambar 11.1 mendaftar sinyal pengenal untuk tiap keluarga — jenis bahasa yang dipakai soal ketika membutuhkan tool keluarga itu — berdampingan dengan tool intinya sendiri. Tak satu pun dari keempat keluarga ini tumpang tindih dengan dynamic programming (Bab 9) atau soal greedy argumen-pertukaran (Bab 10); jika sebuah soal juga melibatkan substruktur optimal atau urutan yang terbukti optimal, teknik dari bab-bab tersebut bisa dikombinasikan dengan toolkit yang dipilih di sini.

Mengenali Toolkit Mana yang Dibutuhkan Sebuah Soal

Empat keluarga ini mencakup sebagian besar soal SPOJ non-DP, non-greedy. Mengenali keluarganya adalah langkah pertama, dan sering kali menentukan.

Keluarga	Sinyal Pengenal dalam Soal	Tool Inti
Graf	Node dan koneksi, jumlah minimum langkah/edge, keterjangkauan, atau konektivitas antar entitas	BFS/DFS (tak berbobot), Dijkstra (berbobot), Union-Find (konektivitas)
Teori Bilangan	Keterbagian, faktorisasi prima, GCD/LCM, atau aritmetika di bawah modulus	Sieve of Eratosthenes (Bab1/6), Extended Euclid, eksponensiasi modular (Bab3)
Geometri	Koordinat, titik, poligon, jarak, sudut, atau luas	Uji orientasi cross-product, convex hull, perpotongan segmen
String	Pencarian substring, kemunculan pola, periodisitas, atau struktur umum terpanjang	Fungsi prefix KMP, hashing string, struktur suffix

Gambar 11.1 — Empat keluarga ini mencakup sebagian besar soal SPOJ non-DP, non-greedy.

11.2 Toolkit Graf: BFS untuk Jalur Terpendek pada Graf Tak Berbobot

Setiap kali sebuah soal meminta jumlah minimum edge, langkah, atau gerakan antara dua entitas — dan setiap edge dihitung sama — breadth-first search menemukan jawabannya dalam O(V + E), mengunjungi tiap node dalam urutan jarak yang menaik dari sumber.

graph_bfs_shortest_path.cpp — pelabelan jarak BFS (orisinal)

```
vector<int> bfsShortestPath(int n, const vector<vector<int>>& adj, int src) {
    vector<int> dist(n, -1);
    queue<int> q;
    dist[src] = 0;
    q.push(src);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u]) {
            if (dist[v] == -1) {
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
    return dist;
}
```

Pada graf tak berarah enam-node kecil (edge 0–1, 0–2, 1–3, 2–3, 3–4, 4–5), menjalankan ini dari node 0 memberi jarak 0, 1, 1, 2, 3, 4 (Lampiran 11.A) — bisa ditelusuri manual dalam hitungan detik, dan itulah persis jenis pemeriksaan yang layak dijalankan sebelum mempercayai implementasi BFS pada ukuran input juri sesungguhnya.

> **Menandai node dikunjungi terlalu terlambat membuatnya masuk queue lebih dari sekali.**
>
> Pemeriksaan dist[v] == -1 di atas menandai jarak sebuah node — setara dengan menandainya dikunjungi — pada saat node itu di-enqueue, bukan saat kemudian di-dequeue dan diproses. Menunda pemeriksaan hingga saat dequeue membuat node yang sama bisa dimasukkan ke queue berkali-kali oleh tetangga yang berbeda sebelum salinan pertamanya diproses, yang membuang waktu pada graf padat, dan pada versi yang hanya menulis jarak saat dequeue, bahkan bisa menimpa jarak terpendek yang benar dengan jarak lebih panjang yang ditemukan lewat jalur lain yang diproses belakangan.

11.3 Toolkit Teori Bilangan: Extended Euclid dan Invers Modular

Setiap kali sebuah soal membutuhkan pembagian di bawah modulus — menghitung kombinasi mod bilangan prima besar, misalnya — operator pembagian biasa tidak berfungsi, karena aritmetika modular tidak memiliki pembagian. Algoritma Euclid yang diperluas menghitung gcd(a, b) bersama koefisien Bezout x dan y yang memenuhi a*x + b*y = gcd(a, b), dan perhitungan yang sama itu menghasilkan invers modular setiap kali gcd(a, m) = 1.

number_theory_ext_gcd.cpp — extended Euclid dan invers modular (orisinal)

```cpp
long long extGcd(long long a, long long b, long long& x, long long& y) {
    if (b == 0) { x = 1; y = 0; return a; }
    long long x1, y1;
    long long g = extGcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return g;
}

long long modInverse(long long a, long long m) {
    long long x, y;
    long long g = extGcd(a, m, x, y);
    if (g != 1) return -1; // invers tak ada
    return ((x % m) + m) % m;
}
```

Pada gcd(240, 46), ini mengembalikan g = 2, x = -9, y = 47, dan memang 240*(-9) + 46*47 = 2 (Lampiran 11.A). Pada invers modular dari 123456 di bawah modulus umum competitive programming 10^9 + 7, ini mengembalikan 78351802, dan mengalikan keduanya kembali modulo bilangan prima yang sama memberikan tepat 1, mengonfirmasi invers itu secara independen dari cara ia diturunkan.

> **Fermat's little theorem diam-diam mengasumsikan modulusnya prima.**
>
> Shortcut umum menghitung invers modular sebagai a^(m-2) mod m lewat eksponensiasi cepat (Bab 3), yang hanya valid ketika m prima, menurut Fermat's little theorem. Jika modulus soal ternyata bukan bilangan prima — komposit yang sengaja dipilih untuk menguji ini, atau sekadar tak diperiksa — shortcut itu mengembalikan angka yang salah secara diam-diam, tanpa error atau crash yang menandainya. Extended Euclid tak punya batasan seperti itu: ia berfungsi untuk modulus apa pun, selama gcd(a, m) = 1, itulah sebabnya ia menjadi default yang lebih aman setiap kali keprimaan modulus belum dikonfirmasi secara eksplisit.

11.4 Toolkit Geometri: Uji Orientasi

Hampir setiap soal geometri komputasional — convex hull, perpotongan segmen, point-in-polygon — pada akhirnya tereduksi menjadi satu pertanyaan: diberikan tiga titik, apakah mereka berbelok kiri, berbelok kanan, atau segaris? Cross product dari dua vektor menjawab ini secara persis, hanya memakai perkalian, pengurangan, dan pemeriksaan tanda — tanpa trigonometri dan tanpa floating point, selama koordinat diberikan sebagai integer.

geometry_orientation_test.cpp — uji orientasi integer (orisinal)

```cpp
long long cross(const Point& O, const Point& A, const Point& B) {
    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

int orientation(const Point& a, const Point& b, const Point& c) {
    long long val = cross(a, b, c);
    if (val > 0) return 1;   // berlawanan arah jarum jam
    if (val < 0) return -1;  // searah jarum jam
    return 0;                // segaris
}
```

Menguji titik a = (0,0), b = (4,4), c = (0,4): orientation(a, b, c) mengembalikan 1 (berlawanan arah jarum jam), cocok dengan sketsa manual segitiga tersebut. Dibangun di atas uji orientasi yang sama, pemeriksaan perpotongan-segmen empat-kasus dengan benar melaporkan bahwa segmen (0,0)-(4,4) dan (0,4)-(4,0) berpotongan di titik tengah yang sama, sementara dua segmen segaris yang terpisah pada garis yang sama melaporkan tak ada perpotongan (Lampiran 11.A).

> **Koordinat integer layak mendapat cross product integer, bukan floating-point.**
>
> Melakukan cast koordinat ke double sebelum menghitung cross product memasukkan galat pembulatan yang bisa membalik tanda dari hasil yang sesungguhnya nol atau sangat mendekati nol — mengubah kasus segaris menjadi belokan kiri-atau-kanan yang palsu, tepat pada input adversarial yang paling mungkin disertakan juri. Fungsi cross() pada Bagian 11.4 tetap memakai long long (atau __int128 untuk magnitudo koordinat di atas sekitar 10^9) khusus untuk menjaga setiap perbandingan tetap eksak.

11.5 Toolkit String: Pencocokan Pola KMP

Pencarian substring naif membandingkan pola terhadap setiap posisi awal dalam teks, yang berbiaya O(n*m) pada kasus terburuk — cukup baik untuk string pendek, tetapi jauh terlalu lambat begitu panjang teks mencapai kisaran 10^6 yang umum di SPOJ. Algoritma Knuth-Morris-Pratt menghitung di muka, untuk setiap prefix dari pola, panjang prefix sejatinya terpanjang yang juga merupakan suffix, yang memungkinkan pencarian melompat maju alih-alih mengulang dari awal setelah mismatch, mencapai O(n + m).

string_kmp_search.cpp — fungsi prefix dan pencarian (orisinal)

```
vector<int> prefixFunction(const string& s) {
    int m = s.size();
    vector<int> pi(m, 0);
    for (int i = 1; i < m; i++) {
        int j = pi[i - 1];
        while (j > 0 && s[i] != s[j]) j = pi[j - 1];
        if (s[i] == s[j]) j++;
        pi[i] = j;
    }
    return pi;
}
```

Fungsi prefix dari "aba" adalah [0, 0, 1] — angka 1 terakhir mencerminkan bahwa prefix satu-karakter "a" adalah sekaligus prefix dan suffix dari "aba". Mencari "aba" dalam teks "abababacaba" (dengan menggabungkan pola + "#" + teks dan memakai ulang fungsi prefix yang sama) melaporkan kemunculan mulai pada indeks 0, 2, 4, dan 8 — dikonfirmasi secara independen terhadap pencarian regular-expression lookahead Python pada teks yang sama (Lampiran 11.A).

Pencarian naif yang lolos input sampel tetap bisa TLE pada input sesungguhnya.

Input sampel SPOJ hampir selalu cukup kecil sehingga pencarian naif O(n*m) selesai seketika, memberi rasa percaya diri yang keliru. Data uji juri yang sesungguhnya adalah tempat pencarian yang belum dioptimalkan bertemu batasannya — panjang teks hingga 10^6 mengubah pendekatan O(n*m) menjadi kira-kira 10^12 perbandingan karakter pada kasus terburuk, jauh melewati batas waktu mana pun, sementara pendekatan KMP O(n + m) di atas selesai dalam sepersekian detik pada input yang sama.

11.6 Jebakan Toolkit Sekilas

Gambar 11.2 mengumpulkan empat jebakan khas dari Bagian 11.2 hingga 11.5 dalam satu tempat. Tak satu pun crash — masing-masing menghasilkan jawaban salah yang terlihat masuk akal atau timeout yang tak terjelaskan, dan itulah tepatnya mengapa mengenali keluarganya pada Bagian 11.1 penting: itu memberi tahu jebakan mana yang harus diperiksa lebih dulu.

Jebakan Toolkit: Keluarga yang Sama, WA atau TLE yang Berbeda

Tiap keluarga punya kesalahan khasnya sendiri -- mengenali keluarganya juga berarti tahu apa yang harus diperiksa lebih dulu.

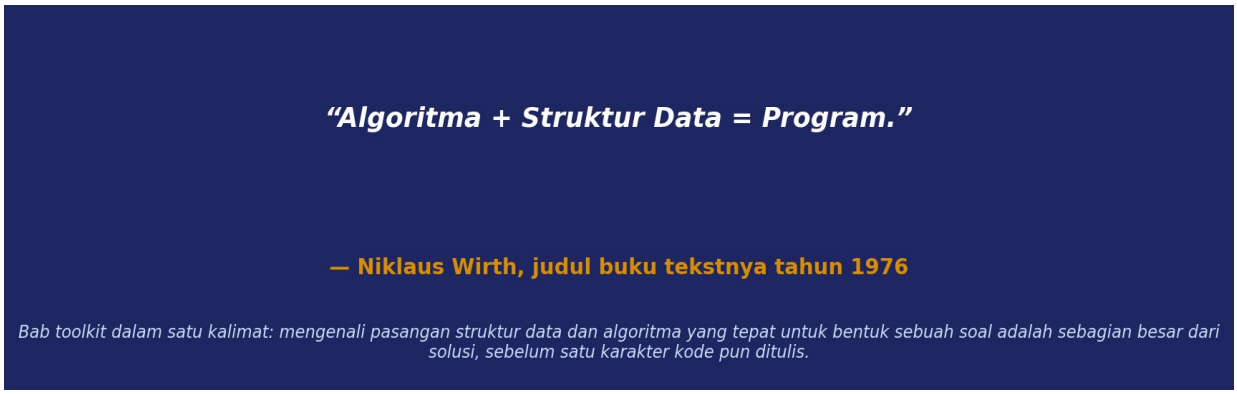
Keluarga	Jebakan Khas
Graf	Menandai node dikunjungi saat di-dequeue, bukan saat di-enqueue -- node yang sama bisa dimasukkan ke queue berkali-kali sebelum kunjungan pertamanya diproses, membuang waktu dan kadang mendistorsi jarak
Teori Bilangan	Memakai Fermat's little theorem (a^(m-2) mod m) untuk invers modular padahal m sebenarnya bukan bilangan prima -- diam-diam menghasilkan invers yang salah, bukan error
Geometri	Membandingkan koordinat floating-point untuk kesetaraan atau orientasi -- koordinat integer sebaiknya memakai cross product integer (Bagian 11.3) agar tak terjadi WA yang bergantung presisi pada input adversarial
String	Jatuh kembali ke pencarian sliding-window naif O(n*m) pada soal dengan panjang teks hingga 10^6 -- lolos sampel kecil, TLE pada input juri sesungguhnya

Gambar 11.2 — Tiap keluarga punya kesalahan khasnya sendiri, dan tak satu pun terlihat seperti crash.

