Conquering SPOJ

First Edition

Irfan Subakti

Profitina

profitina.com

TABLE OF CONTENTS

Preface

Competitive programming is not a course of its own here. SPOJ is where the material of Design and Analysis of Algorithms (DAA) stops being theory and starts being tested: the same sorting, searching, dynamic programming and graph algorithms, now under a judge's time limit. This book is the practical companion to that course -- 14 chapters that take you from a first submission to a habit of mind.

Before you touch a single algorithm, you need to understand exactly what a judge is checking — and why "it works on my machine" is the single most dangerous sentence in competitive programming. This book turns algorithm knowledge already learned elsewhere into fast, correct submissions under a strict clock.

Conquering SPOJ opens with what SPOJ (Sphere Online Judge) actually is and exactly what its automated pipeline checks about a submission, then works through reading a problem like a judge does, picking the right algorithm for the stated constraints, fast I/O in C/C++, compiler pragmas and portability risk, and memory and data-structure choices under a byte budget. A run of chapters on classic implementation gotchas and on debugging TLE and WA without the judge explaining why is followed by algorithmic pattern chapters — dynamic programming, greedy and exchange-argument problems, and a toolkit chapter spanning graph, number theory, geometry, and string algorithms. The book closes with a worked end-to-end example of getting to Accepted, a chapter of responsibly-used cheat codes and shortcuts, and a chapter on where to go next — backed by three reference appendices: a catalog of the book's problem universe, a complexity cheat-sheet, and a glossary of terms.

By the time you reach the last page, you should be able to:

- Read a competitive-programming problem the way a judge's test data will actually exercise it, not the way it first sounds.
- Choose the right algorithm for the stated time and memory constraints, not just any algorithm that produces a correct answer.
- Write fast, portable C/C++ I/O and avoid the compiler-flag and portability traps that silently cost points.
- Diagnose Time Limit Exceeded and Wrong Answer verdicts systematically, without the judge telling you what went wrong.
- Apply dynamic programming, greedy, graph, number-theory, geometry, and string-algorithm patterns to real judge problems.
- Use the book's appendices — a problem catalog, a complexity cheat-sheet, and a glossary — as a working reference during practice.

Who this book is written for: students and self-learners who already know core algorithms and data structure and want to convert that knowledge into fast, correct submissions on SPOJ and similar online judges.

A word on method. Every code example in this book was compiled and run before it was written down, and the appendices carry the verification logs to prove it. Where a number appears, it came from a measurement rather than from memory. If you find an error, or a passage that could be clearer, I would genuinely like to hear about it — that is how the next edition gets better.

Surabaya, September 2026

Irfan Subakti
yifana@gmail.com

Profitina

profitina.com

CHAPTER 1

Welcome to Competitive Programming and SPOJ

Before you touch a single algorithm, you need to understand exactly what a judge is checking — and why "it works on my machine" is the single most dangerous sentence in competitive programming.

Learning Objectives — after this chapter you will be able to:

(1) Explain what an online judge like SPOJ actually does with a submission: compile, run against hidden tests, check correctness/time/memory. (2) Name the six standard verdicts (AC/WA/TLE/MLE/RE/CE) and what each one tells you about your bug. (3) Explain why passing your own sample test case proves far less than beginners assume. (4) Describe how this book uses real SPOJ problems as examples while respecting a strict, explicit confidentiality boundary. (5) Compile and run a first C++ program under competitive-programming conventions (fast I/O, a classic sieve) and read its verification transcript.

One Toolchain, Three Operating Systems — Windows 11, macOS, Ubuntu

OS	One-time install	Compile & run (identical everywhere)
Windows 11	MinGW-w64: winget install BrechtSanders.WinLibs (or WSL2: wsl --install)	g++ -Wall -Wextra -std=c++17 -O2 -o prog prog.cpp then prog.exe / ./prog
macOS	xcode-select --install (Apple clang answers as g++)	g++ -Wall -Wextra -std=c++17 -O2 -o prog prog.cpp then ./prog
Ubuntu/Linux	sudo apt update && sudo apt install build-essential	g++ -Wall -Wextra -std=c++17 -O2 -o prog prog.cpp then ./prog

Every example in this book runs identically on Windows 11, macOS, and Ubuntu/Linux; commands differ only at install time. Pick your row once and follow along on your own laptop.

1.1 What Is Competitive Programming, and What Is SPOJ?

Competitive programming is the sport of solving well-specified algorithmic problems, under a strict time limit, with a program whose correctness and efficiency are checked automatically by a machine called a judge. There is no partial credit for elegant code, no professor to explain what you meant, and no benefit of the doubt for an answer that is "almost right." Either your program produces the exact expected output, within the time and memory the problem allows, on every hidden test case the judge throws at it — or it does not, and you find out which specific way it failed.

SPOJ (Sphere Online Judge) is one of the oldest and most widely used online judges in the world, hosting many thousands of problems spanning every classical algorithmic topic: dynamic programming, greedy algorithms, graph theory, number theory, computational geometry, string algorithms, data structures, and more. It rewards exactly the skills this course's Design and Analysis of Algorithms (DAA) lectures teach — and this book exists because those two things, taught together, reinforce each other far more than either does alone. The course's own SPOJ problem set spans 257 solved problems across nearly every one of those categories; Appendix 15.A catalogs all of them by name, ID, and topic as a reference for what to attempt next.

This book is not a walkthrough of those 257 problems. Reading someone else's accepted solution teaches you what one specific problem's answer looks like; it does not teach you how to recognize the next problem you have never seen. This book is organized the other way around: around the transferable skills — reading constraints, picking an algorithm family, avoiding classic implementation

traps, debugging a rejected submission — that let you attack an unfamiliar SPOJ problem cold and have a real shot at Accepted.

1.2 What a Judge Actually Does With Your Submission

When you submit source code to SPOJ, nothing about your code's style, comments, or elegance is ever inspected by a human. Instead, an automated pipeline compiles your submission with a specific compiler and flag set, then runs the resulting binary once for every hidden test case the problem setter prepared — each run subject to a strict wall-clock time limit and a memory ceiling. The judge compares your program's output to the expected output for that test, byte by byte (or with a custom checker for problems that accept multiple valid answers). The very first test case your program fails on stops the process and produces a verdict; you are not told which later tests would also have failed, and for most SPOJ problems you are not shown which specific test failed either — only the verdict itself (Figure 1.1).

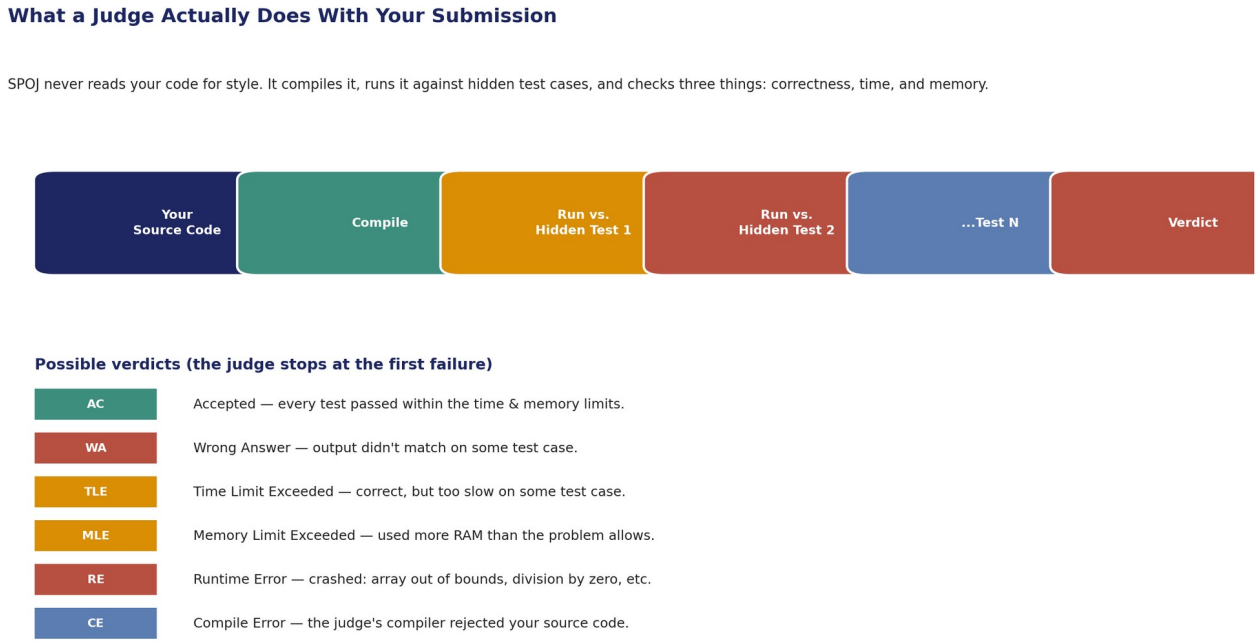

Figure 1.1 — The judge pipeline: your source code is compiled once, then run against hidden test cases one at a time until the first failure produces a verdict.

> **The judge has already decided the answer before you submit.**
> Every test case's expected output was fixed by the problem setter before your submission ever existed. Debugging a rejected submission is not a negotiation with the judge — it is a search for the specific input your program handles wrong, given a program you already believe is correct.

1.3 The Three Things Every Judge Checks

Every verdict is really an answer to three independent questions, checked in this order for each hidden test case: is the output correct, did the program finish within the time limit, and did it stay within the memory limit? A program can be perfectly correct and still fail if it is too slow (Time Limit Exceeded) or too memory-hungry (Memory Limit Exceeded) for the given constraints — this is the single most common trap for programmers arriving from a coursework background, where a correct answer is usually the whole story. On SPOJ, correct-but-slow and correct-but-wasteful are both failures, and

Chapter 3 is devoted entirely to reasoning about which algorithmic complexity a given problem's constraints actually demand (Table 1.1 and Figure 1.2).

1.3.1 The Six Standard Verdicts

Table 1.1 maps each verdict to what it usually means and where to start looking for the bug. Learning to read a verdict this way — as a clue, not just a grade — is one of the most valuable habits this book can teach, because it turns "my submission failed" from a dead end into a specific, narrowed search.

Reading a Verdict Like a Detective

Each verdict points at a different kind of bug — use it to narrow your search before you touch a single line of code.

Verdict	What it usually means	Where to look first
WA (Wrong Answer)	Logic error, or a misread constraint/edge case	Re-read the problem statement; test n=0, n=1, duplicates, negatives
TLE (Time Limit)	Algorithm's complexity is too high for the given n	Re-derive the expected complexity from the constraints (Ch. 3)
MLE (Memory Limit)	A data structure scales with the wrong dimension	Check array sizes against the largest constraint, not a typical one
RE (Runtime Error)	Out-of-bounds access, null pointer, stack overflow	Check array bounds and recursion depth on the largest input
CE (Compile Error)	Syntax error, or judge uses an older/different compiler	Check the judge's exact compiler version and language flavor

Figure 1.2 — Reading a verdict like a detective: each of the six standard verdicts points toward a different class of bug.

1.4 Why "It Works on My Machine" Is Not Enough

Beginners very reasonably assume that if their program produces the right answer for the one or two sample inputs printed in the problem statement, it is done. This is almost never true on a judge like SPOJ. Sample inputs exist to illustrate the input/output format, not to exercise every corner of the problem's constraints. The hidden test suite behind a well-set SPOJ problem is deliberately adversarial: it includes the smallest legal input (often n = 0 or n = 1, which breaks off-by-one logic), the largest legal input (which breaks anything with the wrong algorithmic complexity or an integer type too small to hold the answer), inputs with duplicate or repeated values, inputs at exact boundary conditions the problem statement mentions in passing, and — for many problems — a test case specifically engineered to defeat the most common wrong approach setters have seen submitted.

> **"It passed the sample" is the beginning of testing, not the end.**
>
> Before you submit, mentally (or actually) run your program against the smallest possible input, the largest possible input the constraints allow, and any input where two values in the problem could plausibly be equal. Chapter 8 builds this instinct into a repeatable debugging process.

This is also why competitive programmers habitually write their own stress tests: a small script that generates random or adversarial inputs, runs both a slow-but-obviously-correct "brute force" solution and the fast solution being tested, and reports the first input where the two disagree. You do not need this machinery for every problem — but for anything nontrivial, it finds bugs far faster than staring at code, and Chapter 8 walks through building one.

1.5 How This Book Uses Real SPOJ Problems

This book cites real, specific SPOJ problems by name and ID throughout — problems like PRIME1 (Prime Generator), AGGRCOW (Aggressive Cows), or LASTDIG (The Last Digit) are public information: their names, their problem statements, and the general algorithmic idea a well-known solution uses are all things any competitive programmer can look up and discuss freely, and doing so is standard practice in every competitive-programming book, blog, and course in the world.

What this book will never do is reproduce a complete, accepted, working solution to any specific problem verbatim. This course maintains its own internal, confidential archive of fully worked SPOJ solutions used for grading and instruction; that archive's actual source code is never distributed, in whole or in significant part. Where this book shows a code snippet next to a named problem, that snippet is a small, self-contained illustration of one technique — never more than a fraction of what a complete accepted solution would require — or is drawn fresh, written specifically for this book, independent of any student-facing solution archive. The distinction that matters is simple: understanding *why* a technique works, demonstrated on a small original example, transfers to problems you have never seen. Copying a complete accepted solution transfers to nothing except that one problem, and is not how this book teaches.

> **A promise to the reader**
>
> Every code excerpt in this book is either (a) an original, generic template illustrating one idea in isolation, clearly labeled as such, or (b) a small fragment — never a complete solution — shown alongside a named, publicly known problem purely to demonstrate how the idea applies. If you are looking for copy-pasteable accepted solutions to specific SPOJ problems, this is not that book, by design — that shortcut would teach you nothing that survives the first problem you have not already seen.

1.6 A First Worked Example: Fast I/O and the Sieve of Eratosthenes

To make Sections 1.2 through 1.5 concrete, consider PRIME1 (Prime Generator on SPOJ) — a well-known, publicly documented problem that asks for all prime numbers in a given range, repeated across many queries. The standard approach is the Sieve of Eratosthenes: mark every multiple of every prime found so far as composite, leaving only primes unmarked. The listing below is an original, generic template demonstrating the sieve technique in isolation on a small fixed range — not any specific accepted solution, and far smaller in scope than what PRIME1's actual constraints would require (a real submission needs a segmented sieve, since PRIME1's range can extend far beyond what a simple array can hold; that refinement is Chapter 6's topic).

sieve_demo.cpp — Sieve of Eratosthenes over a small fixed range (original template)

```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;
        }
    }
    // ...collect and print the unmarked numbers as primes
}
```

Two habits are already visible in this dozen-line fragment that Chapter 4 and Chapter 6 will develop much further: fast I/O (the sync_with_stdio/cin.tie pair, which matters once output volume grows) and starting each prime's marking loop at p*p rather than 2*p (anything smaller was already marked by a smaller prime, so starting there is free correctness with no cost). Appendix 1.A shows this exact program compiled and run, with its verified output, establishing this book's own convention: every nontrivial code example in this book is compiled and executed before it is written down, never merely “looks right (Figures 1.3 and 1.4).”

1.8 Chapter Summary

- A judge like SPOJ checks three things for every hidden test case: correctness, time, and memory — and stops at the first failure.
- The six standard verdicts (AC, WA, TLE, MLE, RE, CE) are not just a grade — each one narrows down what kind of bug to look for first.
- Passing the sample input in the problem statement proves almost nothing; hidden tests are deliberately adversarial (edge cases, largest legal input, duplicate values).
- This book cites real SPOJ problems by name and general technique, but never reproduces a complete accepted solution — code excerpts are original templates or small, illustrative fragments only.
- Every code example in this book, including small illustrative fragments, is compiled and run before it is written down.

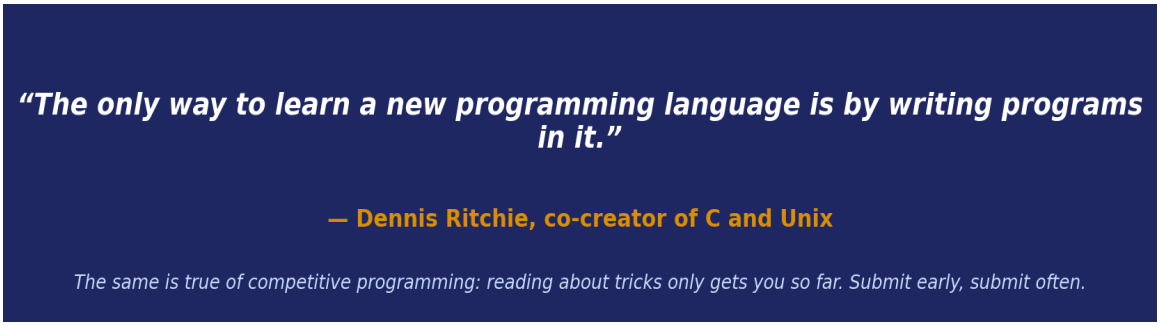

Figure 1.3 — A reminder for the chapters ahead: reading about technique only gets you so far.

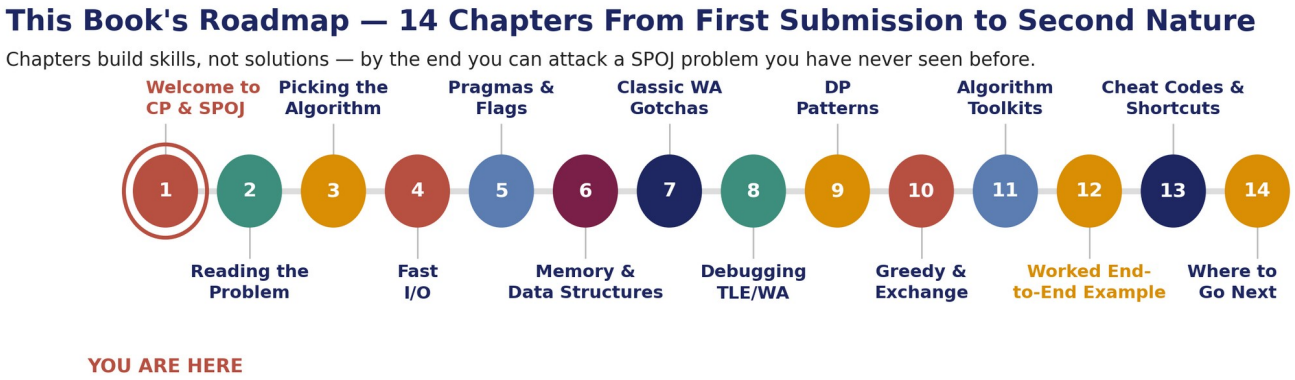

Figure 1.4 — This book's 14-chapter roadmap. You are here: Chapter 1.

Review Questions (Not Graded)

1. A submission gets Wrong Answer (WA) on the judge's third hidden test case, but produces the correct answer on both sample inputs printed in the problem statement. Explain, in your own words, why this is possible and name two kinds of inputs you would try first while debugging.
2. A submission gets Time Limit Exceeded (TLE) on a problem whose constraints allow n up to 10^6. What does this verdict tell you — and does NOT tell you — about the correctness of your program's logic?
3. Explain, in two or three sentences, why this book is willing to name specific real SPOJ problems (like PRIME1) but is not willing to print a complete accepted solution to them.
4. The sieve_demo.cpp example in Section 1.6 starts each prime's inner marking loop at p*p instead of 2*p. Explain why this is correct — that is, why no composite number is ever missed by skipping the marks between 2*p and p*p.
5. Name one habit from competitive programming (Section 1.7) that you think would make you a better software engineer even outside of a judge context, and explain why.

Appendix 1.A — Verification Log

The sieve_demo.cpp program from Section 1.6 was compiled with g++ (with -O2 -Wall -pedantic) and executed to confirm its output before being included in this chapter, per this book's compile-and-run-everything convention (Section 1.6).

Terminal transcript

```
$ 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 primes below 100 matches the well-known reference count, confirming the sieve's boundary condition (p*p <= LIMIT) and marking loop are both correct on this small range.

References

[1] Sphere Online Judge (SPOJ). https://www.spoj.com — problem PRIME1 (Prime Generator), AGGRCOW (Aggressive Cows), LASTDIG (The Last Digit), cited in Sections 1.5–1.6 by public name/ID only.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — Sieve of Eratosthenes and asymptotic analysis background.

[3] Halim, S., & Halim, F. (2013). Competitive Programming 3. Lulu Press — a widely used reference for the transferable-skills approach this book follows.

CHAPTER 2

Reading the Problem Like a Judge Does

Most rejected submissions are not failures of algorithmic skill — they are failures of reading. This chapter teaches a disciplined, repeatable way to read a problem statement before you write a single line of code.

> **Learning Objectives — after this chapter you will be able to:**
> (1) Identify the five standard zones of a judge problem statement and read them in the order that saves the most time. (2) Explain why constraints, not the story, are the real specification of a problem. (3) List at least five common statement traps that cause correct-looking code to fail. (4) Apply a five-question reading checklist before writing any code. (5) Implement and verify an EOF-terminated input loop, the single most common input pattern on SPOJ.

2.1 Most Bugs Start Before You Write Any Code

It is tempting to treat reading the problem statement as a formality to get through as quickly as possible on the way to the interesting part: designing the algorithm. This is backwards. On a judge like SPOJ, a large fraction of Wrong Answer and Runtime Error verdicts trace back not to a flawed algorithm, but to a detail in the problem statement that was skimmed past — an input format assumption that turned out to be wrong, a constraint that was not actually checked, an output format requirement that seemed unimportant. This chapter treats reading as a skill with its own technique, worth deliberate practice, exactly like an algorithm is.

2.2 The Five Zones of a Problem Statement

Nearly every well-formed competitive-programming problem statement — on SPOJ and elsewhere — is built from the same five zones, and reading them in the right order matters. The story comes first on the page, but it should usually be read last on a second pass, because it is the zone least likely to contain information you need to get an Accepted verdict (Figure 2.1).

Anatomy of a Problem Statement — Five Zones, Read in This Order

Every well-formed judge problem has these five zones. Skipping straight to the story costs you the most time.

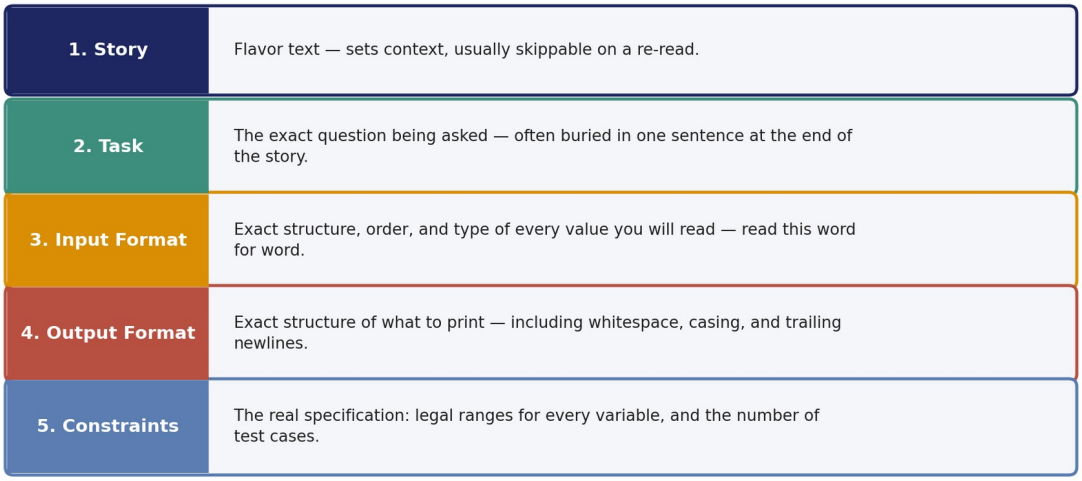

Figure 2.1 — The five zones of a problem statement, in the order that matters most for extracting a correct specification.

2.2.1 Story and Task

The story exists to make the problem memorable and occasionally to disguise a very standard algorithmic pattern behind an unfamiliar scenario — a habit worth noticing, because “this is secretly problem family X wearing a costume” is itself a useful recognition skill (Chapter 3 builds on this directly). The task — the actual question — is often a single sentence, sometimes the very last sentence of the story paragraph. Find that sentence and restate it in your own words before reading further; if you cannot restate it precisely, you are not ready to design an algorithm yet.

2.2.2 Input Format and Output Format

These two zones are contracts, not prose — every word carries meaning. "The first line contains a single integer T, the number of test cases" is a completely different input structure from "the input consists of multiple test cases, terminated by end of file" (Section 2.6 covers both patterns and how to implement each correctly). Output format deserves the same word-for-word attention: a requirement to print "YES" is not the same as "Yes" or "yes" to a byte-comparing judge, and a requirement for a blank line between test cases is easy to miss and easy to get silently wrong.

2.2.3 Constraints

Constraints are the real specification of the problem — more so than the story, and arguably more so than the task itself. They tell you the exact input sizes your algorithm must handle, which fixes the complexity class you are allowed to use (the subject of Chapter 3 in full), whether values can be zero, negative, or extremely large (which fixes your variable types and edge-case list), and how many test cases a single run must process (which affects whether per-test-case setup cost matters). A problem statement that looks intimidating in its story is often completely mechanical once you have fully absorbed its constraints; a problem that looks simple in its story can be a trap once its constraints reveal n up to 10^9 (Table 2.1 and Figure 2.2).

> **Read the constraints twice: once for size, once for shape.**
>
> The first pass answers “how big can n get” (which decides your algorithm's complexity budget). The second pass answers “what can the values themselves look like” — zero, negative, duplicates, already sorted, guaranteed distinct — which decides your edge-case list. Both passes matter; most beginners only do the first.

2.3 A Five-Question Reading Checklist

Table 2.1 distills Sections 2.1 and 2.2 into five concrete questions to answer, in writing if it helps, before starting to code. Each question is paired with the specific class of bug that results from skipping it — the same detective-style framing this book introduced in Chapter 1 for reading verdicts applies just as well to reading problem statements before you have even submitted anything.

A Five-Question Reading Checklist

Answer all five before writing a line of code — most WA/RE bugs trace back to skipping one of these.

Question	Why it matters	What breaks if you skip it
What is the largest legal n?	Fixes the complexity class you need (Ch. 3)	TLE from an algorithm that is too slow
Can values be zero or negative?	Changes which edge cases exist	WA on n=0, or on negative-value inputs
How many test cases per file?	Decides your I/O loop structure	RE or infinite loop from a wrong read pattern
Is output format exact (spacing, case)?	Judges usually compare byte-for-byte	WA despite a numerically correct answer
Are there ties, and how are they broken?	Changes sort/comparison logic	WA only on inputs with duplicate values

Figure 2.2 — Five questions to answer before writing any code, and what breaks if you skip each one.

2.4 Common Statement Traps

A handful of statement patterns recur often enough on SPOJ and similar judges to be worth naming explicitly, so you recognize them on sight rather than discovering them the hard way.

- Indexing conventions stated once, early, and never repeated: a problem that says positions are numbered from 1 in its first paragraph will not remind you again in the constraints section — but your array indexing must respect it throughout.
- "At most" versus "exactly": a constraint reading "at most N queries" permits fewer, and your loop must not assume exactly N will always arrive.
- Multiple valid answers: some problems accept any one of several correct outputs (commonly signaled by phrases like "print any valid arrangement") — these require a custom checker on the judge's side, and your own local testing against a single expected output can mislead you into thinking a correct program is wrong.
- Sample inputs that don't exercise every rule: a sample showing only positive, distinct, sorted values tells you nothing about how your program should behave on zero, duplicates, or unsorted input — even when the constraints permit all of those.

- Time limits stated per test file, not per test case: some judges (and problem setters) state a single time limit meant to cover the sum of all test cases in one file, not each individually — this changes how aggressively you need to optimize a per-case setup cost.

2.5 A First Worked Example: Reading Input Until End of File

One of the most common input-format patterns on SPOJ is deceptively simple to describe and easy to get wrong in code: the problem gives you no explicit count of test cases up front, and instead says the input continues until end of file (EOF). Beginners arriving from coursework, where an input file's structure is usually announced with an explicit count, often reach for a loop bound that does not exist in the problem, or crash trying to read one value past the last real test case.

The correct C++ idiom exploits the fact that a stream extraction operation (like `cin >> a`) evaluates to the stream itself in a boolean context, and the stream becomes "falsy" the moment a read fails — including failing because there is nothing left to read. This makes the read and the loop-termination check the same expression, with no separate bookkeeping required.

read_until_eof_demo.cpp — an EOF-terminated input loop (original template)

