Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

TABLE OF CONTENTS

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 1

Foundations of Algorithm Design and Analysis

From a real-world problem to a proven, running program: the mindset every algorithm designer needs before writing a single line of code.

Learning Objectives — after this chapter you will be able to:
(1) Explain what distinguishes a problem, an algorithm, a program, and a data structure. (2) State and apply the two universal goals of algorithm design — correctness and efficiency. (3) Name the five major algorithm design paradigms and recognize which one a new problem calls for. (4) Apply the Five-Phase Framework (Understand, Abstract, Design, Analyze, Implement) to solve an unfamiliar problem from scratch. (5) Read and write simple correctness arguments and Big-O complexity estimates for elementary algorithms.

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 This Chapter Is About

Welcome to Design and Analysis of Algorithms (DAA). This is not a course about a programming language, nor about a particular library or framework. It is a course about THINKING — about developing a disciplined, repeatable way of turning a messy, real-world problem into a solution that is both provably correct and demonstrably fast.

By the end of this course you will be able to: model a computational problem precisely, given a real-world scenario with clear inputs and outputs; design an algorithm using an appropriate strategy — greedy, divide-and-conquer, dynamic programming, or another paradigm; analyze correctness and complexity, proving that an algorithm always gives the right answer and calculating exactly how fast it runs and how much memory it needs; and implement the design cleanly, translating it into working code using appropriate data structures.

Notice the order.
Understand first, then design, then analyze, then code. The code is the LAST step, not the first. The single most common mistake new students make is jumping straight to the keyboard. This book will train you out of that habit, because the thinking that happens before the code is exactly what separates an engineer who happens to get lucky from one who reliably ships working, efficient software.

1.2 Algorithms and Data Structures — The Two Pillars

1.2.1 What Is an Algorithm?

An algorithm is a step-by-step computational procedure that transforms input into output — a precise recipe: given specific ingredients (the input), follow specific steps, and produce a dish (the output). Formally, every algorithm must satisfy five properties.

- Input — zero or more values are supplied from the outside.
- Output — at least one value is produced as a result.
- Definiteness — every step is precisely specified, with no room for ambiguity.
- Finiteness — the procedure must eventually stop, for every valid input.
- Effectiveness — every step is basic enough that it could, in principle, be carried out by a person with pencil and paper.

A program is simply an implementation of an algorithm in a specific programming language — C++ throughout this book. The algorithm is the idea; the program is its realization. Many different programs, in many different languages, can implement exactly the same algorithm.

1.2.2 What Is a Data Structure?

A data structure is a way of organizing, storing, and managing data so that it can be accessed and modified efficiently. The choice of data structure can be the difference between an algorithm that finishes in milliseconds and one that takes hours on the very same input.

Kitchen analogy
An algorithm is like a cooking recipe — the sequence of steps. A data structure is like the kitchen equipment — the pots, pans, and oven. The exact same recipe, executed with better equipment, produces the same dish faster. Throughout this book you will see the same algorithmic idea run dramatically faster once paired with the right data structure.

Data structure	Typical strength	Example use in this book
Array / vector	O(1) indexed access, contiguous memory	Insertion Sort, Binary Search (Ch.1–2)
Linked list	O(1) insertion/removal at a known position	Dynamic sequences, adjacency lists

Stack / queue	Restricted LIFO / FIFO access	Recursion simulation, BFS traversal
Hash map (unordered_map)	O(1) average lookup by key	Memoization for dynamic programming
Balanced BST (map, set)	O(log n) ordered operations	Sorted dynamic sets, order statistics
Heap / priority queue	O(log n) min/max extraction	Dijkstra's algorithm, Huffman coding
Graph (adjacency list/matrix)	Models relationships/networks	Shortest paths, spanning trees

1.3 The Two Goals of Algorithm Design

Whenever we design an algorithm, there are exactly two properties we must guarantee. Every technique in this book exists in service of one of these two goals.

1.3.1 Goal 1 — Correctness

An algorithm is correct if it produces the right output for EVERY possible valid input — not just the examples you happened to test, and not just small inputs. There are several standard techniques for proving correctness rigorously:

- Direct proof — show that the algorithm's logic always leads to the right answer.
- Proof by contradiction — assume the algorithm gives a wrong answer, and derive a logical contradiction.
- Loop invariant — identify a property that holds before and after every iteration of a loop, and show that it implies the correct result once the loop terminates.
- Induction — prove a base case, then show that if the algorithm is correct for input of size k, it is also correct for size k+1.

Worked example: what does "correct" mean for Sorting?

Consider the sorting problem: given a sequence $a_1, a_2, \ldots, a_n$, produce a permutation $b_1, b_2, \ldots, b_n$ such that $b_1 \le b_2 \le \ldots \le b_n$. A correct sorting algorithm must satisfy TWO conditions simultaneously, for every input: the output must be sorted, and the output must be a permutation of the input — every element in the input appears exactly once in the output, nothing created, nothing destroyed.

If your algorithm turns [2, 5, 4, 10, 7] into [2, 4, 5, 7, 10], it is correct. If it instead produces [1, 2, 3, 4, 5], that output is sorted — but it is not a permutation of the input, so the algorithm is wrong, no matter how "clean" the output looks.

1.3.2 Goal 2 — Efficiency

An algorithm must use as little time and memory as possible. Two algorithms can both be correct and yet behave completely differently as the input grows.

A million numbers, two algorithms

Sorting one million numbers: Bubble Sort, at $O(n^2)$, performs roughly 10^{12} operations — hours of computation. Merge Sort, at $O(n \log n)$, performs roughly 2×10^7 operations — milliseconds. Both are correct. One is about a million times faster.

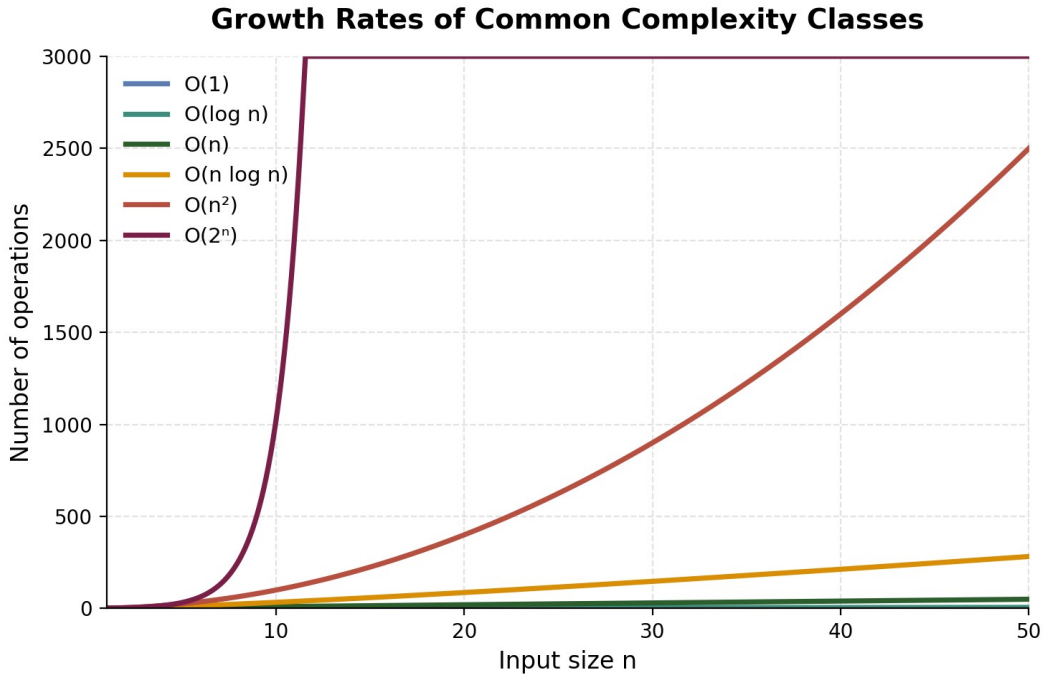

Figure 1.1 — How the number of operations grows with input size n for six common complexity classes. Notice how $O(n^2)$ and $O(2^n)$ explode while $O(\log n)$ and $O(n)$ stay nearly flat.

A practical rule of thumb

Modern computers execute roughly 10^8 to 10^9 simple operations per second. If your algorithm performs more than about 10^8 operations for the input sizes given in a problem's constraints, it will very likely exceed any reasonable time limit — in a course assignment, a competitive-programming judge, or production code.

1.3.3 The Two Goals Work Together

Correctness without efficiency is useless — the program eventually times out or exhausts memory. Efficiency without correctness is worse than useless — a fast program that gives wrong answers can cause real damage. You need both, simultaneously, and this entire book is about how to achieve both at once.

1.4 Algorithm Design Paradigms

There is no single recipe for designing algorithms. Instead, computer science has accumulated a handful of powerful general strategies — paradigms — each suited to broad categories of problems. Recognizing which paradigm a new problem calls for is one of the most valuable skills this course will give you.

Paradigm	Core idea	Canonical example
Incremental (brute force)	Process elements one at a time, maintaining running state.	Scanning an array to find its maximum.
Greedy	At each step, make the locally optimal choice and hope it leads to a global optimum.	Kruskal's algorithm for minimum spanning trees.
Divide and conquer	Split the problem into smaller sub-problems of the same type, solve recursively, combine.	Merge Sort.
Dynamic programming	Like divide-and-conquer, but sub-problems OVERLAP; store results to avoid recomputation.	Computing Fibonacci numbers, or Problem 6 below.
Mathematical / number-theoretic	Exploit a closed-form or algebraic identity that bypasses brute force entirely.	Euclid's algorithm for GCD.

1.5 The Five-Phase Problem-Solving Framework

Every problem in this book — indeed, every problem you will ever face as an algorithm designer — is solved using the same five-phase framework. Internalizing this framework is the single most valuable outcome of this chapter.

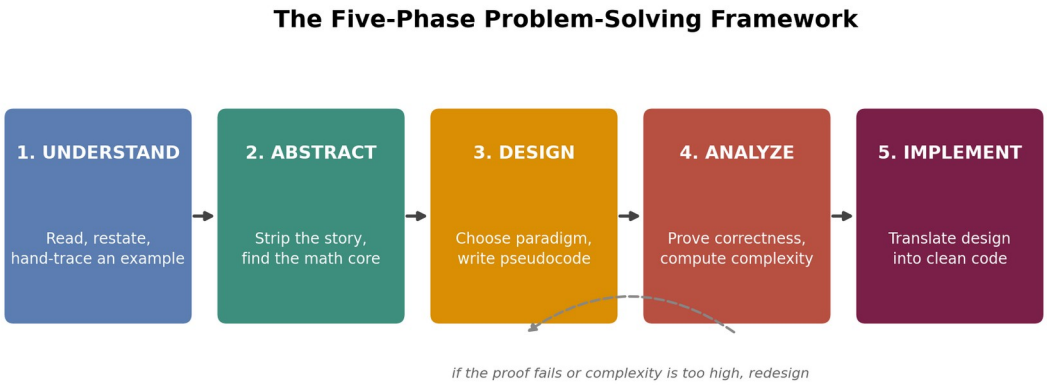

Figure 1.2 — The Five-Phase Framework used throughout this book. Analysis can send you back to redesign; this loop is normal and expected, not a failure.