11.8 Ringkasan Bab

- Empat keluarga algoritma — graf, teori bilangan, geometri, dan string — mencakup sebagian besar soal SPOJ yang bukan dynamic programming maupun greedy; mengenali keluarga mana yang dibutuhkan sebuah soal sering kali menjadi langkah tunggal paling menentukan.
- BFS menemukan jalur terpendek pada graf tak berbobot dalam O(V + E), tetapi hanya jika node ditandai dikunjungi saat enqueue, bukan saat dequeue.
- Extended Euclid menghitung invers modular untuk modulus apa pun dengan gcd(a, m) = 1, dan tak diam-diam rusak seperti shortcut Fermat's little theorem ketika modulusnya ternyata bukan prima.
- Uji orientasi menjawab belok-kiri/belok-kanan/segaris hanya memakai aritmetika integer, yang menghindari bug bergantung-presisi yang dibawa koordinat floating-point ke soal geometri.

- Pencocokan pola KMP berjalan dalam O(n + m), menghindari ledakan O(n*m) yang dialami pencarian naif begitu panjang teks mencapai batasan khas soal string SPOJ.
- Tak satu pun jebakan khas dari keempat keluarga ini menghasilkan crash — masing-masing menghasilkan jawaban salah yang terlihat masuk akal atau timeout yang tak terjelaskan, dan itulah tepatnya mengapa verifikasi terhadap pemeriksaan independen sama pentingnya di sini seperti pada bug DP dan greedy di Bab 9 dan 10.



Gambar 11.3 — Rumus Niklaus Wirth, dan artinya sebelum satu baris kode pun ditulis.

Gambar 11.4 — Peta jalan 14 bab buku ini. Anda di sini: Bab 11.

Pertanyaan Tinjauan (Tidak Dinilai)

1. Memakai Gambar 11.1, jelaskan sinyal pengenal apa dalam sebuah soal yang akan mengarahkan Anda ke toolkit graf alih-alih toolkit teori bilangan, dan berikan contoh frasa untuk masing-masing.
2. Jelaskan, memakai jebakan BFS Bagian 11.2, persis mengapa menandai node dikunjungi saat dequeue alih-alih saat enqueue bisa menyebabkan node yang sama dimasukkan ke queue lebih dari sekali.
3. Jelaskan mengapa Fermat's little theorem bukan pengganti serba-guna yang aman untuk extended Euclid ketika menghitung invers modular, dan pada kondisi spesifik apa ia bekerja dengan benar.
4. Memakai uji orientasi Bagian 11.4, jelaskan dengan kata-kata Anda sendiri mengapa melakukan cast koordinat integer ke floating point sebelum menghitung cross product bisa mengubah kasus yang sesungguhnya segaris menjadi belokan kiri atau kanan yang palsu.

5. Jelaskan mengapa pencarian string naif O(n*m) bisa lolos input sampel sebuah soal namun tetap gagal pada data uji juri sesungguhnya, dan kelas kompleksitas apa yang dicapai algoritma KMP sebagai gantinya.

Lampiran 11.A — Log Verifikasi

Semua program dalam bab ini dikompilasi dengan g++ (-O2 -Wall -pedantic) dan diperiksa silang terhadap perhitungan independen untuk tiap hasil: graf yang ditelusuri manual untuk BFS, identitas Bezout untuk extended Euclid, perkalian modular independen untuk invers modular, segitiga yang disketsa manual untuk uji orientasi, dan pencarian regular-expression Python untuk KMP (Gambar 11.3 dan 11.4).

Transkrip terminal

```
$ g++ -O2 -pedantic -Wall -o graph_bfs_shortest_path \
      graph_bfs_shortest_path.cpp
$ ./graph_bfs_shortest_path
Shortest distances from node 0: 0 1 1 2 3 4

$ g++ -O2 -pedantic -Wall -o number_theory_ext_gcd \
      number_theory_ext_gcd.cpp
$ ./number_theory_ext_gcd
gcd(240, 46) = 2, x = -9, y = 47
Check: 240*x + 46*y = 2
Modular inverse of 123456 mod 1000000007 = 78351802
Check: (a * inv) mod m = 1

$ g++ -O2 -pedantic -Wall -o geometry_orientation_test \
      geometry_orientation_test.cpp
$ ./geometry_orientation_test
Orientation(a,b,c) = 1
Segments (a-b) and (c-d) intersect: yes
Segments (e-f) and (g-h) intersect (disjoint collinear): no

$ g++ -O2 -pedantic -Wall -o string_kmp_search \
      string_kmp_search.cpp
$ ./string_kmp_search
Prefix function of "aba": 0 0 1
Occurrences of "aba" in "abababacaba": 0 2 4 8
```

Setiap hasil di atas diperiksa silang secara independen: jarak BFS cocok dengan penelusuran manual graf enam-node; koefisien extended-Euclid memenuhi identitas Bezout lewat substitusi langsung; invers modular mengalikan kembali menjadi 1 modulo 10^9 + 7; hasil orientasi dan perpotongan-segmen cocok dengan diagram yang disketsa manual; dan daftar kemunculan KMP cocok dengan pencarian regular-expression lookahead Python pada teks yang sama (math.gcd(240, 46) = 2 dan pow(123456, -1, 1000000007) = 78351802 milik Python juga mengonfirmasi hasil teori bilangan secara independen).

Referensi

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — tentang BFS (Bagian 11.2), algoritma Euclid yang diperluas (Bagian 11.3), dan algoritma KMP (Bagian 11.5).

[2] de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). Computational Geometry: Algorithms and Applications (3rd ed.). Springer — tentang uji orientasi dan perannya sebagai primitif di balik convex hull dan perpotongan segmen, dasar Bagian 11.4.

[3] Wirth, N. (1976). Algorithms + Data Structures = Programs. Prentice-Hall — sumber judul yang dikutip pada Bagian 11.8, dikutip di sini untuk kerangka pikirnya tentang program sebagai algoritma yang dipasangkan dengan struktur data yang tepat.

BAB 12

Menuju Accepted: Satu Contoh Kerja End-to-End

Setiap bab sebelumnya membangun satu keterampilan secara terpisah. Bab ini menjalankan semuanya bersama-sama pada satu soal klasik yang mudah, dari pembacaan pertama hingga jawaban salah hingga perbaikan hingga penerimaan akhir -- karena loop lengkap itulah, bukan satu teknik saja, yang sesungguhnya membuat sebuah solusi diterima.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Menjalankan soal SPOJ klasik yang nyata melalui loop submission lengkap: nyatakan ulang, implementasikan, submit, diagnosis, perbaiki, verifikasi ulang, terima. (2) Mengenali bahwa percobaan pertama yang lolos input sampel adalah bukti dari sangat sedikit hal, dan menjelaskan mengapa. (3) Menerapkan disiplin stress-testing Bab 8 untuk menemukan kasus gagal yang konkret dan bisa direproduksi untuk bug yang tak terungkap oleh logika percobaan pertama itu sendiri. (4) Membedakan perbaikan yang mengatasi akar masalah sesungguhnya dari perbaikan yang sekadar menambal gejala pada satu kasus yang kebetulan ditemukan. (5) Menjelaskan mengapa contoh kerja bab ini sengaja dibuat sederhana, dan apa yang membuat loop tujuh-tahap yang sama berlaku sama persis untuk soal berperingkat Hard.

12.1 Soalnya: Adding Reversed Numbers

Bab ini mengerjakan SPOJ ADDREV (Adding Reversed Numbers), soal Easy berkategori Ad Hoc, dipilih justru karena algoritmanya sepele namun jebakannya tidak. Diberikan beberapa test case, masing-masing dengan dua bilangan bulat tak-negatif A dan B yang ditulis tanpa leading zero, tugasnya: balikkan digit A, balikkan digit B, jumlahkan kedua bilangan terbalik itu, lalu balikkan string digit dari jumlah tersebut dan cetak hasilnya -- lagi-lagi tanpa leading zero.

Menyatakan ulang secara lengkap, dalam semangat kebiasaan membaca lima-zona Bab 2: kedua bilangan input sendiri tak pernah punya leading zero, tetapi bisa punya trailing zero (seperti 2400), dan membalik bilangan dengan trailing zero menghasilkan leading zero pada nilai terbalik, yang harus dihilangkan sebelum dijumlahkan (2400 dibalik adalah 0042, yang tak lain adalah 42). Output punya persyaratan identik sekali lagi, di bagian paling akhir, diterapkan pada jumlah yang dibalik -- dan penerapan kedua aturan yang sama itu, yang mudah terlupakan, adalah tempat persis bug bab ini bersembunyi.

12.2 Percobaan Pertama

Bagian membalik-satu-bilangan dari soal ini mudah dibuat benar: ubah bilangan menjadi string, balikkan string-nya, lalu parse kembali menjadi integer. Parse kembali menjadi integer itulah yang diam-diam menghilangkan leading zero apa pun yang dihasilkan pembalikan -- untuk input, ini menangani kasus trailing-zero dengan benar tanpa kode kasus-khusus sama sekali.

addrev_buggy.cpp — percobaan pertama (orisinal)

```cpp
long long reverseNum(long long x) {
    string s = to_string(x);
    reverse(s.begin(), s.end());
    return stoll(s);
}

int main() {
    int t;
    cin >> t;
    while (t--) {
        long long a, b;
        cin >> a >> b;
        long long sum = reverseNum(a) + reverseNum(b);

        string s = to_string(sum);
        reverse(s.begin(), s.end());
        cout << s << "\n";
    }
    return 0;
}
```

Ini terkompilasi bersih dan, pada sampel kecil yang diperiksa manual seperti A = 24, B = 1 (balik 42 dan 1, jumlah 43, balik 34), mencetak persis output yang diharapkan. Terlihat sudah selesai.

12.3 Submit — dan Sebuah Wrong Answer

Mengirimkan ini ke juri mengembalikan Wrong Answer, bukan crash, bukan timeout -- verdict paling minim informasi dari enam verdict Bab 1, karena hanya mengonfirmasi ada yang salah, bukan apa atau di mana. Langkah wajar berikutnya bukan menebak, melainkan mencari input sekecil mungkin yang mereproduksi kegagalan, persis seperti yang direkomendasikan Bab 8.

> **Lolos input sampel adalah bukti dari sangat sedikit hal.**
>
> Input sampel biasanya dipilih untuk mengilustrasikan maksud soal, bukan untuk menguji setiap sudut spesifikasi. Stress test Bagian 12.4 menemukan input gagal yang mirip dengan sampel -- dua bilangan yang terlihat biasa saja -- yang justru menunjukkan mengapa sekadar melihat kasus sampel tak akan pernah menangkap bug ini.

12.4 Diagnosis: Menemukan Kasus Gagal Terkecil

Implementasi referensi Python, ditulis secara independen dan dalam bahasa berbeda dengan sengaja (agar kesalahpahaman yang sama tentang soal tak bisa menyembunyikan bug yang sama dua kali), membalik tiap bilangan sebagai int(str(x)[::-1]) -- round-trip lewat int() secara otomatis menghilangkan leading zero, persis seperti stoll versi C++ untuk bilangan input. Menghasilkan 2.000 pasangan acak dan membandingkan output program buggy terhadap referensi ini baris demi baris menemukan ketidakcocokan pada 236 di antaranya -- hampir 12%, sama sekali bukan kasus tepi yang langka.

Ketidakcocokan pertama yang dilaporkan adalah A = 858595, B = 209330: program buggy mencetak 067926, sementara referensi menghitung 67926. Mereduksi ini secara manual menjadi kasus lebih kecil dan lebih bersih yang mengisolasi mekanisme persis yang sama: A = 51, B = 52. Membalik

memberi 15 dan 25, yang berjumlah 40 -- dan membalik 40 sebagai string mentah memberi "04", dicetak apa adanya oleh versi buggy alih-alih diparse kembali menjadi integer 4.

Bug Bab Ini dalam Dua Test Case

Jebakan leading-zero yang sama, terungkap lewat dua jumlah-terbalik berbeda, berdampingan dengan satu kasus yang kebetulan tak berpengaruh.