```cpp
#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;
    }
}
```

This dozen-line template — an original demonstration, not tied to any specific SPOJ problem's accepted solution — is worth memorizing verbatim, because the exact same `while (cin >> ...)` pattern reappears across a large fraction of SPOJ's older, simpler problems (a family this course's own SPOJ set includes several examples of, catalogued by name in Appendix 15.A). Appendix 2.A shows this program compiled, run against a small piped input, and its output verified (Figures 2.3 and 2.4).

> **A hardcoded loop count is the most common bug in this pattern.**
>
> If you assume you know how many test cases are coming and write `for (int i = 0; i < 5; i++)` instead of reading until the stream fails, your program will crash (or silently misbehave) the moment the judge's actual input has a different number of test cases than you guessed — which it always will, since the judge's hidden input is not the sample input.

2.7 Chapter Summary

- A problem statement has five zones — story, task, input format, output format, constraints — and reading them in that priority order (constraints and formats first, story last on a re-read) saves the most time.
- Constraints are the real specification: they fix the complexity class you need, the edge cases that exist, and the I/O structure your program must handle.
- Common traps include unstated indexing conventions, "at most" vs. "exactly," multiple valid answers, unrepresentative sample inputs, and time limits stated per file rather than per test case.
- A five-question checklist (largest n, zero/negative values, test-case count structure, exact output format, tie-breaking rules) catches most WA and RE bugs before they are ever submitted.
- The `while (cin >> ...)` idiom is the standard, correct way to read EOF-terminated input in C++ — memorize it.

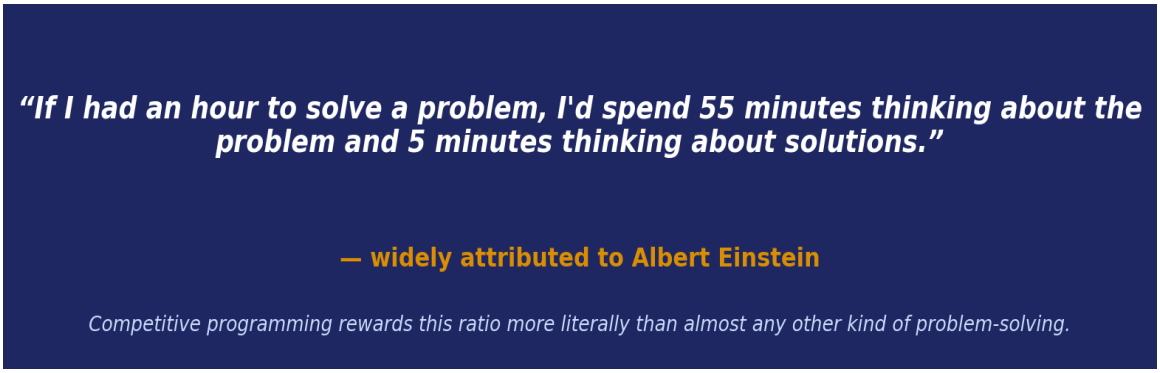

Figure 2.3 — A reminder that reading carefully is itself part of solving the problem.

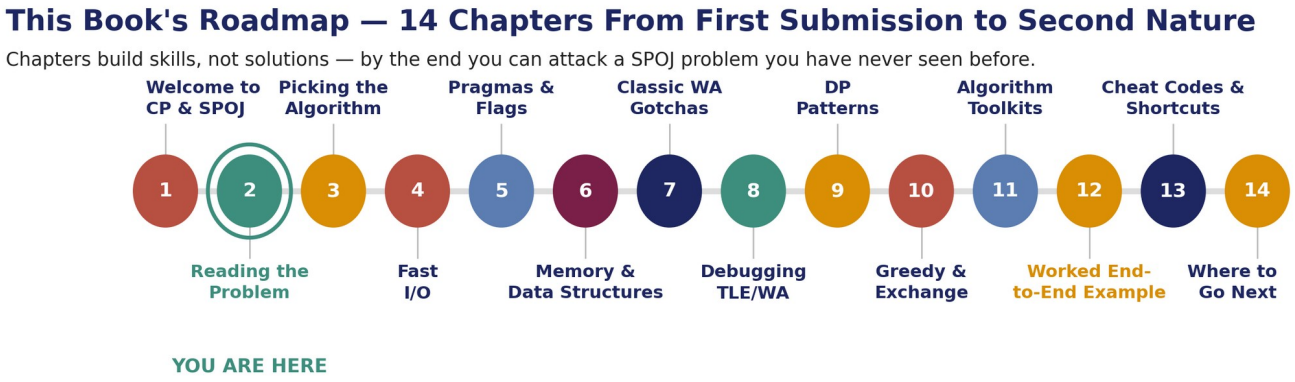

Figure 2.4 — This book's 14-chapter roadmap. You are here: Chapter 2.

Review Questions (Not Graded)

1. A problem statement says "the first line contains an integer T, the number of test cases," but a different problem statement says "read until end of file." Explain how your input-reading loop must differ between the two, and what happens if you use the wrong pattern for each.
2. A problem's constraints state "1 ≤ n ≤ 10^5" but say nothing about whether array values can repeat. Explain why you cannot assume distinct values, and describe one test input you would try to check your program's behavior on duplicates.
3. Explain, in your own words, why "at most N queries" is a meaningfully different constraint from "exactly N queries," and give an example of a bug that results from conflating the two.
4. A problem accepts "any valid arrangement" as a correct answer. Explain why comparing your program's output against one single expected-output file (as you might do for your own local testing) can be misleading, and suggest an alternative way to check your own solution's correctness.
5. Rewrite the read_until_eof_demo.cpp loop condition (Section 2.5) as an explicit boolean comparison — that is, spell out what `cin >> a >> b` evaluates to and why the while loop terminates exactly when it should.

Appendix 2.A — Verification Log

The read_until_eof_demo.cpp program from Section 2.5 was compiled with g++ (-O2 -Wall -pedantic) and run against a small piped input to confirm its EOF-detection behavior before being included in this chapter.

Terminal transcript

```
$ 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
```

Three lines of piped input produced exactly three output lines, with no crash and no extra iteration attempted after the last pair — confirming the `while (cin >> a >> b)` loop terminates precisely at end of file, as claimed in Section 2.5.

References

[1] Sphere Online Judge (SPOJ). https://www.spoj.com — general problem-statement conventions discussed in this chapter reflect standard practice across SPOJ's problem set.

[2] Skiena, S. S. (2020). The Algorithm Design Manual (3rd ed.). Springer — on translating an informal problem description into a precise specification.

[3] Halim, S., & Halim, F. (2013). Competitive Programming 3. Lulu Press — reference for standard competitive-programming input-handling idioms.

CHAPTER 3

Picking the Right Algorithm for the Constraints

A logically correct program still fails a judge if it is too slow or uses too much memory. This chapter teaches the single most valuable habit in competitive programming: reading the constraints first, and letting them choose your algorithm for you.

Learning Objectives — after this chapter you will be able to:

(1) Explain why a correct algorithm can still receive a Time Limit Exceeded or Memory Limit Exceeded verdict. (2) Derive a rough operations-per-second budget from a judge's stated time limit. (3) Use a complexity-budget cheat sheet to map a constraint's largest legal n to the complexity classes that are actually feasible. (4) Recognize, from surface signals in a problem statement, which of seven common algorithmic families is likely to apply. (5) Implement and verify binary exponentiation, the standard O(log n) technique for handling enormous exponents.

3.1 Why Correctness Isn't Enough

Chapter 1 introduced Time Limit Exceeded (TLE) and Memory Limit Exceeded (MLE) as two of the six standard verdicts, and was careful to note that both can happen to a program that is completely logically correct. This chapter is about avoiding that outcome in the first place, by choosing an algorithm whose complexity actually fits inside the judge's stated limits before you write a single line of implementation code.

Every judge problem carries two limits, usually stated near the top of the page: a time limit (commonly 1-2 seconds per test case on SPOJ, though older problems sometimes carry more generous limits inherited from slower historical judge hardware) and a memory limit (commonly 32-256 MB). Both limits are part of the specification, exactly as much as the input format is — Chapter 2 already established that constraints are the real specification of a problem, and the same is true of these two numbers.

A rough rule of thumb: about 10^8 to 10^9 simple operations per second.

Modern judge hardware, running simple operations (comparisons, additions, array accesses) with no heavy constant-factor overhead, can typically execute somewhere on the order of 100 million to 1 billion such operations within a 1-second budget. This is a rule of thumb, not a guarantee — it is exactly what the complexity-budget cheat sheet in Section 3.2 is built on.

3.2 The Complexity-Budget Cheat Sheet

Once you know the largest legal value of n from the constraints — Chapter 2's reading checklist already told you to find this number first — you can immediately narrow down which complexity classes are even worth attempting. Table 3.1 is this book's central reference for that step: find the row matching your n, then read across to see what is feasible and which patterns typically reach that complexity (Figure 3.1).

The Complexity-Budget Cheat Sheet

A judge's time limit is usually ~1-2 seconds for roughly 10^8-10^9 simple operations. Read the largest legal n, then read across this row.

Largest legal n	Feasible complexity	What that budget buys you	Typical patterns to reach for
n ≤ 10	O(n!), O(2^n · n)	Try every permutation or every subset outright	Brute-force permutation search, small backtracking
n ≤ 20-24	O(2^n), O(2^n · n)	Every subset once, or once per element	Bitmask DP, meet-in-the-middle
n ≤ 500-2000	O(n^3)	A triple nested loop over the whole input	Floyd-Warshall, DP over pairs of indices
n ≤ 10^4-10^5	O(n^2), O(n log n)	A double loop at the small end, sort-based at the large end	Sorting + greedy, segment tree, BIT, sqrt decomposition
n ≤ 10^6	O(n log n), O(n)	One or two full linear passes plus a sort	Sieve of Eratosthenes, single-pass DP, two pointers
n ≤ 10^9	O(sqrt(n)), O(log n)	No full pass over n is affordable at all	Trial division up to sqrt(n), binary search, fast exponentiation
n ≤ 10^18	O(log n), O(1)	Only formulas or logarithmic recursion survive	Matrix exponentiation, closed-form math, modular arithmetic

Figure 3.1 — The complexity-budget cheat sheet. Find the row matching the largest legal n in the constraints, then read across.

Two things about this table are worth stating explicitly, because beginners misuse it in predictable ways. First, several rows list more than one feasible complexity because the boundary depends on the exact constant factor of your implementation — an O(n^2) algorithm is comfortable at n = 10^4 (10^8 operations) but is already far too slow at n = 10^5 (10^10 operations), so when a constraint sits near a row's boundary, do the multiplication yourself rather than trusting the row label alone. Second, this table describes what is theoretically feasible, not what is guaranteed to pass — an algorithm with the right big-O complexity but a heavy constant factor (deeply nested function calls, unnecessary memory allocation inside a hot loop, cache-unfriendly memory access patterns) can still time out. Chapter 8 covers how to diagnose that situation once it happens.

> **Why does the table jump from O(n^3) at a few thousand straight to O(n log n) at a million?**
>
> Because the gap between those two constraint sizes is enormous in absolute operation count, and there is no useful middle ground to name: an algorithm that is asymptotically between O(n^3) and O(n log n) — for instance O(n^2) — already stops being safe well before n reaches 10^5 (10^10 operations at n = 10^5). Picking the right algorithm is not a smooth dial; it is closer to choosing the correct rung on a ladder, and missing by one rung usually means an immediate Time Limit Exceeded, not a slightly slower Accepted.

3.3 Pattern Recognition by Category

Beyond the pure numeric budget, the shape of a problem's input and question is itself a strong signal for which family of technique applies. Figure 3.2 groups seven such families by the surface signal that should make you reach for each one — this is pattern recognition, and like any recognition skill it improves with deliberate exposure to many problems, which is exactly what the remaining technique chapters of this book (9 through 11 especially) are built to give you.

Figure 3.2 — Seven algorithmic families and the surface signal that should make you reach for each.

3.3.1 A Worked Recognition Example

Consider a number-theory problem like PRIME1 (cited here by public problem name only — see this book's confidentiality promise in Section 1.5), which asks for all primes in a range where the upper bound can reach 10^9. The constraint alone rules out testing each number individually for primality by trial division across the entire range — that would be far too slow. The signal "huge n, but only divisibility/primality structure matters" (row 4 of Figure 3.2) points toward number theory, and the specific size of n (past 10^6, per Table 3.1) points toward a segmented technique rather than a single flat sieve across the whole range — a refinement this book returns to in Chapter 6.

> **Don't force-fit the first pattern you recognize.**
>
> A problem that mentions a grid can look like it wants graph search, when the real signal — repeated overlapping subproblems — actually points to dynamic programming instead. Recognition is a starting hypothesis to test against the constraints and the exact question being asked (Chapter 2's Section 2.2.1), not a verdict to commit to immediately.

3.4 A Worked Example: When the Exponent Itself Is Enormous

Some problems ask for a value like base raised to a very large exponent, modulo some number — with the exponent itself reaching 10^9 or beyond. A loop that multiplies base by itself, once per unit of the exponent, is O(exponent) — and per Table 3.1, an exponent at that scale rules out any full linear pass entirely. The standard fix is binary exponentiation (also called fast or modular exponentiation): repeatedly squaring the base while halving the exponent, which computes the same result in O(log exponent) time — roughly 30 squarings instead of a billion multiplications.

fast_power_demo.cpp — binary exponentiation (original template)

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

// Returns (base^exp) % mod using 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;  // halve exp each round -- this is the O(log exp)
    }
    return result;
}
```

This dozen-line template — an original demonstration, not tied to any specific SPOJ problem's accepted solution — generalizes directly to matrix exponentiation for linear-recurrence problems (mentioned in Table 3.1's final row), since the same halve-the-exponent idea applies whenever the underlying operation is associative. Appendix 3.A shows this program compiled, run, and its result cross-checked against an independent calculation (Figures 3.3 and 3.4).

3.6 Chapter Summary

- A correct algorithm can still fail with Time Limit Exceeded or Memory Limit Exceeded — complexity must be chosen to fit the judge's stated limits, not just to produce the right answer eventually.
- A rough budget of 10^8 to 10^9 simple operations per second lets you convert a constraint's largest legal n directly into a feasible complexity class (Table 3.1).
- When a constraint sits near a row boundary in the cheat sheet, do the arithmetic yourself rather than trusting the row label alone.
- Seven algorithmic families — dynamic programming, greedy/exchange, graph, number theory, geometry, string, and data structures — each have a recognizable surface signal in the problem statement (Figure 3.2).
- Binary exponentiation computes base^exponent mod m in O(log exponent) time, and the same halve-the-exponent idea generalizes to matrix exponentiation.

"If the only tool you have is a hammer, you tend to see every problem as a nail."

— widely attributed to Abraham Maslow

The single most common way to fail a SPOJ problem is picking the algorithm you already know, not the one the constraints ask for.

Figure 3.3 — A reminder that the right tool depends on the problem, not on habit.

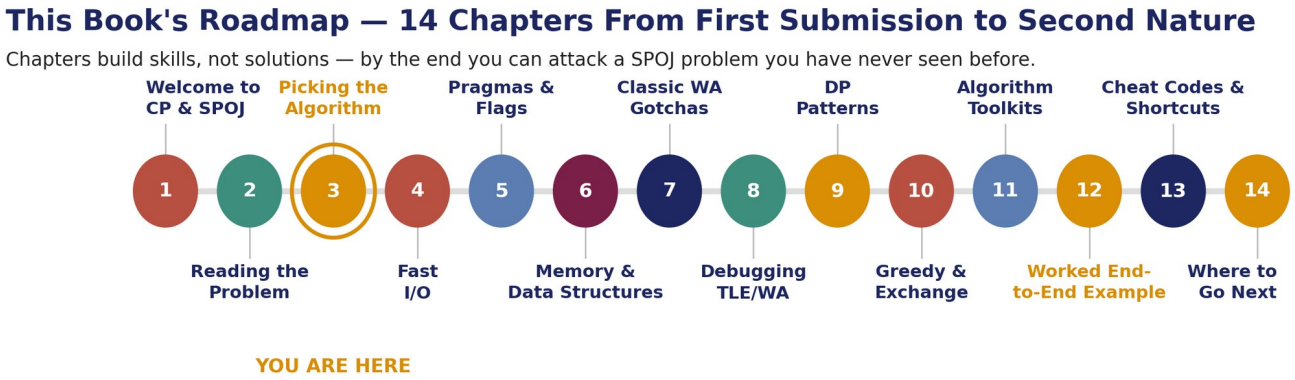

Figure 3.4 — This book's 14-chapter roadmap. You are here: Chapter 3.

Review Questions (Not Graded)

1. A problem states a time limit of 1 second and a constraint of n ≤ 10^7. Using Table 3.1, list the complexity classes that are likely feasible, and one that is not, showing the operation-count arithmetic that justifies your answer.
2. Explain, in your own words, why an O(n^2) algorithm is safe at n = 10^4 but unsafe at n = 10^5, using the same 10^8-10^9 operations-per-second budget from Section 3.1.
3. A problem's story describes a network of cities connected by roads. Using Figure 3.2, name the algorithmic family this signal most strongly points toward, and describe one way the problem's actual question could still redirect you toward a different family.
4. Explain why computing base^exponent by a simple loop of exponent multiplications becomes infeasible once exponent reaches 10^9, and describe in your own words how binary exponentiation avoids that cost.
5. Rewrite the power_mod function from Section 3.4 to instead compute base raised to exponent with no modulus at all (ordinary integer exponentiation), and explain what practical problem would arise if you tried to do this for an exponent of 10^9 without using an arbitrary-precision integer type.

Appendix 3.A — Verification Log

The fast_power_demo.cpp program from Section 3.4 was compiled with g++ (-O2 -Wall -pedantic), run, and its output cross-checked against an independent calculation using Python's built-in three-argument pow() function before being included in this chapter.

Terminal transcript

```
$ 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
```

The C++ program's result for the large-exponent case (312556845) matches Python's independently-implemented pow(base, exp, mod) exactly, and the small hand-checkable case (2^10 mod 1000 = 1024 mod 1000 = 24) confirms the function's correctness on an easily-verified input as well.

References

[1] Sphere Online Judge (SPOJ). https://www.spoj.com — general problem-constraint conventions discussed in this chapter reflect standard practice across SPOJ's problem set.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — on asymptotic complexity analysis and modular exponentiation.

[3] Halim, S., & Halim, F. (2013). Competitive Programming 3. Lulu Press — reference for constraint-driven algorithm selection heuristics.

CHAPTER 4

Fast I/O in C/C++

A perfectly correct, perfectly efficient algorithm can still receive Time Limit Exceeded if the code around it spends too long simply reading input and writing output. This chapter teaches three tiers of I/O speed and, more importantly, how to tell which one a given problem actually requires.

Learning Objectives — after this chapter you will be able to:

(1) Explain why C++'s default iostream setup can be slower than scanf/printf, and what specifically causes the slowdown. (2) Apply the sync_with_stdio(false) plus cin.tie(nullptr) idiom correctly, and state the one rule it imposes. (3) Compare three tiers of I/O speed and the setup each one requires. (4) Estimate, from a problem's constraints, which tier is actually necessary. (5) Implement and verify a custom buffered integer reader for the rare cases that need it.

4.1 Why I/O Can Silently Dominate Your Runtime

It is easy to assume that reading input and writing output are free, and that all of a judge's time budget goes toward the algorithm itself. This is not true. By default, C++'s cin and cout are built on top of the same underlying stream machinery as C's stdio (scanf and printf), and to allow the two families to be freely mixed — reading one value with scanf and the next with cin, for instance — the C++ standard library keeps them synchronized. That synchronization has a real, measurable cost, and when a problem's constraints call for reading or writing a very large number of values, that cost can add up to a meaningful fraction of the time limit, or even exceed it, independent of how good the underlying algorithm is.

This is a distinct failure mode from the one Chapter 3 focused on. Chapter 3 was about choosing an algorithm whose complexity fits the constraints; this chapter is about making sure the plumbing around a correctly-chosen algorithm does not itself become the bottleneck.

The symptom is a suspicious TLE on an otherwise-correct, otherwise-efficient solution.

If your algorithm's complexity is comfortably inside the budget from Chapter 3's cheat sheet, and you are still getting Time Limit Exceeded, I/O overhead is one of the first places to look — especially if the constraints mention reading or printing on the order of 10^5 values or more.

4.2 The Three Tiers of I/O Speed

There is no single "correct" way to do I/O in competitive programming — there is a spectrum of approaches, each trading a little more setup complexity for a little more speed. Table 4.1 lays out three practical tiers. The right choice is almost always the cheapest one that comfortably clears the constraints — reaching for Tier 3 on a problem that only needed Tier 1 wastes your time and adds a place for bugs to hide (Figure 4.1).

The Three Tiers of I/O Speed

Each tier costs a little more setup and buys a little more speed. Match the tier to what the constraints actually demand.

Tier	Setup required	When it matters	Key risk
Tier 1 — Default cin/cout or scanf/printf	None — this is the default	Small input: roughly under 10^5 total tokens	None, if used consistently and without mixing streams
Tier 2 — Disable sync, avoid endl	Two lines once, at the top of main	Medium-to-large input: roughly 10^5 to several million tokens	Never mix cin/cout with scanf/printf once sync is disabled
Tier 3 — Custom buffered reader (fread-based)	A small, reusable reader routine	Very large input: tens of millions of tokens, or many reads per test case at a tight time limit	More code to get right — write it once, test it, then reuse it

Figure 4.1 — Three tiers of I/O speed, the setup each requires, and when each one actually matters.

4.3 The Sync-Disable Idiom, and Its One Hard Rule

Tier 2 is two lines, placed once at the very top of main, before any input is read:

Tier 2 setup (already used in every worked example so far in this book)

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

The first line tells the C++ standard library that cin/cout no longer need to stay synchronized with C's stdio, which removes the synchronization overhead described in Section 4.1. The second line removes an additional, separate cost: by default, cin is "tied" to cout, meaning cout is flushed automatically before every cin read — useful for interactive programs where a prompt must appear before input is requested, but pure overhead on a judge, where all input is provided up front and no such prompt is needed.