- Phase 1 — UNDERSTAND: read the problem, identify input/output/constraints, restate it in your own words, and build a small example by hand.
- Phase 2 — ABSTRACT: strip away the story. What is the mathematical core? What properties govern the answer? This is where the real thinking happens.
- Phase 3 — DESIGN: choose a paradigm from Section 1.4, write the algorithm in pseudocode or plain English, and specify the data structures it needs.
- Phase 4 — ANALYZE: prove correctness (does it work for ALL inputs, not just the ones you tried?), calculate time and space complexity, and check that it fits the problem's constraints.
- Phase 5 — IMPLEMENT: write the code as a direct, clean translation of the Phase 3 design. If the code looks complicated, the design likely needs to be simplified first.

The most common student mistake

Most students skip straight to Phase 5. Resist this urge. The first four phases are precisely what make Phase 5 almost trivial — skipping them is what makes Phase 5 feel hard.

1.6 The Framework in Action: Six Problems Solved from Scratch

The remainder of this chapter applies the Five-Phase Framework to six real problems drawn from the E-Olymp and SPOJ online judges — the same judges you will use for this course's practice problems (see the Chapter 4 exercises and the companion SPOJ problem-set guide). Each problem illustrates a different paradigm from Section 1.4. Full, compiled, and tested C++ source code for every problem is provided in the companion code package (`code/chapter01/`).

1.6.1 Problem 1: Columns (E-Olymp 10281) — Greedy / Direct Computation

Phase 1 — Understand

Input: n columns of unit cubes, column i having height a_i ($1 \le a_i \le 1000$, $1 \le n \le 1000$). Output: the minimum number of colors needed to paint all cubes so that every horizontal run of adjacent cubes (with no gap) and every column receives all-different colors.

```
Row 6:  [X]
Row 5:  [X][X]
Row 4:  [X][X]      [X]      <- gap at column 3 (height 2 < 4)
Row 3:  [X][X]      [X]
Row 2:  [X][X][X][X]         <- all 4 columns present
Row 1:  [X][X][X][X]
        col1 col2 col3 col4   heights = [6, 5, 2, 4]
```

Phase 2 — Abstract

Two constraints must both be satisfied. Vertically, a column of height h needs h distinct colors, so we need at least $\max(a_i)$ colors. Horizontally, row 1 always contains all n columns, forming one substring of length n, so we need at least n colors. Let $C = \max(n, \max(a_i))$. Coloring the cube at (column j, row r) with color $((j + r) \bmod C) + 1$ guarantees distinct colors within any column (at most C cubes tall) and within any horizontal run (at most C cubes wide, since it cannot exceed n).

> **Key insight**
> The answer is exactly $\max(n, \max(a_1, \ldots, a_n))$ — C colors are both necessary (from the two constraints) and sufficient (by the explicit construction above).

Problem 1 · Columns — the coloring argument

Color cube at (column j, row r) with ((j + r) mod C) + 1. C = max(n, max height).

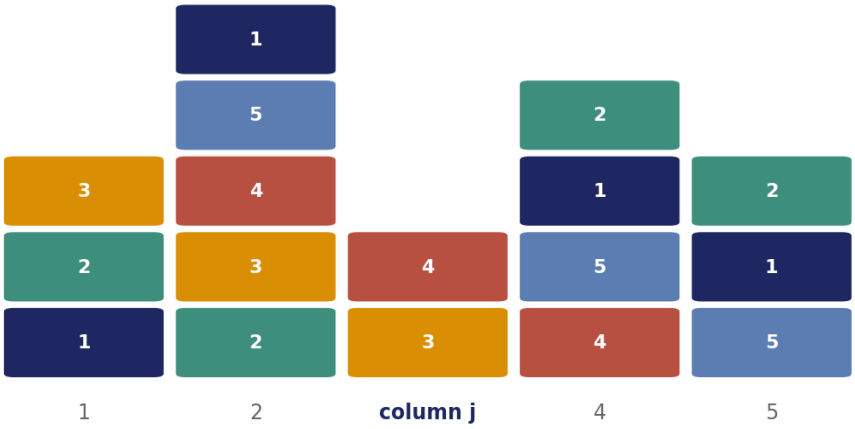

Figure 1.3 — The coloring construction: cube (column j, row r) is painted ((j + r) mod C) + 1. No column and no horizontal run ever repeats a color.

Phase 3 — Design

Read n. Initialize answer = n. For each height a_i, set answer = max(answer, a_i). Output answer.

Phase 4 — Analyze

Time: O(n) — a single left-to-right pass. Space: O(1). Correctness: proven above by explicit construction.

Phase 5 — Implement

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

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

1.6.2 Problem 2: Journey (E-Olymp 10280) — Greedy / Mathematical Observation

Phase 1 — Understand

Input: n cities located on a straight line, city i at coordinate x_i ($1 \le x_i \le 1000$, $1 \le n \le 100$). Task: starting from any city of our choosing, visit every city at least once and return to the start, minimizing total distance travelled.

Phase 2 — Abstract

Strip the story: we have n points on a number line and need the shortest closed tour visiting all of them. On a line you can only move left or right, and every reversal of direction wastes distance; the optimal plan is to travel from one extreme point to the other and back. Let x_{min} and x_{max} be the smallest and largest coordinates. Traversing $[x_{min}, x_{max}]$ in both directions visits every point, and algebraically the starting position cancels out of the total, which is always $2 \times (x_{max} - x_{min})$.

Key insight

Answer = $2 \times (\max(x_i) - \min(x_i))$. No search over starting cities or orderings is needed — the geometry of a line makes the optimum a closed-form expression.

Problem 2 · Journey — traverse the span twice

Any closed tour visiting every point in [xmin, xmax] crosses the span at least twice.

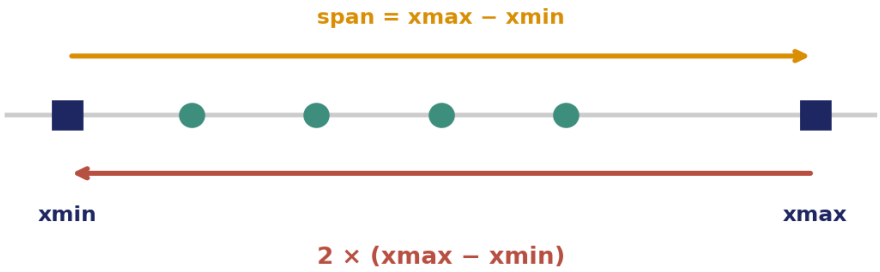

Figure 1.4 — Any closed tour on a line must cross the full [xmin, xmax] span at least twice — once outbound, once back.

Phase 3 & 4 — Design and Analyze

Track running dmin and dmax while reading coordinates in one pass; output 2 × (dmax − dmin). Time O(n), space O(1). Correctness: any closed tour on a line must traverse the full [dmin, dmax] range at least twice — once in each direction — and this plan achieves exactly that lower bound.

Phase 5 — Implement

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

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

1.6.3 Problem 3: Unique Numbers (SPOJ DCEPC901 / 12073) — Number Theory (GCD)

Phase 1 — Understand

Start with the set {0, 1, …, n−1}. Perform k operations in sequence, each acting on the result of the previous one: type 0 with value x replaces every number i with (i + x) mod n; type 1 with value x replaces every number i with (i × x) mod n. After every operation, report how many distinct numbers remain.

Phase 2 — Abstract

Adding a constant modulo n is a bijection — if a ≠ b then a+x ≠ b+x (mod n) — so addition never changes the count of distinct values. Multiplication is different: the image of f(i) = i×x mod n, for i = 0..n−1, is exactly the set of multiples of g = gcd(n, x), namely {0, g, 2g, …, n−g}, which has n/g elements. Chaining operations, we can therefore track only a running “effective size” n′: each multiply by x sets n′ ← n′/gcd(n′, x), and each add leaves n′ unchanged.

Key insight

Track n as a single running variable. On a multiply operation, n ← n / gcd(n, x); on an add operation, n is unchanged. Print n after every operation.

Problem 3 · Unique Numbers — multiplication shrinks the universe

Multiplying by x maps {0..n−1} onto multiples of gcd(n, x): the universe shrinks to n / gcd(n, x).

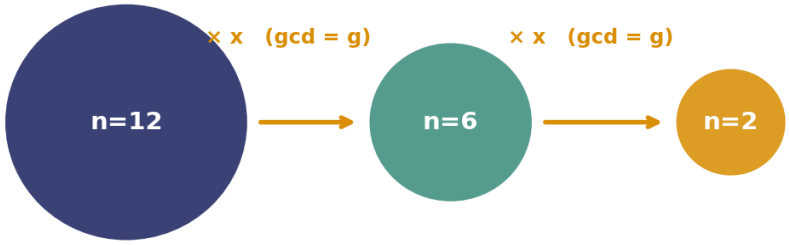

Figure 1.5 — Each multiply-by-x operation shrinks the universe of distinct values from n to n / gcd(n, x); add operations leave it unchanged.

Phase 3 & 4 — Design and Analyze

Read n and k. For each operation (p, x): if p = 1 (multiply), update n ← n / gcd(n, x); print n either way. Time: O(k log n) — one GCD computation per operation. Space: O(1).

Phase 5 — Implement

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

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

1.6.4 Problem 4: Minimum & Maximum Numbers (SPOJ 7667) — Greedy Digit Construction

Phase 1 — Understand

For each of T test cases, given N (number of digits, 1 ≤ N ≤ 18) and K (number of distinct digits required, 1 ≤ K ≤ 10), output the smallest and the largest positive N-digit integers (no leading zero) that use exactly K distinct digits.

Phase 2 — Abstract

If K = 1, the answer is a single digit repeated N times — it cannot be 0 because of the leading-zero rule, so the minimum is 111…1 and the maximum is 999…9. For K ≥ 2, magnitude is controlled almost entirely by the most significant digits, so a greedy strategy applies: for the minimum, place the smallest usable leading digit (1) first, pad the next (N−K+1) positions with 0, and place the remaining K−2 required distinct digits (2, 3, …, K−1) at the very end, where they affect magnitude least. For the maximum, mirror the strategy with 9's.

> **Verification (N=5, K=3)**
> Minimum: 1 + 000 + 2 = 10002 (digits used: {0,1,2}, exactly 3 distinct). Maximum: 999 + 87 = 99987 (digits used: {7,8,9}, exactly 3 distinct). Both checks pass.

Problem 4 · Min / Max Numbers — extremes at the ends

N = 5 digits, K = 3 distinct digits: lead digit + filler extreme, then the required digits appended last.

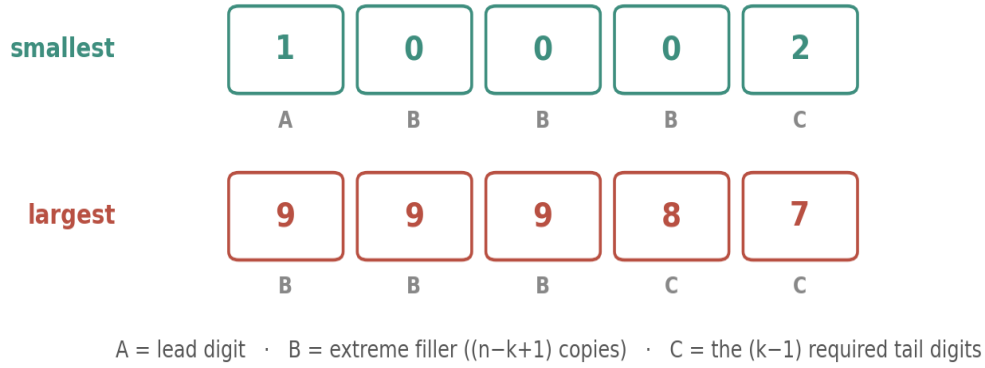

Figure 1.6 — Digit placement order for N=5, K=3: the lead digit and extreme filler go first (they dominate magnitude), the required distinct digits are appended last.