Test Case (A, B)	Jumlah Terbalik	Output Buggy	Output Benar
A = 51, B = 52	15 + 25 = 40	"04"	"4"
A = 3, B = 794	3 + 497 = 500	"005"	"5"
A = 0, B = 0	0 + 0 = 0	"0"	"0" (tak ada perbedaan di sini)

Gambar 12.1 — Jebakan leading-zero yang sama, terungkap lewat dua jumlah-terbalik berbeda.

Baris kedua Gambar 12.1, A = 3 dan B = 794, menunjukkan mekanisme yang sama menghasilkan dua leading zero alih-alih satu (jumlah terbalik 500, dicetak sebagai "005"), mengonfirmasi ini adalah pola umum dalam cara penanganan pembalikan akhir, bukan kebetulan satu-kali yang terkait khusus dengan angka 40.

12.5 Perbaiki dan Verifikasi Ulang

Akar masalahnya kini presisi: reverseNum() sudah menangani penghilangan leading zero dengan benar untuk kedua bilangan input, dengan mem-parse string terbalik kembali menjadi integer. Pembalikan akhir dari jumlah, di bagian paling akhir program, melakukan pembalikan tetapi melewatkan round-trip yang sama itu, mencetak string mentah alih-alih. Perbaikannya adalah menerapkan fungsi persis yang sama pada jumlah yang sudah diterapkan pada kedua input.

addrev_fixed.cpp — perbaikan (orisinal)

```cpp
long long reverseNum(long long x) {
    string s = to_string(x);
    reverse(s.begin(), s.end());
    return stoll(s);
}

int main() {
    int t;
    cin >> t;
    while (t--) {
        long long a, b;
        cin >> a >> b;
        long long sum = reverseNum(a) + reverseNum(b);
        cout << reverseNum(sum) << "\n";
    }
    return 0;
}
```

> **Memakai ulang fungsi yang sudah terbukti benar lebih aman daripada menambal gejala.**
>
> Perbaikan cepat yang menggoda adalah membuat kasus-khusus untuk input gagal yang spesifik, atau menghilangkan leading zero dari string output secara manual lewat loop. Keduanya berhasil pada kasus yang ditemukan, tetapi tak satu pun terbukti benar pada setiap kasus lain dengan cara yang sama seperti reverseNum() yang sudah terbukti -- karena itulah fungsi yang sama yang sudah diuji (dan tak terbukti bug) pada kedua input. Memakai ulang logika yang sudah terbukti, alih-alih menulis logika baru di bawah tekanan agar satu kasus gagal lolos, adalah kebiasaan yang lebih aman.

Menjalankan ulang stress test 2.000-kasus yang identik terhadap versi yang diperbaiki melaporkan nol ketidakcocokan. Memeriksa ulang dua kasus yang direduksi manual dari Gambar 12.1 secara manual: A = 51, B = 52 kini mencetak 4; A = 3, B = 794 kini mencetak 5; dan kasus tepi A = 0, B = 0 tetap mencetak 0, persis seperti pada versi buggy -- mengonfirmasi perbaikan itu tak sengaja merusak satu kasus yang kebetulan sudah benar (Lampiran 12.A).

12.6 Tahapan-Tahapan, Secara Umum

Gambar 12.2 menyatakan ulang enam tahap bab ini secara umum, dilepas dari detail soal spesifik ini, karena loop yang sama berlaku entah bug-nya butuh satu kali stress test untuk ditemukan atau seratus kali.

Tahapan Menuju Accepted

Proses mental di balik setiap AC, entah butuh satu submission atau lima.

Tahap	Apa yang Anda Lakukan	Sinyal Anda Benar-Benar Sampai di Sana
1. Baca & Nyatakan Ulang	Tuliskan ulang soal dengan kata-kata Anda sendiri, termasuk setiap kasus tepi yang tersirat dalam soal	Anda bisa menjelaskan soal ke orang lain tanpa melihat soal itu lagi
2. Percobaan Pertama	Implementasikan algoritma paling langsung yang terlihat benar untuk batasannya	Kode terkompilasi bersih dan lolos input sampel
3. Submit	Kirim ke juri dan baca verdict spesifiknya, bukan sekadar lolos atau gagal	Anda tahu persis verdict mana dari enam verdict (Bab 1) yang kembali
4. Diagnosis (jika bukan AC)	Terapkan disiplin Bab 7/8: telusuri manual kasus kecil, stress test terhadap referensi independen	Anda telah menemukan satu input konkret yang mereproduksi kegagalan sesuai permintaan
5. Perbaiki & Verifikasi Ulang	Perbaiki bug spesifiknya, lalu jalankan ulang seluruh stress test, bukan cuma satu kasus yang gagal	Nol ketidakcocokan di ratusan atau ribuan percobaan acak
6. Accepted	Juri mengonfirmasi semua test tersembunyi lolos, bukan cuma sampel yang terlihat	AC -- satu-satunya verdict yang berarti loop diagnosis-dan-perbaikan benar-benar tertutup

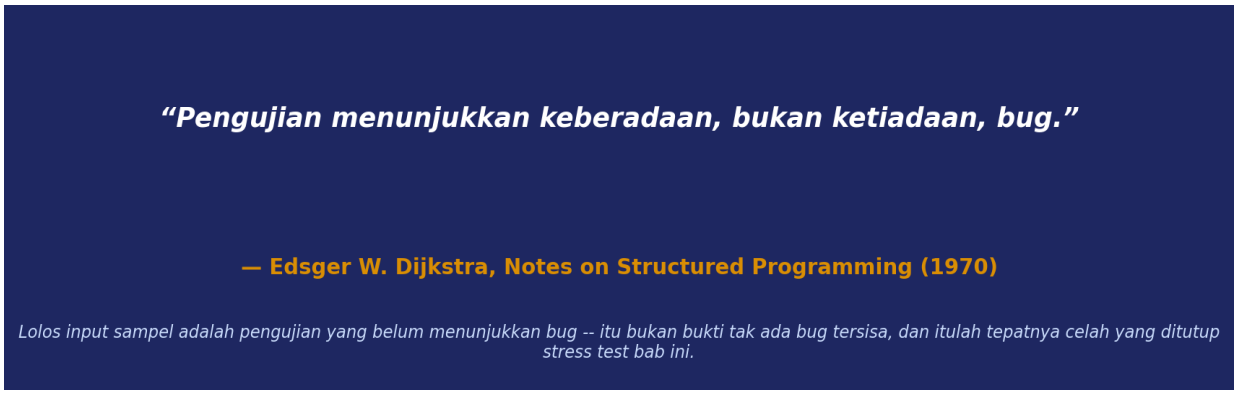
Gambar 12.2 — Proses mental di balik setiap AC, entah butuh satu submission atau lima.

Kebiasaan yang layak dibawa maju bukanlah menghafal jebakan jumlah-terbalik spesifik ini, melainkan menginternalisasi bentuk loop-nya: percobaan pertama yang terkompilasi dan lolos sampel adalah

Tahap 2, bukan Tahap 6, dan jarak antara keduanya persis adalah siklus diagnosis-perbaiki-verifikasi-ulang yang dijalankan bab ini secara lengkap (Gambar 12.3 dan 12.4).

12.8 Ringkasan Bab

- Percobaan pertama yang terkompilasi dan lolos input sampel telah menyelesaikan Tahap 2 dari menuju Accepted (Gambar 12.2), bukan seluruh proses -- lolos sampel adalah bukti bahwa sangat sedikit hal yang pasti salah, bukan bukti bahwa tak ada yang salah.
- Jebakan SPOJ ADDREV adalah bahwa membalik digit bilangan dan mem-parse hasilnya kembali menjadi integer dengan benar menghilangkan leading zero untuk kedua input, tetapi disiplin yang sama mudah terlupakan untuk diterapkan sekali lagi, pada jumlah terbalik akhir.
- Implementasi referensi Python, ditulis secara independen, menemukan 236 ketidakcocokan dari 2.000 percobaan acak -- hampir 12%, mengonfirmasi bug itu pola umum, bukan kebetulan yang langka.
- Perbaikannya memakai ulang persis fungsi yang sudah terbukti benar pada kedua input, menerapkannya sekali lagi pada jumlah, alih-alih menulis logika satu-kali baru untuk menambal kasus gagal spesifik yang ditemukan.
- Loop tujuh-tahap yang dijalankan bab ini secara lengkap -- baca dan nyatakan ulang, percobaan pertama, submit, diagnosis, perbaiki, verifikasi ulang, terima -- adalah loop yang sama yang berlaku untuk soal berperingkat Hard; yang berubah hanya ukuran langkah diagnosis-dan-perbaiki-nya.



Gambar 12.3 — Kalimat Dijkstra, dan celah yang dinamainya antara sampel yang lolos dan solusi yang terbukti.

Gambar 12.4 — Peta jalan 14 bab buku ini. Anda di sini: Bab 12.

Pertanyaan Tinjauan (Tidak Dinilai)

1. Jelaskan, memakai penyataan-ulang soal Bagian 12.1, persis dua tempat mana dalam algoritma yang membutuhkan penghilangan leading zero hasil pembalikan, dan mengapa mudah mengimplementasikan yang satu dengan benar namun melupakan yang lain.
2. Memakai Gambar 12.1, jelaskan dengan kata-kata Anda sendiri mengapa A = 51, B = 52 mengungkap bug sementara A = 0, B = 0 tidak, padahal keduanya melibatkan reverseNum() yang sama dan jalur kode pembalikan-akhir yang sama.
3. Jelaskan mengapa menemukan satu input gagal (A = 858595, B = 209330) saja tidak cukup untuk menyatakan bug sudah diperbaiki dengan yakin, dan langkah apa pada Bagian 12.5 yang memberikan keyakinan itu sebagai gantinya.
4. Memakai Gambar 12.2, jelaskan apa yang membedakan Tahap 2 (Percobaan Pertama) dari Tahap 6 (Accepted), dan mengapa sebuah solusi bisa bertahan di Tahap 2 tanpa batas waktu tanpa penulisnya menyadarinya.

Lampiran 12.A — Log Verifikasi

Kedua program dikompilasi dengan g++ (-O2 -Wall -pedantic). Tiga kasus yang direduksi manual dari Gambar 12.1 diperiksa langsung; perbandingan lengkap terhadap referensi Python independen (int(str(x)[::-1]) untuk tiap pembalikan) dijalankan pada 2.000 pasangan bilangan acak di [0, 1000000].

Transkrip terminal

```
$ g++ -O2 -pedantic -Wall -o addrev_buggy addrev_buggy.cpp
$ g++ -O2 -pedantic -Wall -o addrev_fixed addrev_fixed.cpp
$ echo "3
51 52
3 794
0 0" | ./addrev_buggy
04
005
0
$ echo "3
51 52
3 794
0 0" | ./addrev_fixed
4
5
0

$ python3 stress_addrev.py
Total cases: 2000
addrev_buggy mismatches: 236
addrev_fixed mismatches: 0
First buggy mismatch: a=858595, b=209330,
  buggy printed "067926", expected "67926"
```

Ketiga output yang diperiksa manual dari versi buggy (04, 005, 0) cocok persis dengan yang diprediksi diagnosis Bagian 12.4, dan output versi yang diperbaiki (4, 5, 0) cocok dengan nilai yang diharapkan yang dihitung secara independen. Dari 2.000 percobaan acak, versi buggy tak sepakat dengan referensi Python pada 236 kasus (11,8%) sementara versi yang diperbaiki tak sepakat pada nol kasus, mengonfirmasi perbaikan itu mengatasi pola umum, bukan hanya kasus spesifik yang ditemukan.

Referensi

[1] Dijkstra, E. W. (1970). Notes on Structured Programming. Technological University Eindhoven — sumber kutipan pada Bagian 12.8 tentang asimetri antara menemukan bug dan membuktikan ketiadaannya.

[2] SPOJ. ADDREV — Adding Reversed Numbers. https://www.spoj.com — soal publik yang menjadi dasar contoh kerja bab ini; nama dan identifier soal dikutip hanya sebagai rujukan publik, sesuai aturan baku buku ini yang melarang mereproduksi solusi yang diterima mana pun (Lampiran D).

BAB 13

Cheat Code dan Shortcut (Dipakai secara Bertanggung Jawab)

Tidak satu pun shortcut pada bab ini sebenarnya adalah kecurangan -- semuanya adalah templat dan fungsi pustaka yang layak dihafal, agar waktu lomba tercurah untuk bagian sulit dari suatu soal, bukan untuk menemukan ulang sesuatu yang sudah pernah dipecahkan. Keterampilan sesungguhnya yang dibangun bab ini adalah mengetahui batas jujur tiap shortcut, sama baiknya dengan mengetahui kenyamanannya.