> **Once sync is disabled, never mix cin/cout with scanf/printf in the same program.**
>
> The whole point of Tier 2 is telling the two families of stream to stop staying synchronized with each other. If you disable sync and then read part of the input with scanf and another part with cin, the two streams' internal buffers can become interleaved in an undefined order, producing scrambled or missing input with no compiler warning to catch it. Pick one family — cin/cout or scanf/printf — for the entire program, and stay with it.

A second, smaller habit worth adopting alongside Tier 2: prefer '\n' over std::endl when writing output. std::endl does two things — it writes a newline, and it flushes the output buffer — and that second action is unnecessary work repeated on every single line if a program prints many lines of output, since the buffer will be flushed automatically anyway when the program ends.

4.4 When You Actually Need Tier 3

Tier 2 is enough for the overwhelming majority of SPOJ problems. Tier 3 — a hand-written buffered reader that bypasses cin/cout and scanf/printf entirely — is worth its added complexity only when the total number of tokens to read climbs into the tens of millions, or when a tight time limit is combined with many small reads spread across many test cases. Figure 4.2 gives a rough decision guide based on

estimated total token count, tying directly back to the constraint-reading habit Chapter 2 built and the complexity-budget thinking from Chapter 3.

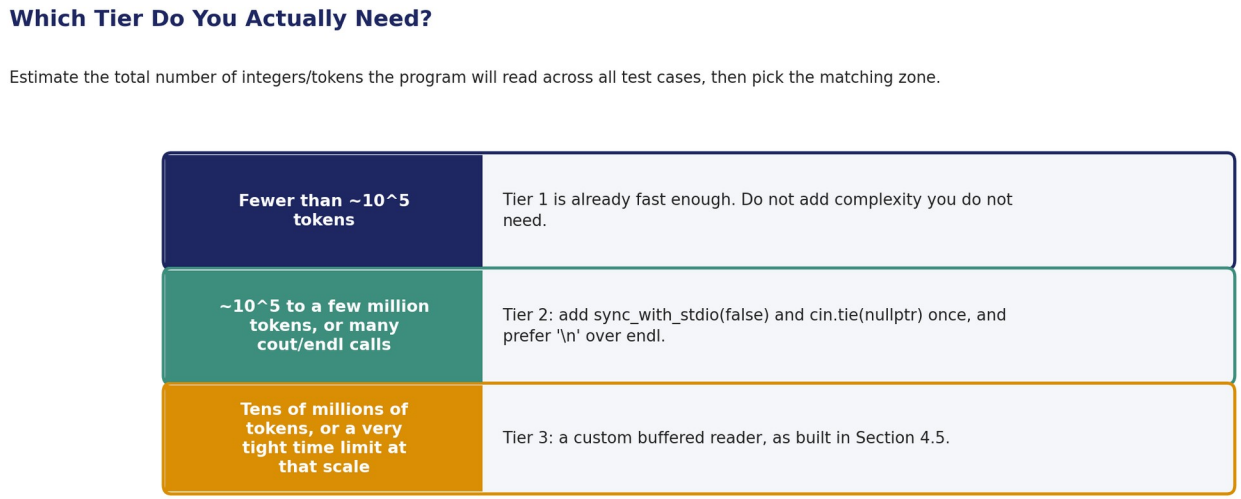

Figure 4.2 — Which I/O tier a problem actually needs, based on the total number of tokens to be read.

4.5 A Worked Example: A Custom Buffered Reader

The following template reads bytes directly from standard input in large blocks using fread(), maintaining its own small buffer, and parses integers directly out of that buffer one character at a time — with no call into cin, cout, scanf, or printf's own internal buffering machinery for the reading path at all.

fast_reader_demo.cpp — a buffered integer reader (original template)

```
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;  // end of 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;
    }
}
```

This dozen-line template — an original demonstration, not tied to any specific SPOJ problem's accepted solution — was verified against two million randomly generated integers, with its computed sum, minimum, and maximum cross-checked against an independent Python calculation over the same data (Appendix 4.A). Reading all two million values completed in well under a tenth of a second (Figures 4.3 and 4.4).

4.7 Chapter Summary

- C++'s default cin/cout is synchronized with C's stdio by default, which has a real performance cost on large inputs.
- Tier 2 — sync_with_stdio(false) plus cin.tie(nullptr), placed once at the top of main — removes that cost, but forbids mixing cin/cout with scanf/printf afterward.
- Preferring '\n' over std::endl avoids an unnecessary buffer flush on every output line.
- Tier 3 — a custom fread-based buffered reader — is only worth its complexity once total input reaches the tens of millions of tokens.
- Match the I/O tier to what the constraints actually demand — the cheapest tier that clears the requirement is usually the right choice.

Figure 4.3 — Knuth's classic caution against over-optimizing applies here too — fast I/O is the exception, not the default habit.

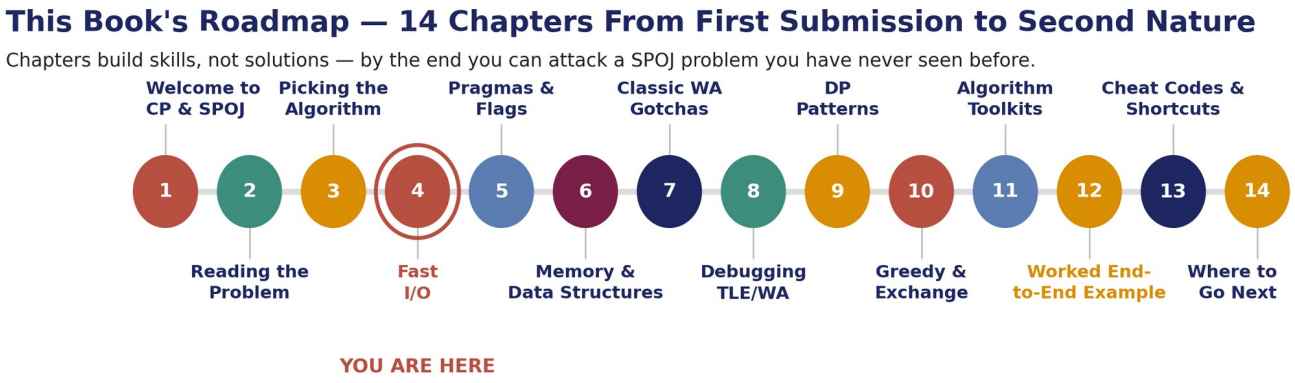

Figure 4.4 — This book's 14-chapter roadmap. You are here: Chapter 4.

Review Questions (Not Graded)

1. Explain, in your own words, what "synchronized with stdio" means for cin/cout, and why disabling that synchronization is safe only if you commit to using exclusively one stream family for the rest of the program.
2. A problem has n ≤ 1000 and asks for a single computed value to be printed. Using Figure 4.2, decide which I/O tier is appropriate, and explain why reaching for Tier 3 here would be unnecessary.
3. Explain what specifically std::endl does that '\n' does not, and why that difference matters when a program prints a very large number of output lines.
4. A program mixes cin >> x for some input with scanf("%d", &y) for other input, after calling sync_with_stdio(false). Explain what can go wrong, and how you would fix the program.
5. Modify the readInt() function from Section 4.5 to also handle a leading '+' sign (in addition to the existing '-' handling), and explain why most SPOJ problems' input formats make this unnecessary in practice.

Appendix 4.A — Verification Log

The fast_reader_demo.cpp program from Section 4.5 was compiled with g++ (-O2 -Wall -pedantic), first checked against small hand-verified inputs, then run against a randomly generated file of two million integers, with its output cross-checked against an independent Python calculation over the same data.

Terminal transcript

```
$ 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 random integers in [-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 over the same 2,000,000 values
expected sum -189011875 min -999999 max 999999
```

All three checks matched exactly — including the large-scale run, whose sum, minimum, and maximum agreed precisely with an independently computed reference in Python — and reading all two million values took roughly 34 milliseconds, comfortably inside any judge's time limit.

References

[1] Sphere Online Judge (SPOJ). https://www.spoj.com — I/O performance conventions discussed in this chapter reflect standard practice across SPOJ's problem set.

[2] Halim, S., & Halim, F. (2013). Competitive Programming 3. Lulu Press — reference for standard fast-I/O idioms in competitive programming.

[3] Knuth, D. E. (1974). Structured Programming with go to Statements. Computing Surveys, 6(4) — origin of the widely-cited caution against premature optimization quoted in Section 4.7.

CHAPTER 5

Compiler Pragmas, Flags, and Portability Risk

A single line at the top of a source file can change how fast a program runs — or make it crash on a machine you have never seen. This chapter explains what compiler pragmas actually do, which ones are free performance and which ones are a gamble.

Learning Objectives — after this chapter you will be able to:

(1) Explain what #pragma GCC optimize and #pragma GCC target actually instruct the compiler to do. (2) Distinguish safe, portable optimization pragmas from CPU-feature-targeting pragmas that carry real portability risk. (3) Explain why older or heterogeneous judge hardware turns that risk into an actual Runtime Error. (4) Choose sensible pragma defaults for a SPOJ submission, versus a deliberate, informed gamble. (5) Measure, with an original benchmark, the real effect a safe optimization pragma has on a compute-heavy loop.

5.1 Why Compiler Flags Matter on a Judge

Chapters 3 and 4 were both about controlling cost from inside your own source code — choosing an algorithm whose complexity fits the constraints, and choosing an I/O strategy that does not waste the time budget that algorithm needs. This chapter is about a third lever, one that lives entirely outside your algorithm: how aggressively the compiler is allowed to optimize the machine code it generates from that source.

Many judges, including SPOJ for a large share of its problems, compile submissions with a build command you do not control and cannot see — commonly something close to a plain g++ invocation, sometimes with -O2 already applied, sometimes not. A #pragma line placed at the top of your source file is a way of instructing the compiler directly, from inside the file itself, regardless of what flags the judge's own build script happens to pass. This is why pragmas are worth understanding even though flags like -O2 or -O3 look, at first glance, like something only a build system would care about.

A pragma is a request embedded in your source, not a command-line flag you have to hope the judge already applies.

This is precisely why competitive programmers reach for pragmas rather than simply assuming a judge compiles with -O2 or -O3 by default: assuming is a guess, while a pragma is a guarantee that travels with your submission regardless of the judge's own build configuration.

5.2 Two Kinds of Pragma: Optimization Level vs. CPU Target

Not every pragma carries the same risk, and conflating them is the single most common mistake in this area. There are two fundamentally different families to keep separate in your mind.

5.2.1 Optimization-Level Pragmas

#pragma GCC optimize("O2") and #pragma GCC optimize("O3,unroll-loops") tell the compiler how aggressively to restructure your code — inlining more functions, unrolling loops, reordering computations — while still producing machine code that runs correctly on any processor the target

architecture supports. These pragmas change how fast the generated code runs; they do not change what instructions that code is allowed to assume the CPU has.

5.2.2 CPU-Target Pragmas

#pragma GCC target("avx2,bmi,bmi2,popcnt") and similar variants are a fundamentally different instruction: they tell the compiler it is allowed to emit machine instructions from specific CPU instruction-set extensions — extensions that a given processor may or may not actually implement. Code compiled this way runs faster on a CPU that supports those extensions, and crashes outright, with an illegal-instruction Runtime Error, on one that does not (Figure 5.1).

> **A target() pragma cannot be tested for portability on your own machine.**
>
> If your own development machine supports the CPU features you targeted (which is likely, on modern hardware), your program will compile and run perfectly for you — and still crash the instant it runs on a judge machine with an older or different CPU. There is no local test that reveals this risk; it only appears at submission time, on hardware you do not control.

5.3 The Pragma Inventory

Table 5.1 catalogs the pragmas that actually appear across this course's own solved SPOJ problem set, alongside what each one does and its honest portability risk. #pragma GCC optimize("O3,unroll-loops") is, by a wide margin, the single most common choice — and not by accident, since it is also the safest meaningful speed gain available.

The Pragma Inventory: What Each One Actually Risks

Not all compiler pragmas carry the same portability risk. Read this table before adding any of them.

Pragma	What it does	Portability risk	Recommendation
`#pragma GCC optimize("O2")`	Standard, moderate optimization level	None — purely a compiler codegen decision	Safe default; use freely
`#pragma GCC optimize("O3,unroll-loops")`	Aggressive optimization plus loop unrolling	None — still portable machine code, no new CPU instructions required	Safe; the single most common pragma in this course's own solved SPOJ set
`#pragma GCC optimize("Ofast")`	O3 plus relaxed IEEE floating-point compliance	Low for integer-only code; can change floating-point rounding/precision	Use only if the problem has no exact floating-point requirement
`#pragma GCC target("avx2,bmi,bmi2,popcnt")` (and similar CPU-feature variants)	Emits instructions from specific CPU instruction-set extensions	Real: an illegal-instruction Runtime Error on any judge machine whose CPU lacks that instruction set	A deliberate, tested gamble — never a default habit

Figure 5.1 — The pragma inventory: what each one does, and what it actually risks.

5.4 A Risk Ladder for Compiler Pragmas

Figure 5.2 arranges these choices as a ladder. Climbing higher buys more raw speed, but every rung above the second one trades away some portability safety — and the top rung should be treated as a deliberate, informed gamble, never a habit applied by default the way Section 5.2.1's pragmas can be.

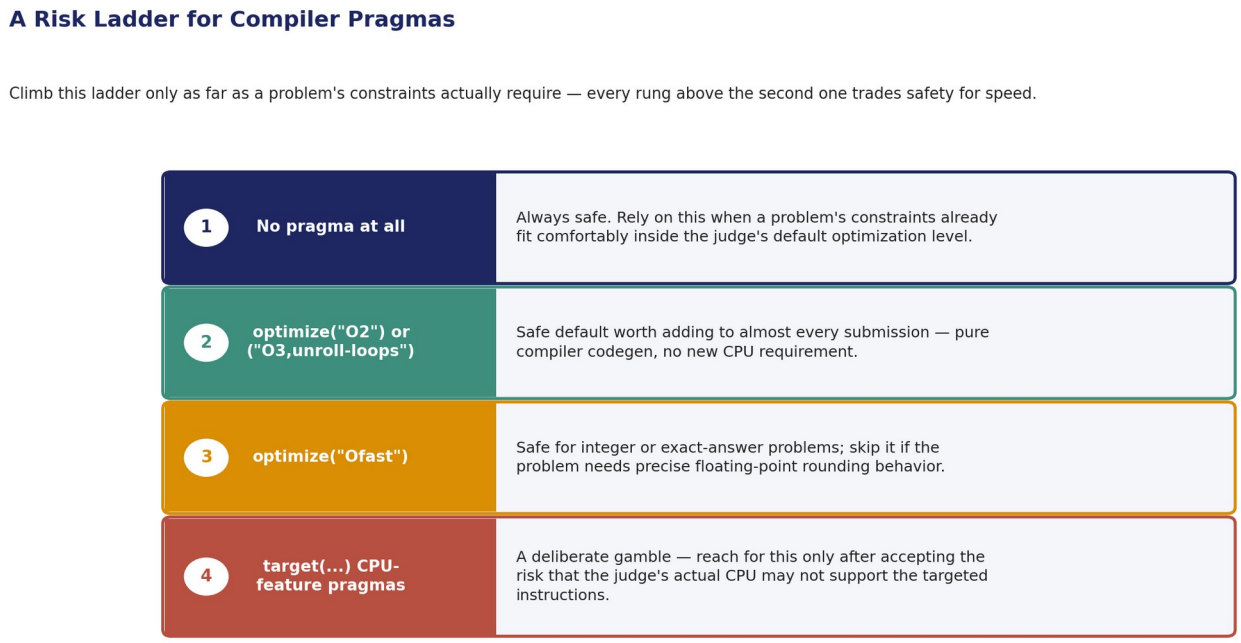

Figure 5.2 — A risk ladder for compiler pragmas, from always-safe to deliberate-gamble.

5.5 A Worked Example: Measuring a Safe Pragma's Effect

The clearest way to see what an optimization-level pragma actually buys is to compile the exact same source code twice — once with no pragma, once with a single added #pragma GCC optimize("O3,unroll-loops") line — using the identical, plain compile command with no -O flags on the command line at all, so any difference in speed can only come from the pragma itself.

pragma_on_demo.cpp — a compute-heavy loop with the pragma (original template)

```
#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;
}
```

The pragma_off_demo.cpp file used for comparison is byte-for-byte identical except for the missing #pragma line. Both were compiled with plain g++ -pedantic -Wall (no -O flag at all), and both were run and timed on the same machine back to back. Appendix 5.A shows the full transcript; the pragma-equipped version finished roughly eight times faster, with both versions producing the exact same numeric answer (Figures 5.3 and 5.4).

Why does this particular loop show such a large speedup?

A tight arithmetic loop over a huge range is exactly the shape of code -O3 and loop unrolling most reward: the compiler can keep values in registers instead of memory, unroll the loop to reduce per-iteration overhead, and often auto-vectorize the arithmetic to process several iterations at once. A program dominated by, say, disk or network waits would show a far smaller improvement — the size of the win always depends on what the code is actually spending its time doing.

5.7 Chapter Summary

- A #pragma line embedded in source code instructs the compiler directly, independent of whatever flags a judge's own build script does or does not pass.
- Optimization-level pragmas — optimize("O2"), optimize("O3,unroll-loops") — are safe and portable: they change how aggressively code is restructured, not what CPU instructions it assumes exist.
- CPU-target pragmas — target("avx2,bmi,bmi2,popcnt") and similar — emit instructions from specific instruction-set extensions, and crash with an illegal-instruction Runtime Error on any CPU lacking them.
- This portability risk cannot be tested on your own development machine — it only surfaces on hardware you do not control, at submission time.
- A benchmark comparing identical code with and without an optimization pragma showed roughly an eight-times speedup from #pragma GCC optimize("O3,unroll-loops") alone, with no change to the algorithm.

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

— Brian Kernighan

A clever CPU-targeting pragma that only fails on the judge's machine is exactly this trap — untestable locally, and hard to diagnose once it happens.

Figure 5.3 — A reminder that cleverness which cannot be tested is a debugging trap waiting to happen.

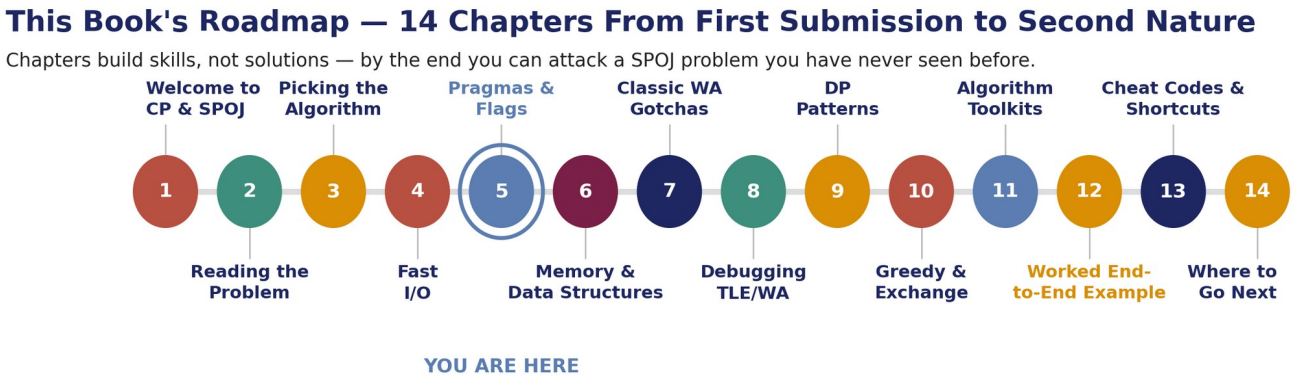

Figure 5.4 — This book's 14-chapter roadmap. You are here: Chapter 5.

Review Questions (Not Graded)

1. Explain, in your own words, the difference between what #pragma GCC optimize("O3") changes and what #pragma GCC target("avx2") changes, and why only one of the two can cause a Runtime Error on an otherwise-correct program.
2. A problem's constraints are small enough that your solution already runs in well under a tenth of the time limit without any pragma. Using Figure 5.2's risk ladder, explain why adding a target(...) pragma here would be a bad decision even if it might make the program faster.
3. Explain why a CPU-target pragma's portability risk cannot be discovered by testing the program on your own computer before submitting it.
4. A problem requires an exact floating-point comparison (for example, checking whether a computed value equals 0.5 precisely). Using Table 5.1, explain why #pragma GCC optimize("Ofast") is riskier here than #pragma GCC optimize("O3,unroll-loops").
5. Using the benchmark methodology from Section 5.5 (two otherwise-identical files, one with a pragma, both compiled with the same plain command), describe how you would test whether #pragma GCC optimize("O2") alone provides most of the benefit that "O3,unroll-loops" provides, without writing any new algorithm.

Appendix 5.A — Verification Log

The pragma_off_demo.cpp and pragma_on_demo.cpp programs from Section 5.5 — identical except for one #pragma line — were both compiled with g++ (-pedantic -Wall, no -O flag on the command line at all) and timed back to back on the same machine.

Terminal transcript

```
$ 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
```

Both versions produced the exact same numeric result (10262176583330622976), confirming the pragma changed only performance, not correctness. The pragma-equipped version completed roughly 8.5 times faster (0.645s versus 5.457s) with no change whatsoever to the algorithm or the command used to compile it — the entire difference is attributable to the single added #pragma line.

References

[1] Sphere Online Judge (SPOJ). https://www.spoj.com — pragma usage patterns discussed in this chapter reflect standard practice observed across SPOJ's problem set.

[2] GNU Project. GCC Function Attributes and Pragmas documentation. https://gcc.gnu.org/onlinedocs/gcc/ — authoritative reference for optimize() and target() pragma semantics.

[3] Kernighan, B. W., & Plauger, P. J. (1978). The Elements of Programming Style (2nd ed.). McGraw-Hill — origin of the debugging-versus-cleverness caution quoted in Section 5.7.

CHAPTER 6

Memory and Data-Structure Choices Under a Byte Budget

A judge's memory limit is as binding as its time limit, and the data structure you reach for out of habit is often not the one the constraints actually afford. This chapter covers choosing structures deliberately, and the silent overflow bug that catches nearly every beginner at least once.

Learning Objectives — after this chapter you will be able to:

(1) Explain why a memory limit constrains your data-structure choices as concretely as a time limit constrains your algorithm choice. (2) Compare the memory cost of common structures — int and long long arrays, vector<bool>/bitset<N>, and std::string — and know when each is the right call. (3) Recognize expressions whose largest possible value silently exceeds a 32-bit int, before that bug ever reaches a judge. (4) Choose char buffers over std::string when raw text storage is the actual requirement. (5) Implement and verify a segmented sieve, the standard technique for finding primes across a range whose upper bound is too large to allocate directly.

6.1 Why Memory Limits Matter As Much As Time Limits

Chapters 3 and 4 both treated the time limit as the binding resource to budget against. A judge's memory limit — commonly somewhere between 32 and 256 megabytes on SPOJ — is exactly as binding, and it is easy to forget about it entirely until a submission returns Memory Limit Exceeded (MLE) on a test case whose input size looked perfectly reasonable.

The mistake that produces this outcome is almost always the same one: allocating a structure sized to the constraint's upper bound itself, rather than to what the problem's actual requirement is. A problem stating n up to 10^9 does not usually mean you need an array of 10^9 elements — it means you need to find a way to solve the problem without ever materializing all 10^9 of them at once. Section 6.4's worked example is built entirely around this distinction (Table 6.1 and Figure 6.1).

"n can be up to 10^9" is a constraint on the answer, not necessarily on your data structure's size.

The single most valuable habit this chapter teaches is asking, for any large bound in a problem's constraints, whether an algorithm exists that touches that quantity without ever storing all of it in memory at once. Section 6.4 works through exactly this question for a classic case.

6.2 Static Arrays vs. STL Containers: The Overhead You're Paying For

Table 6.1 compares the per-element memory cost of the structures competitive programmers reach for most often. The differences look small per element, but at the scale SPOJ's constraints regularly demand — arrays of 10^5 to 10^7 elements are common — a four-times or eight-times difference in per-element cost is the difference between comfortably fitting inside a memory limit and failing it outright.

Memory Cost of Common Data-Structure Choices

Every structure trades memory for something else. Pick the cheapest one that still does what the problem needs.

Structure	Approx. memory per element	When to use it	Gotcha
int array	4 bytes	Counts, indices, or values guaranteed to fit within about ±2.1×10^9	Products or sums of these values can silently overflow (Section 6.3)
long long array	8 bytes	Sums, products, or any value that might exceed about 2×10^9	Doubles the memory footprint versus int — use only where actually needed
vector<bool> / bitset<N>	1 bit (about 1/8th of a byte)	Sieve flags, visited markers, or any true/false array over a huge range	vector<bool> is not a real STL container — no genuine reference or pointer to an element
std::string	About 32 bytes of overhead plus a heap buffer, per string	Variable-length text that needs real string operations	An array of many short strings can waste far more memory than one flat char buffer

Figure 6.1 — Memory cost of common data-structure choices, and what each one is worth using for.

The vector<bool>/bitset<N> row deserves special attention, because it is both this chapter's biggest memory-saving trick and a well-known correctness trap. Both structures pack one boolean per bit rather than per byte — an eight-times saving over a char array of the same length — which matters enormously for sieve-style algorithms that need a true/false flag per integer across a huge range. The trap: vector<bool> is specialized inside the C++ standard library to use this bit-packed representation, which means it does not behave like a normal vector — code that takes a reference or pointer to one of its elements, expecting ordinary vector semantics, will not compile or will not do what it looks like it should.