Phase 3 & 4 — Design and Analyze

Time: O(N) per test case, built by direct digit placement rather than search. Space: O(1). Correctness follows from the general principle that greedy choices at the most-significant digit dominate magnitude, so fixing them first and only touching the least-significant digits to satisfy the distinct-digit-count requirement cannot be beaten by any other arrangement.

Phase 5 — Implement

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

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

1.6.5 Problem 5: Interesting Sequence (E-Olymp 4894) — Bit Manipulation / Pattern

Phase 1 — Understand

A sequence is built starting from [0]: repeatedly copy the current sequence, replace every 0→1, 1→2, 2→0 in the copy, and append it to the original — producing [0] → [0,1] → [0,1,1,2] → [0,1,1,2,1,2,2,0] → …. Given m (1-indexed, up to $2^{63}-1$), find the m-th element.

Phase 2 — Abstract

The replacement rule 0→1→2→0 is simply “add 1 mod 3.” Because the sequence doubles at each construction step, the 0-indexed position p can be examined bit by bit: each time the current most-significant bit of p is 1, we are in the “second half” produced by the copy-and-increment step, contributing one more +1 (mod 3). Summing this contribution over every 1-bit of p gives value(p) = popcount(p) mod 3.

Verification

p=0 (000_2): popcount=0 → 0. p=3 (011_2): popcount=2 → 2. p=7 (111_2): popcount=3 → 0. All three match the sequence values obtained by direct construction.

Problem 5 · Interesting Sequence — popcount mod 3

Doubling construction: copy the sequence, add 1 (mod 3) to the copy → value(p) = popcount(p) mod 3.

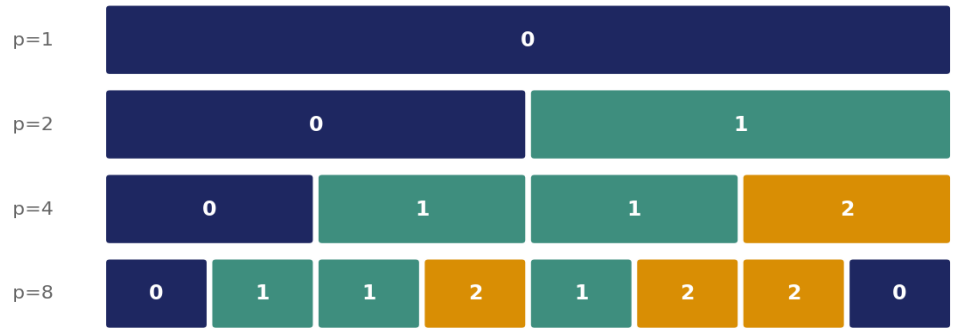

Figure 1.7 — Each doubling step copies the row and adds 1 (mod 3) to the copy, so value(p) always equals popcount(p) mod 3.

Phase 3 & 4 — Design and Analyze

Convert the 1-indexed query m to the 0-indexed position p = m−1, compute popcount(p) using a hardware popcount instruction, and output the result mod 3. Time: O(1) per query. Space: O(1).

Phase 5 — Implement

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

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

1.6.6 Problem 6: Recursive Function 1 (E-Olymp 10296) — Dynamic Programming / Memoization

Phase 1 — Understand

A function is defined by f(0) = 1 and, for n > 0, f(n) = f(⌊n/2⌋) + f(⌊n/3⌋). Given one integer n (up to 10^{18}), compute f(n) within a one-second time limit.

Phase 2 — Abstract

Naive recursion branches into two calls at every level, which is exponential without caching. The key observation is that every value ever queried has the form $\lfloor n / (2^a \times 3^b) \rfloor$ for some non-negative integers a and b, with a bounded by roughly $\log_2(10^{18}) \approx 60$ and b by roughly $\log_3(10^{18}) \approx 38$. That gives at most about $60 \times 38 \approx 2{,}280$ distinct sub-problems — tiny, even though n itself can be enormous. An array indexed by n is impossible (it would need 10^{18} entries), but a hash map that stores only the sub-problems actually visited makes memoization trivial.

```
f(12) = f(6) + f(4)
  f(6) = f(3) + f(2)
    f(3) = f(1) + f(1)
      f(1) = f(0) + f(0) = 1 + 1 = 2      [cache f(1)=2]
    f(3) = 2 + 2 = 4                      [cache f(3)=4]
    f(2) = f(1) + f(0) = 2 + 1 = 3        [cache f(2)=3]
  f(6) = 4 + 3 = 7                        [cache f(6)=7]
  f(4) = f(2) + f(1) = 3 + 2 = 5          [both already cached]
f(12) = 7 + 5 = 12
```

Key insight

Despite n reaching 10^{18}, only about 2,000 distinct sub-problems ever exist. A map-based memo table turns an exponential recursion into a fast, tiny computation.

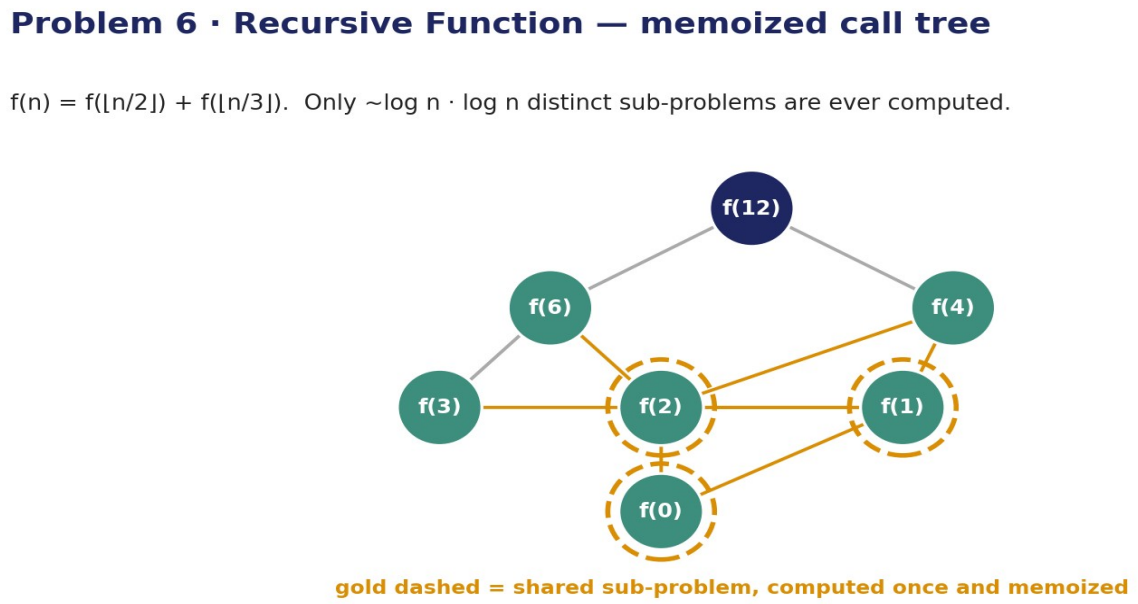

Figure 1.8 — The call tree for f(12). Gold dashed nodes are reached via more than one path — exactly the sub-problems memoization saves from being recomputed.

Phase 3 & 4 — Design and Analyze

Time: $O(\log^2 n \cdot \log \log n)$ — roughly 2,000 sub-problems, each with an O(log) map lookup. Space: $O(\log^2 n)$ for the memo table.

Phase 5 — Implement

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

map<ll, ll> memo;

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

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

1.7 How Each Problem Maps to a Design Paradigm

One of the central goals of this course is learning to recognize, quickly, which paradigm a new problem calls for. The table below summarizes the six problems just solved.

Problem	Paradigm	Recognizing signal
Columns	Greedy / direct computation	Answer reduces to a single max() over two simple constraints.
Journey	Greedy / mathematical observation	1-D geometry; only two directions of travel are possible.
Unique Numbers	Number theory (GCD)	Repeated modular operations; track an invariant, not the full state.
Min & Max Numbers	Greedy digit construction	Magnitude is dominated by the most significant digits.
Interesting Sequence	Bit manipulation / pattern	Recursive doubling construction; look for a closed bit-level formula.
Recursive Function 1	Dynamic programming / memoization	Recursive calls with a small set of DISTINCT reachable arguments.

1.8 Profitina in Practice: Where This Chapter Shows Up in a Real Product

> **About this box**
> Throughout this book, boxes like this one connect course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author. These boxes describe WHERE and WHY a concept is used in a live, running system — never the proprietary implementation details, business logic, or source code of Profitina itself, which remain confidential. See the companion document 'Profitina: A Non-Confidential Technical Overview' for the full picture.

Profitina's scanning engine must repeatedly decide, for a business name and domain, which of several candidate signal-extraction strategies to run first — an everyday instance of choosing a paradigm. Cheap, high-value checks are treated greedily (Section 1.4): run the check most likely to confirm or rule out a hypothesis first, exactly the same reasoning used to justify the greedy step in the Columns and Journey problems above. Where Profitina must avoid repeating expensive computation across many similar business queries (for example, re-estimating category or locale signals for businesses in the same city and sector), the system uses a memoization-style cache keyed by a normalized fingerprint of the input — conceptually the same idea as Problem 6's map-based memo table, applied to a production workload instead of a competitive-programming judge. Correctness and efficiency (Section 1.3) are treated as equally non-negotiable requirements in Profitina's own engineering practice: a scan that runs fast but returns a wrong signal is exactly as unacceptable as one that is correct but too slow to serve a user interactively.

1.9 Chapter Summary and Key Takeaways

- An algorithm must be both CORRECT and EFFICIENT — neither alone is sufficient.
- Always follow the Five Phases: Understand → Abstract → Design → Analyze → Implement. Never skip straight to code.
- The ABSTRACTION phase is where the real thinking happens: reduce the problem to its mathematical core.
- Know your paradigms — greedy, divide-and-conquer, dynamic programming, number theory, bit manipulation — each has recognizable signals.
- The right data structure can turn an impossible problem into a trivial one.

"The essence of algorithm design is not memorizing solutions. It is learning to see the hidden mathematical structure in problems — the abstraction that reduces something complex into something simple."

1.10 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Lecture 2.

1. Give an example (other than sorting) of a problem where an incorrect-but-fast algorithm and a correct-but-slow algorithm might both be tempting to submit, and explain which one you would choose and why.
2. For the Columns problem, why is it not enough to guarantee only the vertical constraint (colors ≥ max column height)? Construct a small counter-example where ignoring the horizontal constraint gives too few colors.
3. In Problem 3 (Unique Numbers), why does an ADD operation never change the count of distinct values, while a MULTIPLY operation can? Answer in terms of bijections.
4. For Problem 6 (Recursive Function 1), estimate how many distinct sub-problems would exist if the recurrence were instead f(n) = f(⌊n/2⌋) + f(⌊n/2⌋) (i.e., both terms divide by 2). Would memoization still help as much? Why or why not.
5. Identify one part of a system you use every day (a maps app, a messaging app, a search engine) where you suspect a greedy strategy is being used, and describe what the 'locally optimal choice' might be.