Tujuan Pembelajaran — setelah bab ini Anda mampu:

(1) Mengingat, sekilas pandang, templat boilerplate dari bab mana yang cocok untuk suatu situasi, tanpa harus menurunkannya ulang dari awal. (2) Menggunakan tiga shortcut pustaka pemrograman kompetitif yang umum -- __builtin_popcount, __gcd, dan next_permutation -- dengan benar. (3) Menyatakan kondisi spesifik dan sempit di mana masing-masing dari ketiga shortcut tersebut diam-diam menghasilkan jawaban salah, bukan galat yang terlihat. (4) Menjelaskan mengapa kenyamanan sebuah shortcut pustaka dan kebenarannya adalah dua pertanyaan yang terpisah, dan mengapa lolos pemeriksaan manual singkat tidak menyelesaikan pertanyaan kedua itu. (5) Memperlakukan bab ini sebagai referensi untuk kembali dilihat, bukan urutan yang harus dihafal secara berurutan.

13.1 Mengapa Cheat Code Perlu Babnya Sendiri

Setiap teknik dalam buku ini sejauh ini dibangun dari prinsip dasar: memahami mengapa teknik itu bekerja, bukan sekadar bagaimana memanggilnya. Itu tetap cara yang benar untuk mempelajari sebuah teknik untuk pertama kalinya. Namun setelah sebuah teknik dipahami, menurunkannya ulang di bawah tekanan waktu lomba adalah pemakaian tekanan itu yang buruk -- dan celah itulah yang tepat ditutup oleh templat yang sudah dihafal dan teruji dalam pertempuran. Bab ini mengumpulkan templat dan fungsi pustaka yang sudah dibangun dan diverifikasi buku ini, murni sebagai titik referensi cepat, dan memasangkannya dengan daftar kedua: shortcut pustaka yang tampak seperti kebenaran gratis, masing-masing dengan satu kondisi spesifik di mana kebenaran gratis itu diam-diam berhenti menjadi gratis.

Boilerplate yang Layak Dihafal

Sembilan template dari bab-bab sebelumnya, dan situasi yang seharusnya membuat masing-masing terlintas di pikiran.

Template	Dari	Kapan Memakainya
Eksponensiasi cepat (binary power)	Bab3	Eksponensiasi modular atau perhitungan pangkat besar mana pun di bawah modulus
Buffered fast I/O reader	Bab4	Ukuran input di atas sekitar 10^5 baris, tempat cin/scanf menjadi bottleneck
Baris pragma compiler yang aman	Bab5	Peningkatan kecepatan portabel dengan risiko portabilitas nyaris nol (anak tangga 1 risk ladder)
Kebiasaan casting aman-overflow	Bab6	Perhitungan mana pun yang hasil antaranya bisa melampaui rentang int
Segmented sieve	Bab6	Query bilangan prima pada rentang yang terlalu besar untuk di-sieve dari nol langsung
Pola sentinel memoisasi	Bab9	DP top-down di mana nilai terhitung harus bisa dibedakan dari belum-terhitung
Checklist argumen-pertukaran	Bab10	Memverifikasi aturan greedy yang diusulkan benar-benar optimal, bukan sekadar masuk akal
Tabel pengenal empat-keluarga	Bab11	Memutuskan toolkit klasik mana -- graf, teori bilangan, geometri, atau string -- yang dibutuhkan soal
Loop tujuh-tahap menuju AC	Bab12	Menyusun proses dari percobaan pertama hingga penerimaan akhir pada soal apa pun

Gambar 13.1 — Sembilan templat dari bab-bab sebelumnya, dan situasi yang seharusnya membuat masing-masing teringat.

Gambar 13.1 sengaja bukan berupa kumpulan listing kode -- setiap templat ini sudah dibangun, diverifikasi, dan ditampilkan secara lengkap di bab asalnya. Memperlakukannya sebagai indeks, bukan pengulangan, adalah intinya: begitu bentuk suatu soal lomba cocok dengan salah satu baris ini, kode yang sudah terverifikasi dari bab yang bersangkutan adalah jawabannya, bukan implementasi baru yang ditulis di bawah tekanan waktu.

13.2 __builtin_popcount: Cepat, dan Spesifik terhadap Lebar Bit

Menghitung jumlah bit yang bernilai satu pada sebuah bilangan bulat muncul terus-menerus -- DP bitmask subset-sum, teori bilangan terkait XOR, perbandingan bergaya jarak Hamming. Keluarga __builtin_popcount dari GCC menghitung ini dalam sejumlah instruksi mesin yang kecil dan konstan, jauh lebih cepat daripada loop penghitung bit yang ditulis tangan.

popcount_pitfall_demo.cpp — pemotongan yang spesifik terhadap lebar bit (orisinal)

```
long long big = 1LL << 40; // bit ke-40 aktif, jauh melebihi 32 bit int

// BUG: __builtin_popcount mengharapkan argumen *unsigned int*. Jika
// diberi long long, nilainya diam-diam dipotong ke 32 bit terendah.
int wrongCount = __builtin_popcount((unsigned int)big);

// PERBAIKAN: gunakan varian berlebar-64-bit untuk nilai 64-bit.
int rightCount = __builtin_popcountll(big);
```

Varian 32-bit dan 64-bit tidak dapat saling dipertukarkan, dan kompiler tidak akan memberi peringatan.

__builtin_popcount menerima unsigned int; __builtin_popcountll menerima unsigned long long. Memberikan nilai 64-bit ke versi 32-bit tetap terkompilasi dengan bersih dan begitu saja membuang semua bit di atas posisi 31 sebelum menghitung. Ini tidak terlihat pada nilai uji mana pun yang kebetulan muat dalam 32 bit, dan justru karena itulah bug ini cenderung lolos dari pemeriksaan cepat dan baru muncul pada data uji tersembunyi juri yang lebih besar.

13.3 __gcd: Benar, Kecuali pada Tanda

Faktor persekutuan terbesar (GCD) muncul di seluruh teori bilangan (Bab 11) dan penyederhanaan pecahan. __gcd dari GCC (ekstensi non-standar yang berada di <algorithm>) dan std::gcd standar C++17 (di <numeric>) sama-sama mengimplementasikan algoritma Euclid yang sama, tetapi keduanya berbeda pada satu bentuk masukan tertentu.

gcd_pitfall_demo.cpp — penanganan tanda berbeda antarversi (orisinal)

```
cout << __gcd(12, 18) << "\n";     // 6
cout << __gcd(-12, 18) << "\n";    // 6
cout << __gcd(-12, -18) << "\n";   // -6  <-- negatif!

cout << gcd(-12, 18) << "\n";      // 6  (std::gcd, <numeric>)
cout << gcd(-12, -18) << "\n";     // 6  (selalu non-negatif)
```

__gcd menerapkan langkah modulo-dan-tukar dari algoritma Euclid langsung pada tanda apa pun yang diberikan, tanpa langkah nilai-mutlak terlebih dahulu -- pada dua argumen negatif, ini dapat mengembalikan hasil negatif. std::gcd dispesifikasikan untuk mengambil nilai mutlak dari kedua argumen sebelum melanjutkan, sehingga selalu mengembalikan hasil non-negatif terlepas dari tanda masukan (Lampiran 13.A mengonfirmasi kedua perilaku ini melalui eksekusi langsung).

Fungsi yang benar pada setiap kasus uji positif tetap bisa salah pada masukan yang sengaja dipilih juri.

Jika batasan suatu soal mengizinkan nilai negatif -- secara eksplisit, atau melalui pengurangan yang tandanya tidak diperiksa -- perilaku tanda __gcd persis jenis kasus tepi yang tidak pernah muncul pada pemeriksaan manual cepat dengan bilangan positif kecil, tetapi remeh bagi generator data uji juri untuk disertakan dengan sengaja.

13.4 next_permutation: Hanya Maju dari Titik Ini

Melakukan iterasi atas setiap permutasi dari suatu urutan kecil umum dilakukan dalam verifikasi brute-force (teknik yang sama yang telah dipakai berulang kali oleh buku ini, di Bab 9, 10, dan 12, untuk membangun referensi yang jelas-jelas benar sebagai pembanding). next_permutation dari pustaka standar menyusun ulang suatu rentang menjadi permutasi berikutnya secara leksikografis, mengembalikan false begitu tidak ada lagi permutasi berikutnya -- yang membuat loop do/while sederhana tampak seolah-olah mencakup semuanya.

next_permutation_pitfall_demo.cpp — titik awal itu penting (orisinal)

```
int countPermutations(vector<int> v) {
    int count = 0;
    do { count++; } while (next_permutation(v.begin(), v.end()));
    return count;
}

vector<int> notSorted = {2, 1, 3};
int wrongCount = countPermutations(notSorted);   // 4, bukan 6

vector<int> sorted = {2, 1, 3};
sort(sorted.begin(), sorted.end());              // {1, 2, 3}
int rightCount = countPermutations(sorted);      // 6, benar
```

Dimulai dari {2, 1, 3} -- bukan susunan terkecil secara leksikografis dari ketiga nilai ini -- loop di atas hanya mengunjungi 4 dari 6 total permutasi dari 3 elemen berbeda, karena next_permutation hanya berjalan maju dari titik awalnya dan tidak pernah berputar kembali untuk mengunjungi permutasi yang seharusnya muncul sebelumnya. Mengurutkan menaik terlebih dahulu menjamin titik awal adalah susunan terkecil secara leksikografis, sehingga setiap pemanggilan berikutnya mencapai keenamnya (Lampiran 13.A) (Gambar 13.2 dan 13.3).

> **Kontrak next_permutation mengatakan "berikutnya," bukan "semua" -- membaca hanya nama fungsinya mengundang bug ini.**
>
> Perbaikannya hanya berharga satu baris (pemanggilan sort) dan mudah dinyatakan begitu diketahui, tetapi bug yang dicegahnya tidak menyebabkan crash dan tidak tampak jelas salah -- ia hanya diam-diam menghitung atau mendaftar terlalu sedikit tergantung susunan awal masukan yang kebetulan dimulai, yang sulit disadari hanya dari keluarannya saja.

13.6 Ringkasan Bab

- Sembilan templat pada Gambar 13.1 adalah indeks referensi cepat menuju kode yang sudah terverifikasi dari bab-bab sebelumnya -- inti menghafalnya adalah mengenali situasinya, bukan menurunkan ulang implementasinya di bawah tekanan waktu lomba.
- __builtin_popcount dan __builtin_popcountll tidak dapat saling dipertukarkan: memberikan nilai 64-bit ke varian 32-bit diam-diam memotongnya alih-alih memunculkan galat, dan ini tidak terlihat pada nilai uji mana pun yang kebetulan muat dalam 32 bit.
- __gcd (ekstensi GCC) tidak mengambil nilai mutlak terlebih dahulu dan dapat mengembalikan hasil negatif pada dua argumen negatif, tidak seperti std::gcd, yang dispesifikasikan untuk selalu mengembalikan hasil non-negatif.

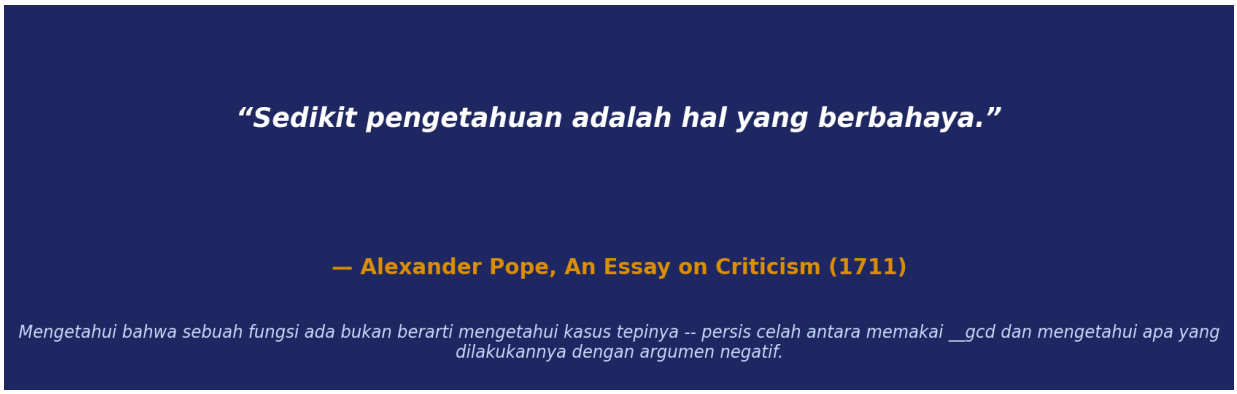
- next_permutation hanya mendaftar permutasi maju dari susunan awalnya -- rentangnya harus diurutkan menaik terlebih dahulu untuk menjamin setiap permutasi terkunjungi.
- Setiap shortcut pada bab ini sungguh-sungguh berguna dan sungguh-sungguh benar di dalam kontrak terdokumentasinya -- disiplin yang dibangun bab ini adalah memeriksa apakah bentuk masukan sebenarnya dari suatu soal berada di dalam kontrak itu, bukan mengasumsikannya.