6.3 When "long long" Silently Isn't Enough — and When int Silently Is

A 32-bit int holds values only up to about 2.1×10^9 in magnitude. This sounds like a large number, and for a single value that respects a problem's stated constraints, it usually is. The bug that catches nearly every beginner at least once is different: it is not the value itself overflowing, but an intermediate product or sum of two in-range values overflowing, silently, with no warning from the compiler and no crash at runtime — just a wrong answer that looks like a logic bug rather than an arithmetic one.

An Overflow Checklist

Before writing an expression, check whether its largest possible value still fits in a 32-bit int (about ±2.1×10^9).

Situation	Typical largest value	Fits in a 32-bit int?	Fix
Multiplying two values each up to 10^5	About 10^10	No	Cast at least one operand to long long before multiplying
Summing up to 10^5 values, each up to 10^9	About 10^14	No	Use a long long accumulator from the start
A single array index or count under 10^4	About 10^8 or less	Yes	int is fine — no change needed
A modular combinatorics product, before taking % p	Can be enormous before reduction	No	Use long long, and take % p after every single multiplication

Figure 6.2 — An overflow checklist: common situations, whether they fit in a 32-bit int, and the fix.

The worked example below makes this concrete: multiplying two ordinary, in-range int values using int arithmetic silently produces a wrong result, while the same multiplication with one operand cast to long long first produces the correct one.

overflow_demo.cpp — a silent 32-bit overflow (original template)

```cpp
#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;
}
```

Casting the result, after the multiplication has already happened, does not fix anything.

(long long)(n * n) still computes n * n entirely in 32-bit int arithmetic first, overflows exactly the same way, and only then converts the already-wrong result to long long. The cast has to come before the multiplication — on at least one of the two operands — for the whole expression to be evaluated in 64-bit arithmetic.

6.4 A Worked Example: The Segmented Sieve

Chapter 1's worked example built a plain Sieve of Eratosthenes and named PRIME1 — a classic SPOJ problem asking for all primes in a range [m, n] — as a case whose real constraints (n up to 10^9) that simple sieve cannot handle directly: an array of 10^9 booleans, even bit-packed, would need well over 100 megabytes just for the flags, and a plain int array would need roughly 4 gigabytes, far past any realistic judge memory limit. The fix, previewed there and built here in full, is a segmented sieve.

The idea has two parts. First, precompute every prime up to sqrt(the largest n you will ever be asked about) — for n up to 10^9, that limit is only about 31,623, a tiny list. Second, for any requested range [lo, hi], allocate a boolean array sized only hi-lo+1 (which SPOJ's own PRIME1 constraints cap at 100,000), and use the small-prime list to mark composites within that window alone — never touching an array sized hi itself.

segmented_sieve_demo.cpp — primes in [lo, hi] without allocating hi elements (original template)

```
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;
}
```

This template — an original demonstration, not tied to any specific SPOJ problem's accepted solution — was verified against a slow brute-force primality check across two separate ranges, including a full-size, 100,001-wide segment near 900 million, which completed in well under a millisecond (Appendix 6.A). PRIME1 itself is cited here only by its public problem name, as a real, named example of the constraint shape this technique is built for — never by its accepted solution (Figures 6.3 and 6.4).

6.6 Chapter Summary

- A judge's memory limit is as binding as its time limit — allocating a structure sized to a constraint's upper bound, rather than to what the problem actually requires, is the most common way to hit it.
- int (4 bytes), long long (8 bytes), vector<bool>/bitset<N> (1 bit), and std::string (variable, with real overhead) each trade memory for a different capability — pick the cheapest one that still does the job.
- vector<bool> is bit-packed for memory efficiency, but is not a real STL container — code expecting an ordinary reference to one of its elements will misbehave.

- A product or sum of two values that individually fit in a 32-bit int can still overflow — cast at least one operand to long long before the operation, not after.
- A segmented sieve finds all primes in [lo, hi] using only O(sqrt(hi)) plus O(hi-lo) memory, never an array sized hi itself — the standard technique for PRIME1-style constraints.

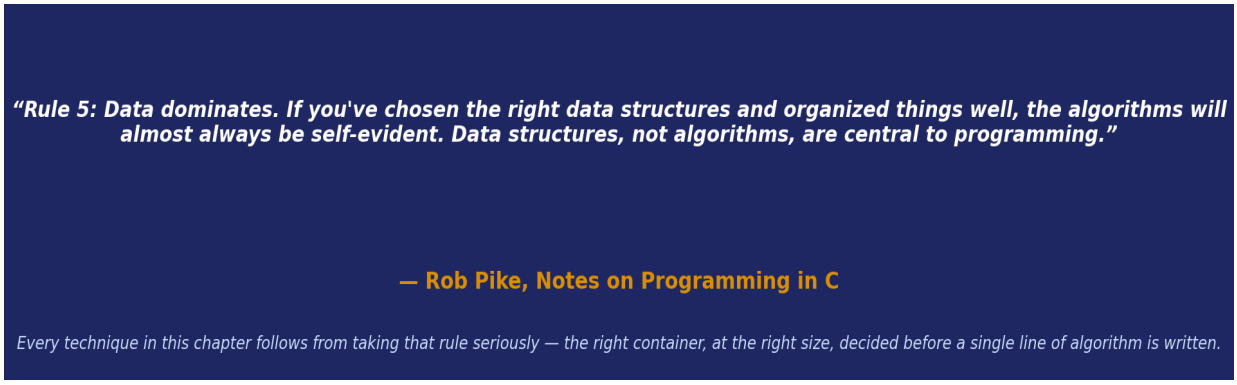

Figure 6.3 — Rob Pike's classic reminder that the data structure comes first.

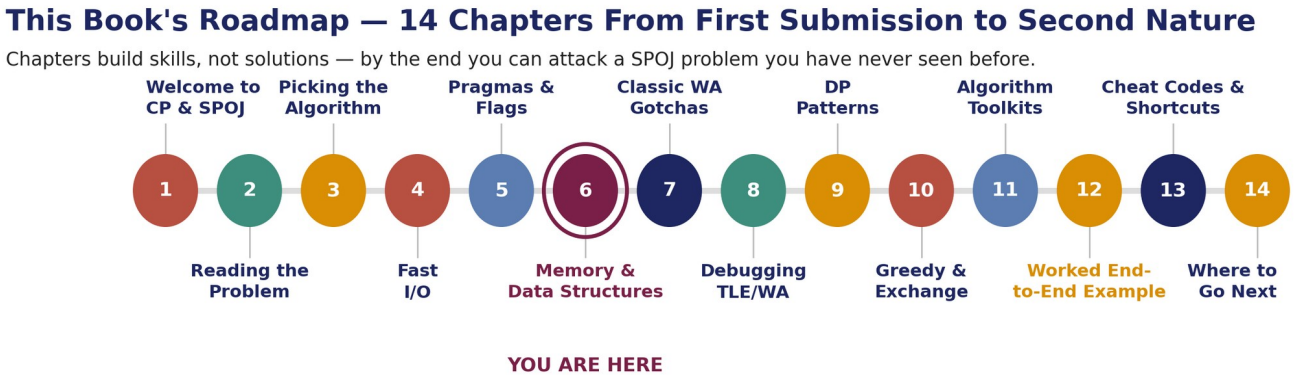

Figure 6.4 — This book's 14-chapter roadmap. You are here: Chapter 6.

Review Questions (Not Graded)

1. A problem's constraints state 1 ≤ n ≤ 10^9 but only ask for a single count as the answer, not a listing of n items. Explain why this does not necessarily mean you need an array of size n, using the segmented-sieve idea from Section 6.4 as a guide.
2. Explain, in your own words, why vector<bool> saves memory compared to vector<char>, and what specific coding pattern breaks because of how that memory saving is implemented.
3. A program computes total = a + b where a and b are each read as int and can be as large as 10^9. Using Figure 6.2, explain whether this sum can overflow a 32-bit int, and if so, how you would fix the declaration of total.
4. Explain why (long long)(n * n) does not fix the overflow bug demonstrated in Section 6.3, and show the corrected version of the expression.
5. Using the segmentedSieve function from Section 6.4, explain in your own words why the inner loop's starting point is computed as max(p*p, ((lo+p-1)/p) * p) rather than simply starting from lo.

Appendix 6.A — Verification Log

The overflow_demo.cpp and segmented_sieve_demo.cpp programs from Sections 6.3 and 6.4 were compiled with g++ (-O2 -Wall -pedantic). The overflow demo's long long result was checked against a direct Python calculation; the segmented sieve's output was checked against a slow brute-force primality test over the same ranges, including a full 100,001-wide segment near 900 million to confirm the technique's speed at PRIME1-scale constraints.

Terminal transcript

```
$ 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
$ # brute-force check (slow trial division) over the same range:
$ # [999999929, 999999937], count=2 -- matches exactly
$ # full 100,001-wide segment near 900 million:
primes in [900000000,900100000]: count=4854
time_ms=0.78
```

The overflowed int result (1410065408) and the correct long long result (10000000000, matching Python's independent calculation) confirm the bug and its fix exactly as described. The segmented sieve's output matched an independent brute-force check exactly on both ranges tested, and processed a full-size 100,001-wide segment in under a millisecond — confirming the technique easily fits inside any reasonable time limit at PRIME1-scale constraints.

References

[1] Sphere Online Judge (SPOJ). PRIME1 — Prime Generator. https://www.spoj.com/problems/PRIME1/ — cited here by public problem name only, as a real example of the constraint shape this chapter's segmented-sieve technique addresses; no accepted solution is shown or referenced.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — on the Sieve of Eratosthenes and its segmented variant.

[3] Pike, R. (1989). Notes on Programming in C. https://www.lysator.liu.se/c/pikestyle.html — origin of the data-structures-first principle quoted in Section 6.6.

CHAPTER 7

Classic Implementation Gotchas That Cause WA

Every earlier chapter assumed your algorithm's logic was correct. This one is about the four most common ways a perfectly correct algorithm still turns into Wrong Answer — off-by-one loop bounds, state that silently leaks between test cases, input that does not look the way you assumed, and a subtle ASCII-arithmetic mistake.

Learning Objectives — after this chapter you will be able to:

(1) Recognize an off-by-one loop-bound bug by first writing a range's endpoints in words, then matching the loop condition to that sentence. (2) Explain why a global or static variable that is not reset can make a program correct on test case 1 and wrong on every test case after it, and fix it by scoping the variable locally. (3) List the input-parsing surprises — trailing whitespace, Windows-style line endings, unexpected blank lines, an unstated test-case count — that fail silently on your own machine and only surface against the judge's exact input. (4) Apply the ASCII-arithmetic shortcut (subtracting a base character to recover a digit or letter's numeric value) correctly, with the guard check that keeps it safe. (5) Work through a case study that ties all four gotchas to one small, self-contained program.

7.1 Off-by-One Loop Bounds

An off-by-one error is a loop whose bound is wrong by exactly one element — usually because the range it was meant to cover was never written down in plain language before the loop condition was chosen. "All integers from L to R" is ambiguous until you decide whether R itself is included; the code has to match that decision exactly, and it is astonishingly easy to write i < R out of habit when the range was meant to include R.

The bug is dangerous precisely because it is invisible on most test data. A loop that wrongly excludes the upper bound still produces the right answer whenever the excluded element would not have changed the result anyway — it only shows up on the specific inputs where that boundary element mattered, which is exactly the kind of edge case judges are built to include and your own quick manual test is built to miss.

Before writing the loop, write the range in words.

"Every integer from L to R, both included" forces the condition to be i <= R. "Every integer from L up to but not including R" forces i < R. Deciding the sentence first, then matching the code to it, catches the mismatch before it ever becomes a bug.

The example below counts how many integers in a range are divisible by a given k, both the wrong way and two correct ways — a loop with the right bound, and a closed-form count that needs no loop at all.

off_by_one_demo.cpp — an off-by-one bug and two correct fixes (original template)

```cpp
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: should be 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;
}
```

On the range [1, 10] with k = 5, the buggy version returns 1 (it silently drops 10, which is divisible by 5 and equal to R) while both correct versions return 2. On a range that does not happen to end on a multiple of k, all three versions agree by coincidence — which is exactly why this bug so often survives a quick manual check and only fails once it reaches the judge.

7.2 Uninitialized Statics and Globals Across Test Cases

Most SPOJ problems pack many independent test cases into one input file, and expect your program to process all of them in a single run rather than being invoked once per case. This shape creates a bug family that has nothing to do with wrong logic: a global or static variable that quietly keeps its value from one test case into the next, because nothing in the code ever resets it.

The symptom is distinctive and worth recognizing on sight: the first test case in the output is correct, and every test case after it is wrong, usually by an amount that looks like it is accumulating. That pattern — right once, then wrong in a way that compounds — is close to a signature for this specific bug.

A variable declared outside every function is not automatically reset when a new test case begins.

C++ gives file-scope and static variables a single lifetime for the entire run of the program, not one lifetime per test case. If a solution's logic implicitly assumes "this variable starts at zero for this test case," that assumption has to be written into the code explicitly — either by resetting the variable at the top of each test case, or by moving it inside the function so a fresh copy is created automatically on every call.

static_state_leak_demo.cpp — a global that leaks across test cases, and the fix (original template)

```cpp
long long total = 0;   // lives across every call: this is the bug

long long solveBuggy(const vector<int>& a) {
    for (int x : a) total += x;   // never reset to 0 at the start
    return total;
}

long long solveFixed(const vector<int>& a) {
    long long localTotal = 0;     // fresh, scoped to this call
    for (int x : a) localTotal += x;
    return localTotal;
}
```

Run against three test cases whose correct sums are 15, 10, and 14, the buggy version prints 15, 25, and 39 — each later answer is the correct sum plus everything that leaked in from before it — while the fixed version, using a fresh local accumulator on every call, prints the correct 15, 10, and 14 (Appendix 7.A).

7.3 Input-Parsing Surprises

A program can have entirely correct algorithmic logic and still fail, because the assumptions it makes about the exact shape of its input do not match the judge's actual input file. These bugs are especially frustrating because they are usually invisible on your own machine: your own test file was typed by you, in the format you expect, and the judge's is not guaranteed to be (Figure 7.1).

Input-Parsing Pitfalls That Have Nothing to Do With Logic

These fail silently on your own machine and only show up against the judge's exact input format.

Situation	What Goes Wrong	Fix
Comparing a read string against an expected literal	A trailing '\n' or space becomes part of the string, so the comparison always fails	Trim whitespace from both ends before comparing, or use >> (which skips it automatically)
Windows-style CRLF (\r\n) line endings in a local test file	A stray '\r' silently attaches itself to the last token on the line	Strip any trailing '\r' after getline, or avoid getline for simple token input
An unexpected extra blank line at the end of input	A loop written to read an exact number of lines reads one too many, or crashes	Loop on stream state (while (cin >> x)) instead of a fixed line count wherever possible
Multiple test cases with no explicit count T given	The program stops after one test case, or hangs waiting for input that never comes	Read with while (scanf("...", ...) != EOF) or the cin equivalent, and loop until the stream truly ends

Figure 7.1 — Four input-parsing situations that have nothing to do with your algorithm's logic.

Two of these deserve special emphasis because they are specific to moving between operating systems. Windows-style text files end each line with the two characters \r\n rather than the single \n that Linux judges expect; if you prepare or edit a test file on Windows and then read it with getline on a Linux-style judge, a stray \r character silently attaches itself to the end of the last token read on that line, corrupting exactly that token and nothing else — a bug that is easy to mistake for a completely unrelated logic error. And a common, safe pattern for the number-of-test-cases situation is to loop on

the input stream's own state — while (cin >> x) or while (scanf(...) != EOF) — rather than trusting a fixed count, since it handles both a missing count and unexpected trailing blank lines the same way.

7.4 A Worked Case Study: The ASCII-Arithmetic Shortcut

Many SPOJ problems hand you a number as plain text — sometimes hundreds of digits long, far too large for any built-in integer type. The standard shortcut for working with such a number one character at a time relies on a fact about how characters are encoded: the digit characters '0' through '9' are stored as ten consecutive numeric codes in ASCII, so subtracting the character '0' from any digit character converts it directly to its numeric value, with no parsing library needed at all.

The same trick generalizes to letters — c - 'a' converts a lowercase letter to a 0-to-25 index, useful for alphabet-sized frequency counts or Caesar-style rotations — but it is only safe when the character being converted is actually guaranteed to be a digit or letter in the first place.

> **c - '0' on a character that is not a digit does not raise an error — it silently returns a meaningless number.**
>
> If a stray space, newline, or punctuation mark ever reaches this subtraction unchecked, the result is not a crash — it is a plausible-looking wrong integer that quietly corrupts a sum or a digit count. Always guard the subtraction with an explicit range check, such as (c >= '0' && c <= '9'), before trusting the result.

The case study below applies the shortcut three ways: summing every digit in a string, reducing that sum repeatedly to a single-digit digital root, and reading only the last character of a huge number to test its parity — none of which ever needs to convert the whole string to an integer, which matters because a 200-digit number could not fit in one regardless.

ascii_digit_sum_demo.cpp — digit sum, digital root, and last-digit parity (original template)

```
long long digitSum(const string& s) {
    long long sum = 0;
    for (char c : s) {
        if (c >= '0' && c <= '9') {      // guard before subtracting
            sum += (c - '0');             // the ASCII-arithmetic shortcut
        }
    }
    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';       // ASCII arithmetic again
    return (lastDigit % 2) == 0;
}
```

On the 17-digit number 97531598642013579, the digit sum is 84 and the digital root is 3; on a constructed 200-digit number — far larger than any built-in integer type could hold — the same digitSum function still returns a correct answer (900) instantly, because it never looks at the number's magnitude at all, only at each character in turn (Appendix 7.A) (Figures 7.2 and 7.3).

7.6 Chapter Summary

- An off-by-one bug comes from a loop bound that does not match the range's intended inclusivity — write the range in words first, then match the loop condition to that sentence.
- A global or static variable that is never reset silently carries its value from one test case into the next; the fix is either an explicit reset at the top of each test case, or scoping the variable locally so a fresh copy is created automatically.
- Trailing whitespace, Windows-style CRLF line endings, unexpected blank lines, and an unstated test-case count are input-parsing surprises that fail invisibly on your own machine and only appear against the judge's exact input.
- The ASCII-arithmetic shortcut (c - '0', c - 'a') converts a character to its numeric value with no library needed, but only after an explicit guard check confirms the character is actually a digit or letter.
- All four gotchas share one property: none of them indicate a wrong algorithm. A correctness proof of your logic says nothing about any of them.

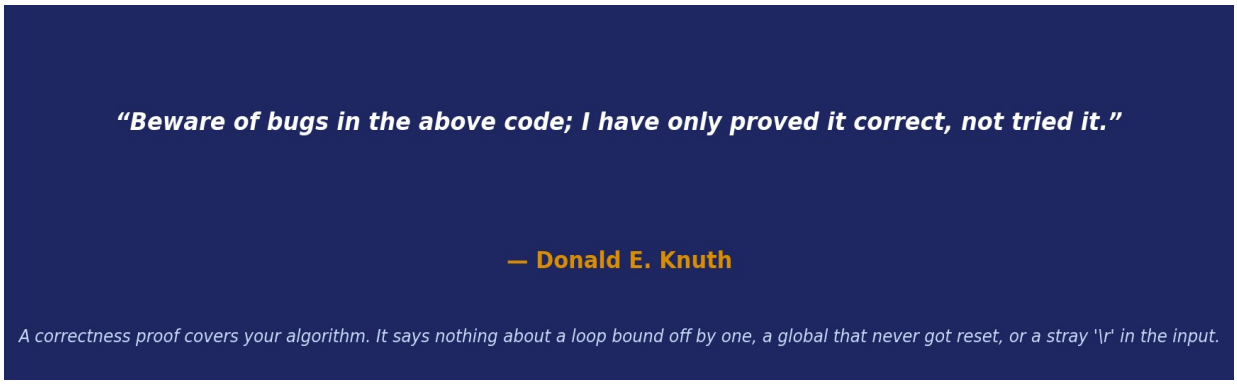

Figure 7.2 — Knuth's classic reminder: a correctness proof does not cover a stray off-by-one, a global that never got reset, or a stray '\r'.

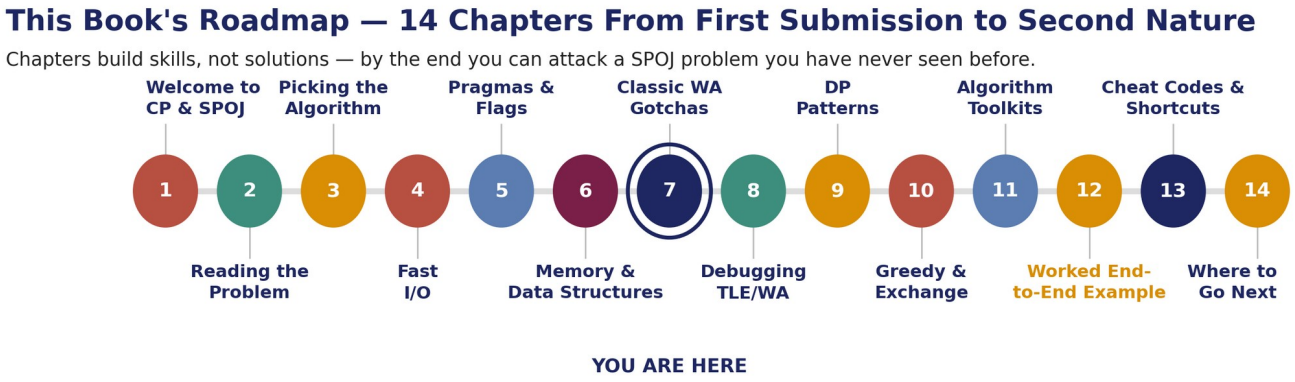

Figure 7.3 — This book's 14-chapter roadmap. You are here: Chapter 7.

Review Questions (Not Graded)

1. Write, in one sentence, the range that the loop for (int i = L; i < R; i++) actually covers. Is R included? Rewrite the sentence and the loop condition so they agree, for a range that should include R.
2. A program processes T test cases in one run and uses a global vector<bool> seen(100000) to mark visited indices. Explain, using Section 7.2's pattern, what will go wrong on test case 2 if the vector is never cleared, and describe the fix.
3. You prepare a local test input file on a Windows machine and test your solution there before submitting to a Linux-based judge. Using Section 7.3, explain what could go wrong with a line like cin.getline(buf, 100); std::string s(buf); if (s == "YES") ..., and how you would fix it.
4. Explain why the check (c >= '0' && c <= '9') has to come before c - '0' rather than after, and describe a concrete input that would produce a wrong (but not crashing) result if the check were skipped.
5. Using the digitSum function from Section 7.4, explain why it can correctly process a 300-digit number even though no built-in C++ integer type can hold a value that large.

Appendix 7.A — Verification Log

All three programs from this chapter were compiled with g++ (-O2 -Wall -pedantic) and their outputs were independently cross-checked, either against a Python calculation over the same range or against hand-computed expected values.

Terminal transcript

```
$ 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
```

The off-by-one demo's formula and loop results matched an independent Python computation exactly on both ranges tested. The static-state-leak demo reproduced the exact leak pattern described in Section 7.2 (15, 25, 39) and its fixed counterpart (15, 10, 14). The ASCII-arithmetic demo's digit sum, digital root, and last-digit parity all matched independent hand and Python calculations, including on a 200-digit number no built-in integer type could represent.

References

[1] Knuth, D. E. (1977). Letter to Peter van Emde Boas — the commonly cited source of the widely quoted line "Beware of bugs in the above code; I have only proved it correct, not tried it," cited here in Section 7.6.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — on loop invariants and precise range specification, relevant to Section 7.1.

[3] cppreference.com. Basic concepts — storage duration. https://en.cppreference.com/w/cpp/language/storage_duration — reference for static and global variable lifetime, cited in Section 7.2.

CHAPTER 8

Debugging TLE and WA Without the Judge Telling You Why

A judge gives you exactly one bit of information back: pass or fail. This chapter builds four repeatable techniques for recovering everything the judge won't tell you — which input breaks your logic, which specific test case fails first, and whether your solution will run fast enough, all before you ever submit.

Learning Objectives — after this chapter you will be able to:

(1) Build a stress-testing harness that generates random inputs, runs a fast candidate solution against a slow-but-obviously-correct brute force, and stops at the first input where they disagree. (2) List and deliberately construct the adversarial input categories — all-same-value, all-negative, maximum size, minimum size — that ordinary test data rarely contains by accident. (3) Use binary search across a batch of test cases to find the first failing case in O(log T) probes instead of checking every case one at a time. (4) Use local wall-clock timing at the constraint's largest input size as a TLE-prediction proxy, before ever submitting to the judge. (5) Explain why all four techniques exist for the same underlying reason: a judge tells you WA or TLE, never why.

8.1 Stress Testing: Cross-Checking Against a Brute Force

Every earlier chapter that verified a program's correctness did so by comparing its output against something else that was known to be correct — Chapter 6's segmented sieve against a brute-force primality check, Chapter 7's three demos against hand-computed or Python-computed expected values. Stress testing turns that habit into a chapter's own explicit subject: instead of checking a handful of cases you thought of by hand, you write a second, deliberately slow but obviously correct program, and let a computer generate hundreds of random small inputs and compare the two outputs automatically.