Appendix 1.A — Verification Log for Chapter 1 Source Code

Every program shipped with this book is compiled and run against a hand-verified example before being included, as a matter of policy. All six programs below were compiled with `g++ -O2 -std=c++17 -Wall` (only benign scanf return-value warnings, zero errors) and produced the exact output predicted in Section 1.6.

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

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

$ printf "6 2\n1 2\n0 1\n" | ./03_unique_numbers
3
3                                   # expected 6/gcd(6,2)=3, unchanged by the
add

$ printf "1\n5 3\n" | ./04_min_max_numbers
10002 99987                         # matches the Section 1.6.4 worked
verification

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

$ printf "12\n" | ./06_recursive_function
12                                  # matches the Section 1.6.6 walkthrough of
f(12)
```

References

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

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

[3] R. Soelaiman, S. Candra, M. M. I. Subakti. “Efficient Approach for Deterministic Data Extrapolation from a Clean Periodic Function with Periodic Components Representation by a System of Linear Equations.” Journal of Theoretical and Applied Information Technology (JATIT), Vol. 97, No. 8, 2019.

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

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

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

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 2

Analyzing Algorithms: The RAM Model, Insertion Sort, and Searching

How to measure "fast" without a stopwatch, why Insertion Sort is the perfect first case study, and how a sorted array turns O(n) search into O(log n).

Learning Objectives — after this chapter you will be able to:
(1) Explain the RAM model and use it to derive a running-time function T(n) for a simple algorithm. (2) State and prove a loop invariant for an iterative algorithm, using Insertion Sort as the running example. (3) Distinguish best-case, worst-case, and average-case running time, and explain why this course focuses on the worst case. (4) Compare growth rates (O(1), O(log n), O(n), O(n log n), O(n^2), O(2^n)) and predict which algorithms will finish within a time limit. (5) Implement and compare Linear Search and Binary Search, and explain the divide-and-conquer idea behind the latter.

2.1 Recap: From Lecture 1 to Lecture 2

Lecture 1 gave you the Five-Phase Framework and used it to solve six concrete problems. This chapter steps back and builds the theoretical machinery that explains WHY those solutions worked: a precise model of what "one step" means, a standard technique (the loop invariant) for proving an iterative algorithm correct, and the vocabulary of growth rates that turns "fast" and "slow" into precise, comparable statements.

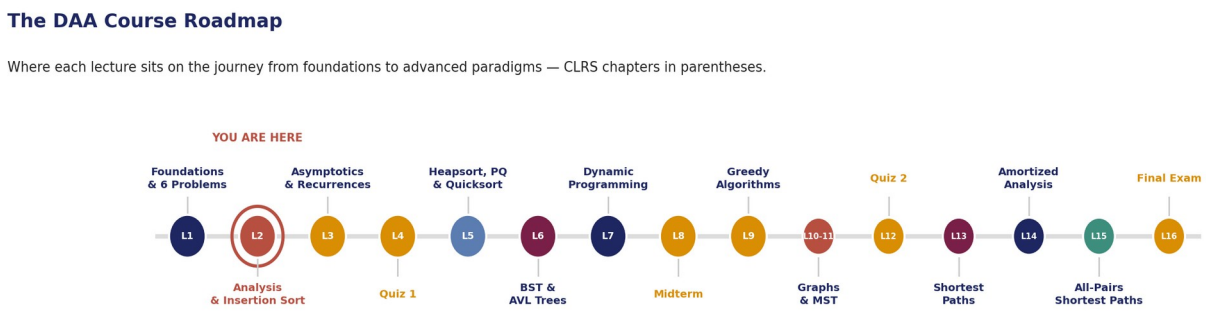

Figure 2.1 — Where this chapter sits in the sixteen-lecture DAA roadmap. Lecture 3 will formalize the asymptotic notation this chapter uses informally.

2.2 A Brief History of Algorithms

The word "algorithm" comes from the name of the Persian mathematician Muhammad al-Khwarizmi (c. 780–850 AD), whose name was Latinized to Algorismus; his treatises on algebra and arithmetic shaped how the world computes. The first algorithm most students ever meet is far older still: Euclid's method for the greatest common divisor, dating to roughly 300 BC — over 2,300 years old, and still the method you used in Lecture 1's Unique Numbers problem. The 19th century gave us Charles

Babbage's Analytical Engine and Ada Lovelace's algorithm intended for machine execution — the first program ever written for a machine that did not yet exist. The 20th century then formalized computation itself: Alan Turing's Turing Machine (1936), Alonzo Church's lambda calculus, and John von Neumann's stored-program architecture, which essentially every computer since has used.

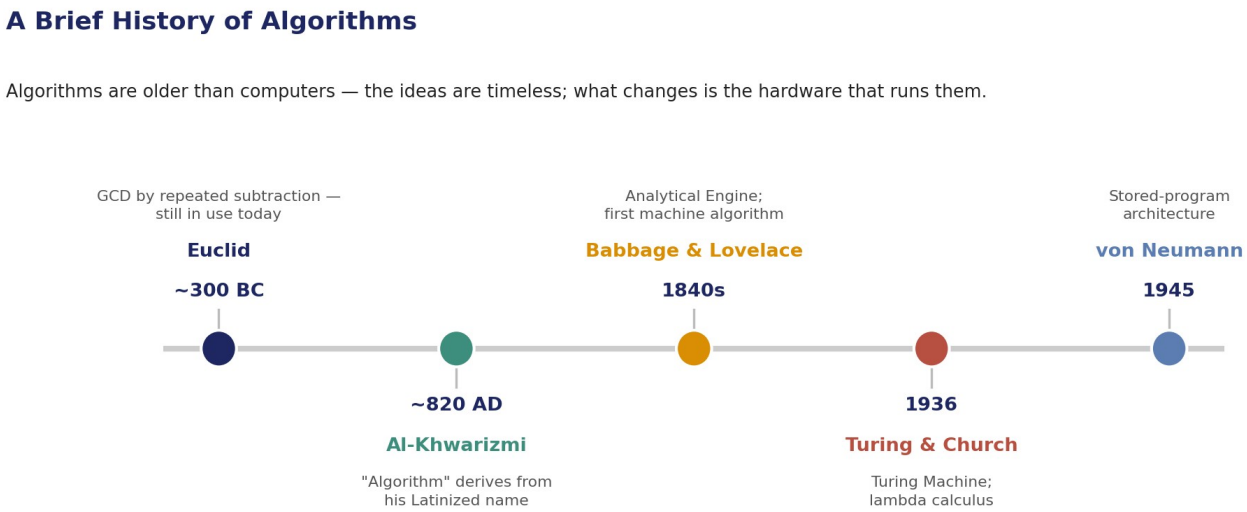

Figure 2.2 — Algorithms are older than computers. The ideas are timeless; what changes is the hardware that runs them.

> **Key insight**
> Algorithms are older than computers by over two thousand years. The IDEAS are timeless — what changes across centuries is only the hardware that executes them.

2.3 Core Definitions, Precisely

2.3.1 What Is an Algorithm, Formally?

An algorithm is a well-defined computational procedure that takes some value (or set of values) as input and produces some value (or set of values) as output — the “essence” of a computation, stripped of any specific programming language. Every algorithm must satisfy five properties, first given in Lecture 1 and restated here in full:

- Input — zero or more values are supplied externally.
- Output — at least one value is produced.
- Definiteness — every step is precisely and unambiguously specified.
- Finiteness — the algorithm terminates after a finite number of steps for every valid input.
- Effectiveness — every step is basic enough that a person could, in principle, carry it out with pencil and paper.

2.3.2 Algorithmic Problems vs. Algorithmic Solutions

An algorithmic problem specifies the desired input–output relationship — WHAT needs to be computed, not HOW — and admits infinitely many valid inputs. The SORTING problem, for instance, is: input a sequence of n numbers, output a permutation of those numbers in non-decreasing order. An algorithmic solution (an algorithm) describes HOW to compute the output from the input; for any given problem there are infinitely many correct algorithms, some fast, some slow, some elegant, some ugly.

> **The central challenge of this course**
> Among the infinitely many correct algorithms for a given problem, find the ones that are EFFICIENT. Correctness alone is the easy half of the job.

2.4 The Five Design Goals, Revisited

Lecture 1 introduced two primary goals — correctness and efficiency — plus three secondary goals that matter more in a software-engineering context: robustness (handling bad input gracefully), adaptability (being easy to modify), and reusability (being usable in other programs).

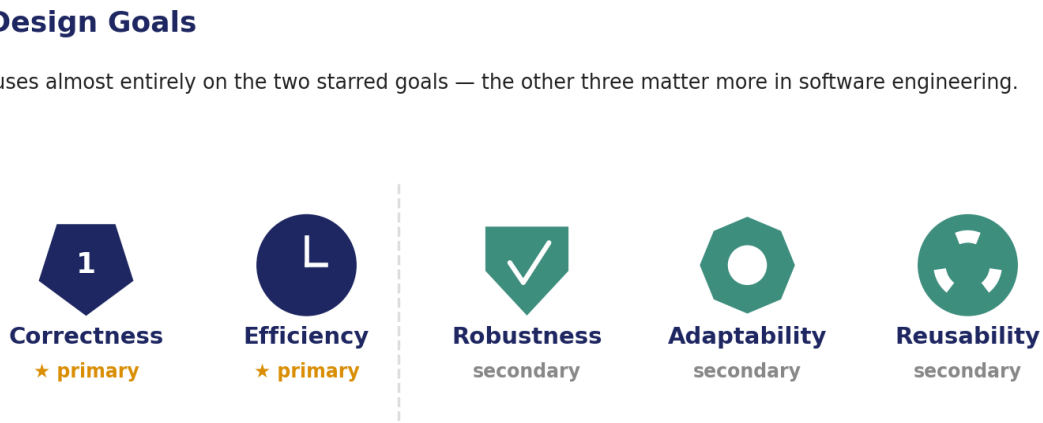

Figure 2.3 — This course focuses almost entirely on the two starred goals. The other three are the proper subject of a software-engineering course.

> **Key insight**
> In this course we focus almost entirely on Correctness and Efficiency. The other three goals are essential in practice, but they are the subject of a different course.

2.5 The RAM Model of Computation

> **Before we can measure speed...**
> ...we need to agree on what “one step” means. How do we count?

We use the Random Access Machine (RAM) model — an idealized computer with simple rules: each “simple” instruction (arithmetic, data movement, control flow) takes exactly one time step; each

memory access — reading or writing any cell — takes one time step, regardless of location; and data are integers and floating-point numbers of reasonable size. The RAM model is a simplification — real computers have caches, pipelines, and multiple cores — but it is accurate enough to predict performance and, crucially, to compare algorithms fairly.

> **Why the RAM model, and not a stopwatch?**
> If we measured actual wall-clock time, the same algorithm would appear "faster" on a newer laptop. The RAM model is machine-independent: it counts OPERATIONS, not seconds, which lets us compare algorithms fairly, forever, regardless of whose computer runs them.

2.6 How to Analyze Running Time

We express running time as a function T(n) of the input size n. To find T(n): count how many times each line of the algorithm executes; assign a cost to each line (a constant, under the RAM model); and sum all the costs. The input size n itself depends on the problem — the number of elements for sorting, the dimension for matrix multiplication, the number of vertices (and edges, m) for graph algorithms, or the number of bits for number-theoretic problems. Running time is the total number of primitive operations executed — NOT wall-clock time, but a count proportional to actual time on any reasonable machine.

2.7 Case Study: Insertion Sort — A Complete Analysis

Insertion Sort is the first "real" algorithm we analyze in full detail. It demonstrates every concept this chapter needs: a correctness proof, the best/worst/average-case distinction, and asymptotic analysis — all in one small, honest algorithm.

2.7.1 The Intuition: Sorting a Hand of Cards

Imagine holding playing cards in your left hand, already sorted. With your right hand you pick up a new card from the table, then slide it into its correct position among the sorted cards by comparing from right to left. That is exactly how Insertion Sort works on an array — Figure 2.4 shows the analogy directly, and Figure 2.5 shows the same idea in code, one array position at a time.

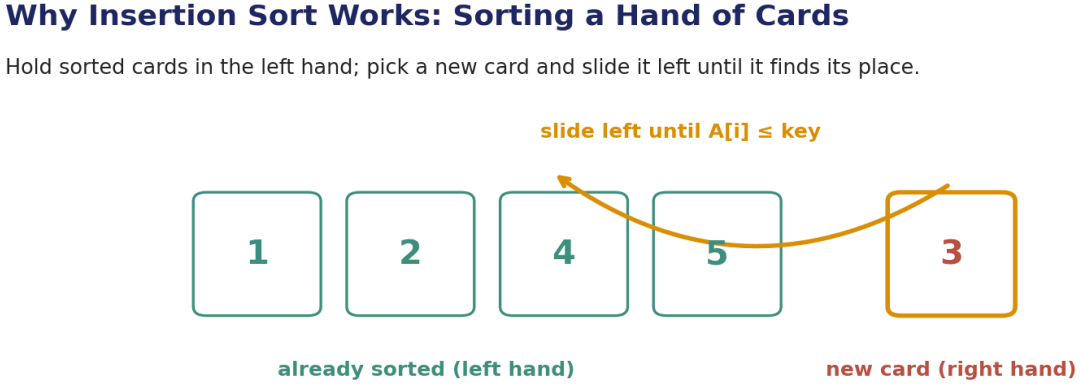

Figure 2.4 — The physical intuition behind Insertion Sort: a new card slides left until it finds its correct place among the already-sorted cards.

2.7.2 The Algorithm

Strategy: process elements from left to right. For each new element (the “key”), shift larger elements one position to the right to make room, then insert the key in its correct place.