Fungsi Library yang Terasa Seperti Cheat Code

Masing-masing sungguh berguna -- dan masing-masing punya batasan spesifik yang sering terlewat pada percobaan pertama.

Fungsi	Apa yang Dilakukan	Batasan yang Tak Terbaca Siapa Pun
`__builtin_popcount(x) / popcountll(x)`	Menghitung bit yang bernilai 1 dalam sebuah integer	Spesifik-lebar: versi 32-bit diam-diam memotong nilai 64-bit alih-alih memunculkan error
`__gcd(a, b) (ekstensi GCC)`	Pembagi persekutuan terbesar via algoritma Euclid	Tak mengambil nilai absolut lebih dulu -- bisa mengembalikan hasil negatif untuk input negatif, tak seperti std::gcd
`next_permutation(first, last)`	Memajukan rentang ke permutasi leksikografis berikutnya	Hanya mengenumerasi permutasi dari susunan saat ini dan seterusnya -- rentang harus mulai terurut menaik agar mengunjungi semuanya

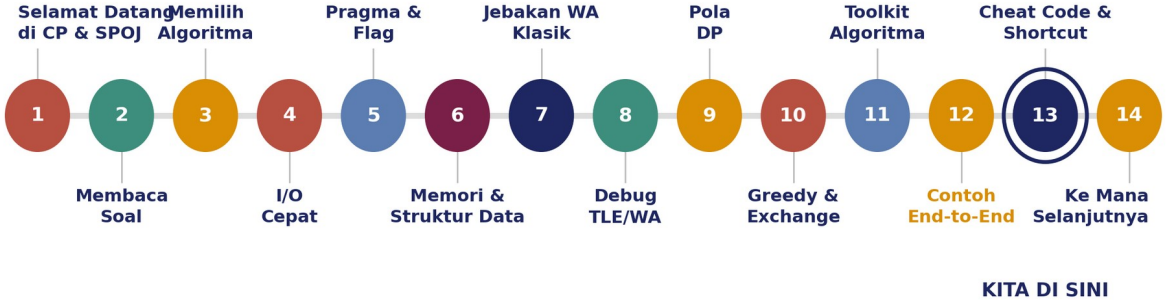
Gambar 13.2 — Setiap shortcut sungguh berguna, dan masing-masing memiliki batas spesifik yang cenderung terlewat pada percobaan pertama.



Gambar 13.3 — Ungkapan Pope, dan celah yang dinamainya antara memakai sebuah fungsi dan mengetahui kasus-kasus tepinya.

Peta Jalan Buku Ini — 14 Bab Dari Submission Pertama Hingga Jadi Insting

Tiap bab membangun keterampilan, bukan solusi — di akhir Anda bisa menghadapi soal SPOJ yang belum pernah dilihat.



Gambar 13.4 — Peta jalan 14 bab buku ini. Kita di sini: Bab 13.

Pertanyaan Tinjauan (Tidak Dinilai)

1. Menggunakan Gambar 13.1, pilih dua baris mana pun dan jelaskan, dengan kata-kata Anda sendiri, susunan kata soal seperti apa yang seharusnya membuat templat baris itu teringat.
2. Jelaskan, menggunakan demonstrasi Bagian 13.2, secara tepat mengapa bug pada pemberian nilai 64-bit ke __builtin_popcount tidak terlihat pada nilai uji kecil dan baru muncul pada nilai besar.
3. Menggunakan Bagian 13.3, jelaskan perbedaan perilaku spesifik antara __gcd dan std::gcd, dan gambarkan bentuk masukan di mana sebuah soal dapat mengungkap perbedaan itu.
4. Jelaskan, menggunakan demonstrasi Bagian 13.4, mengapa memulai next_permutation dari {2, 1, 3} hanya mengunjungi 4 permutasi alih-alih 6, dan mengapa mengurutkan terlebih dahulu memperbaikinya.

Lampiran 13.A — Log Verifikasi

Ketiga program dikompilasi dengan g++ (-O2 -Wall -pedantic) dan dijalankan langsung; keluaran setiap demonstrasi dibandingkan secara manual dengan nilai yang diharapkan sebagaimana dinyatakan pada bagiannya masing-masing (Gambar 13.4).

Transkrip terminal

```
$ g++ -O2 -pedantic -Wall -o popcount_pitfall_demo \
      popcount_pitfall_demo.cpp
$ ./popcount_pitfall_demo
big = 2^40, popcount via 32-bit builtin (truncated): 0
big = 2^40, popcount via 64-bit builtin (correct):   1
small = 0b1011, 32-bit builtin: 3, 64-bit builtin: 3

$ g++ -O2 -pedantic -Wall -o gcd_pitfall_demo gcd_pitfall_demo.cpp
$ ./gcd_pitfall_demo
__gcd(12, 18) = 6
__gcd(-12, 18) = 6
__gcd(-12, -18) = -6
std::gcd(-12, 18) = 6
std::gcd(-12, -18) = 6

$ g++ -O2 -pedantic -Wall -o next_permutation_pitfall_demo \
      next_permutation_pitfall_demo.cpp
$ ./next_permutation_pitfall_demo
Starting from {2,1,3} (not sorted): 4 permutations visited
Starting from {1,2,3} (sorted first): 6 permutations visited
True total permutations of 3 distinct elements (3!): 6
```

Demo popcount mengonfirmasi mekanisme persis yang dijelaskan di Bagian 13.2: 2^40 memiliki tepat satu bit bernilai satu, dilaporkan dengan benar sebagai 1 oleh builtin 64-bit, tetapi dilaporkan sebagai 0 oleh builtin 32-bit karena bit ke-40 terpotong, sementara nilai kecil (0b1011, 3 bit bernilai satu) dilaporkan identik pada keduanya, menunjukkan mengapa bug ini tersembunyi pada data uji kecil. Demo gcd mengonfirmasi __gcd(-12,-18) mengembalikan -6 sementara std::gcd tidak pernah mengembalikan nilai negatif pada masukan yang sama. Demo permutasi mengonfirmasi hitungan 4-berbanding-6 persis seperti yang dijelaskan Bagian 13.4, cocok dengan 3! = 6 hanya setelah masukan diurutkan terlebih dahulu.

Referensi

[1] Pope, A. (1711). An Essay on Criticism. — sumber kutipan pada Bagian 13.6 tentang celah antara pengetahuan parsial dan lengkap terhadap sebuah alat.

[2] Stroustrup, B. (2013). The C++ Programming Language (edisi ke-4). Addison-Wesley — mengenai fasilitas pustaka standar yang dibahas di Bagian 13.3 dan 13.4, termasuk std::gcd dan kontrak terdokumentasi next_permutation.

BAB 14

Ke Mana Selanjutnya

Ini adalah bab terakhir, bukan karena tidak ada lagi yang perlu dipelajari, melainkan karena semua yang tersisa untuk dipelajari kini dibangun di atas fondasi yang telah selesai diletakkan buku ini. Bab ini adalah peta untuk apa yang datang setelahnya: soal mana yang layak dicoba selanjutnya, topik mana yang memberi hasil dari perkakas yang sudah Anda miliki, dan bagaimana kebiasaan yang dibangun sepanjang empat belas bab terakhir terus berlipat ganda jauh setelah buku ini ditutup.

Tujuan Pembelajaran — setelah bab ini Anda mampu:

(1) Menjelaskan strategi konkret berbasis tag untuk memilih soal SPOJ mana yang akan dicoba selanjutnya, tanpa bergantung pada diberi tahu soal mana yang harus dikerjakan. (2) Menyebutkan setidaknya tiga topik di luar cakupan buku ini, dan menjelaskan bab yang sudah ada mana yang menjadi dasar langsung masing-masing. (3) Menjelaskan, dengan kata-kata sendiri, bagaimana keterampilan pemrograman kompetitif buku ini terhubung kembali ke mata kuliah algoritma formal seperti DAA. (4) Menjelaskan setidaknya dua kebiasaan belajar yang berlipat ganda selama berbulan-bulan, bukan hanya membantu dalam satu soal saja. (5) Menyatakan, dalam satu kalimat, apa yang sebenarnya dilatih buku ini -- dan menjelaskan mengapa latihan itu tidak berakhir saat bukunya berakhir.

14.1 Di Mana Posisi Anda Sekarang

Tiga belas bab lalu, buku ini dibuka dengan memperlakukan SPOJ sebagai wilayah asing: selembar aturan, seorang juri yang mengembalikan vonis, sebuah kotak teks kosong menunggu program. Tidak ada yang berubah dari susunan itu. Yang berubah adalah apa yang terjadi antara membaca soal dan mengirimkan solusi: tabel pengenal untuk menentukan toolkit klasik mana yang dibutuhkan sebuah soal (Bab 11), checklist untuk memverifikasi aturan greedy sebelum mempercayainya (Bab 10), insting untuk mengetahui di mana kompleksitas pendekatan naif akan gagal (Bab 6 hingga 9), dan kebiasaan membaca kontrak fungsi pustaka sebelum bersandar padanya (Bab 13). Tidak satu pun dari ini membuat soal asing berikutnya menjadi akrab. Ia membuat soal asing itu bisa ditangani, dan itu selalu menjadi tujuan sebenarnya.

14.2 Memilih Soal SPOJ Berikutnya

Cara umum untuk terhenti setelah menyelesaikan buku seperti ini adalah melihat daftar lengkap soal SPOJ dan tidak tahu harus mulai dari mana. Solusinya bukan daftar soal spesifik untuk dikerjakan -- melainkan strategi untuk menemukan soal Anda sendiri, disesuaikan dengan apa yang sebenarnya dibangun buku ini.

Rencana Latihan Sepuluh Soal

Progresi berdasarkan tag dan tingkat kesulitan, bukan ID soal spesifik -- pilih sendiri dari pencarian tag SPOJ setelah tahu apa yang dicari.

Soal	Tag / Fokus	Tujuan
1-2	Ad hoc / implementasi, mudah	Membangun kembali kecepatan dan kepercayaan diri dengan I/O cepat Bab 4 dan pragma aman Bab 5
3-4	Greedy, mudah-ke-sedang	Melatih checklist argumen-pertukaran Bab 10 pada aturan yang belum pernah dilihat sebelumnya
5-6	Graf (BFS/DFS atau lintasan terpendek), mudah-ke-sedang	Menerapkan tabel pengenal empat-keluarga Bab 11 di bawah tekanan waktu nyata
7-8	Dynamic programming, sedang	Menerapkan pola pengenalan DP Bab 9 pada ruang state yang harus dirancang sendiri
9-10	Pilihan sendiri, sedang-ke-sulit	Pilih soal pada area dari Gambar 15.1 yang belum pernah dipelajari, dan jadikan kebuntuan sebagai intinya

Gambar 14.1 — Progresi berdasarkan tag dan tingkat kesulitan, bukan ID soal spesifik.

Inti dari progresi Gambar 14.1 bukan jumlah soal spesifiknya -- melainkan bentuknya: mulai dari yang paling mudah diverifikasi cepat (ad hoc, mudah), lalu lewati toolkit yang paling baru dan paling menyeluruh dibangun buku ini (greedy, graf, DP), dan akhiri dengan memilih wilayah asing Anda sendiri. Langkah terakhir itu penting lebih dari yang terlihat: setiap teknik dalam buku ini pun awalnya asing, dan nilainya tidak pernah terletak pada teknik spesifik yang dibahas, melainkan pada bagaimana rasanya membuat teknik asing menjadi akrab lewat proses disiplin yang sama, yang bisa diulang kapan pun dibutuhkan.

14.3 Melampaui Buku Ini: Lima Topik yang Layak Dipelajari Berikutnya

Toolkit buku ini -- struktur data, greedy, DP, graf, teori bilangan, geometri, string, dan shortcut pustaka dari Bab 6 hingga 13 -- mencakup sebagian besar dari apa yang muncul pada judge pemrograman kompetitif pada umumnya, tetapi tidak semuanya. Kelima topik di bawah ini adalah lapisan berikutnya yang alami, dipilih secara spesifik karena masing-masing merupakan perluasan langsung dari toolkit yang sudah dibangun buku ini, bukan awal yang baru sama sekali.

Melampaui Buku Ini: Topik yang Layak Dipelajari Berikutnya

Masing-masing dibangun di atas perkakas yang sudah diberikan buku ini -- tak satu pun dimulai dari nol.