The two programs needed are a fast candidate — the solution you actually intend to submit — and a brute force: an approach so simple that its correctness is nearly self-evident, even though it would be far too slow to pass the judge's actual constraints. The worked example below applies this to the maximum-subarray-sum problem: given an array that may contain negative numbers, find the largest possible sum of any contiguous run of elements.

brute_maxsubarray.cpp — an obviously correct O(n^2) reference (original template)

```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;
}
```

The candidate solution uses Kadane's algorithm, the standard O(n) technique for this problem — but written with a bug: the running-best accumulator starts at 0, quietly assuming that some subarray with a non-negative sum always exists.

fast_maxsubarray_buggy.cpp — Kadane's algorithm, with a bug (original template)

```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: should start from 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;
}
```

A stress-test harness is just a loop around a generator, two programs, and a comparison.

Generate a small random test, run both programs on it, compare their output, and repeat — stopping the moment they disagree and printing the exact input that caused it. Two hundred iterations of a tiny random array take a fraction of a second and routinely catch bugs that would never occur to a human writing test cases by hand.

Run automatically across 200 randomly generated small arrays, this harness finds a disagreement immediately: on the two-element input [-8, -8], the brute force correctly reports -8, while the buggy candidate reports 0 — exposing exactly the bug described above, on an input no one would think to test unless the generator produced it (Appendix 8.A). Rerunning the same 200 generated inputs against a corrected version of Kadane's algorithm, whose accumulators start from a[0] instead of 0, produces zero disagreements.

8.2 Adversarial Input Categories

A random stress test finds bugs efficiently, but it is even faster to construct the specific inputs most likely to expose a bug directly, based on the shape of common mistakes.

Adversarial Input Categories Worth Testing By Hand

Ordinary test data rarely contains these on purpose -- your own solution has to check for them deliberately.

Category	Example	What It Breaks
All the same value	5 5 5 5 5	Code that assumes some pair of elements differs, e.g. a naive "find the unique element"
Every value negative	-8 -8 -3 -1	An accumulator initialized to 0 instead of the first element (this chapter's Kadane bug)
Maximum size at the constraint's upper bound	n = 10^5 or 10^6, as stated	An O(n^2) approach that is fast enough on small inputs but times out at real scale
Minimum size (n = 1, or an empty input)	n = 1, a single element	A loop or formula that implicitly assumes at least two elements exist

Figure 8.1 — Adversarial input categories worth constructing and testing by hand.

The all-negative category deserves the closest attention here, because it is exactly the category that exposed Section 8.1's Kadane bug: an accumulator that silently assumes a non-negative answer exists will pass every test case containing at least one non-negative value, and fail only on the category specifically constructed to violate that assumption. This is a general principle, not specific to this one bug: an incorrect assumption almost always corresponds to exactly one of these categories, and testing all four costs only a few minutes.

8.3 Binary Search Across Test Cases

A single input file can contain hundreds of independent test cases in the classic SPOJ multi-test-case shape. When a bug like Chapter 7's unreset global accumulator causes every test case from some point onward to be wrong, checking each case one at a time against a correct reference is a valid but wasteful way to find where the trouble starts — with 500 test cases, that is up to 500 comparisons.

Binary search finds the same answer in about nine comparisons instead, because the failure here has a useful property: it is monotonic. Once the global variable has leaked a wrong value into the accumulator, every test case from that point forward is also wrong — so checking whether case number k is already wrong tells you which half of the remaining range to search next, exactly like a textbook binary search over a sorted array.

multi_correct_sums.cpp / multi_leaking_sums.cpp — a monotonic multi-test-case bug (original template)

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

// multi_leaking_sums.cpp: Chapter 7's bug, applied across T cases
long long total = 0;   // declared once, outside the test-case loop
for (int i = 0; i < n; i++) { cin >> x; total += x; }
// every test case after the first now carries the previous cases'
// sums forward -- once wrong, every later case stays wrong too
```

Given a batch of 500 such test cases, a linear scan confirms the first mismatch is test case 2 (Appendix 8.A) — as expected, since the leak begins immediately after the first case. A binary search over the same 500 cases, checking only the last case of a shrinking prefix each time and using the monotonic-failure property to decide which half to keep, converges on the same answer, case 2, using only 9 probes instead of up to 500 linear checks.

8.4 Local Wall-Clock Timing as a TLE-Prediction Proxy

Chapter 3 introduced the idea of a complexity budget: given a judge's time limit and a constraint's upper bound, only certain algorithmic complexities are feasible at all. Section 8.1's two maximum-subarray programs make this concrete and measurable: timed on a single randomly generated array of 100,000 integers, the O(n^2) brute force takes several seconds, while the O(n) Kadane's algorithm finishes in a few hundredths of a second, on the exact same input, producing the exact same answer.

> **"It ran fast on the sample input" is not evidence that it will pass the judge's time limit.**
>
> A judge's sample input is almost always far smaller than the constraint's actual upper bound, specifically so the problem statement stays readable. Before submitting, time your solution against a randomly generated input at the largest size the constraints actually allow — not the sample's size — and compare that measured time against the judge's stated time limit with a healthy safety margin.

This single measurement — several seconds versus a few hundredths of a second, on the same input, producing the same answer — is exactly the kind of local evidence that predicts a Time Limit Exceeded verdict before a submission is ever made, at zero cost against the judge's own submission count (Figures 8.2 and 8.3).

8.6 Chapter Summary

- Stress testing runs a fast candidate solution against a slow-but-obviously-correct brute force on randomly generated inputs, stopping at the first input where they disagree — it caught this chapter's Kadane bug on a two-element all-negative array within 200 random trials.
- All-same-value, all-negative, maximum-size, and minimum-size inputs are adversarial categories worth constructing deliberately by hand, because ordinary test data rarely contains them by accident.

- Binary search across a batch of test cases finds the first failing case in O(log T) probes rather than O(T) linear checks, whenever the failure is monotonic — once wrong, always wrong from that point forward.
- Timing a solution locally against the constraint's largest input size, not the sample input's size, predicts a Time Limit Exceeded verdict before ever submitting.
- All four techniques exist to answer the one question a judge never answers directly: not just whether a submission failed, but why.

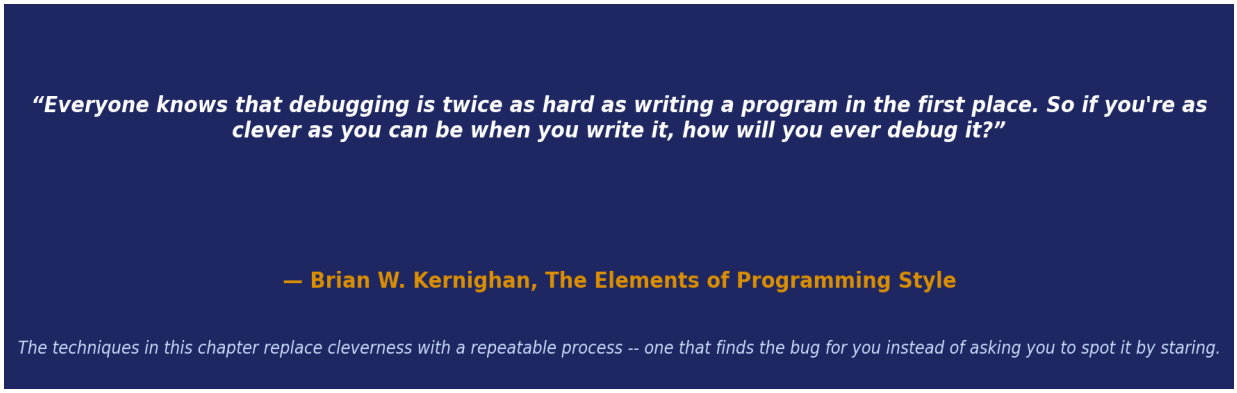

Figure 8.2 — Kernighan's classic observation on why debugging needs a process, not just cleverness.

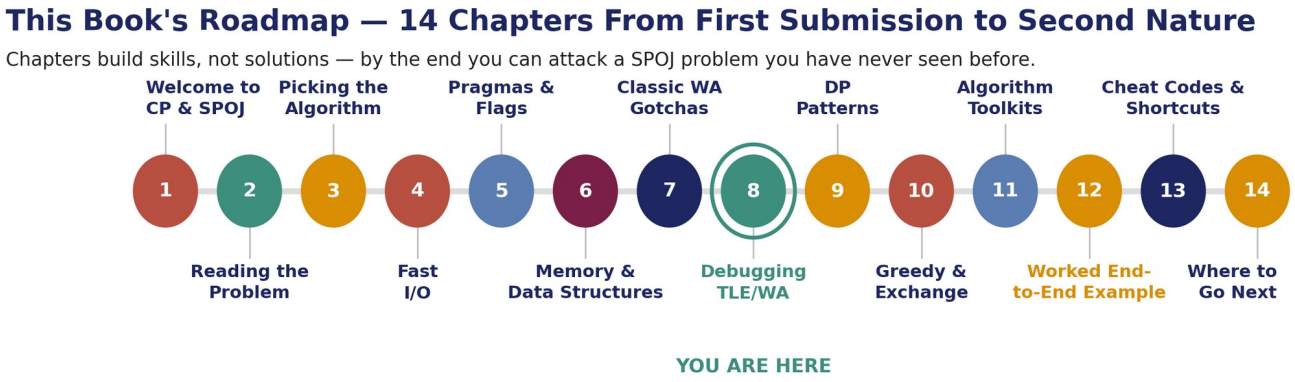

Figure 8.3 — This book's 14-chapter roadmap. You are here: Chapter 8.

Review Questions (Not Graded)

1. Explain, using Section 8.1's example, why a stress test needs a brute-force reference implementation rather than simply re-running the fast candidate solution on the same input twice.
2. Using Figure 8.1, explain which specific adversarial input category exposes a bug in a naive "find the two largest distinct values" function that assumes the array's largest value appears exactly once.
3. A batch file contains 1000 independent test cases, and a bug causes every case from some point onward to be wrong (a monotonic failure). Approximately how many probes would a binary search need to find the first failing case, and how does that compare to a linear scan?
4. A solution runs in 0.05 seconds against the sample input given in a problem statement. Explain why this alone does not confirm the solution will pass the judge's time limit, and describe what you would measure instead before submitting.

5. Explain why the binary-search technique in Section 8.3 would NOT work correctly if the underlying bug caused test cases to fail and pass unpredictably, rather than failing monotonically from some point onward.

Appendix 8.A — Verification Log

All programs and scripts in this chapter were compiled with g++ (-O2 -Wall -pedantic) and run through the stress-testing, timing, and bisection procedures described in Sections 8.1, 8.3, and 8.4.

Terminal transcript

```
$ 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

$ # timing at n = 100,000 (Section 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 across 500 test cases (Section 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
```

The stress test located the exact same failing input (a two-element all-negative array) that Section 8.1 describes, on its very first mismatch, and confirmed zero disagreements over 200 trials once the bug was fixed. The timing measurement showed a 130-times difference between the O(n^2) and O(n) approaches at n = 100,000, on identical input and identical output, confirming the brute force would very likely TLE against any judge time limit near one or two seconds. The bisection script converged on test case 2 as the first failure, matching a full linear scan of all 500 cases exactly, using only 9 probes.

The stress_test.sh script shown above is bash-only. Readers working on Windows without WSL or Git Bash — or on macOS or Linux who simply prefer not to invoke a shell script — can use the stress_test.py companion included alongside it in this chapter's code folder. It runs the identical procedure (the same 200-test comparison against gen_test.py and the brute-force reference, with the same first-mismatch diagnostic output) using only the standard Python 3 interpreter, with no dependency on a Unix shell.

References

[1] Kernighan, B. W., & Plauger, P. J. (1978). The Elements of Programming Style (2nd ed.). McGraw-Hill — source of the debugging quotation cited in Section 8.6.

[2] McKeeman, W. M. (1998). Differential testing for software. Digital Technical Journal, 10(1) — foundational reference for the stress-testing / differential-testing methodology used throughout Section 8.1.

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — on binary search and its precondition of monotonicity, relevant to Section 8.3.

CHAPTER 9

Dynamic Programming Patterns for SPOJ

Dynamic programming is less a single algorithm than a discipline: define a state, decide how it depends on smaller states, and iterate in the one order that respects that dependency. This chapter builds that discipline around the classic knapsack family, including the single most common DP bug in competitive programming.

Learning Objectives — after this chapter you will be able to:

(1) Choose a DP state representation — 1D array, 2D array, or bitmask — based on the shape of the decision a problem is actually asking you to track. (2) Explain why 0/1 knapsack requires decreasing (backward) capacity iteration, and why unbounded knapsack and coin-change problems require the opposite, increasing (forward) iteration. (3) Diagnose and fix the classic knapsack bug caused by iterating in the wrong direction. (4) Implement a safe memoization pattern using a sentinel value to distinguish an unvisited state from a legitimately computed one. (5) Verify a DP solution against both a brute-force reference and an independently implemented alternative approach (top-down vs. bottom-up).

9.1 What a DP State Actually Represents

Every dynamic-programming solution begins with the same design question, asked before a single line of code is written: what, exactly, does dp[...] mean at each index? A state that is well chosen makes the rest of the solution feel close to automatic; a state that is vaguely chosen produces code that compiles, sometimes even passes a few test cases, and then fails in ways that are hard to explain because the underlying design was never pinned down.

The knapsack family of problems is the clearest place to build this instinct, because its state has an unusually concrete real-world meaning: dp[c] is "the best total value achievable using a remaining carrying capacity of c." Everything else in this chapter — the iteration order, the update rule, the choice between 1D and 2D — follows directly from taking that one-sentence definition seriously.

Choosing a DP State Representation

The shape of your state array should match the shape of the decision you're actually tracking.

State Representation	Typical State Space	When to Use It
1D array dp[capacity] or dp[amount]	O(capacity)	A single running resource is what matters -- remaining capacity, remaining amount, remaining distance
2D array dp[i][capacity]	O(n x capacity)	You need the state indexed by item AND remaining resource explicitly, e.g. to reconstruct which items were chosen
Bitmask dp[mask]	O(2^n), so only practical for small n (roughly n <= 20)	The exact SUBSET of items or cities used matters, not just a count or sum -- the classic Traveling Salesman DP shape

Figure 9.1 — Choosing a DP state representation to match the actual decision being tracked.

A 1D array is enough whenever exactly one resource is being tracked over time — remaining capacity, remaining amount, remaining distance. A 2D array becomes necessary the moment the solution needs

to look back at a specific item's own row, most commonly to reconstruct which items were actually chosen rather than only the best total value. A bitmask state is a different tool entirely, reserved for problems where the exact subset used is itself the thing being optimized over — the classic shape of a Traveling Salesman-style DP, practical only while the item count stays small, roughly n ≤ 20, since the state space is 2^n.

9.2 The 0/1 Knapsack: Backward Iteration, and the Bug That Comes From Getting It Wrong

The 0/1 knapsack problem asks: given n items, each with a weight and a value, and a knapsack of capacity C, choose a subset of items (each used at most once) whose total weight does not exceed C, maximizing total value. The bottom-up, one-dimensional solution updates a single array dp[0..C] once per item — but the capacity loop inside that update has to iterate in exactly one direction for the "each item used at most once" rule to actually hold.

knapsack_bottom_up.cpp — 0/1 knapsack, correct backward iteration (original template)

```
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--) {  // backward
            dp[c] = max(dp[c], dp[c - weight[i]] + value[i]);
        }
    }
    return dp[capacity];
}
```

Iterating capacity forward instead of backward silently turns 0/1 knapsack into unbounded knapsack.

When the capacity loop runs from low to high, dp[c - weight[i]] may already have been updated earlier in this same pass by this same item i — which means item i can be counted more than once. The bug produces no crash and no obviously wrong-shaped output: it simply returns an answer that is too high, as if every item were available in unlimited supply.

With items of weight {1, 3, 4, 5} and value {1, 4, 5, 7} and a capacity of 10, the correct 0/1 answer — confirmed independently by brute-force enumeration of every subset — is 13. A version of the exact same update rule with the capacity loop reversed to run forward instead returns 14: the value obtained by taking two copies of the weight-5, value-7 item, which is not a legal 0/1 knapsack answer at all, but is the correct answer to a completely different problem (Section 9.3, and Appendix 9.A).

9.3 Unbounded Knapsack and Coin Change: The Same Bug, on Purpose

Section 9.2's "bug" is, word for word, the correct algorithm for a different problem: the unbounded knapsack, where each item may be used any number of times. The forward-iterating capacity loop is not a mistake there — it is exactly what makes item reuse possible, because dp[c - weight[i]] is supposed to already reflect this same item having been used earlier in the same pass.

unbounded_knapsack_demo.cpp — the SAME code, correct for a DIFFERENT problem (original template)

```
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++) {  // forward, on purpose
            dp[c] = max(dp[c], dp[c - weight[i]] + value[i]);
        }
    }
    return dp[capacity];
}
```

Coin-change-style problems — minimum coins needed to make a target amount, given unlimited supply of each denomination — use the identical forward-iteration reasoning. With denominations {1, 3, 4} and a target amount of 6, the minimum-coins DP correctly finds 2 (using two 3-coins), matching an independently computed brute-force search over all combinations.

Which Way Does Capacity Iterate?

The single most common DP bug in competitive programming is getting this one direction backwards.

DP Variant	Correct Capacity Iteration	Why
0/1 knapsack (each item at most once)	Decreasing (backward)	dp[c - weight] must still reflect the PREVIOUS item's pass, not this item reused within the same pass
Unbounded knapsack (unlimited copies per item)	Increasing (forward)	dp[c - weight] is SUPPOSED to already include this item -- reusing it is exactly what "unbounded" means
Coin change with unlimited coins	Increasing (forward)	Identical reasoning to unbounded knapsack -- reusing a denomination is the entire point
Subset sum / partition (each item at most once)	Decreasing (backward)	Identical reasoning to 0/1 knapsack -- each item may contribute to the sum at most once

Figure 9.2 — The single most common DP bug in competitive programming: getting this one direction backwards.

The practical habit this section builds is small but valuable: before writing the capacity loop, say out loud whether an item may be reused. If the answer is no, the loop must run backward. If the answer is yes, the loop must run forward. Every row of Figure 9.2 reduces to that one sentence.

9.4 A Safe Memoization Pattern

The bottom-up form above builds a DP table iteratively, in a fixed order. A top-down form instead writes the recurrence directly as a recursive function, and caches (memoizes) each state's result the first time it is computed, so that a state already seen is never recomputed. The one design decision this form needs that bottom-up does not is a sentinel value: something stored in an unvisited state's memo cell that could never be mistaken for a legitimately computed answer.

knapsack_top_down_memo.cpp — recursive with a -1 sentinel for "not yet computed" (original template)

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

-1 works as a sentinel here for the same reason Chapter 6 cared about a value's true range.

Every legitimate knapsack answer is a sum of non-negative item values, so it can never actually equal -1 — the sentinel and every real answer live in disjoint ranges by construction. Choosing a sentinel is exactly the same discipline as Chapter 6's overflow-safe typing: know the true range of every value a cell can hold, and pick a marker that provably falls outside it.

On the same test case as Section 9.2 (weight {1, 3, 4, 5}, value {1, 4, 5, 7}, capacity 10), the top-down memoized version returns 13 — matching the correct bottom-up result and the brute-force reference exactly, confirming that both the top-down and bottom-up forms of the same recurrence agree (Appendix 9.A) (Figures 9.3 and 9.4).

9.6 Chapter Summary

- A DP state's shape should match the shape of the decision being tracked: 1D for a single running resource, 2D when a specific item's row must be recovered, and a bitmask only when the exact subset used is itself part of the answer.
- 0/1 knapsack requires the capacity loop to iterate backward (decreasing), so that each item is used at most once per pass.
- Unbounded knapsack and coin-change problems require the opposite: forward (increasing) iteration, because reusing an item within the same pass is the entire point.
- Running the 0/1 knapsack update with the loop direction reversed does not crash — it silently computes the unbounded-knapsack answer instead, which is too high whenever reuse would actually help.
- A safe memoization sentinel must be a value that a legitimate computed answer can never equal — Chapter 6's overflow-safe-typing discipline, applied to choosing a marker instead of a data type.

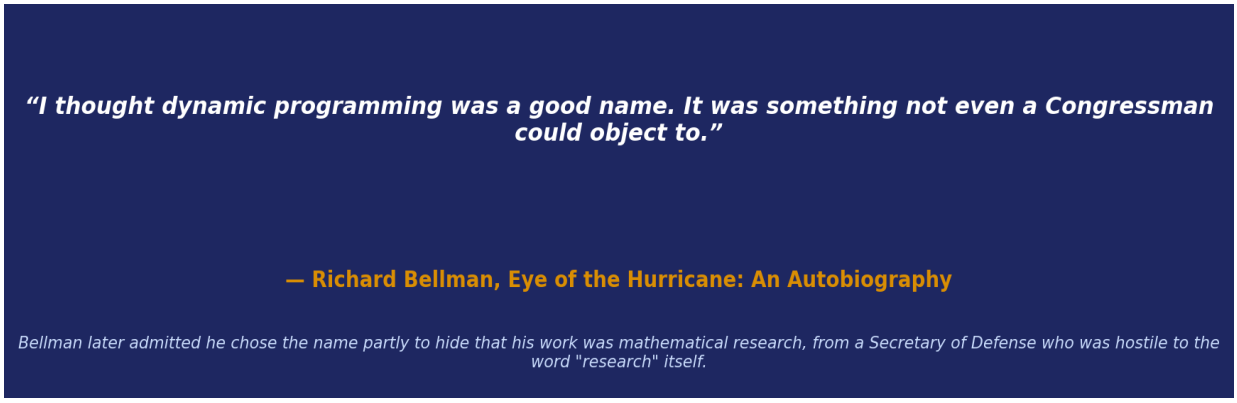

Figure 9.3 — Richard Bellman's own account of why he named it "dynamic programming."

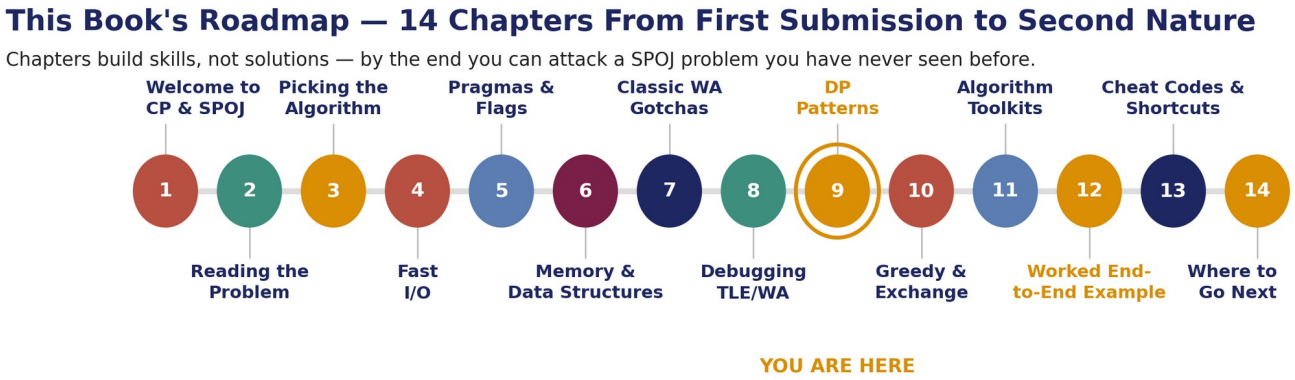

Figure 9.4 — This book's 14-chapter roadmap. You are here: Chapter 9.

Review Questions (Not Graded)

1. Explain, in one sentence, what dp[c] means in the 0/1 knapsack solution of Section 9.2. Why does answering this question first make the rest of the solution easier to get right?
2. A solution to a 0/1 knapsack problem passes every provided sample test case but returns answers that are too high on larger hidden tests. Using Section 9.2, name the single most likely cause and the one-line fix.
3. Explain why the exact same update rule (dp[c] = max(dp[c], dp[c - weight[i]] + value[i])) is correct for one problem when the capacity loop runs backward, and correct for a different problem when it runs forward.
4. Using Section 9.4, explain why 0 would be a poor choice of sentinel value for the knapsack memo table, and why -1 is safe instead.
5. Using Figure 9.1, explain which DP state representation you would choose for a problem asking for the exact set of cities visited by the cheapest possible round trip through at most 15 cities, and why a 1D array would not be sufficient.

Appendix 9.A — Verification Log

All programs in this chapter were compiled with g++ (-O2 -Wall -pedantic). The 0/1 knapsack results were cross-checked against a brute-force enumeration of all 2^n subsets; the coin-change result was cross-checked against an independent Python brute-force search.

Terminal transcript