```
INSERTION-SORT(A)
  for j = 2 to A.length              // start from 2nd element
      key = A[j]                     // pick up the card
      i = j - 1                      // start comparing from the left
      while i > 0 and A[i] > key      // shift bigger elements right
          A[i + 1] = A[i]
          i = i - 1
      A[i + 1] = key                  // insert the card
```

2.7.3 Tracing Through an Example

Let us sort A = [5, 2, 4, 6, 1, 3] step by step. Figure 2.5 shows the array before the loop begins (key = –, nothing sorted yet) and after each iteration j = 2, 3, 4, 5, 6 — the shaded prefix is exactly the sorted region the loop invariant guarantees, and the gold cell is the key just inserted.

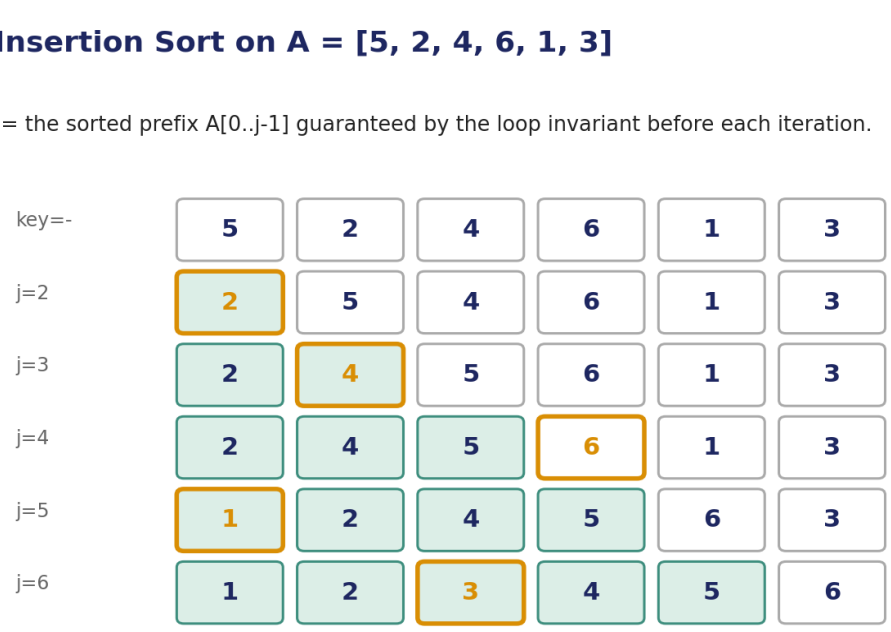

Figure 2.5 — Tracing Insertion Sort on A = [5, 2, 4, 6, 1, 3]. The array is fully sorted after j = 6.

2.7.4 Proving Correctness: Loop Invariant

> **How do we PROVE Insertion Sort always works?**
> Not just for this example — for every possible input array, of every possible length.

We use a loop invariant — a property that is true before and after every iteration of the loop — and prove it exactly like mathematical induction: a base case (Initialization), an inductive step (Maintenance), and a conclusion (Termination).

Loop Invariant for Insertion Sort

At the start of each iteration of the for loop (for index j), the subarray A[1..j−1] consists of the elements originally in A[1..j−1], but in sorted order.

- Initialization (j = 2): A[1..1] contains just one element. A single element is trivially sorted. ✓
- Maintenance: assume A[1..j−1] is sorted. The inner while loop shifts every element larger than key one position right, then places key in the correct spot. After this, A[1..j] is sorted, maintaining the invariant for the next iteration. ✓
- Termination (j = n + 1): the loop ends when j > n. At that point the invariant tells us A[1..n] consists of the original elements in sorted order — exactly the definition of a correct sort. ✓

Key insight

Loop invariants are the standard proof technique for iterative algorithms. Think of them as: "What is always true about the state of the computation at this point?" You will use this technique throughout the course.

2.7.5 Analyzing the Running Time

In the RAM model, each line has a constant cost. Let t_j be the number of times the inner while-loop body executes for a given j. The total running time is $T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4\Sigma t_j + c_5\Sigma(t_j-1) + c_6\Sigma(t_j-1) + c_7(n-1)$. The key variable is t_j — how many comparisons the inner loop makes — and this depends entirely on the INPUT, which is exactly why we need the best/worst/average-case distinction of the next section.

Phase 5 — Implement

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

void insertionSort(int A[], int n) {
    for (int j = 1; j < n; j++) {
        int key = A[j];
        int i = j - 1;
        while (i >= 0 && A[i] > key) {
            A[i + 1] = A[i];
            i = i - 1;
        }
        A[i + 1] = key;
    }
}

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

2.8 Best Case, Worst Case, and Average Case

The same algorithm can take very different amounts of time depending on the specific input. We distinguish three cases.

2.8.1 Best Case: Already Sorted

If A is already sorted, then for each j, A[j−1] ≤ key, so the while-loop body never executes: $t_j = 1$ for all j. This gives $T(n) = c_1 n + (c_2+c_3+c_4+c_7)(n-1) = an + b$ — a LINEAR function of n.

Key insight
Best case of Insertion Sort: T(n) is LINEAR — O(n). It just scans through, confirming everything is already in place.

2.8.2 Worst Case: Reverse-Sorted

If A is in strictly decreasing order (e.g., [6, 5, 4, 3, 2, 1]), every new element must be compared against ALL previously sorted elements: $t_j = j$ for each j. Since $\Sigma(j=2 \text{ to } n)\ j = n(n+1)/2 - 1$, this gives $T(n) = an^2 + bn + c$ — a QUADRATIC function of n.

Key insight
Worst case of Insertion Sort: T(n) is QUADRATIC — $O(n^2)$. Sorting a reverse-sorted array of 1,000,000 elements requires roughly 10^{12} operations — far too slow for any real time limit.

2.8.3 Average Case

On average, assuming a random permutation, each element is compared against about half of the previously sorted elements: $t_j \approx j/2$. This gives $T(n) = an^2/2 + \ldots$ — still $O(n^2)$. The average case has the SAME order of growth as the worst case.

2.8.4 Why This Course Focuses on the Worst Case

We almost always analyze the worst case, for four reasons: it gives a guaranteed upper bound — the algorithm will NEVER take longer than this; safety-critical systems (air-traffic control, surgical robots) MUST know the worst case; for many algorithms the worst case occurs frequently in practice; and the average case is often the same asymptotic order as the worst case anyway, while computing the TRUE average requires knowing the input distribution, which is often unknown.

A delivery-company analogy
“Your package arrives in at most 5 days” (a worst-case guarantee) is more useful for planning than “it usually arrives in 3 days” (an average-case claim). You can plan reliably around a guarantee; you cannot plan reliably around an average.

2.9 Growth of Functions: Why Asymptotics Matter

Not all functions grow at the same rate, and as n gets large the differences become dramatic. Table 2.1 makes this concrete for n = 1,000,000, assuming a modern machine executing roughly 10^9 simple operations per second.

Complexity	Operations at $n = 10^6$	Time at 10^9 ops/sec
O(log n)	≈ 20	instant
O(n)	1,000,000	0.001 s
O(n log n)	≈ 19,930,000	0.02 s

Complexity	Operations at n = 10^6	Time at 10^9 ops/sec
$O(n^2)$	1,000,000,000,000	≈ 17 minutes
$O(2^n)$	≈ 10^{301030}	far beyond the ~10^{80} atoms in the observable universe

Key insight

As n grows, the RATE OF GROWTH dominates everything. Constants and lower-order terms become irrelevant — this is exactly why we use asymptotic (Big-O) notation: it captures the growth rate and discards the noise.

2.9.1 The Hierarchy of Growth Rates

From fastest to slowest growing (best to worst for running times): $O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$.

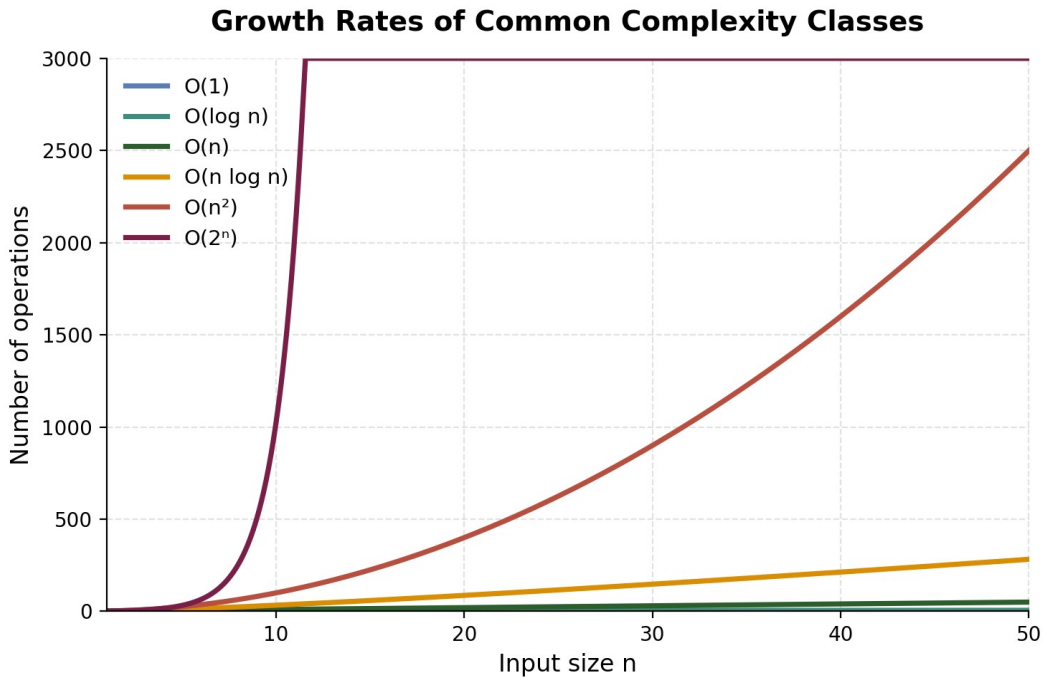

Figure 2.6 — How the number of operations grows with input size n (reproduced from Figure 1.1). Notice how $O(n^2)$ and $O(2^n)$ explode while $O(\log n)$ and $O(n)$ stay nearly flat.

A practical rule of thumb

If your algorithm is $O(n^2)$ and n = 1,000,000, it needs about 10^{12} operations — roughly 1,000 seconds, or 16 minutes and 40 seconds — way over almost any time limit. You need a better algorithm, not a faster computer.

2.10 Searching: The Fundamental Problem

2.10.1 Problem Definition

Input: a sequence A = [a_1, a_2, …, a_n] and a query value q. Output: an index j such that A[j] = q, or NIL if q is not in A.

2.10.2 Linear Search (Unsorted Array)

If the array is not sorted, we have no choice but to check every element in turn.

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

Worst case (q not found): we check all n elements, O(n). Average case (q at a random position): about n/2 elements, still O(n). There is no way to do better on an unsorted array — this is a LOWER BOUND, not merely an upper bound.

> **Key insight**
> Linear search on an unsorted array is O(n), and this is optimal — you cannot find an element faster without additional structure, such as sorting.

Phase 5 — Implement

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

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

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