Topik	Apa yang Anda Peroleh	Dibangun Dari
Fenwick tree / segment tree	Query jumlah, min, atau max pada rentang dengan update titik atau rentang dalam O(log n), bukan O(n) per query	Kebiasaan struktur data berbasis array Bab 6 dan rekursi dari DP Bab 9
Union-Find (DSU)	Query konektivitas nyaris-O(1) -- "apakah kedua simpul ini dalam grup yang sama" -- dan minimum spanning tree Kruskal	Toolkit graf Bab 11 dan kebiasaan sinyal-pengenalannya
Network flow (max flow, bipartite matching)	Satu kerangka kerja yang memecahkan berbagai soal penugasan dan kapasitas begitu dikenali	BFS Bab 11, dibingkai ulang sebagai pencarian augmenting-path di dalam algoritma flow
Suffix array / suffix automaton	Query substring dan pola berulang di luar yang bisa dijawab efisien oleh KMP pola tunggal (Bab 11)	Toolkit string Bab 11, digeneralisasi ke banyak pola atau semua substring sekaligus
Bitmask DP dan digit DP	Dua bentuk DP spesifik -- state sebagai subset, atau state sebagai posisi digit -- yang membuka keluarga besar soal yang sebelumnya sulit dipecahkan	Pola pengenalan DP Bab 9 dan kebiasaan sentinel memoisasi

Gambar 14.2 — Masing-masing dibangun di atas perkakas yang sudah diberikan buku ini.

Perhatikan pola pada kolom ketiga Gambar 14.2: setiap satu dari kelima topik ini adalah generalisasi dari sesuatu yang sudah dibahas buku ini, bukan subjek baru yang tak berhubungan. Fenwick tree adalah apa yang menjadi sebuah array ketika update titik dan query rentang sama-sama harus cepat. Union-Find adalah apa yang menjadi konektivitas graf ketika Anda perlu bertanya pertanyaan itu berkali-kali seiring sisi ditambahkan satu per satu. Ini layak disebutkan secara eksplisit, karena mudah melihat nama sebuah topik asing dan berasumsi ia membutuhkan mulai dari nol -- jarang sekali begitu.

14.4 Kembali ke DAA: Apa yang Memberi Makan Apa

Bagi pembaca yang juga sedang menempuh mata kuliah Perancangan dan Analisis Algoritma (DAA) dari penulis yang sama, hubungan antara mata kuliah itu dan buku ini berjalan dua arah. Enam belas kuliah DAA membangun fondasi formal -- analisis asimtotik, pembuktian kebenaran, keluarga algoritma klasik -- yang diam-diam diasumsikan setiap bab buku ini. Buku ini, sebaliknya, adalah wujud fondasi formal itu di bawah tekanan waktu, diterapkan pada soal yang bentuk pastinya tidak Anda ketahui sebelumnya dan dipertahankan melawan juri yang tidak akan memberi nilai parsial untuk analisis yang hampir benar. Tak satu pun menggantikan yang lain: DAA mengajarkan Anda membuktikan sebuah algoritma benar dan efisien; buku ini mengajarkan Anda mengenali, di bawah tekanan, bukti mana yang berlaku.

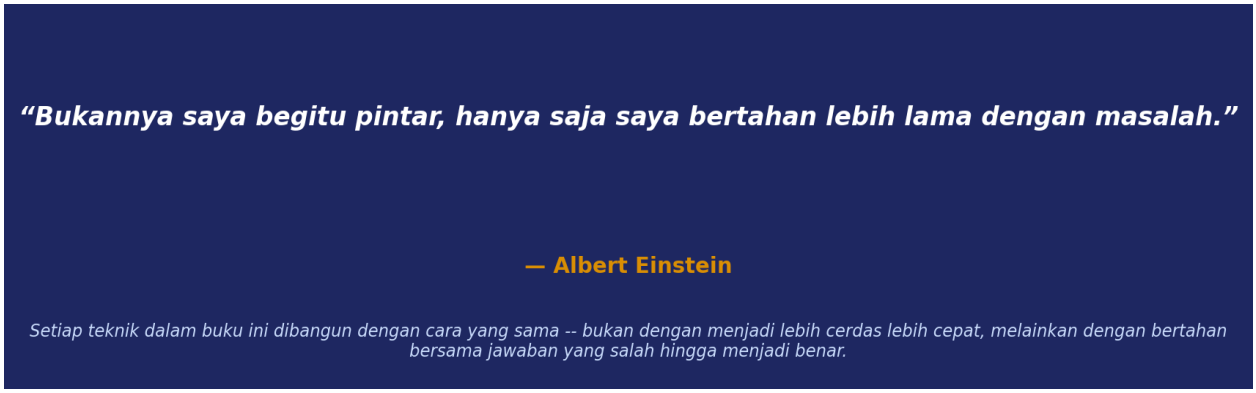
> **Juri adalah uji tekanan bagi pemahaman, bukan pengganti pemahaman itu.**
>
> Algoritma sebuah kuliah, dipahami cukup baik untuk direproduksi dalam situasi terkendali, dan algoritma yang sama, dikenali dengan benar dalam hitungan detik setelah membaca pernyataan soal yang asing, adalah dua tingkat pemahaman yang berbeda dari hal yang sama. SPOJ (dan judge sejenisnya) adalah cara melatih tingkat kedua begitu tingkat pertama sudah terbentuk -- ia tidak pernah menjadi pengganti tingkat pertama.

14.5 Kebiasaan Belajar yang Berlipat Ganda

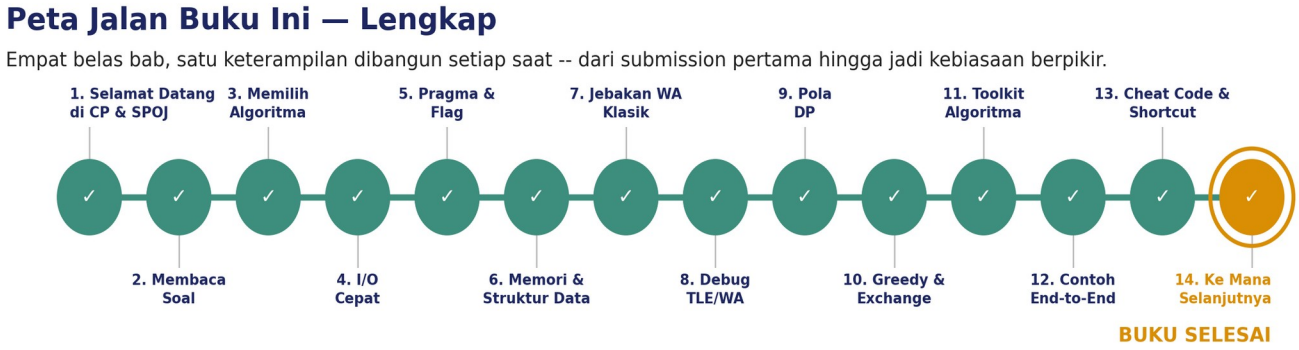
Dua kebiasaan, lebih daripada teknik spesifik apa pun dalam buku ini, layak dibawa terus karena nilainya berlipat ganda semakin lama dipraktikkan. Yang pertama adalah menyimpan catatan pribadi setiap kesalahan yang menghabiskan waktu nyata -- bukan algoritmanya, melainkan kesalahan penalaran spesifiknya: tanda yang tak diperiksa, batas yang tak pernah diuji, kontrak pustaka yang setengah dibaca. Bab 7, 8, 13, dan 14 masing-masing menunjukkan wujud berbeda dari pola yang sama ini, dan pola itu sendiri, begitu diberi nama, mulai menangkap dirinya sendiri lebih awal setiap kalinya. Yang kedua adalah dengan sengaja meninjau ulang soal lama yang sudah terpecahkan berbulan-bulan kemudian dan memecahkannya kembali tanpa melihat solusi lama -- bukan karena jawabannya diragukan, melainkan karena versi diri Anda yang mengerjakannya kedua kali memiliki perkakas yang lebih besar, dan perbandingan itulah tempat pertumbuhan menjadi terlihat (Gambar 14.3 dan 14.4).

14.6 Ringkasan Bab

- Progresi berdasarkan tag dan tingkat kesulitan, bukan daftar soal spesifik, adalah cara yang tepat untuk memilih apa yang dicoba selanjutnya -- Gambar 14.1 memberikan bentuk konkret sepuluh soal untuk memulai.
- Fenwick/segment tree, Union-Find, network flow, struktur suffix, dan DP bitmask/digit adalah lapisan alami berikutnya melampaui buku ini, dan masing-masing merupakan generalisasi dari toolkit yang sudah dibangun di sini, bukan awal yang baru sama sekali.
- Buku ini dan mata kuliah algoritma formal seperti DAA saling memberi makan dalam dua arah: satu memasok pembuktian, yang lain melatih pengenalan bukti mana yang berlaku di bawah tekanan waktu.
- Menyimpan catatan kesalahan penalaran spesifik, dan secara berkala memecahkan ulang soal lama dengan perkakas yang lebih besar, adalah dua kebiasaan yang paling layak dibawa terus karena nilainya berlipat ganda selama berbulan-bulan, bukan hanya dalam satu sesi.
- Juri selalu menjadi alat latihan: keterampilan yang dilatihnya -- membuktikan aturan sebelum mempercayainya, mendaftar kasus tepi, membaca kontrak sebelum bergantung padanya -- bertahan lebih lama daripada juri itu sendiri dan berpindah ke apa pun yang datang setelah buku ini.



Gambar 14.3 — Ungkapan Einstein, dan kebiasaan yang sebenarnya menjadi dasar setiap teknik dalam buku ini.



Gambar 14.4 — Peta jalan 14 bab buku ini, lengkap.

Pertanyaan Tinjauan (Tidak Dinilai)

1. Menggunakan Gambar 14.1, jelaskan mengapa progresinya berakhir dengan "pilihan sendiri" alih-alih kategori keenam yang ditentukan, dan mengapa pilihan spesifik itu penting.
2. Pilih satu baris mana pun dari Gambar 14.2 dan jelaskan, dengan kata-kata Anda sendiri, toolkit bab mana yang sudah ada yang digeneralisasinya dan bagaimana caranya.
3. Dengan kata-kata Anda sendiri, jelaskan klaim Bagian 14.4 bahwa DAA dan buku ini saling memberi makan dalam dua arah, menggunakan satu contoh spesifik teknik dari masing-masing.
4. Jelaskan, menggunakan Bagian 14.5, mengapa menyimpan catatan kesalahan penalaran spesifik lebih berguna dalam jangka panjang daripada menyimpan catatan algoritma apa saja yang telah Anda pelajari.
5. Dalam satu atau dua kalimat, nyatakan apa yang Anda yakini sebenarnya dilatih buku ini untuk Anda lakukan, dengan kata-kata Anda sendiri, terlepas dari teknik tunggal apa pun yang diajarkannya.

Referensi

[1] Einstein, A. — dikutip pada gambar penutup Bagian 14.6 tentang kegigihan sebagai mekanisme sebenarnya di balik setiap teknik yang dibangun buku ini.

[2] Subakti, I. — materi mata kuliah Design and Analysis of Algorithms (DAA), dirujuk sepanjang Bagian 14.4 sebagai pasangan formal bagi latihan terapan buku ini yang bertekanan waktu.

[3] LAMPIRAN A

Katalog Semesta Soal Buku Ini

Lampiran ini mengatalogkan 162 soal representatif dari semesta judge yang menjadi rujukan buku ini, diorganisasikan berdasarkan tingkat kesulitan dan ditandai berdasarkan topik -- peta yang bisa dijelajahi untuk strategi "pilih soal berikutnya sendiri" dari Bab 14. Tidak ada solusi, petunjuk menuju solusi, atau detail implementasi khas judge yang muncul di sini: hanya nama publik sebuah soal, tag topiknya, dan posisinya pada tangga kesulitan. Gunakan seperti yang dimaksudkan Gambar 14.1 -- sebagai cara menemukan soal berikutnya sendiri pada topik yang asing, bukan sebagai daftar centang untuk diselesaikan dari atas ke bawah.

> **Cara membaca katalog ini**
>
> "Kode" adalah pengenal yang akan Anda cari di sebuah judge (soal SPOJ dengan kode singkatnya, soal E-Olymp dengan ID numeriknya). "Topik" sering mencantumkan lebih dari satu tag -- kebanyakan soal nyata berada di persimpangan dua atau tiga ide klasik, dan ini sendiri layak diperhatikan: tabel pengenal empat-keluarga dari Bab 11 adalah titik awal pencarian, bukan satu label yang sepenuhnya mendeskripsikan sebuah soal.

Mudah · 71 soal

Soal di mana langkah pengenalan (tabel empat-keluarga Bab 11) adalah keseluruhan tantangannya -- begitu toolkit klasik yang tepat teridentifikasi, implementasinya singkat.