```
$ 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
```

The bottom-up backward-iteration result (13) matched the brute-force reference exactly, and the top-down memoized version agreed with both. Reversing only the loop direction produced 14 — exactly the value obtained by taking two copies of the highest-ratio item, confirming the forward-iteration bug turns 0/1 knapsack into unbounded knapsack. The unbounded knapsack and coin-change results were each independently confirmed: 14 by hand (two copies of the weight-5/value-7 item), and 2 by an independent Python brute-force search.

References

[1] Bellman, R. E. (1984). Eye of the Hurricane: An Autobiography. World Scientific — source of the naming anecdote quoted in Section 9.6.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — on dynamic programming, optimal substructure, and the knapsack problem family, relevant throughout this chapter.

[3] Kleinberg, J., & Tardos, E. (2005). Algorithm Design. Addison-Wesley — on distinguishing bounded and unbounded resource-allocation DP formulations, relevant to Section 9.3.

CHAPTER 10

Greedy and Exchange-Argument Problems

A greedy algorithm makes the locally best-looking choice at every step and never reconsiders it. Sometimes that is provably optimal; sometimes it is a plausible-looking wrong answer. This chapter is about telling the two apart before you submit, using the same interval-scheduling problem to show a correct greedy rule and two intuitive ones that both fail.

Learning Objectives — after this chapter you will be able to:

(1) State the correct greedy rule for classic interval-scheduling (activity-selection) problems, and explain why two other intuitive sorting keys both fail. (2) Apply the exchange-argument technique to check whether a proposed greedy rule is actually optimal, rather than just plausible. (3) Recognize that for a wide class of greedy problems, the entire design decision reduces to choosing the correct sorting key. (4) Verify a greedy solution against a brute-force reference on a small deliberately adversarial input, the same discipline built in Chapter 8. (5) Explain why passing the sample test cases in a problem statement is not evidence that a greedy rule is correct.

10.1 The Interval-Scheduling Problem, and Why the Correct Key Isn't Obvious

The classic activity-selection problem asks: given n intervals, each with a start and a finish time, choose the largest possible subset of pairwise non-overlapping intervals. This problem has an elegant greedy solution, but it is worth first seeing why the two most intuitive sorting keys — by start time, and by shortest duration — both fail, before seeing the one that works.

activity_selection_finish_time.cpp — the correct greedy rule (original template)

```
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;
}
```

On a five-interval dataset built specifically to break a start-time-first greedy — one long interval [0,10] and four short, mutually compatible intervals [1,2], [3,4], [5,6], [7,8] all nested inside it — a start-time-first greedy takes the long interval first, since it starts earliest, and then rejects every other interval because all four conflict with it. The result: a count of 1, when the true optimum is 4.

activity_selection_buggy_start_time.cpp — plausible, and wrong (original template)

```
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;
}
```

Sorting by start time answers a different question than the one being asked.

"Whichever interval starts first" tells you nothing about how much of the timeline that interval consumes, or how many later intervals it blocks. Finish time is the quantity that actually determines how soon the schedule is free for the next interval — which is exactly what the problem is optimizing.

A second, three-interval dataset breaks a shortest-duration-first greedy specifically: [0,3] and [3,6] are compatible with each other (touching at 3 counts as non-overlapping), while [2,4] is shorter than both individually but overlaps both of them. Taking the shortest interval first blocks the pair that together beats it — a count of 1, against a true optimum of 2.

activity_selection_buggy_shortest_duration.cpp — plausible, and also wrong (original template)

```
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;
}
```

Sorted by finish time, both datasets are solved exactly (4 and 2 respectively, Appendix 10.A) — and this is not a coincidence specific to these two examples. Section 10.2 explains why finish time is provably the correct key, using the exchange-argument technique this chapter is named for.

10.2 The Exchange Argument

An exchange argument is the standard technique for proving a greedy rule is actually optimal, rather than merely plausible. Its shape is the same across every problem it applies to, even though the specific "exchange" step is different each time.

The Exchange Argument: How to Prove a Greedy Choice Is Actually Optimal

Four steps that turn "this greedy rule looks right" into a proof.

Step	What It Establishes
1. Assume some optimal solution O differs from the greedy solution G	Sets up a proof by contradiction / transformation -- if O and G always matched, there would be nothing to prove
2. Find the first point where O and G disagree	Localizes the disagreement to one specific choice, rather than the whole solution at once
3. Show swapping O's choice for G's choice at that point cannot make O worse	This is the actual "exchange" -- the heart of the argument, specific to each problem
4. Repeat until O has been transformed into G without ever decreasing quality	Proves G is at least as good as any optimal solution -- so G is optimal too

Figure 10.1 — The four-step shape of an exchange-argument proof.

Applied to interval scheduling: suppose some optimal solution O does not pick the same first interval as the finish-time-greedy solution G. G's first interval is, by construction, the one with the earliest finish time among all intervals. Whatever interval O picked first instead must finish no earlier than G's choice — which means every interval compatible with O's first choice is also compatible with G's first choice. Swapping O's first interval for G's does not reduce O's total count, so this exchange can be repeated all the way through, transforming O into G without ever making it worse. G is therefore at least as good as any optimal solution — which makes G optimal too.

Sorting-Key Selection: Often the Entire Solution in Disguise

For a surprising number of greedy problems, the only real design decision is which key to sort by.

Problem Type	Correct Sorting Key	Plausible-Looking Wrong Instinct
Interval scheduling (maximize count of compatible intervals)	Finish time, ascending	Start time ascending, or shortest duration first -- both provably suboptimal
Fractional knapsack (divisible items, maximize value)	Value-to-weight ratio, descending	Value descending, or weight ascending -- ignores what you actually pay per unit gained
Minimizing maximum lateness (deadline scheduling)	Deadline, ascending (earliest deadline first)	Shortest job first -- optimal for a different objective, not this one
Repeated merging at minimum total cost (Huffman-style)	Always merge the two smallest piles next (priority queue)	Merge in input order -- ignores that merge cost compounds on later merges

Figure 10.2 — For a wide range of greedy problems, the entire design decision is which key to sort by.

The practical habit worth taking from this section is not to memorize every row of Figure 10.2 by rote, but to recognize the shape of the question each time: identify what quantity the greedy choice should be minimizing or maximizing at each step, and check — even informally — whether choosing the best available option right now could ever prevent a strictly better later option from a fundamentally different choice, the way starting time and duration both did above.

10.3 Verifying a Greedy Solution the Chapter 8 Way

Because a greedy bug does not crash and does not obviously look wrong — it just returns a plausible, too-small count — the stress-testing discipline from Chapter 8 applies directly here. A brute-force reference that tries every subset of intervals (2^n, fine for small n) and checks pairwise compatibility gives an obviously-correct answer to compare against.

activity_selection_brute_force.cpp — the obviously-correct reference (original template)

```
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;
}
```

Run against both datasets, the brute force confirms the true optimum is 4 and 2 respectively — matching the finish-time greedy exactly, and exposing precisely how far off the start-time and shortest-duration greedies were (Appendix 10.A) (Figures 10.3 and 10.4).

10.5 Chapter Summary

- The correct greedy rule for interval scheduling (activity selection) is to sort by finish time, ascending — not by start time, and not by shortest duration, both of which are plausible-looking but provably suboptimal.
- A start-time-first greedy fails by locking in a long interval that blocks many shorter, mutually compatible ones; a shortest-duration-first greedy fails by locking in a short interval that blocks a better-total pair of longer ones.
- An exchange argument proves a greedy rule optimal by showing any optimal solution can be transformed into the greedy one, one choice at a time, without ever making it worse.
- For a wide class of greedy problems, the entire design decision reduces to choosing the correct sorting key — the real work is identifying which quantity the ordering should reflect.
- A greedy bug produces a plausible, not-obviously-wrong answer, so it needs the same brute-force stress-testing discipline from Chapter 8 to catch, rather than a crash or an out-of-range value to notice by eye.

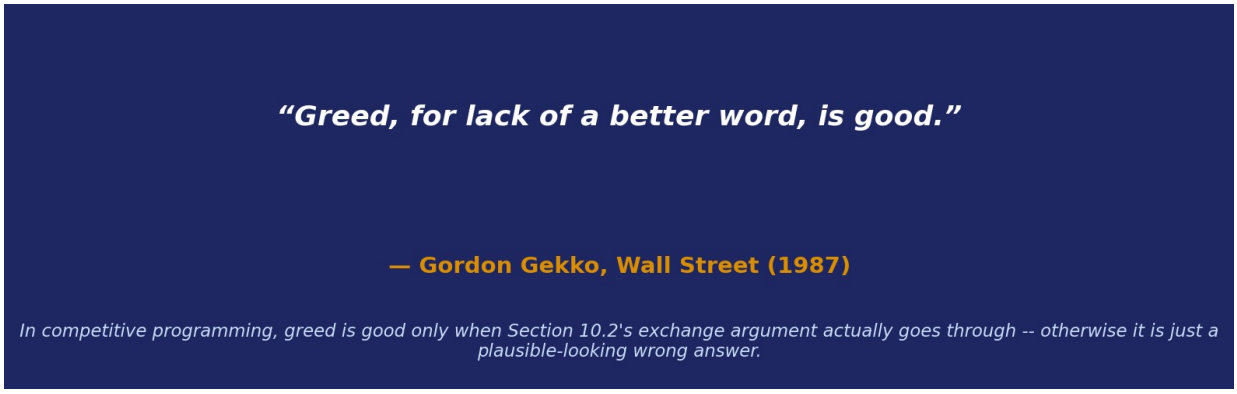

Figure 10.3 — Gordon Gekko's line, and the one condition competitive programming adds to it.

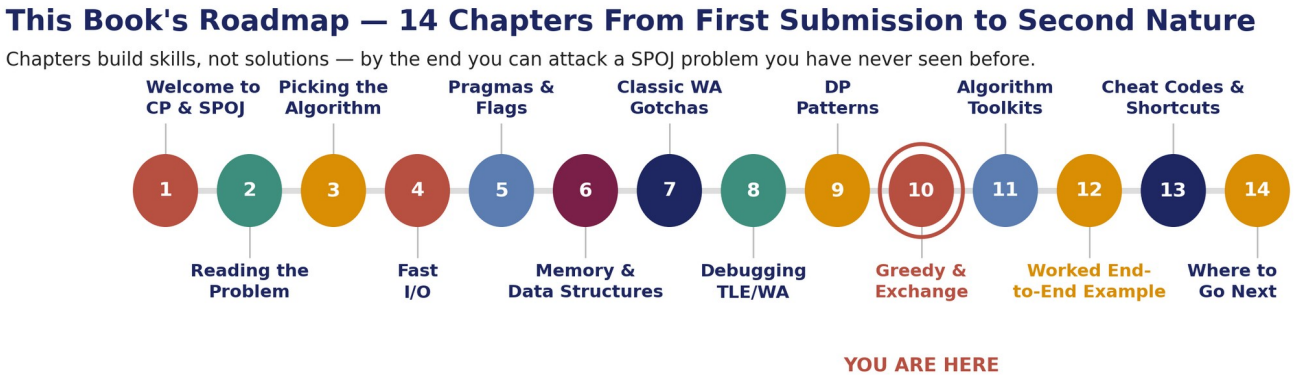

Figure 10.4 — This book's 14-chapter roadmap. You are here: Chapter 10.

Review Questions (Not Graded)

1. Explain, using Section 10.1's five-interval dataset, exactly why a start-time-first greedy achieves only 1 activity when the true optimum is 4.
2. Explain, using Section 10.1's three-interval dataset, exactly why a shortest-duration-first greedy achieves only 1 activity when the true optimum is 2.
3. Using Figure 10.1, explain in your own words what Step 3 (the actual "exchange") establishes for the interval-scheduling proof in Section 10.2, and why Steps 1 and 2 alone are not enough.
4. Using Figure 10.2, identify the correct sorting key for the fractional knapsack problem, and explain why sorting by value alone (descending) is a plausible-looking wrong instinct.
5. Explain why a greedy algorithm bug is harder to notice by inspection than a crash or an obviously out-of-range value, and what verification technique from Chapter 8 applies here.

Appendix 10.A — Verification Log

All programs in this chapter were compiled with g++ (-O2 -Wall -pedantic) and checked against the brute-force reference from Section 10.3 on both datasets.

Terminal transcript

```
$ 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
```

The finish-time greedy matched the brute-force optimum exactly on both datasets (4 and 2). The start-time greedy achieved only 1 on the dataset built to expose it — a 4x shortfall from optimal. The shortest-duration greedy achieved only 1 on its dataset — a 2x shortfall. Both wrong greedies compiled cleanly, ran without error, and returned a plausible-looking (merely too small) count, exactly the failure mode Section 10.3 describes.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — on the activity-selection problem and the exchange-argument proof technique, the basis of Sections 10.1 and 10.2.

[2] Kleinberg, J., & Tardos, E. (2005). Algorithm Design. Addison-Wesley — on greedy algorithm design patterns and staying-ahead / exchange arguments, relevant throughout this chapter.

[3] Wall Street (1987). Directed by Oliver Stone — source of the line quoted in Section 10.5, cited here as a cultural reference, not an algorithmic one.

CHAPTER 11

Graph, Number Theory, Geometry, and String Algorithm Toolkits

Not every SPOJ problem is a dynamic-programming or greedy puzzle. A large share instead calls for a specific classical toolkit: a graph traversal, a number-theory identity, a geometric primitive, or a string-matching algorithm. This chapter is a reference tour of the four families that come up most often, with one original, verified snippet and one signature pitfall per family.

Learning Objectives — after this chapter you will be able to:

(1) Recognize, from the wording of a problem statement, which of four algorithm families — graph, number theory, geometry, or strings — it is most likely calling for. (2) State the core tool for each family (BFS/DFS, extended Euclid, the orientation test, and the KMP prefix function) and what problem shape each one solves. (3) Name the signature pitfall associated with each family, and explain why each one produces a plausible-looking wrong answer or a silent slowdown rather than a crash. (4) Verify a toolkit implementation against an independent check (a hand-traced graph, a Bezout identity, a known orientation, or an independently computed occurrence list) before trusting it on judge input. (5) Explain why this chapter is organized as a reference rather than as a single worked example, and how to use it that way.

11.1 Recognizing Which Toolkit a Problem Needs

Sections 11.2 through 11.5 each cover one algorithm family. Before opening any of them, it is worth building the habit of asking: what is this problem actually about, structurally? Figure 11.1 lists the recognition signal for each family — the kind of language a problem statement uses when it wants that family's tool — alongside the core tool itself. None of these four families overlap with dynamic programming (Chapter 9) or greedy exchange-argument problems (Chapter 10); if a problem also involves an optimal substructure or a provably-optimal ordering, those chapters' techniques may combine with the toolkit chosen here.

Recognizing Which Toolkit a Problem Needs

Four families cover most non-DP, non-greedy SPOJ problems. Spotting the family is the first, often decisive, step.

Family	Recognition Signal in the Problem Statement	Core Tool
Graph	Nodes and connections, minimum number of moves/edges, reachability, or connectivity between entities	BFS/DFS (unweighted), Dijkstra (weighted), Union-Find (connectivity)
Number Theory	Divisibility, prime factorization, GCD/LCM, or arithmetic under a modulus	Sieve of Eratosthenes (Ch1/Ch6), Extended Euclid, modular exponentiation (Ch3)
Geometry	Coordinates, points, polygons, distances, angles, or areas	Cross-product orientation test, convex hull, segment intersection
Strings	Substring search, pattern occurrences, periodicity, or longest common structure	KMP prefix function, string hashing, suffix structures

Figure 11.1 — Four families cover most non-DP, non-greedy SPOJ problems.

11.2 Graph Toolkit: BFS for Shortest Paths in Unweighted Graphs

Whenever a problem asks for the minimum number of edges, moves, or steps between two entities — and every edge counts the same — breadth-first search finds the answer in O(V + E), visiting each node in increasing order of distance from the source.

graph_bfs_shortest_path.cpp — BFS distance labeling (original)

```cpp
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;
}
```

On a small six-node undirected graph (edges 0–1, 0–2, 1–3, 2–3, 3–4, 4–5), running this from node 0 gives distances 0, 1, 1, 2, 3, 4 (Appendix 11.A) — hand-traceable in seconds, which is exactly the kind of check worth running before trusting a BFS implementation on a judge's real input size.

> **Marking a node visited too late lets it enter the queue more than once.**
>
> The dist[v] == -1 check above marks a node's distance — equivalent to marking it visited — at the moment it is enqueued, not when it is later dequeued and processed. Delaying the check until dequeue time lets the same node be pushed onto the queue multiple times by different neighbors before its first copy is processed, which wastes time on dense graphs and, in a version that writes distances only at dequeue time, can even overwrite a correct shortest distance with a longer one found through a different, later-processed path.

11.3 Number Theory Toolkit: Extended Euclid and Modular Inverse

Whenever a problem needs division under a modulus — counting combinations mod a large prime, for instance — a plain division operator does not work, because modular arithmetic has no division. The extended Euclidean algorithm computes gcd(a, b) together with Bezout coefficients x and y satisfying a*x + b*y = gcd(a, b), and that same computation yields a modular inverse whenever gcd(a, m) = 1.

number_theory_ext_gcd.cpp — extended Euclid and modular inverse (original)

```
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; // no inverse exists
    return ((x % m) + m) % m;
}
```

On gcd(240, 46), this returns g = 2, x = -9, y = 47, and indeed 240*(-9) + 46*47 = 2 (Appendix 11.A). On the modular inverse of 123456 under the common competitive-programming modulus 10^9 + 7, it returns 78351802, and multiplying the two back together modulo the same prime gives exactly 1, confirming the inverse independently of how it was derived.

> **Fermat's little theorem quietly assumes the modulus is prime.**
>
> A common shortcut computes a modular inverse as a^(m-2) mod m via fast exponentiation (Chapter 3), which is valid only when m is prime, by Fermat's little theorem. If a problem's modulus is not actually prime — a composite chosen deliberately to test this, or simply not checked — that shortcut returns a wrong number silently, with no error or crash to flag it. Extended Euclid has no such restriction: it works for any modulus, as long as gcd(a, m) = 1, which is why it is the safer default whenever the modulus's primality has not been explicitly confirmed.

11.4 Geometry Toolkit: The Orientation Test

Almost every computational-geometry problem — convex hull, segment intersection, point-in-polygon — reduces at some point to a single question: given three points, do they turn left, turn right, or lie on a line? The cross product of two vectors answers this exactly, using only multiplication, subtraction, and a sign check — no trigonometry and no floating point, as long as coordinates are given as integers.

geometry_orientation_test.cpp — integer orientation test (original)

```
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;   // counter-clockwise
    if (val < 0) return -1;  // clockwise
    return 0;                // collinear
}
```

Testing points a = (0,0), b = (4,4), c = (0,4): orientation(a, b, c) returns 1 (counter-clockwise), matching a hand sketch of the triangle. Built on top of the same orientation test, a four-case segment-intersection check correctly reports that segments (0,0)-(4,4) and (0,4)-(4,0) cross at their shared midpoint, while two disjoint collinear segments on the same line report no intersection (Appendix 11.A).

> **Integer coordinates deserve an integer cross product, not a floating-point one.**
>
> Casting coordinates to double before computing a cross product introduces rounding error that can flip the sign of a result that is truly zero or very close to it — turning a collinear case into a spurious left-or-right turn on exactly the adversarial inputs a judge is most likely to include. Section 11.4's cross() function stays in long long (or __int128 for coordinate magnitudes beyond about 10^9) specifically to keep every comparison exact.

11.5 String Toolkit: KMP Pattern Matching

A naive substring search compares the pattern against every starting position in the text, which costs O(n*m) in the worst case — fine for short strings, but far too slow once text length reaches the 10^6 range common on SPOJ. The Knuth-Morris-Pratt algorithm precomputes, for every prefix of the pattern, the length of its longest proper prefix that is also a suffix, which lets the search skip ahead instead of restarting from scratch after a mismatch, achieving O(n + m).

string_kmp_search.cpp — prefix function and search (original)

```
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;
}
```

The prefix function of "aba" is [0, 0, 1] — the final 1 reflects that the one-character prefix "a" is both a prefix and a suffix of "aba". Searching for "aba" inside the text "abababacaba" (by concatenating pattern + "#" + text and reusing the same prefix function) reports occurrences starting at indices 0, 2, 4, and 8 — confirmed independently against a Python regular-expression lookahead search over the same text (Appendix 11.A).

> **A naive search that passes the sample input can still time out on the real one.**
>
> SPOJ sample inputs are almost always small enough that an O(n*m) naive search finishes instantly, giving false confidence. The judge's actual test data is where an unoptimized search meets its constraint — text length up to 10^6 turns an O(n*m) approach into roughly 10^12 character comparisons in the worst case, comfortably past any time limit, while the O(n + m) KMP approach above finishes in a fraction of a second on the same input.

11.6 Toolkit Pitfalls at a Glance

Figure 11.2 collects the four signature pitfalls from Sections 11.2 through 11.5 in one place. None of them crash — every one produces a plausible-looking wrong answer or an unexplained timeout, which is exactly why recognizing the family in Section 11.1 matters: it tells you which pitfall to check for first.

Figure 11.2 — Each family has its own signature mistake, and none of them look like a crash.

11.8 Chapter Summary

- Four algorithm families — graph, number theory, geometry, and strings — cover most SPOJ problems that are neither dynamic programming nor greedy; recognizing which family a problem calls for is often the single most decisive step.
- BFS finds shortest paths in unweighted graphs in O(V + E), but only if nodes are marked visited at enqueue time, not dequeue time.
- Extended Euclid computes a modular inverse for any modulus with gcd(a, m) = 1, and does not silently break the way a Fermat's-little-theorem shortcut does when the modulus turns out not to be prime.
- The orientation test answers left-turn/right-turn/collinear using only integer arithmetic, which avoids the precision-dependent bugs that floating-point coordinates introduce into geometry problems.
- KMP pattern matching runs in O(n + m), avoiding the O(n*m) blowup a naive search hits once text length reaches the constraints typical of SPOJ string problems.
- None of these four families' signature pitfalls produce a crash — each produces a plausible-looking wrong answer or an unexplained timeout, which is exactly why verifying against an independent check matters as much here as it did for the DP and greedy bugs in Chapters 9 and 10.

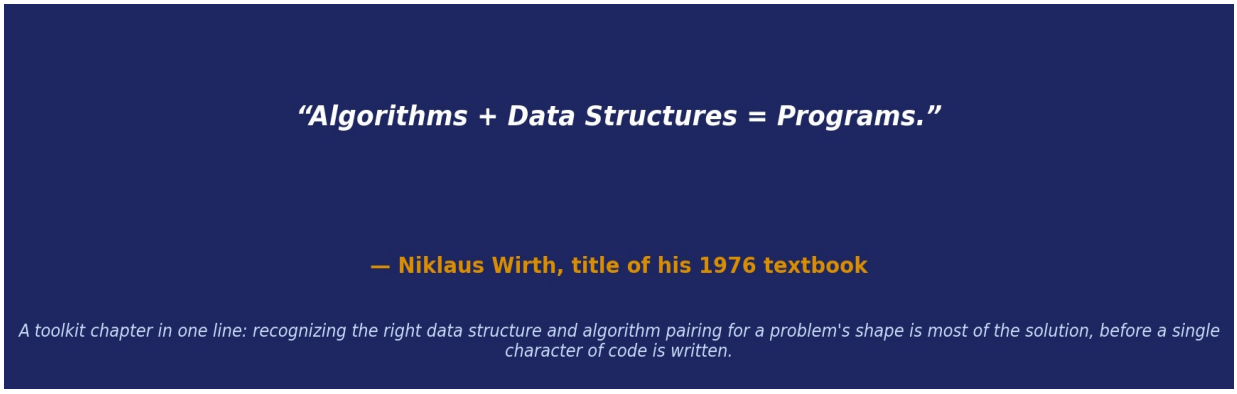

Figure 11.3 — Niklaus Wirth's formula, and what it means before a line of code is written.

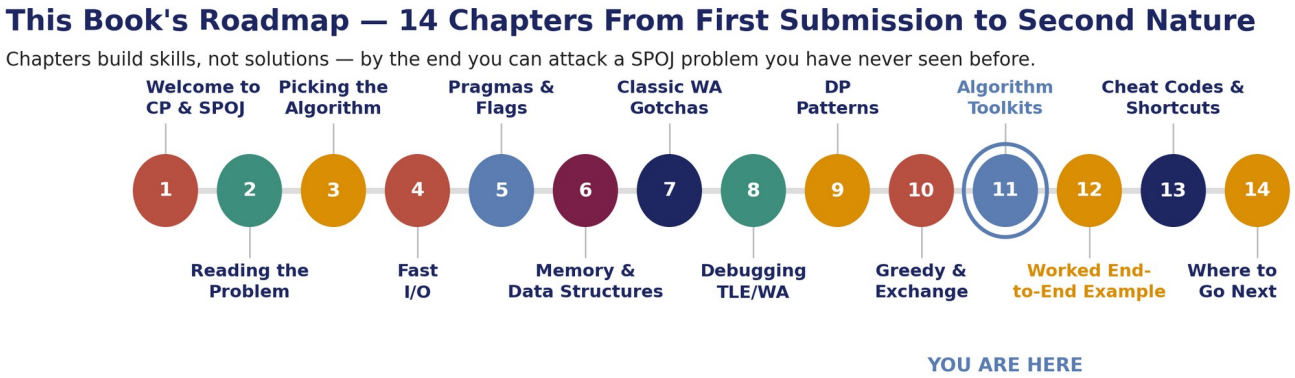

Figure 11.4 — This book's 14-chapter roadmap. You are here: Chapter 11.

Review Questions (Not Graded)

1. Using Figure 11.1, explain what recognition signal in a problem statement would point you toward the graph toolkit rather than the number-theory toolkit, and give an example phrase for each.
2. Explain, using Section 11.2's BFS pitfall, exactly why marking a node visited at dequeue time instead of enqueue time can cause the same node to be pushed onto the queue more than once.
3. Explain why Fermat's little theorem is not a safe general-purpose replacement for extended Euclid when computing a modular inverse, and under what specific condition it does work correctly.
4. Using Section 11.4's orientation test, explain in your own words why casting integer coordinates to floating point before computing a cross product can turn a truly collinear case into a spurious left or right turn.
5. Explain why a naive O(n*m) string search can pass a problem's sample input yet still fail on the judge's real test data, and what complexity class the KMP algorithm achieves instead.

Appendix 11.A — Verification Log

All programs in this chapter were compiled with g++ (-O2 -Wall -pedantic) and cross-checked against an independent computation for each result: a hand-traced graph for BFS, the Bezout identity for extended Euclid, an independent modular multiplication for the modular inverse, a hand-sketched

triangle for the orientation test, and a Python regular-expression search for KMP (Figures 11.3 and 11.4).

Terminal transcript