2.10.3 Binary Search: Divide and Conquer for Searching

> **Can we do better?**
> What if the array is SORTED? Yes — sorted order lets us exploit structure for a fundamentally faster search.

Input: a SORTED non-decreasing sequence A = [$a_1 \le a_2 \le \ldots \le a_n$] and a query q. Compare q with the MIDDLE element: if A[mid] = q, we are done; if A[mid] > q, q must be in the left half, so discard the right half; if A[mid] < q, discard the left half instead. Each comparison HALVES the search space — the essence of divide and conquer.

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

Tracing a search for q = 7 in A = [2, 4, 5, 7, 10]: Figure 2.7 shows both steps. Binary search finds the answer in just 2 comparisons, instead of 4 for linear search on the same array.

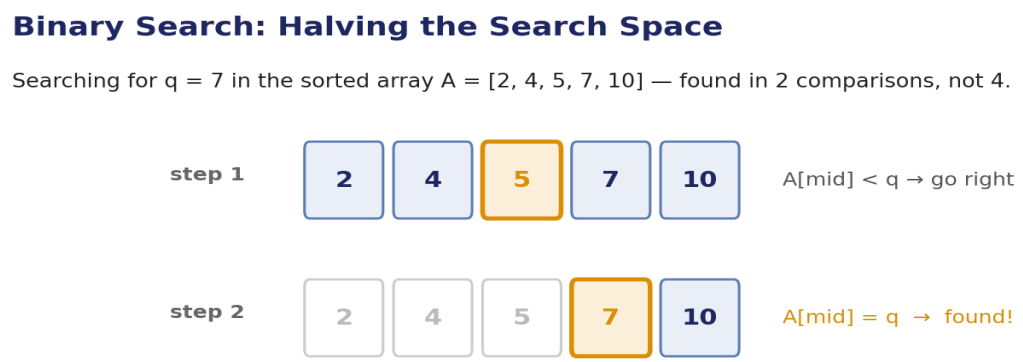

Figure 2.7 — Binary Search on A = [2, 4, 5, 7, 10] searching for q = 7. Each step discards half of what remains.

How many times can you cut n in half before reaching 1?

Start with n elements. After 1 comparison: n/2. After 2: n/4. After k: $n/2^k$. We stop when $n/2^k \leq 1$, i.e. $2^k \geq n$, i.e. $k \geq \log_2(n)$.

Key insight

Binary search runs in O(log n) time. For n = 1,000,000, it needs at most 20 comparisons — compare this to 1,000,000 comparisons for linear search. The sorted structure of the data gives an exponential speedup.

The tradeoff is preprocessing: binary search requires a sorted array, and sorting costs O(n log n). The total cost is O(n log n) to sort plus O(log n) per search — worth it whenever you search many times: for one search, linear is cheaper; for 100 searches, binary starts winning for large n; for 10,000 searches, binary search is overwhelmingly better.

Phase 5 — Implement

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

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

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

2.11 Design Paradigms Revisited

We have now seen concrete examples of the two foundational paradigms named in Chapter 1.

Incremental vs. Divide-and-Conquer

Two ways to grow a solution: one element at a time, or by splitting the problem in half and combining results.

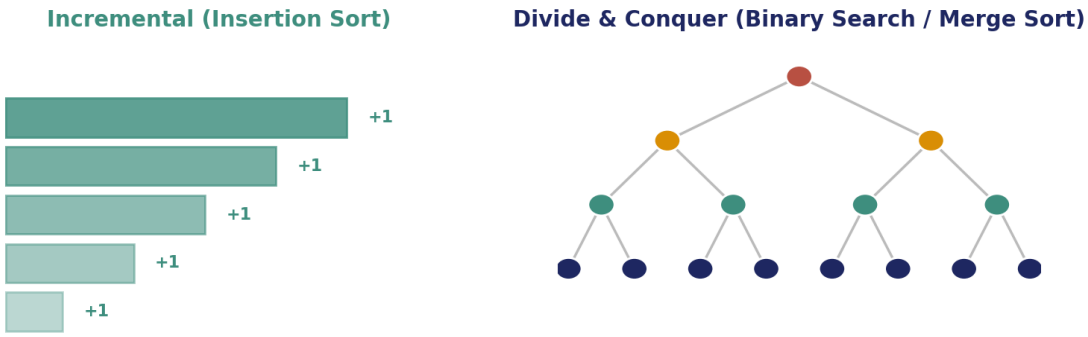

Figure 2.8 — Incremental algorithms grow the solution one element at a time; divide-and-conquer algorithms split the problem and combine sub-results.

2.11.1 Incremental Approach

Build the solution one piece at a time, processing one element per iteration. Insertion Sort is the classic example: at each step, we “insert” one more element into the already-sorted portion. Strength: simple, easy to understand and implement, and works well for small or nearly-sorted inputs. Weakness: often leads to $O(n^2)$ algorithms, because each insertion may require scanning the entire sorted portion.

2.11.2 Divide and Conquer

Break the problem into smaller sub-problems of the same type, solve them recursively, then combine the results. Binary Search is a perfect example: each comparison divides the search space in half. Strength: often leads to O(n log n) or O(log n) algorithms, naturally suited to recursion. You will soon study Merge Sort, which splits the array in half, sorts each half recursively, then merges the two sorted halves in O(n log n) — dramatically faster than Insertion Sort's $O(n^2)$ for large n.

2.11.3 Ad Hoc Algorithms

The Latin phrase ad hoc means “for this” — a solution designed for one specific problem that does not generalize. Lecture 1's Columns problem (answer = max(n, max(a_i))) and Interesting Sequence (answer = popcount(m−1) mod 3) are both ad hoc: powerful, but hard to teach systematically. The paradigms — incremental, divide-and-conquer, dynamic programming, greedy — are what give you GENERAL problem-solving tools that transfer to new problems.

2.12 Profitina in Practice: Where This Chapter Shows Up in a Real Product

> **About this box**
> As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details. See the companion document 'Profitina: A Non-Confidential Technical Overview' for the full picture.

Profitina's own engineering has to make the same best/worst/average-case judgment call this chapter teaches. When Profitina looks up a previously-scanned business by its normalized domain key inside an in-memory index, that lookup behaves like binary search over sorted keys in the worst case — O(log n) — rather than a linear scan of every record. But when a fresh signal must be merged into an already-processed result set one business at a time (for instance, incrementally folding in newly discovered citations for a business as they are found), the update pattern is structurally identical to Insertion Sort's incremental approach: each new item is "inserted" into its correct place among items already processed, and Profitina's engineers reason about that step's worst case exactly the way Section 2.8 does here — by asking what happens if every new item must be compared against everything already stored.

2.13 Chapter Summary and Key Takeaways

- An ALGORITHM is a step-by-step procedure; a PROGRAM is its implementation; a DATA STRUCTURE is how data is organized.
- We analyze algorithms using the RAM model, counting operations as a function of input size n.
- Insertion Sort is O(n) in the best case (already sorted) and $O(n^2)$ in the worst/average case (reverse-sorted / random).
- Loop invariants prove correctness for iterative algorithms: Initialization, Maintenance, Termination.
- We focus on WORST-CASE analysis because it provides a guaranteed upper bound.
- Growth-rate hierarchy: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$ — the differences become dramatic for large n.
- Linear search is O(n) and optimal for unsorted arrays; binary search is O(log n) for sorted arrays — an exponential improvement from exploiting structure.
- Design paradigms (incremental, divide-and-conquer) are general strategies; ad hoc solutions are problem-specific.

> **“An algorithm must be seen to be believed.”**
>
> **— Donald E. Knuth**
>
> *Trace every algorithm by hand. The understanding comes from the doing.*

Next lecture, we formalize everything used informally in this chapter: precise definitions of O, Θ, and Ω; how to formally prove f(n) = O(g(n)); and how to solve recurrences (the Master Theorem and substitution method) — the tools needed to analyze Merge Sort's T(n) = 2T(n/2) + O(n) = O(n log n).

2.14 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Lecture 3.

1. Prove the loop invariant for Insertion Sort in your own words, without looking at Section 2.7.4, using a fresh example array of your choosing.
2. Construct an input array of size 5 that produces the WORST case for Insertion Sort, and one that produces the BEST case. Compute t_j for every j in both cases.
3. Explain, in terms of the RAM model, why an array access A[i] costs the same (O(1)) regardless of how large i is or where in memory the array lives.
4. For n = 1,000, roughly how many operations does an O(n log n) algorithm perform, compared to an $O(n^2)$ algorithm? Use Table 2.1's method to justify your estimate.
5. Binary search requires a sorted array. If you need to perform only 3 searches on an array of 10,000 elements, is it worth sorting first? Justify using the O(n log n) sort + O(log n) per search cost model from Section 2.10.3.

Appendix 2.A — Verification Log for Chapter 2 Source Code

As in Chapter 1, every program shipped with this book is compiled and run against a hand-verified example before being included. All three programs below were compiled with `g++ -O2 -std=c++17 -Wall` (only benign scanf return-value warnings, zero errors) and produced the exact output predicted in the text above.