Kode	Nama	Topik
ADDREV	Adding Reversed Numbers	Ad Hoc
MGSINFOA	Big J Walks Around the School	Ad Hoc / Array Sum
LONER	The Loner	Ad Hoc / Greedy / Parity
MUSVIOMG	Robert Plays The Viola	Ad Hoc / Hash Map
PCSKREAB	Minesweeper	Ad Hoc / Simulation
PCSKREAA	The 3n + 1 Problem	Ad Hoc / Simulation + Memoization
MGSINFOB	Russian Year 5 Problem	Array / Complement Lookup
MPPZC	Pierwiastkowanie (Square Root)	Big Integer / Binary Search
BOOKS1	Copying Books	Binary Search / Greedy
STACKOFBOXES	Stacks of Boxes	Binary Search / Greedy
BT69	BitPlay69	Bitwise / Greedy
XUDOKU	Where Proofless Assumption	Bitwise / Greedy
CMPLS	Complete the Sequence!	Classical
TEST	Life, the Universe, and Everything	Classical
TETRA	Sphere in a Tetrahedron	Classical
CRYPTO1	The Bytelandian Cryptographer (Act I)	Classical
CRYPTO2	The Bytelandian Cryptographer (Act II)	Classical
SHPATH	The Shortest Path	Classical
ONP	Transform the Expression	Classical
TRICENTR	Triangle From Centroid	Classical
SP2004X	Bankomat	DP + Bitset
DEV_CP	Ambitious XOXO	Struktur Datas (BIT/Fenwick Tree)
HMRO	Help the Military Recruitment Office!	Struktur Datas (Union-Find + Hash Map)
EKO	Eko	Divide & Conquer
ACODE	Alphacode	Dynamic Programming
COINS	Bytelandian Gold Coins	Dynamic Programming
WTPZ2001G	Diamonds	Dynamic Programming
MORIA	Mines of Moria	Dynamic Programming
PARTY	Party Schedule	Dynamic Programming
BYTESM2	Philosophers Stone	Dynamic Programming
TABY	TABTABTAB	Dynamic Programming
KNAPSACK	Tutorial 3321 KNAPSACK - The Knapsack Problem	Dynamic Programming
PRIZES	Precious Prizes	Dynamic Programming (0/1 Knapsack)
PIR	Pyramids	Geometry
BSHEEP	Build the Fence	Geometry (Convex Hull)

Kode	Nama	Topik
BLINNET	Bytelandian Blingors Network	Graph / MST (Kruskal)
PT07Y	Is It a Tree	Graph Algorithms
NAKANJ	Minimum Knight Moves !!!	Graph Algorithms
LABYR1	Labyrinth	Graph Algorithms / DFS / Tree Diameter
WORDS1	Play on Words	Graph Theory / Euler Path
SCYPHER	Substitution Cipher	Graph Theory / Topological Sort
BAISED	Biased Standings	Greedy
CANDY	Candy I	Greedy
DEFKIN	Defense of a Kingdom	Greedy
E-Olymp 10280	Journey	Greedy
E-Olymp 10281	Columns	Greedy
BUSYMAN	I Am Very Busy	Greedy
BALIFE	Load Balancing	Greedy
MASUM	Maximum Sum of the Array	Greedy
MWPZ017	Card Stacking	Greedy / Longest Increasing Subsequence
SPAM_ATX	Internet Spamming	Greedy / Segment Tree / Sorted Structure
CHOCOLA	Chocolate	Greedy / Sorting
HASHIT	Hash it!	Hash Table / Hashing
TLPNGEM2	Teleporters and Gems II	Interactive / Math
CPDUEL5A	Comet Number	Math / Number Theory
ROBOWORLD	Robot World	Math / Number Theory
STONE	Lifting the Stone	Math / Plane Geometry
SLNCRD	Selling Greeting Cards	Math, Combinatorics, Modular Arithmetic
CEQU	Crucial Equation	Number Theory
E-Olymp 273	Modular Exponentiation	Number Theory
FCTRL	Factorial (Trailing Zeros)	Number Theory
LASTDIG	The Last Digit	Number Theory
FLT	Unique Sequence	Number Theory / Fermat's Little Theorem
EQBOX	Equipment Box	Plane Geometry
TEWBMCUL	Hugo's Tennis Training	Prefix Sum
HOTELS	Hotels Along the Croatian Coast	Searching & Sorting
SBANK	Sorting Bank Accounts	Searching & Sorting
TUTORIAL	TUTORIAL 16579 PAIRS1 - Count the Pairs	Searching & Sorting
FESTIVAL	Crowded Music Festival	Sliding Window Maximum
NHAY	A Needle in the Haystack	Strings / KMP
BPF1	Brenden Politeness Filter	Text-processing

Sedang · 50 soal

Soal yang membutuhkan satu wawasan tidak-jelas di atas toolkit klasik -- state DP yang tidak biasa, aturan greedy yang butuh checklist argumen-pertukaran (Bab 10), atau kasus batas yang cenderung terlewat pada percobaan pertama.

Kode	Nama	Topik
CATOW	Cairo Tower	Simulation / Discrete-Event / Priority Queue
POUR1	Pouring Water	Ad Hoc
BSS	Single buffer (cannibalize input) Halved BSS footprint. Less	Ad Hoc
FCTRL2	Small Factorials	Ad Hoc
TRIMOD2	The Triangle of Pascal	Number Theory / Combinatorics
NQUEEN	Yet Another N-Queen Problem	Backtracking
E-Olymp 4894	Interesting Sequence	Bit Manipulation
CMEXPR	Complicated Expressions	Classical
IKEYB	I-Keyboard	Classical
ARITH	Simple Arithmetics	Classical
SBSTR1	Substring Check (Bug Funny)	Classical
MMIND	The Game of Master-Mind	Classical
PALIN	The Next Palindrome	Classical
CODE1	Secret Code	Complex Number Arithmetic / Number Theory
MORIA2	Mines of Moria II	DP / Convex Hull Trick
CPDUEL5C	Phasmophobia	DP / Permutations / Bitmask
HORRIBLE	Horrible Queries	Struktur Datas / Lazy Segment Tree
FAKETSP	Traveling Salesman	Distance Tracking
AGGRCOW	Aggressive Cows	Divide & Conquer
INVCNT	Inversion Count	Divide & Conquer
MCOINS	Coins Game	Dynamic Programming
WTPZ2002C	Diamond Collector	Dynamic Programming / Greedy Transition
DIEHARD	Die Hard	Dynamic Programming
E-Olymp 10296	Recursive Function 1	Dynamic Programming
TSPMOHUG	Hu-Gi-Oh	Dynamic Programming
MAXSUB	Maximum Subset of Array	Dynamic Programming
TRIP	Trip	Dynamic Programming
JRIDE	Jill Rides Again	Dynamic Programming / Max Subarray + Index Tracking
MORIA3	Watching over the Mines of	Dynamic Programming / Tree / Set Cover
BITMAP	Bitmap	Graph Algorithms
PPATH	Prime Path	Graph Algorithms / BFS on State Space
DST	See you again?	Graph Theory + DP (Subsequence on Tree)
RDR2_1	Escape the New America	Graph Theory / BFS

Kode	Nama	Topik
ARRANGE	Arranging Amplifiers	Greedy
ANADIV	Anagrams and Divisibility	Greedy / Bitmask DP / Number Theory
TOHU	Help Tohu	Math / Sequence Sum Formula
CPDUEL5B	Asacoco Prescription	Math, Number Theory, Greedy
MUL	Fast Multiplication	Mathematics / FFT
AFS	Amazing Factor Sequence	Number Theory
BSEXP	Base Exploration	Number Theory
ETF	Euler Totient Function	Number Theory
PRIME1	Prime Generator	Number Theory
CRYPTO3	The Bytelandian Cryptographer (Act III)	Number Theory / Brainfuck
GCD2	GCD2	Number Theory / GCD of Large Numbers
UPDTARR	Arithmetic Progression	Number Theory / Sqrt Decomposition
JPM	Just Primes	Number Theory, Dynamic Programming
SQPENT	Square Pentagon	Number Theory, Math
JNEXT	Just Next !!!	Searching & Sorting / Next Permutation
DIGSHIFT	Digit Shifts	Segment Tree / Hashing / Fast I/O
PROPKEY	The Proper Key	Simulation / Bitset Arrays

Sulit · 35 soal

Soal yang menggabungkan dua toolkit atau lebih, atau membutuhkan struktur yang kurang umum yang hanya disinggung buku ini (segment tree, teori bilangan lanjut, algoritma graf yang lebih berat) -- lihat topik "melampaui buku ini" Bab 14.

Kode	Nama	Topik
BATTLECRY	Battle Of The Bastards	Berlekamp-Massey / Lagrange Interpolation
SUDOKU	Sudoku	Backtracking / (Dancing Links / DLX)
WM06	Soccer Choreography	Bioinformatics / Signed Permutation Sorting / Hannenhalli-Pevzner
DIRVS	Direct Visibility	Classical
HOTLINE	Hotline	Classical
BULK	The Bulk!	Classical
CRYPTO4	The Bytelandian Cryptographer (Act IV)	Classical
CFONISMG	Seating Plan	Combinatorial Optimization / Backtracking + Bitmask DP
SPY3	Spy Reloaded	Combinatorics / Generating Functions / Modular Arithmetic
RUNAWAY	Run Away	Computational Geometry (Voronoi / Delaunay)
GSS1	Can You Answer These Queries I	Struktur Datas
GSS3	Can You Answer These Queries III	Struktur Datas
MKTHNUM	K-th Number	Struktur Datas / Merge Sort Tree

Kode	Nama	Topik
LIS2	Another Longest Increasing Subsequence Problem	Dynamic Programming
DSUBSEQ	Distinct Subsequences	Dynamic Programming
VILUNIT	Village Unity	Geometry / Graph / MST
CYCLINGS	Distinct Cyclings	Geometry, Combinatorics, Hashing
FASTFLOW	Maximum Flow	Graph Algorithms / Dinic's Algorithm
CHIPSLL	Chips Challenge	Graph Theory / Shortest Path (Dial's Algorithm)
HBD	Birthday Present	Graph Theory, Binary Search, Struktur Datas
AHASHREV	The Revenge Of Anti Hash	Hashing / Constructive Algorithms
CBNAK	Charu and Coin Distribution	Math
CIVIL	Civil Engineering	Math
CRCLE_UI	Colorful Circle (EASY)	Math / Chromatic Polynomial
MONONUM	Monotonous Numbers	Math / Ratio of Decreasing to Increasing
ASSIEVE	A Simple Sieve	Min25 Sieve
ANAGCD	Anagrams and GCD	Number Theory
GCDPRDSM	GCD Product Sum	Number Theory
PRLCM	Personal LCM	Number Theory & Math
JPM2	Just Primes II	Number Theory / FFT
HISTOGRA	Largest Rectangle in a Histogram	Searching & Sorting
LCS	Longest Common Substring	Searching & Sorting
ABCDEF	ABCDEF	Searching & Sorting / Meet in the Middle
MARTIA_AURIS	Martia Auris 3020	String Matching / Bioinformatics
DISUBSTR	Distinct Substrings	Strings / Suffix Array + LCP

Sangat Sulit · 6 soal

Ujung terjauh katalog: soal yang membutuhkan iterasi serius bahkan bagi penyelesai berpengalaman, sering membutuhkan topik dari daftar lapisan-berikutnya Bab 14 (segment tree, network flow, heavy-light decomposition, FFT) secara penuh.

Kode	Nama	Topik
CONTEST	Fixed Partition Contest Management	Branch & Bound
CHALLENGE	CHALLENGE 155 SOLVING - Solving the Puzzle	Challenge
GSS2	Can You Answer These Queries II	Struktur Datas
QTREE	Query on a Tree	Struktur Datas / Heavy-Light Decomposition
SEGTREE	Segment Tree	Struktur Datas / Nearest Neighbor MST
ASUMEXTR	A Summatory (Extreme)	Math, Lagrange Interpolation, Number Theory

LAMPIRAN B

Lembar Contekan Kompleksitas

Lampiran ini mengumpulkan tabel referensi Big-O yang dibangun sedikit demi sedikit oleh bab-bab buku ini menjadi satu halaman yang mudah dijelajahi. Dimaksudkan untuk dibaca mundur dari batasan (constraint) sebuah soal, bukan maju dari nama sebuah algoritma -- cara yang dijelaskan Bagian B.2 adalah cara penyelesai berpengalaman benar-benar menggunakan tabel seperti ini di bawah tekanan waktu.

B.1 Referensi Laju Pertumbuhan

Setiap baris di bawah adalah kelas kompleksitas yang benar-benar digunakan buku ini, beserta ukuran input tempatnya masih nyaman berada di bawah batas waktu judge yang tipikal satu-hingga-dua detik, dan bab tempat kelas tersebut dibangun secara detail.