```
$ 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
```

Every result above was cross-checked independently: the BFS distances match a hand trace of the six-node graph; the extended-Euclid coefficients satisfy the Bezout identity by direct substitution; the modular inverse multiplies back to 1 modulo 10^9 + 7; the orientation and segment-intersection results match a hand-sketched diagram; and the KMP occurrence list matches a Python regular-expression lookahead search over the same text (Python's math.gcd(240, 46) = 2 and pow(123456, -1, 1000000007) = 78351802 independently confirm the number-theory results as well).

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press — on BFS (Section 11.2), the extended Euclidean algorithm (Section 11.3), and the KMP algorithm (Section 11.5).

[2] de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). Computational Geometry: Algorithms and Applications (3rd ed.). Springer — on the orientation test and its role as the primitive underneath convex hull and segment intersection, the basis of Section 11.4.

[3] Wirth, N. (1976). Algorithms + Data Structures = Programs. Prentice-Hall — source of the title quoted in Section 11.8, cited here for its framing of a program as an algorithm paired with the right data structure.

CHAPTER 12

Getting to Accepted: A Worked End-to-End Example

Every previous chapter built one skill in isolation. This chapter runs all of them together on a single, classic, easy problem, from first reading to a wrong answer to a fix to final acceptance -- because that full loop, not any one technique alone, is what actually gets a solution accepted.

Learning Objectives — after this chapter you will be able to:

(1) Walk a real, classic SPOJ problem through the full submission loop: restate, implement, submit, diagnose, fix, re-verify, accept. (2) Recognize that a first attempt passing the sample input is evidence of very little, and explain why. (3) Apply Chapter 8's stress-testing discipline to find a concrete, reproducible failing case for a bug that a first attempt's own logic does not reveal. (4) Distinguish a fix that addresses the specific root cause from one that merely patches the symptom on the one case you happened to find. (5) Explain why this chapter's worked example is deliberately simple, and what makes the same seven-stage loop apply just as much to a Hard-rated problem.

12.1 The Problem: Adding Reversed Numbers

This chapter works through SPOJ ADDREV (Adding Reversed Numbers), an Ad Hoc-rated Easy problem, chosen precisely because its algorithm is trivial and its trap is not. Given several test cases, each with two non-negative integers A and B written without leading zeros, the task is: reverse the digits of A, reverse the digits of B, add the two reversed numbers together, then reverse the digit string of that sum and print the result -- again without leading zeros.

Restating it fully, in the spirit of Chapter 2's five-zones reading habit: the two input numbers themselves never have leading zeros, but they may have trailing zeros (like 2400), and reversing a number with trailing zeros produces leading zeros in the reversed value, which must be dropped before adding (2400 reversed is 0042, which is just 42). The output has the identical requirement one more time, at the very end, applied to the reversed sum -- and that second, easy-to-forget application of the same rule is exactly where this chapter's bug lives.

12.2 First Attempt

The reversing-a-single-number half of the problem is simple to get right: convert the number to a string, reverse the string, and parse it back into an integer. Parsing back into an integer is what quietly strips any leading zeros the reversal introduced -- for input, this handles the trailing-zero case correctly without any special-case code at all.

addrev_buggy.cpp — first attempt (original)

```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;
}
```

This compiles cleanly and, on a small hand-checked sample like A = 24, B = 1 (reverse 42 and 1, sum 43, reverse 34), prints exactly the expected output. It looks finished.

12.3 Submit — and a Wrong Answer

Submitting this to a judge returns Wrong Answer, not a crash, not a timeout -- the least informative verdict of Chapter 1's six, because it confirms only that something is off, not what or where. The natural next step is not to guess, but to go looking for the smallest possible input that reproduces the failure, exactly as Chapter 8 recommends.

> **Passing the sample input is evidence of very little.**
>
> A sample input is usually chosen to illustrate the problem's intent, not to stress every corner of the specification. Section 12.4's stress test finds a failing input that resembles the sample closely -- two ordinary-looking numbers -- which is precisely why eyeballing sample cases alone would never have caught this bug.

12.4 Diagnose: Finding the Smallest Failing Case

A Python reference implementation, written independently and in a different language on purpose (so a shared misunderstanding of the problem can't hide the same bug twice), reverses each number as int(str(x)[::-1]) -- the round-trip through int() strips leading zeros automatically, exactly like the C++ version's stoll does for the input numbers. Generating 2,000 random pairs and comparing the buggy program's output against this reference line by line finds mismatches on 236 of them -- just under 12%, not a rare corner case at all.

The first mismatch reported is A = 858595, B = 209330: the buggy program prints 067926, while the reference computes 67926. Reducing this by hand to a smaller, cleaner case that isolates exactly the

same mechanism: A = 51, B = 52. Reversing gives 15 and 25, which sum to 40 -- and reversing 40 as a raw string gives "04", printed as-is by the buggy version instead of being parsed back into the integer 4.

This Chapter's Bug in Two Test Cases

The same leading-zero trap, exposed by two different reversed sums, alongside one case where it happens not to matter.

Test Case (A, B)	Reversed Sum	Buggy Output	Correct Output
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" (no divergence here)

Figure 12.1 — The same leading-zero trap, exposed by two different reversed sums.

Figure 12.1's second row, A = 3 and B = 794, shows the same mechanism producing two leading zeros instead of one (reversed sum 500, printed as "005"), confirming this is a general pattern in how the final reversal is handled, not a one-off fluke tied to the specific number 40.

12.5 Fix and Re-verify

The root cause is now precise: reverseNum() already handles leading-zero stripping correctly for the two input numbers, by parsing the reversed string back into an integer. The final reversal of the sum, at the very end of the program, does the reversal but skips that same round-trip, printing a raw string instead. The fix is to apply the exact same function to the sum that is already applied to the two inputs.

addrev_fixed.cpp — fix (original)

```
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;
}
```

> **Reusing the function that already got it right is safer than patching the symptom.**
>
> A tempting quick fix is to special-case the specific failing input, or to manually strip leading zeros from the output string with a loop. Both work on the case that was found, but neither is obviously correct on every other case the same way reverseNum() already provably is, since it is the same function already exercised (and not found buggy) on both inputs. Reusing proven logic, rather than writing new logic under pressure to make one failing case pass, is the safer habit.

Re-running the identical 2,000-case stress test against the fixed version reports zero mismatches. Re-checking the two hand-reduced cases from Figure 12.1 by hand: A = 51, B = 52 now prints 4; A = 3, B = 794 now prints 5; and the edge case A = 0, B = 0 still prints 0, exactly as it did even in the buggy version -- confirming the fix did not accidentally break the one case that already happened to be correct (Appendix 12.A).

12.6 The Stages, in General

Figure 12.2 restates this chapter's six stages in general terms, stripped of this specific problem's details, because the same loop applies whether the bug takes one stress-test run to find or a hundred.

Stages of Reaching Accepted

The mental process behind every AC, whether it takes one submission or five.

Stage	What You Do	Signal You're Actually There
1. Read & Restate	Rewrite the problem in your own words, including every edge case the statement implies	You could explain the problem to someone else without looking at it again
2. First Attempt	Implement the most direct correct-looking algorithm for the constraints	Code compiles cleanly and passes the sample input
3. Submit	Send it to the judge and read the specific verdict, not just pass or fail	You know exactly which of the six verdicts (Chapter 1) came back
4. Diagnose (if not AC)	Apply Chapter 7/8 discipline: hand-trace a small case, stress test against an independent reference	You've found one concrete input that reproduces the failure on demand
5. Fix & Re-verify	Correct the specific bug, then re-run the full stress test, not just the one failing case	Zero mismatches across hundreds or thousands of random trials
6. Accepted	The judge confirms all hidden tests passed, not just the visible sample	AC -- the only verdict that means the diagnosis-and-fix loop is actually closed

Figure 12.2 — The mental process behind every AC, whether it takes one submission or five.

The habit worth carrying forward is not memorizing this specific reversed-sum trap, but internalizing the shape of the loop: a first attempt that compiles and passes the sample is Stage 2, not Stage 6, and the distance between them is exactly the diagnose-fix-reverify cycle this chapter walked through in full (Figures 12.3 and 12.4).

12.8 Chapter Summary

- A first attempt that compiles and passes the sample input has completed Stage 2 of reaching Accepted (Figure 12.2), not the whole process -- passing the sample is evidence that very little is definitely wrong, not evidence that nothing is.
- SPOJ ADDREV's trap is that reversing a number's digits and parsing the result back into an integer correctly strips leading zeros for the two inputs, but the same discipline is easy to forget to apply one more time, to the final reversed sum.
- A Python reference implementation, written independently, found 236 mismatches out of 2,000 random trials -- just under 12%, confirming the bug is a general pattern, not a rare fluke.
- The fix reuses the exact function already proven correct on the two inputs, applying it a second time to the sum, rather than writing new one-off logic to patch the specific failing case that was found.
- The seven-stage loop this chapter walked through in full -- read and restate, first attempt, submit, diagnose, fix, re-verify, accept -- is the same loop that applies to a Hard-rated problem; only the size of the diagnose-and-fix step changes.

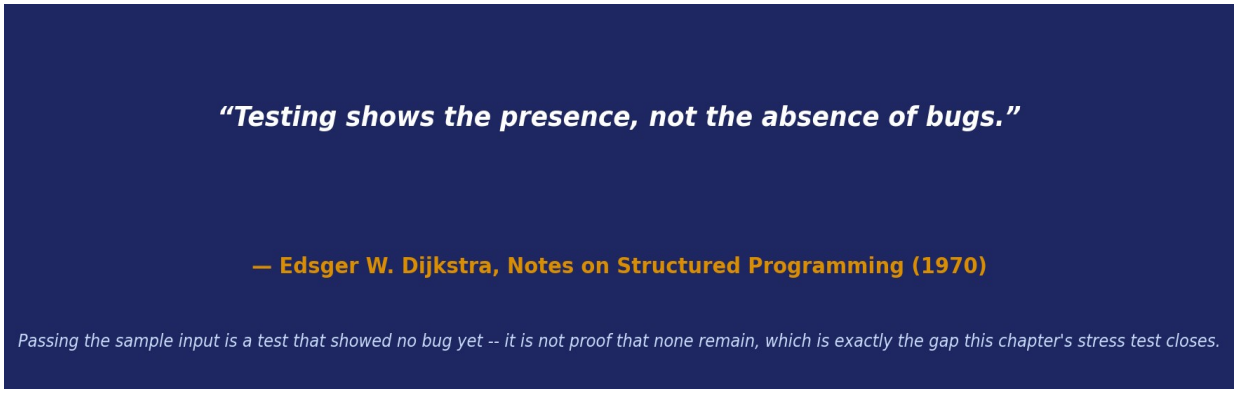

Figure 12.3 — Dijkstra's line, and the gap it names between a passing sample and a proven solution.

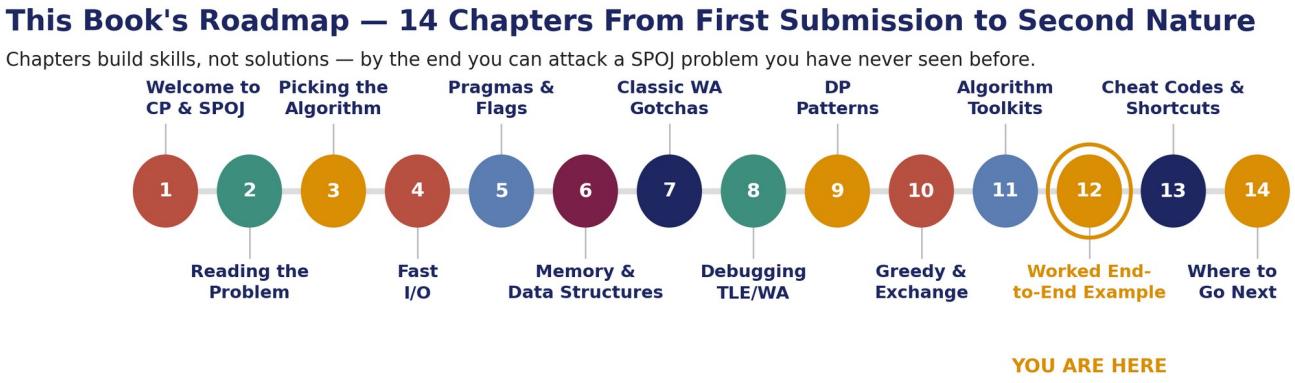

Figure 12.4 — This book's 14-chapter roadmap. You are here: Chapter 12.

Review Questions (Not Graded)

1. Explain, using Section 12.1's restatement of the problem, exactly which two places in the algorithm require stripping leading zeros produced by a reversal, and why it is easy to implement one correctly and forget the other.
2. Using Figure 12.1, explain in your own words why A = 51, B = 52 exposes the bug while A = 0, B = 0 does not, even though both involve the same reverseNum() and the same final-reversal code path.
3. Explain why finding one failing input (A = 858595, B = 209330) was not, by itself, enough to confidently call the bug fixed, and what step in Section 12.5 supplied that confidence instead.
4. Using Figure 12.2, explain what distinguishes Stage 2 (First Attempt) from Stage 6 (Accepted), and why a solution can sit at Stage 2 indefinitely without the author realizing it.

Appendix 12.A — Verification Log

Both programs were compiled with g++ (-O2 -Wall -pedantic). The three hand-reduced cases from Figure 12.1 were checked directly; the full comparison against an independent Python reference (int(str(x)[::-1]) for each reversal) ran across 2,000 random pairs of integers in [0, 1000000].

Terminal transcript

```
$ 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"
```

The buggy version's three hand-checked outputs (04, 005, 0) match exactly what Section 12.4's diagnosis predicted, and the fixed version's outputs (4, 5, 0) match the independently computed expected values. Across 2,000 random trials, the buggy version disagreed with the Python reference on 236 cases (11.8%) while the fixed version disagreed on none, confirming the fix addresses the general pattern rather than only the specific case that was found.

References

[1] Dijkstra, E. W. (1970). Notes on Structured Programming. Technological University Eindhoven — source of the quote in Section 12.8 on the asymmetry between finding bugs and proving their absence.

[2] SPOJ. ADDREV — Adding Reversed Numbers. https://www.spoj.com — the public problem this chapter's worked example is based on; problem name and identifier cited by public reference only, per this book's standing rule against reproducing any accepted solution (Appendix D).

CHAPTER 13

Cheat Codes and Shortcuts (Used Responsibly)

None of this chapter's shortcuts are actually cheating -- they are templates and library functions worth having memorized, so contest time goes to the hard part of a problem instead of reinventing a solved one. The real skill this chapter builds is knowing each shortcut's honest limit as well as its convenience.

Learning Objectives — after this chapter you will be able to:

(1) Recall, at a glance, which earlier chapter's boilerplate template fits a given situation, without re-deriving it from scratch. (2) Use three common competitive-programming library shortcuts -- __builtin_popcount, __gcd, and next_permutation -- correctly. (3) State the specific, narrow condition under which each of those three shortcuts silently produces a wrong answer instead of an error. (4) Explain why a library shortcut's convenience and its correctness are two separate questions, and why passing a quick manual check does not settle the second one. (5) Treat this chapter as a reference to return to, not a sequence to memorize in order.

13.1 Why Cheat Codes Need a Chapter of Their Own

Every technique in this book so far was built from first principles: understanding why it works, not just how to invoke it. That remains the right way to learn a technique for the first time. But once a technique is understood, re-deriving it under contest time pressure is a poor use of that pressure -- which is exactly the gap a memorized, battle-tested template closes. This chapter collects the templates and library functions this book has already built and verified, purely as a quick-reference point, and pairs them with a second list: library shortcuts that look like free correctness, each with one specific condition under which that free correctness quietly stops being free.

Boilerplate Worth Memorizing

Nine templates from earlier chapters, and the situation that should make each one come to mind.

Template	From	When to Reach For It
Fast exponentiation (binary power)	Ch3	Modular exponentiation or any large power computation under a modulus
Buffered fast I/O reader	Ch4	Input sizes beyond roughly 10^5 lines, where cin/scanf becomes the bottleneck
Safe compiler pragma line	Ch5	A portable speed boost with near-zero portability risk (rung 1 of the risk ladder)
Overflow-safe casting habit	Ch6	Any computation whose intermediate result might exceed int range
Segmented sieve	Ch6	Prime queries over a range too large to sieve from zero directly
Memoization sentinel pattern	Ch9	Top-down DP where a computed value must be distinguishable from not-yet-computed
Exchange-argument checklist	Ch10	Verifying a proposed greedy rule is actually optimal, not just plausible
Four-family recognition table	Ch11	Deciding which classical toolkit -- graph, number theory, geometry, or strings -- a problem needs
Seven-stage AC loop	Ch12	Structuring the process from first attempt to final acceptance on any problem

Figure 13.1 — Nine templates from earlier chapters, and the situation that should make each one come to mind.

Figure 13.1 is deliberately not a set of code listings -- every one of these templates was already built, verified, and shown in full in its originating chapter. Treating it as an index rather than a restatement is the point: the moment a contest problem's shape matches one of these rows, the corresponding chapter's already-verified code is the answer, not a fresh implementation written under time pressure.

13.2 __builtin_popcount: Fast, and Width-Specific

Counting the number of set bits in an integer comes up constantly -- subset-sum bitmask DP, XOR-related number theory, Hamming-distance-style comparisons. GCC's __builtin_popcount family computes this in a small, constant number of machine instructions, far faster than a hand-written bit-counting loop.

popcount_pitfall_demo.cpp — width-specific truncation (original)

```
long long big = 1LL << 40; // bit 40 set, well beyond int's 32-bit range

// BUG: __builtin_popcount expects an *unsigned int* argument. Passing
// a long long silently truncates it to the low 32 bits first.
int wrongCount = __builtin_popcount((unsigned int)big);

// FIX: use the 64-bit-width variant for a 64-bit value.
int rightCount = __builtin_popcountll(big);
```

The 32-bit and 64-bit variants are not interchangeable, and the compiler will not warn you.

__builtin_popcount takes an unsigned int; __builtin_popcountll takes an unsigned long long. Passing a 64-bit value to the 32-bit version compiles cleanly and simply discards every bit above position 31 before counting. This is invisible on any test value that happens to fit in 32 bits, which is exactly why it tends to survive a quick sanity check and only surface on a judge's larger hidden test.

13.3 __gcd: Correct, Except on Sign

The greatest common divisor comes up throughout number theory (Chapter 11) and fraction simplification. GCC's __gcd (a non-standard extension living in <algorithm>) and the C++17 standard std::gcd (in <numeric>) both implement the same Euclidean algorithm, but they differ on one specific input shape.

gcd_pitfall_demo.cpp — sign handling differs by version (original)

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

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

__gcd applies the Euclidean algorithm's modulo-and-swap steps directly to whatever sign it is given, without an absolute-value step first -- on two negative arguments, this can return a negative result. std::gcd is specified to take the absolute value of both arguments before proceeding, so it always returns a non-negative result regardless of input sign (Appendix 13.A confirms both behaviors by direct execution).

A function that is correct on every positive test case can still be wrong on the input a judge chooses deliberately.

If a problem's constraints allow negative values -- explicitly, or through a subtraction that was not checked for sign -- __gcd's sign behavior is exactly the kind of edge case that never shows up in a quick manual check with small positive numbers, but is trivial for a judge's test-data generator to include on purpose.

13.4 next_permutation: Only From Here Onward

Iterating over every permutation of a small sequence is common in brute-force verification (the same technique this book has used repeatedly, in Chapters 9, 10, and 12, to build an obviously-correct reference to test against). The standard library's next_permutation rearranges a range into its lexicographically next permutation, returning false once there is no next one -- which makes a simple do/while loop look like it enumerates everything.

next_permutation_pitfall_demo.cpp — starting point matters (original)

```
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, not 6

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

Starting from {2, 1, 3} -- not the lexicographically smallest arrangement of these three values -- the loop above visits only 4 of the 6 total permutations of 3 distinct elements, because next_permutation only walks forward from wherever it starts and never wraps around to visit the permutations that would have come before it. Sorting ascending first guarantees the starting point is the lexicographically smallest arrangement, so every subsequent call reaches all 6 (Appendix 13.A) (Figures 13.2 and 13.3).

> **next_permutation's contract says "next," not "all" -- reading only the function name invites this bug.**
>
> The fix costs one line (a sort call) and is easy to state once known, but the bug it prevents does not crash and does not obviously look wrong -- it just silently under-counts or under-enumerates on whatever arrangement the input happened to start in, which is a difficult thing to notice from the output alone.

13.6 Chapter Summary

- Figure 13.1's nine templates are a quick-reference index to already-verified code from earlier chapters -- the point of memorizing them is recognizing the situation, not re-deriving the implementation under contest time pressure.
- __builtin_popcount and __builtin_popcountll are not interchangeable: passing a 64-bit value to the 32-bit variant silently truncates it instead of raising an error, and this is invisible on any test value that happens to fit in 32 bits.
- __gcd (the GCC extension) does not take an absolute value first and can return a negative result on two negative arguments, unlike std::gcd, which is specified to always return a non-negative result.

- next_permutation only enumerates permutations forward from its starting arrangement -- the range must be sorted ascending first to guarantee every permutation is visited.
- Every shortcut in this chapter is genuinely useful and genuinely correct within its documented contract -- the discipline this chapter builds is checking that a problem's actual input shape falls inside that contract, not assuming it does.

Library Functions That Feel Like Cheat Codes

Each one is genuinely useful -- and each one has a specific limit that a first attempt tends to overlook.

Function	What It Does	The Limit Nobody Reads
`__builtin_popcount(x) / popcountll(x)`	Counts the set bits in an integer	Width-specific: the 32-bit version silently truncates a 64-bit value instead of raising an error
`__gcd(a, b) (GCC extension)`	Euclidean-algorithm greatest common divisor	Does not take an absolute value first -- can return a negative result for negative inputs, unlike std::gcd
`next_permutation(first, last)`	Advances a range to its lexicographically next permutation	Only enumerates permutations from the current arrangement onward -- the range must start sorted ascending to visit all of them

Figure 13.2 — Each shortcut is genuinely useful, and each has a specific limit a first attempt tends to overlook.

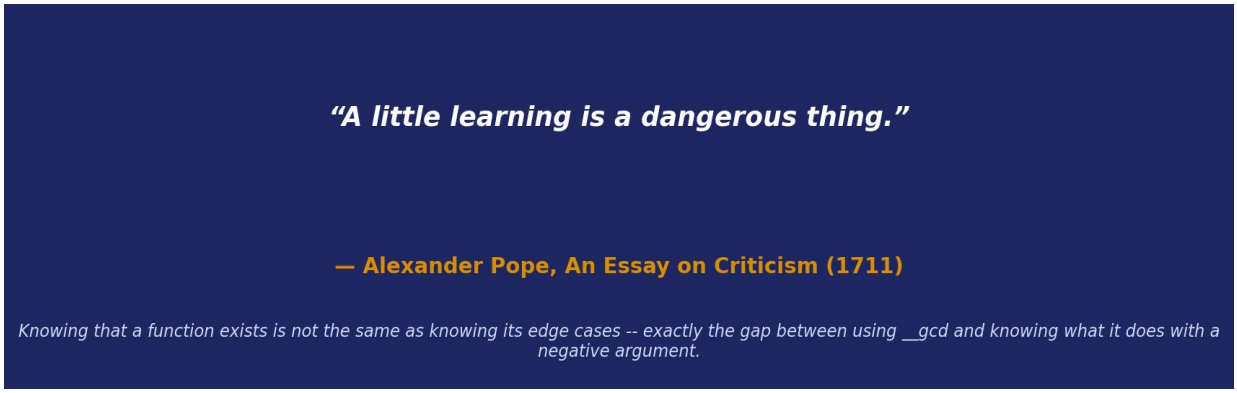

Figure 13.3 — Pope's line, and the gap it names between using a function and knowing its edge cases.

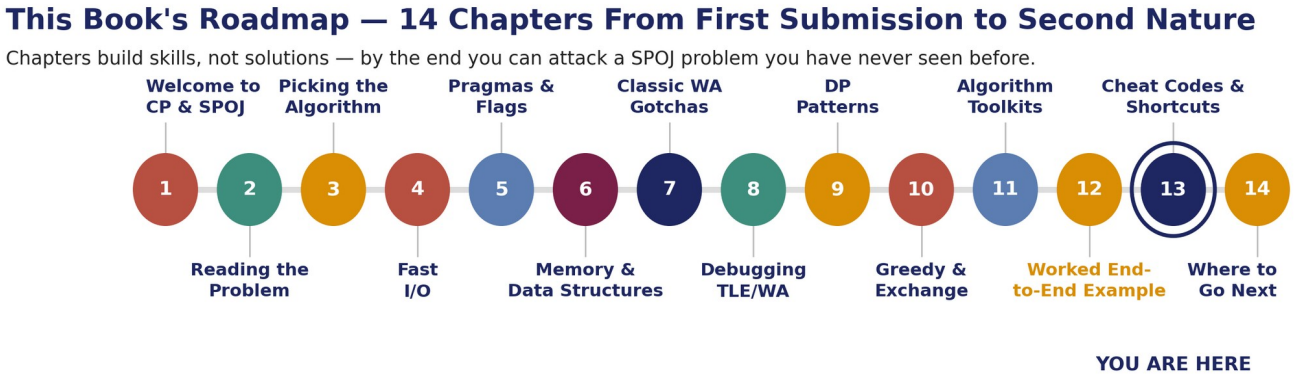

Figure 13.4 — This book's 14-chapter roadmap. You are here: Chapter 13.

Review Questions (Not Graded)

1. Using Figure 13.1, pick any two rows and explain, in your own words, what problem-statement wording should bring that row's template to mind.
2. Explain, using Section 13.2's demonstration, exactly why the bug in passing a 64-bit value to __builtin_popcount is invisible on small test values and only surfaces on a large one.
3. Using Section 13.3, explain the specific difference in behavior between __gcd and std::gcd, and describe an input shape where a problem could expose that difference.
4. Explain, using Section 13.4's demonstration, why starting next_permutation from {2, 1, 3} visits only 4 permutations instead of 6, and why sorting first fixes this.

Appendix 13.A — Verification Log

All three programs were compiled with g++ (-O2 -Wall -pedantic) and run directly; each demonstration's output was compared by hand against the expected value stated in its section (Figure 13.4).

Terminal transcript

```
$ 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
```

The popcount demo confirms the exact mechanism described in Section 13.2: 2^40 has exactly one bit set, correctly reported as 1 by the 64-bit builtin, but reported as 0 by the 32-bit builtin because bit 40 is truncated away, while a small value (0b1011, 3 bits set) reports identically under both, showing why this bug hides on small test data. The gcd demo confirms __gcd(-12,-18) returns -6 while std::gcd never returns a negative value on the same inputs. The permutation demo confirms the 4-versus-6 count exactly as Section 13.4 describes, matching 3! = 6 only once the input is sorted first.

References

[1] Pope, A. (1711). An Essay on Criticism. — source of the quote in Section 13.6 on the gap between partial and complete knowledge of a tool.

[2] Stroustrup, B. (2013). The C++ Programming Language (4th ed.). Addison-Wesley — on the standard library facilities discussed in Sections 13.3 and 13.4, including std::gcd and next_permutation's documented contract.

CHAPTER 14

Where to Go Next

This is the last chapter, not because there is nothing left to learn, but because everything left to learn now builds on a foundation this book has finished laying. This chapter is a map for what comes after: which problems to pick next, which topics reward the toolkit you already have, and how the habits built across the last fourteen chapters keep compounding long after this book is closed.

Learning Objectives — after this chapter you will be able to:

(1) Describe a concrete, tag-based strategy for choosing which SPOJ problems to attempt next, without relying on being told which ones. (2) Name at least three topics beyond this book's scope, and explain what existing chapter each one builds directly on. (3) Explain, in your own words, how this book's competitive-programming skills connect back to a formal algorithms course such as DAA. (4) Describe at least two study habits that compound over months rather than helping only within a single problem. (5) State, in one sentence, what this book was actually training -- and explain why that training does not end when the book does.

14.1 Where You Stand Now

Thirteen chapters ago, this book opened by treating SPOJ as unfamiliar territory: a page of rules, a judge that returns a verdict, a blank text box waiting for a program. Nothing about that setup has changed. What has changed is what happens between reading the problem and submitting a solution: a recognition table for which classical toolkit a problem needs (Chapter 11), a checklist for verifying a greedy rule before trusting it (Chapter 10), an instinct for where a naive approach's complexity will fail (Chapters 6 through 9), and a habit of reading a library function's contract before leaning on it (Chapter 13). None of this makes the next unfamiliar problem familiar. It makes an unfamiliar problem tractable, which was always the actual goal.

14.2 Picking Your Next SPOJ Problems

A common way to stall out after finishing a book like this one is to look at SPOJ's full problem list and not know where to start. The fix is not a list of specific problems to solve -- it is a strategy for finding your own, matched to what this book actually built.

A Ten-Problem Practice Plan

A tag-and-difficulty progression, not specific problem IDs -- pick your own from SPOJ's tag search once you know what to look for.

Problems	Tag / Focus	Goal
1-2	Ad hoc / implementation, easy	Rebuild speed and confidence with Chapter 4's fast I/O and Chapter 5's safe pragmas
3-4	Greedy, easy-to-medium	Practice Chapter 10's exchange-argument checklist on a rule you have not seen before
5-6	Graph (BFS/DFS or shortest path), easy-to-medium	Apply Chapter 11's four-family recognition table under real time pressure
7-8	Dynamic programming, medium	Apply Chapter 9's DP recognition patterns to a state space you have to design yourself
9-10	Your choice, medium-to-hard	Pick a problem in an area from Figure 15.1 you have not studied yet, and treat getting stuck as the point

Figure 14.1 — A tag-and-difficulty progression, not specific problem IDs.

The point of Figure 14.1's progression is not the specific number of problems -- it is the shape: start with what is easiest to verify quickly (ad hoc, easy), move through the toolkits this book built most recently and most thoroughly (greedy, graphs, DP), and end by choosing your own unfamiliar territory. That last step matters more than it looks: every technique in this book was originally unfamiliar too, and the value was never in the specific techniques covered, but in what it feels like to make an unfamiliar technique familiar through the same disciplined process, repeatable on demand.

14.3 Beyond This Book: Five Topics Worth Learning Next

This book's toolkit -- Chapters 6 through 13's data structures, greedy, DP, graphs, number theory, geometry, strings, and library shortcuts -- covers a large fraction of what appears on a typical competitive-programming judge, but not all of it. The five topics below are the natural next layer, chosen specifically because each one is a direct extension of a toolkit this book already built, not a fresh start.

Beyond This Book: Topics Worth Learning Next

Each one builds on a toolkit this book already gave you -- none of them starts from zero.

Topic	What It Buys You	Builds On
Fenwick tree / segment tree	Range sum, min, or max queries with point or range updates in O(log n) instead of O(n) per query	Chapter 6's array-based data structure habits and recursion from Chapter 9's DP
Union-Find (DSU)	Near-O(1) connectivity queries -- "are these two nodes in the same group" -- and Kruskal's minimum spanning tree	Chapter 11's graph toolkit and its recognition-signal habit
Network flow (max flow, bipartite matching)	A single framework that solves a surprising range of assignment and capacity problems once recognized	Chapter 11's BFS, reframed as the augmenting-path search inside a flow algorithm
Suffix array / suffix automaton	Substring and repeated-pattern queries beyond what single-pattern KMP (Chapter 11) can answer efficiently	Chapter 11's string toolkit, generalized to many patterns or all substrings at once
Bitmask DP and digit DP	Two specific DP shapes -- state as a subset, or state as a digit position -- that unlock a large family of otherwise-intractable problems	Chapter 9's DP recognition patterns and memoization sentinel habit

Figure 14.2 — Each one builds on a toolkit this book already gave you.

Notice the pattern in Figure 14.2's third column: every one of these five topics is a generalization of something this book already covered, not an unrelated new subject. A Fenwick tree is what an array becomes when point updates and range queries both need to be fast. Union-Find is what graph connectivity becomes when you need to ask the question many times as edges are added one at a time. This is worth naming explicitly, because it is easy to look at an unfamiliar topic's name and assume it requires starting from zero -- it rarely does.

14.4 Back to DAA: What Feeds What

For readers also working through this author's Design and Analysis of Algorithms (DAA) course, the relationship between that course and this book runs in both directions. DAA's sixteen lectures build the formal foundation -- asymptotic analysis, correctness proofs, the classical algorithm families -- that this book's every chapter has quietly assumed. This book, in turn, is what that formal foundation looks like under time pressure, applied to problems whose exact shape you do not know in advance and defended against a judge that will not partially credit an almost-right analysis. Neither replaces the other: DAA teaches you to prove an algorithm is correct and efficient; this book teaches you to recognize, under pressure, which proof applies.

> **A judge is a stress test for understanding, not a substitute for it.**
>
> A lecture's algorithm, understood well enough to reproduce it in a controlled setting, and the same algorithm, recognized correctly within seconds of reading an unfamiliar problem statement, are two different levels of the same understanding. SPOJ (and judges like it) is a way to train the second level once the first is in place -- it was never a replacement for the first.

14.5 Study Habits That Compound

Two habits, more than any specific technique in this book, are worth carrying forward because their value compounds the longer they are practiced. The first is keeping a personal record of every mistake that cost real time -- not the algorithm, but the specific reasoning error: an unchecked sign, a boundary never tested, a library contract half-read. Chapters 7, 8, 13, and 14 each demonstrated a different instance of this same pattern, and the pattern itself, once named, starts catching itself earlier each time. The second is deliberately revisiting an old, solved problem months later and re-solving it without looking at the old solution -- not because the answer is in doubt, but because the version of you doing it the second time has a bigger toolkit, and the comparison is where growth becomes visible (Figures 14.3 and 14.4).

14.6 Chapter Summary

- A tag-and-difficulty progression, not a list of specific problems, is the right way to choose what to attempt next -- Figure 14.1 gives a concrete ten-problem shape to start from.
- Fenwick/segment trees, Union-Find, network flow, suffix structures, and bitmask/digit DP are the natural next layer beyond this book, and each one is a generalization of a toolkit already built here, not a fresh start.
- This book and a formal algorithms course such as DAA feed each other in both directions: one supplies the proofs, the other trains recognizing which proof applies under time pressure.
- Keeping a record of specific reasoning errors, and periodically re-solving an old problem with a bigger toolkit, are the two habits most worth carrying forward because their value compounds over months, not just within one sitting.
- The judge was always a training device: the skills it trained -- proving a rule before trusting it, enumerating edge cases, reading a contract before depending on it -- outlast it and transfer to whatever comes after this book.

“It is not that I'm so smart, it is just that I stay with problems longer.”

— Albert Einstein

Every technique in this book was built the same way -- not by being clever faster, but by staying with a wrong answer until it became a right one.

Figure 14.3 — Einstein's line, and the habit every technique in this book was actually built on.

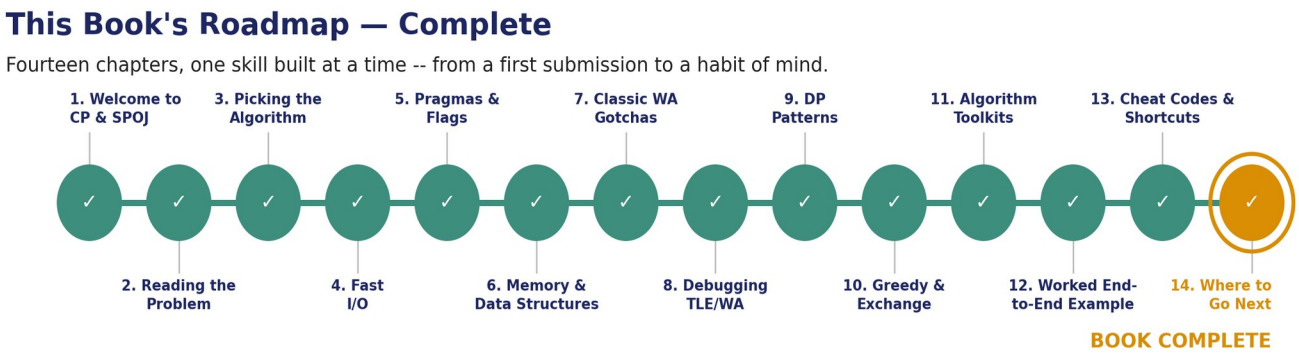

Figure 14.4 — This book's 14-chapter roadmap, complete.

Review Questions (Not Graded)

1. Using Figure 14.1, explain why the progression ends with "your choice" rather than a sixth prescribed category, and why that specific choice matters.
2. Pick any one row of Figure 14.2 and explain, using your own words, what existing chapter's toolkit it generalizes and how.
3. In your own words, explain Section 14.4's claim that DAA and this book feed each other in both directions, using a specific example of a technique from each.
4. Explain, using Section 14.5, why keeping a record of specific reasoning errors is more useful long-term than keeping a record of which algorithms you have learned.
5. In one or two sentences, state what you believe this book was actually training you to do, in your own words, independent of any single technique it taught.

References

[1] Einstein, A. — quoted in Section 14.6's closing figure on persistence as the actual mechanism behind every technique this book built.

[2] Subakti, I. — Design and Analysis of Algorithms (DAA) course materials, referenced throughout Section 14.4 as the formal counterpart to this book's applied, time-pressured practice.

[3] APPENDIX A

A Catalog of This Book's Problem Universe

This appendix catalogs 162 representative problems from the judge universe this book draws on, organized by difficulty tier and tagged by topic -- a browsable map for Chapter 14's "pick your own next problem" strategy. No solutions, hints toward solutions, or judge-specific implementation details appear here: only a problem's public name, its topic tag, and where it sits on the difficulty ladder. Use it the way Figure 14.1 intended -- as a way to find your own next problem in an unfamiliar topic, not as a checklist to clear top to bottom.

> **Reading this catalog**
>
> "Code" is the identifier you would search for on a judge (SPOJ problems by their short code, E-Olymp problems by their numeric ID). "Topic" often lists more than one tag -- most real problems sit at the intersection of two or three classical ideas, which is itself worth noticing: the four-family recognition table from Chapter 11 is a starting point for a search, not a single label that fully describes a problem.

Easy · 71 problems

Problems where the recognition step (Chapter 11's four-family table) is the whole battle -- once the right classical toolkit is identified, the implementation is short.

Code	Name	Topic
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	Data Structures (BIT/Fenwick Tree)
HMRO	Help the Military Recruitment Office!	Data Structures (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)

Code	Name	Topic
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

Medium · 50 problems

Problems that need one non-obvious insight on top of a classical toolkit -- an unusual state for a DP, a greedy rule that needs the exchange-argument checklist (Chapter 10), or a boundary case that a first attempt tends to miss.

Code	Name	Topic
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	Data Structures / 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

Code	Name	Topic
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

Hard · 35 problems

Problems that combine two or more toolkits, or need a less common structure this book only gestures toward (segment trees, advanced number theory, heavier graph algorithms) -- see Chapter 14's "beyond this book" topics.

Code	Name	Topic
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	Data Structures
GSS3	Can You Answer These Queries III	Data Structures
MKTHNUM	K-th Number	Data Structures / Merge Sort Tree

Code	Name	Topic
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, Data Structures
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

Super Hard · 6 problems

The far end of the catalog: problems that took serious iteration even for an experienced solver, often needing a topic from Chapter 14's next-layer list (segment trees, network flow, heavy-light decomposition, FFT) in full force.

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

APPENDIX B

A Complexity Cheat-Sheet

This appendix collects the Big-O reference tables this book's chapters build up piece by piece into one browsable page. It is meant to be read backward from a problem's constraints, not forward from an algorithm's name -- the way Section B.2 describes is how experienced solvers actually use a table like this one under time pressure.

B.1 Growth-Rate Reference

Each row below is a complexity class this book actually uses, with the input size it stays comfortable at under a typical one-to-two-second judge time limit, and the chapter where that class's algorithms are built in detail.

Complexity	Name	Comfortable n	Typical algorithm	Chapter
O(1)	Constant	any n	Array/hash lookup, arithmetic	Ch. 6
O(log n)	Logarithmic	up to 10^18	Binary search, balanced BST op	Ch. 6, 11
O(√n)	Square root	up to 10^14	Trial-division primality, sqrt decomp.	Ch. 11, 13
O(n)	Linear	up to 10^8	Single pass, prefix sums, counting	Ch. 4, 7
O(n log n)	Linearithmic	up to 10^6–10^7	Sorting, sort-based greedy, merge	Ch. 10
O(n√n)	Root-n times n	up to 10^5	Sqrt-decomposition query structures	Ch. 13
O(n^2)	Quadratic	up to 5,000–10,000	Nested loops, simple 2-D DP	Ch. 9
O(n^2 log n)	Quadratic-log	up to 2,000–3,000	n^2 with a sort or heap inside	Ch. 9, 10
O(n^3)	Cubic	up to 300–500	Floyd–Warshall, cubic DP, matrix mult.	Ch. 11
O(2^n)	Exponential	up to 20–25	Subset enumeration, bitmask DP	Ch. 9
O(n!)	Factorial	up to 10–11	Permutation brute force	Ch. 9

B.2 Reading Constraints Backward

A judge's constraints are a message about which complexity class is expected, and reading that message before writing any code is one of this book's most repeated habits (Chapters 3, 6, and 9 each return to it). The rule of thumb: a modern judge machine executes on the order of 10^8 to 10^9 simple operations per second, so an algorithm whose operation count clears the constraint's n by that ratio is safe, and one that does not is a TLE waiting to happen.

If the constraint is...	...this complexity is safe	...because
n ≤ 11	O(n!) or O(2^n · n)	Full permutation search is affordable
n ≤ 25	O(2^n)	Subset / bitmask enumeration
n ≤ 500	$O(n^3)$	Cubic DP, Floyd–Warshall-style all-pairs work
n ≤ 5,000	$O(n^2)$	Simple nested-loop or 2-D DP solutions
n ≤ 10^6	O(n log n)	Sort-based greedy, most graph algorithms
n ≤ 10^8	O(n) or O(n log n)	Single-pass scans, careful constant factors
n ≤ 10^18	O(log n) or O(1)	Binary search, closed-form / modular math

Constraints are a specification, not decoration.

A problem stating "1 ≤ n ≤ 10^5" is not incidental -- it is telling you, explicitly, that an $O(n^2)$ solution (10^10 operations) will not finish in time and an O(n log n) one (about 1.7 × 10^6 operations) comfortably will. Chapter 3 builds this habit from scratch; this table is what to reach for once the habit is automatic.

B.3 Data Structure Operation Costs

Complexity classes describe algorithms; the table below describes the data structures this book's chapters build those algorithms on top of, and what each core operation costs.

Structure	Access	Search	Insert / Update	Chapter
Array / vector	O(1)	O(n)	O(n) (O(1) at the end)	Ch. 6
Sorted array	O(1)	O(log n)	O(n)	Ch. 6, 11
Hash map / set	O(1) avg	O(1) avg	O(1) avg	Ch. 6
std::set / std::map	O(log n)	O(log n)	O(log n)	Ch. 6
Fenwick tree (BIT)	—	O(log n) prefix query	O(log n) point update	Ch. 13, 15
Segment tree	—	O(log n) range query	O(log n) point/range update	Ch. 15
Union-Find (DSU)	—	O(α(n)) find	O(α(n)) union	Ch. 15

B.4 One Rule Worth Memorizing

If nothing else on this page is memorized, this is the one rule to keep: look at the largest constraint in the problem, decide what complexity class is comfortable for it using Section B.2, and only then start thinking about which algorithm actually delivers that complexity. Chapters 3 through 14 are, in one sense, sixteen different answers to the second half of that process -- but the first half, reading the constraint, is what makes it possible to ask the right question before writing a single line of code.

APPENDIX C

Glossary of Terms

This glossary collects the competitive-programming and algorithms terms this book uses, in one alphabetical list, each tagged with the chapter (or appendix) where it is properly introduced and explained in context. Use it as a quick lookup while reading, not as a substitute for reading the chapter itself -- a one-sentence definition is a reminder, not a replacement for the worked examples that make a term actually usable under time pressure.

> **How to use this glossary**
>
> Every entry names the specific chapter it belongs to on purpose: a definition alone rarely transfers to a new problem, but a definition plus the worked chapter it came from usually does. If a term here feels unfamiliar even after reading its entry, that is a signal to revisit its chapter, not a gap in this page.

Accepted (AC) *[Ch. 1]*

The verdict a judge returns when a submission produces the correct output within the time and memory limits, for every test case the judge holds.

Ad Hoc *[Ch. 2, 3]*

A problem with no standard named algorithm behind it -- the difficulty is entirely in careful reading and careful implementation, not in recalling a technique.

Amortized complexity *[Ch. 6]*

A cost quoted per operation averaged over a whole sequence of operations, even though any single operation in the sequence might cost more (e.g. vector push_back is amortized O(1)).

Asymptotic complexity *[Ch. 3]*

How an algorithm's running time or memory grows as input size grows without bound, written in Big-O notation and independent of any specific machine.

Backtracking *[Ch. 9]*

A search strategy that builds a solution incrementally and abandons ("backtracks" from) a partial solution as soon as it cannot possibly be extended to a valid one.

Big-O notation *[Ch. 3]*

A way of describing an algorithm's worst-case growth rate that ignores constant factors and lower-order terms -- O(n) and O(2n + 5) describe the same class.

Binary search *[Ch. 6, 11]*

A search technique that repeatedly halves a sorted search space, achieving O(log n) time; also applicable to any monotonic answer space ("binary search on the answer").

Bitmask DP *[Ch. 9, 14]*

Dynamic programming where the state includes a bitmask representing a subset of items -- practical only while the number of items stays small (roughly n ≤ 20-25).

Brute force *[Ch. 3, 9]*

A solution that tries every possibility directly, with no algorithmic shortcut -- correct by construction, but only fast enough when the input size is small.

Compile Error (CE) *[Ch. 1]*

A judge verdict meaning the submitted source failed to compile -- the judge never even ran the program.

Complexity class *[Ch. 3, Appx. B]*

A category of growth rate (O(1), O(log n), O(n), O(n log n), O(n^2), and so on) that an algorithm's running time belongs to.

DFS / BFS *[Ch. 11]*

Depth-First Search and Breadth-First Search: the two fundamental graph traversal orders, DFS via a stack (often recursion), BFS via a queue, both O(V + E).

Dynamic Programming (DP) *[Ch. 9]*

A technique for problems with overlapping subproblems and optimal substructure, solved by storing (memoizing) subproblem answers instead of recomputing them.

Edge case *[Ch. 7, 8]*

An input at the boundary of what a problem's constraints allow (empty input, single element, maximum size, all-equal values) -- disproportionately likely to expose a bug.

Exchange argument *[Ch. 10]*

A proof technique for greedy algorithms: show that any optimal solution can be transformed, without making it worse, into one that matches the greedy choice.

Fast I/O *[Ch. 4]*

Techniques (e.g. disabling C++ stream synchronization, or scanf/printf) that remove the default overhead of cin/cout, often necessary when reading large inputs.

Fenwick tree (BIT) *[Ch. 13, 14]*

Binary Indexed Tree: a compact structure supporting point updates and prefix-sum queries in O(log n), simpler to implement than a full segment tree.

Greedy algorithm *[Ch. 10]*

An algorithm that makes the locally best choice at each step and never reconsiders it -- correct only for problems where a greedy-choice property and optimal substructure both hold.

Hashing *[Ch. 6]*

Mapping data to a fixed-size value (a hash) for fast average-case lookup, equality checking, or deduplication -- O(1) average, not worst case.

Judge *[Ch. 1]*

The automated system (SPOJ, Codeforces, and similar sites) that compiles a submission, runs it against hidden test cases, and returns a verdict.

Memoization *[Ch. 9]*

Caching the result of a function call keyed by its arguments, so a repeated call with the same arguments returns instantly instead of recomputing.

Number theory (in CP) *[Ch. 11]*

The branch of this book's toolkit covering primality, GCD/LCM, modular arithmetic, and modular inverses -- the recurring building blocks behind many "Classical" problems.

Overflow *[Ch. 7]*

The error that occurs when an arithmetic result exceeds the range of its integer type -- a frequent, quiet cause of Wrong Answer in competitive programming.

Prefix sum *[Ch. 6]*

A precomputed running total that turns repeated range-sum queries into O(1) lookups after an O(n) one-time setup.

Runtime Error (RE) *[Ch. 1, 7]*

A judge verdict meaning the program crashed during execution -- common causes include array out-of-bounds access, division by zero, and stack overflow from deep recursion.

Segment tree *[Ch. 14]*

A binary-tree structure supporting range queries and range or point updates in O(log n), more general than a Fenwick tree at the cost of more code.

Stress testing *[Ch. 12]*

Generating many small random test cases and comparing a fast solution's output against a slow, obviously-correct brute-force solution, to catch bugs a single hand-written test would miss.

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

A judge verdict meaning the program ran correctly in principle but did not finish within the time limit -- almost always a complexity problem, not a small bug.

Time complexity *[Ch. 3]*

See Complexity class -- specifically the growth rate of an algorithm's running time as a function of input size.

Two pointers *[Ch. 6]*

A technique using two indices that move through a sequence (often from opposite ends, or both forward) to solve certain O(n) problems that a naive approach would solve in O(n^2).

Union-Find (DSU) *[Ch. 14]*

Disjoint Set Union: a structure tracking a partition of elements into disjoint sets, supporting near-O(1) find and union operations via path compression and union by rank.

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

A judge verdict meaning the program ran and finished in time, but produced incorrect output on at least one hidden test case.