Kompleksitas	Nama	n yang nyaman	Algoritma tipikal	Bab
O(1)	Konstan	n berapa pun	Akses array/hash, aritmetika	Bab 6
O(log n)	Logaritmik	hingga 10^18	Binary search, operasi BST seimbang	Bab 6, 11
O(√n)	Akar kuadrat	hingga 10^14	Primalitas trial-division, sqrt decomp.	Bab 11, 13
O(n)	Linear	hingga 10^8	Sekali lintas, prefix sum, penghitungan	Bab 4, 7
O(n log n)	Linearitmik	hingga 10^6–10^7	Sorting, greedy berbasis sort, merge	Bab 10
O(n√n)	Akar-n kali n	hingga 10^5	Struktur query sqrt-decomposition	Bab 13
O(n^2)	Kuadratik	hingga 5.000–10.000	Nested loop, DP 2-D sederhana	Bab 9
O(n^2 log n)	Kuadratik-log	hingga 2.000–3.000	n^2 dengan sort atau heap di dalamnya	Bab 9, 10
O(n^3)	Kubik	hingga 300–500	Floyd–Warshall, DP kubik, perkalian matriks	Bab 11
O(2^n)	Eksponensial	hingga 20–25	Enumerasi subset, bitmask DP	Bab 9
O(n!)	Faktorial	hingga 10–11	Brute force permutasi	Bab 9

B.2 Membaca Batasan (Constraint) Secara Mundur

Batasan sebuah soal pada judge adalah pesan tentang kelas kompleksitas mana yang diharapkan, dan membaca pesan itu sebelum menulis kode apa pun adalah salah satu kebiasaan yang paling sering diulang buku ini (Bab 3, 6, dan 9 masing-masing kembali membahasnya). Aturan praktis: mesin judge modern mengeksekusi sekitar 10^8 hingga 10^9 operasi sederhana per detik, sehingga algoritma yang

jumlah operasinya berada dalam rasio itu terhadap n pada batasan aman, dan yang tidak adalah TLE yang menunggu untuk terjadi.

Jika batasannya...	...kompleksitas ini aman	...karena
n ≤ 11	O(n!) atau O(2^n · n)	Pencarian permutasi penuh masih terjangkau
n ≤ 25	O(2^n)	Enumerasi subset / bitmask
n ≤ 500	$O(n^3)$	DP kubik, gaya Floyd–Warshall all-pairs
n ≤ 5.000	$O(n^2)$	Solusi nested-loop sederhana atau DP 2-D
n ≤ 10^6	O(n log n)	Greedy berbasis sort, sebagian besar algoritma graf
n ≤ 10^8	O(n) atau O(n log n)	Scan sekali lintas, faktor konstanta yang hati-hati
n ≤ 10^18	O(log n) atau O(1)	Binary search, matematika modular/bentuk tertutup

Batasan adalah spesifikasi, bukan hiasan.

Soal yang menyatakan "1 ≤ n ≤ 10^5" bukanlah kebetulan -- itu memberi tahu, secara eksplisit, bahwa solusi $O(n^2)$ (10^10 operasi) tidak akan selesai tepat waktu dan solusi O(n log n) (sekitar 1,7 × 10^6 operasi) akan selesai dengan nyaman. Bab 3 membangun kebiasaan ini dari nol; tabel ini adalah yang perlu dirujuk begitu kebiasaan itu sudah otomatis.

B.3 Biaya Operasi Struktur Data

Kelas kompleksitas mendeskripsikan algoritma; tabel di bawah mendeskripsikan struktur data yang menjadi fondasi algoritma-algoritma tersebut di bab-bab buku ini, dan biaya setiap operasi intinya.

Struktur	Akses	Pencarian	Sisip / Update	Bab
Array / vector	O(1)	O(n)	O(n) (O(1) di akhir)	Bab 6
Array terurut	O(1)	O(log n)	O(n)	Bab 6, 11
Hash map / set	O(1) rata-rata	O(1) rata-rata	O(1) rata-rata	Bab 6
std::set / std::map	O(log n)	O(log n)	O(log n)	Bab 6
Fenwick tree (BIT)	—	O(log n) query prefix	O(log n) update titik	Bab 13, 15
Segment tree	—	O(log n) query rentang	O(log n) update titik/rentang	Bab 15
Union-Find (DSU)	—	O(α(n)) find	O(α(n)) union	Bab 15

B.4 Satu Aturan yang Layak Dihafal

Jika tidak ada yang lain di halaman ini yang dihafal, inilah satu aturan yang perlu dipegang: lihat batasan terbesar pada soal, tentukan kelas kompleksitas mana yang nyaman untuknya menggunakan Bagian B.2, dan baru setelah itu mulai memikirkan algoritma mana yang benar-benar mencapai

kompleksitas tersebut. Bab 3 hingga 14, dalam satu pengertian, adalah enam belas jawaban berbeda untuk paruh kedua proses itu -- tetapi paruh pertama, membaca batasan, adalah yang membuat mungkin untuk mengajukan pertanyaan yang tepat sebelum menulis satu baris kode pun.

LAMPIRAN C

Daftar Istilah

Lampiran ini mengumpulkan istilah pemrograman kompetitif dan algoritma yang digunakan buku ini, dalam satu daftar alfabetis, masing-masing ditandai dengan bab (atau lampiran) tempat istilah tersebut diperkenalkan dan dijelaskan secara semestinya dalam konteks. Gunakan sebagai rujukan cepat saat membaca, bukan sebagai pengganti membaca bab itu sendiri -- definisi satu kalimat adalah pengingat, bukan pengganti contoh-contoh yang dikerjakan lengkap yang membuat sebuah istilah benar-benar dapat dipakai di bawah tekanan waktu.

> **Cara menggunakan daftar istilah ini**
>
> Setiap entri sengaja menyebutkan bab spesifik tempatnya berada: definisi saja jarang bisa diterapkan ke soal baru, tetapi definisi ditambah bab yang mengerjakannya biasanya bisa. Jika sebuah istilah di sini masih terasa asing bahkan setelah membaca entrinya, itu adalah sinyal untuk mengunjungi kembali babnya, bukan kekurangan pada halaman ini.

Accepted (AC) *[Bab 1]*

Verdict yang dikembalikan judge ketika sebuah submission menghasilkan output yang benar dalam batas waktu dan memori, untuk setiap test case yang dimiliki judge.

Ad Hoc *[Bab 2, 3]*

Soal tanpa algoritma bernama standar di baliknya -- kesulitannya sepenuhnya ada pada ketelitian membaca dan ketelitian implementasi, bukan pada mengingat suatu teknik.

Amortized complexity *[Bab 6]*

Biaya yang dinyatakan per operasi, dirata-rata sepanjang keseluruhan rangkaian operasi, meskipun satu operasi tertentu dalam rangkaian itu bisa jadi lebih mahal (mis. vector push_back bersifat amortized O(1)).

Asymptotic complexity *[Bab 3]*

Bagaimana waktu eksekusi atau memori sebuah algoritma tumbuh seiring ukuran input membesar tanpa batas, ditulis dalam notasi Big-O dan tidak bergantung pada mesin tertentu.

Backtracking *[Bab 9]*

Strategi pencarian yang membangun solusi secara bertahap dan meninggalkan ("backtrack" dari) solusi parsial segera setelah tidak mungkin lagi diperluas menjadi solusi yang valid.

Big-O notation *[Bab 3]*

Cara mendeskripsikan laju pertumbuhan kasus terburuk sebuah algoritma yang mengabaikan faktor konstanta dan suku orde-lebih-rendah -- O(n) dan O(2n + 5) mendeskripsikan kelas yang sama.

Binary search *[Bab 6, 11]*

Teknik pencarian yang berulang kali membelah dua ruang pencarian yang terurut, mencapai waktu O(log n); juga berlaku pada ruang jawaban monoton apa pun ("binary search pada jawaban").

Bitmask DP *[Bab 9, 14]*

Dynamic programming di mana state-nya mencakup sebuah bitmask yang merepresentasikan subset dari suatu himpunan item -- praktis hanya ketika jumlah item masih kecil (kira-kira n ≤ 20-25).

Brute force *[Bab 3, 9]*

Solusi yang mencoba setiap kemungkinan secara langsung, tanpa jalan pintas algoritmik -- benar berdasarkan konstruksinya, tetapi hanya cukup cepat ketika ukuran input kecil.

Compile Error (CE) *[Bab 1]*

Verdict judge yang berarti source code yang dikirim gagal dikompilasi -- judge bahkan tidak pernah menjalankan programnya.

Complexity class *[Bab 3, Lampiran B]*

Kategori laju pertumbuhan (O(1), O(log n), O(n), O(n log n), O(n^2), dan seterusnya) tempat waktu eksekusi sebuah algoritma berada.

DFS / BFS *[Bab 11]*

Depth-First Search dan Breadth-First Search: dua urutan penelusuran graf yang paling fundamental, DFS melalui stack (seringkali rekursi), BFS melalui queue, keduanya O(V + E).

Dynamic Programming (DP) *[Bab 9]*

Teknik untuk soal dengan overlapping subproblems dan optimal substructure, diselesaikan dengan menyimpan (memoisasi) jawaban subproblem alih-alih menghitungnya ulang.

Edge case *[Bab 7, 8]*

Input pada batas terluar yang diizinkan oleh batasan sebuah soal (input kosong, satu elemen, ukuran maksimum, semua nilai sama) -- jauh lebih mungkin menyingkap sebuah bug.

Exchange argument *[Bab 10]*

Teknik pembuktian untuk algoritma greedy: tunjukkan bahwa solusi optimal mana pun dapat diubah, tanpa membuatnya lebih buruk, menjadi solusi yang cocok dengan pilihan greedy.

Fast I/O *[Bab 4]*

Teknik (mis. menonaktifkan sinkronisasi stream C++, atau scanf/printf) yang menghilangkan overhead bawaan cin/cout, sering diperlukan saat membaca input berukuran besar.

Fenwick tree (BIT) *[Bab 13, 14]*

Binary Indexed Tree: struktur ringkas yang mendukung update titik dan query prefix-sum dalam O(log n), lebih sederhana diimplementasikan dibanding segment tree penuh.

Greedy algorithm *[Bab 10]*

Algoritma yang membuat pilihan terbaik secara lokal pada setiap langkah dan tidak pernah mempertimbangkannya kembali -- benar hanya untuk soal yang memenuhi greedy-choice property dan optimal substructure.

Hashing *[Bab 6]*

Memetakan data ke sebuah nilai berukuran tetap (hash) untuk pencarian, pengecekan kesetaraan, atau deduplikasi rata-rata cepat -- O(1) rata-rata, bukan kasus terburuk.

Judge *[Bab 1]*

Sistem otomatis (SPOJ, Codeforces, dan situs sejenis) yang mengompilasi sebuah submission, menjalankannya terhadap test case tersembunyi, dan mengembalikan sebuah verdict.

Memoization *[Bab 9]*

Menyimpan cache hasil pemanggilan fungsi berdasarkan argumennya, sehingga pemanggilan berulang dengan argumen yang sama langsung mengembalikan hasil alih-alih menghitung ulang.

Number theory (dalam CP) *[Bab 11]*

Cabang toolkit buku ini yang mencakup primalitas, GCD/LCM, aritmetika modular, dan invers modular -- blok bangunan yang terus berulang di balik banyak soal "Classical".

Overflow *[Bab 7]*

Kesalahan yang terjadi ketika hasil aritmetika melebihi jangkauan tipe integernya -- penyebab Wrong Answer yang sering dan senyap dalam pemrograman kompetitif.

Prefix sum *[Bab 6]*

Total berjalan yang dihitung di muka, yang mengubah query jumlah-rentang berulang menjadi lookup O(1) setelah setup satu kali sebesar O(n).

Runtime Error (RE) *[Bab 1, 7]*

Verdict judge yang berarti program crash saat dieksekusi -- penyebab umum termasuk akses array di luar batas, pembagian dengan nol, dan stack overflow akibat rekursi yang terlalu dalam.

Segment tree *[Bab 14]*

Struktur pohon biner yang mendukung query rentang dan update rentang atau titik dalam O(log n), lebih umum dibanding Fenwick tree dengan konsekuensi kode yang lebih banyak.

Stress testing *[Bab 12]*

Menghasilkan banyak test case acak berukuran kecil dan membandingkan output solusi cepat terhadap solusi brute-force lambat yang jelas benar, untuk menangkap bug yang terlewat oleh satu test case buatan tangan.

Time Limit Exceeded (TLE) *[Bab 1, 3, 8]*

Verdict judge yang berarti program pada prinsipnya berjalan benar tetapi tidak selesai dalam batas waktu -- hampir selalu masalah kompleksitas, bukan bug kecil.

Time complexity *[Bab 3]*

Lihat Complexity class -- secara spesifik laju pertumbuhan waktu eksekusi sebuah algoritma sebagai fungsi dari ukuran input.

Two pointers *[Bab 6]*

Teknik yang menggunakan dua indeks yang bergerak melintasi sebuah sequence (sering dari ujung yang berlawanan, atau keduanya maju) untuk menyelesaikan soal O(n) tertentu yang secara naif akan diselesaikan dalam O(n^2).

Union-Find (DSU) *[Bab 14]*

Disjoint Set Union: struktur yang melacak partisi elemen menjadi himpunan-himpunan terpisah, mendukung operasi find dan union yang mendekati O(1) melalui path compression dan union by rank.

Wrong Answer (WA) *[Bab 1, 7, 8]*

Verdict judge yang berarti program berjalan dan selesai tepat waktu, tetapi menghasilkan output yang salah pada setidaknya satu test case tersembunyi.