```
$ printf "6\n5 2 4 6 1 3\n" | ./01_insertion_sort
1 2 3 4 5 6                          # matches the Section 2.7.3 trace (Figure
2.5)

$ printf "5\n2 5 4 10 7\n2\n5\n9\n" | ./02_linear_search
1                                    # q=5 found at 0-indexed position 1
NIL                                  # q=9 is not present

$ printf "5\n2 4 5 7 10\n3\n7\n2\n6\n" | ./03_binary_search
3                                    # q=7 found at 0-indexed position 3, in 2
comparisons
0                                    # q=2 found at 0-indexed position 0
NIL                                  # q=6 is not present
```

SPOJ Practice — the companion book
This chapter's techniques are tested for real on the SPOJ online judge. Open Chapter 2–3 of the companion book “SPOJ Competitive Programming” (SPOJ folder, BI & EN) for insertion sort, linear & binary search under real constraints — complete with the tips, tricks, and hints that get solutions Accepted, without spoiling answers.

References

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

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

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

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

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 3

Asymptotic Notation, Recurrences, and Merge Sort

Making "O(n log n)" a precise, provable claim instead of a vibe — and meeting the first divide-and-conquer sorting algorithm that actually beats Insertion Sort.

Learning Objectives — after this chapter you will be able to:
(1) State the precise, constants-and-threshold definitions of O, Ω, and Θ notation, and use them to prove simple asymptotic claims directly. (2) Recognize a recurrence relation arising from a recursive algorithm, and solve it using the recursion-tree method, the substitution method, or the Master Theorem. (3) Trace, implement, and prove the correctness of Merge Sort, including the MERGE subroutine. (4) Derive Merge Sort's recurrence T(n) = 2T(n/2) + Θ(n) and solve it to show T(n) = Θ(n log n). (5) Explain, with concrete numbers, why an O(n log n) algorithm is not just "a bit faster" than an O(n^2) one but categorically different at scale.

3.1 Recap: From Lecture 2 to Lecture 3

Lecture 2 used Big-O notation constantly — "O(n)", "O(n^2)", "O(log n)" — but always informally, leaning on intuition about growth rates. That informality was fine for building intuition, but it cannot support a rigorous proof, and it cannot tell us how to solve the strange-looking equations that recursive algorithms produce, like T(n) = 2T(n/2) + Θ(n). This chapter fixes both gaps. First we make O, Ω, and Θ precise mathematical definitions with constants and thresholds, the kind you can use in a proof. Then we build three tools for solving recurrences — the recursion tree, substitution, and the Master Theorem — and use every one of them on Merge Sort, our first genuine divide-and-conquer sorting algorithm.

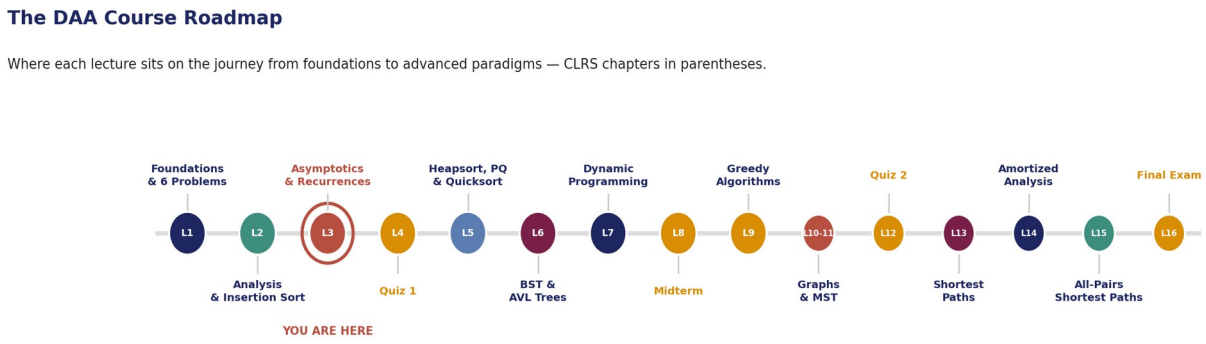

Figure 3.1 — Where this chapter sits in the sixteen-lecture DAA roadmap. Lecture 5 will apply the divide-and-conquer recipe built here to Quicksort; Lectures 6 and 7 then turn to new territory entirely — balanced search trees, and dynamic programming.

3.2 Formalizing Asymptotic Notation: O, Ω, and Θ

Lecture 2 used phrases like "T(n) is roughly n^2" without ever pinning down what "roughly" means. Here is the precise version, for a function f(n) that is asymptotically nonnegative (nonnegative for all sufficiently large n).

Big-O (upper bound)
f(n) = O(g(n)) if there exist positive constants c and n_0 such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$. In words: from some point on, f never exceeds a constant multiple of g. O(g(n)) is an upper bound — it may or may not be tight.

Big-Omega (lower bound)
f(n) = Ω(g(n)) if there exist positive constants c and n_0 such that $0 \le c \cdot g(n) \le f(n)$ for all $n \ge n_0$. In words: from some point on, f is never smaller than a constant multiple of g. Ω(g(n)) is a lower bound.

Big-Theta (tight bound)
f(n) = Θ(g(n)) if there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for all $n \ge n_0$. Equivalently: f(n) = O(g(n)) AND f(n) = Ω(g(n)). Θ is the tight bound — the one we actually want whenever we can get it.

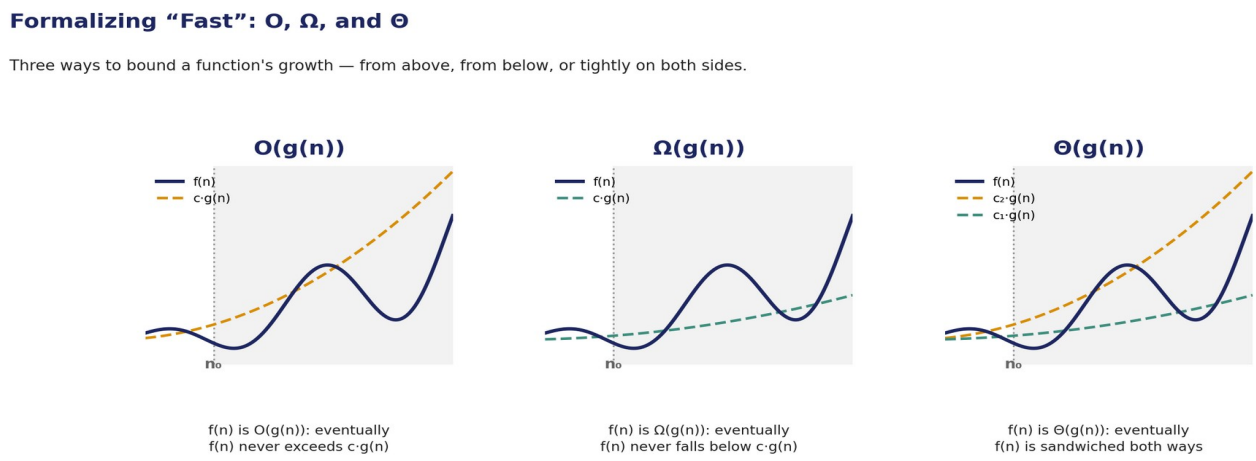

Figure 3.2 — All three notations only make a claim from n_0 onward; nothing is required for small n. O bounds from above, Ω from below, Θ sandwiches f(n) between two scaled copies of g(n).

Key insight
Every one of these definitions has the same shape: “eventually ($n \ge n_0$), up to a constant factor (c), f behaves like g.” The constants and the threshold are exactly what lets us ignore lower-order terms and machine-dependent constants — the very thing Lecture 2 did informally by eye.

3.3 Proving Asymptotic Bounds from the Definition

Let's prove a claim from Lecture 2 that we previously only asserted: $3n^2 + 2n + 5 = O(n^2)$.

Worked proof

Claim: $3n^2 + 2n + 5 = O(n^2)$. Proof: for all $n \geq 1$, $2n \leq 2n^2$ and $5 \leq 5n^2$, so $3n^2 + 2n + 5 \leq 3n^2 + 2n^2 + 5n^2 = 10n^2$. Choosing $c = 10$ and $n_0 = 1$, we have $0 \leq 3n^2 + 2n + 5 \leq 10n^2$ for all $n \geq n_0 = 1$. By the definition of O, $3n^2 + 2n + 5 = O(n^2)$. ∎

The same function is also $\Omega(n^2)$ — for all $n \geq 1$, $3n^2 + 2n + 5 \geq 3n^2$, so $c = 3$ and $n_0 = 1$ work — and therefore $\Theta(n^2)$. Notice that the constants c and n_0 are not unique; any $c \geq 10$ and $n_0 \geq 1$ would also work in the O proof above. A proof only needs to exhibit ONE valid pair (c, n_0), not the best possible one.

A common student mistake

"$f(n) = O(n^2)$" does NOT mean f grows exactly like n^2. It means f grows AT MOST like n^2, up to a constant. Both $f(n) = n$ and $f(n) = 0.001n^2$ are $O(n^2)$ — Big-O is an upper bound on a whole family of functions, not a single curve. This is precisely why we prefer Θ whenever we can prove it: Θ pins the growth rate down exactly.

3.4 Properties of Asymptotic Notation

A short list of facts, all provable directly from the definitions above, that make everyday asymptotic reasoning legal rather than just intuitive:

- Transitivity: if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$. (Also holds for Ω and Θ.)
- Reflexivity: $f(n) = \Theta(f(n))$ for any f.
- Transpose symmetry: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
- Sums: if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$. This is why, in $T(n) = c_1n + c_2n^2 + c_3$, the n^2 term alone determines the Θ-class — lower-order terms and constants are absorbed.
- Constants don't matter, growth rate does: for any constant $k > 0$, $k \cdot f(n) = \Theta(f(n))$.

Key insight

These properties are exactly why Lecture 2 could safely drop lower-order terms and constant factors when comparing algorithms — an operation like “add a constant number of extra array accesses” never changes the Θ-class of a correct running-time analysis.

3.5 Recurrences: Where They Come From

A recurrence is an equation that describes a function in terms of its own value on smaller inputs, together with a base case. Recurrences arise directly from recursive algorithms: to analyze a recursive algorithm, you first write down a recurrence for its running time T(n), then solve that recurrence to get a closed form like $\Theta(n \log n)$.

For a divide-and-conquer algorithm that divides a problem of size n into a subproblems, each of size n/b, and spends f(n) time dividing and combining (outside of the recursive calls), the running time satisfies:

> **The general divide-and-conquer recurrence**
> $T(n) = a \cdot T(n/b) + f(n)$, for n larger than some constant threshold, with $T(n) = \Theta(1)$ for the base case (small n). Here $a \geq 1$ is the number of subproblems, $b > 1$ is the factor by which the input shrinks, and f(n) is the cost of dividing the problem and combining the subproblem solutions.

This chapter's three solution techniques — recursion tree, substitution, and the Master Theorem — are three different ways of turning this equation into a closed-form Θ-bound.

3.6 Solving Recurrences I: The Recursion Tree Method

A recursion tree makes the total cost of a recurrence visible by drawing one node per recursive call, labeled with the cost of that call NOT counting its children, then summing every node in the tree. Consider $T(n) = 2T(n/2) + cn$ — the shape Merge Sort will turn out to have.

At the root, the non-recursive cost is cn. The root has 2 children, each solving a subproblem of size n/2, each with non-recursive cost c(n/2) — so the two children together also cost cn. Their children (4 of them, size n/4 each) again cost cn in total. This pattern — every level costing exactly cn — continues until the subproblems reach size 1, which takes $\log_2 n$ levels below the root, for $\log_2 n + 1$ levels in total (including the root).

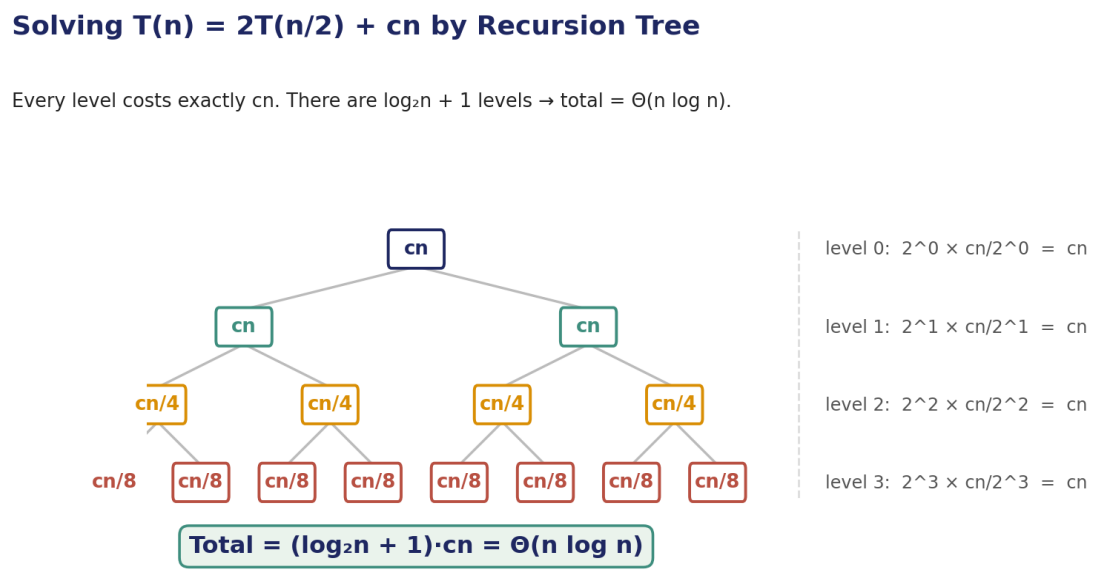

Figure 3.3 — The recursion tree for $T(n) = 2T(n/2) + cn$. Every level costs exactly cn regardless of depth, because each level has twice as many nodes as the level above, each doing half the work.

> **Recursion tree conclusion**
> Total cost = (number of levels) × (cost per level) = $(\log_2 n + 1) \times cn = cn \cdot \log_2 n + cn = \Theta(n \log n)$.

Key insight

The recursion tree method is intuitive and hard to get wrong once drawn correctly, but it is informal — it does not, by itself, constitute a proof. In practice it is used to GUESS the answer, which is then verified rigorously by substitution (Section 3.7) or read off directly from the Master Theorem (Section 3.8).

3.7 Solving Recurrences II: The Substitution Method

The substitution method proves a guessed bound by mathematical induction: guess the closed form, then verify it satisfies the recurrence by direct substitution into both sides.

Worked proof by substitution

Claim: $T(n) = 2T(n/2) + n$ is $O(n \log n)$. Guess: $T(n) \le cn \log_2 n$ for a constant c to be determined, for $n \ge 2$. Inductive step: assume $T(n/2) \le c(n/2)\log_2(n/2)$ holds for the smaller subproblem. Then $T(n) = 2T(n/2) + n \le 2 \cdot c(n/2)\log_2(n/2) + n = cn \cdot \log_2(n/2) + n = cn \cdot (\log_2 n - 1) + n = cn \cdot \log_2 n - cn + n = cn \cdot \log_2 n - (c-1)n$. This is $\le cn \cdot \log_2 n$ exactly when $(c-1)n \ge 0$, i.e. whenever $c \ge 1$. Choosing $c = 2$ (checking the base case $T(1) = \Theta(1) \le c \cdot 1 \cdot \log_2 1 = 0$ separately requires starting the induction at $n = 2$, where a direct check confirms the bound holds for a suitably large c), the inductive step goes through for all $n \ge 2$. Therefore $T(n) = O(n \log n)$. ∎

The subtlety everyone trips on

The inductive step above needed $(c-1)n \ge 0$ — the '$-cn$' term was ESSENTIAL to absorb the '$+n$' and keep the inequality tight. A common mistake is guessing $T(n) \le cn$ (linear) for this same recurrence: the induction then produces $T(n) \le cn + n$, which is NOT $\le cn$ for any fixed c — the guess was simply wrong, and no amount of algebra can force it through. Substitution proves a guess correct or reveals it false; it does not generate the guess for you (that's what the recursion tree is for).

3.8 Solving Recurrences III: The Master Theorem

The recursion tree and substitution methods both work, but both require some cleverness. For the common case $T(n) = aT(n/b) + f(n)$, the Master Theorem gives a direct answer by comparing $f(n)$ — the cost of dividing and combining — against $n^{(\log_b a)}$ — the cost that would result if the leaves of the recursion tree did all the work.

The Master Theorem (CLRS form)

Let a ≥ 1 and b > 1 be constants and let f(n) be a function, and let T(n) = aT(n/b) + f(n) (with T(n) = Θ(1) for n below some constant threshold). Then: Case 1: if f(n) = O(n^(log_b a − ε)) for some constant ε > 0 (f grows POLYNOMIALLY slower than n^(log_b a)), then T(n) = Θ(n^(log_b a)). Case 2: if f(n) = Θ(n^(log_b a)) (f grows at the SAME rate), then T(n) = Θ(n^(log_b a) · log n). Case 3: if f(n) = Ω(n^(log_b a + ε)) for some constant ε > 0 (f grows POLYNOMIALLY faster), AND the regularity condition a·f(n/b) ≤ c·f(n) holds for some constant c < 1 and all sufficiently large n, then T(n) = Θ(f(n)).

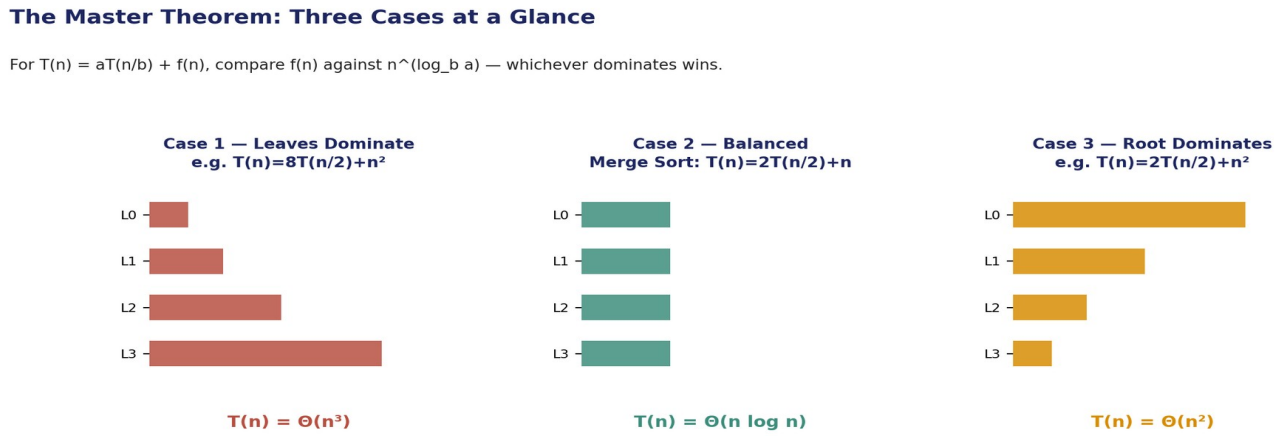

Figure 3.4 — The three cases correspond to where the recursion tree's cost is concentrated: at the leaves (Case 1), spread evenly across all levels (Case 2), or at the root (Case 3).

The intuition behind the three cases is exactly what Figure 3.4 shows. In Case 1, f(n) is small relative to n^(log_b a), so the leaves — where the a^(depth) recursive calls bottom out — contribute the most total cost, and that leaf cost dominates. In Case 2, every level of the recursion tree contributes the same amount (as in Merge Sort, Section 3.6's example), so the total is (cost per level) × (number of levels) = Θ(n^(log_b a) log n). In Case 3, f(n) is large enough that the very first call already dominates everything its children could possibly add, provided the regularity condition rules out f(n) oscillating in a way that breaks this — a technical condition satisfied by every polynomial f(n) you will meet in this course.

The Master Theorem does not always apply

There is a gap between Cases 1 and 2 (f(n) polynomially smaller is Case 1, but merely asymptotically smaller without a polynomial gap is neither case) and a similar gap between 2 and 3. A recurrence like T(n) = 2T(n/2) + n/log n falls into none of the three cases — the Master Theorem simply gives no answer, and you must fall back to the recursion tree or substitution method instead.

Applying this to Merge Sort's recurrence, derived in Section 3.9.5: T(n) = 2T(n/2) + Θ(n). Here a = 2, b = 2, so n^(log_b a) = n^(log₂2) = n¹ = n. Since f(n) = Θ(n) = Θ(n^(log_b a)), we are in Case 2, giving T(n) = Θ(n log n) — matching exactly what the recursion tree in Section 3.6 found by direct summation.

3.9 Case Study: Merge Sort — A Complete Analysis

Merge Sort is our first genuine divide-and-conquer sorting algorithm, and the algorithm that makes every tool built in this chapter concrete.

3.9.1 The Strategy

Divide the n-element array into two halves of about n/2 elements each. Conquer by recursively sorting each half. Combine the two now-sorted halves by merging them into a single sorted array. The base case is an array of 0 or 1 elements, which is trivially already sorted.

```
MERGE-SORT(A, p, r)
  if p < r                          // more than one element?
      q = floor((p + r) / 2)        // midpoint
      MERGE-SORT(A, p, q)           // conquer left half
      MERGE-SORT(A, q + 1, r)       // conquer right half
      MERGE(A, p, q, r)             // combine
```

3.9.2 Tracing the Divide-and-Combine Process

Let's sort A = [5, 2, 4, 7, 1, 3, 2, 6]. Figure 3.5 shows the full recursion: each node's top (gray) label is the subarray before recursing, and the bottom (bold, colored) label is the sorted result once MERGE returns. Reading the tree top-down shows the DIVIDE phase splitting the array down to single elements; reading it bottom-up shows the COMBINE phase merging sorted runs back into the fully-sorted array.

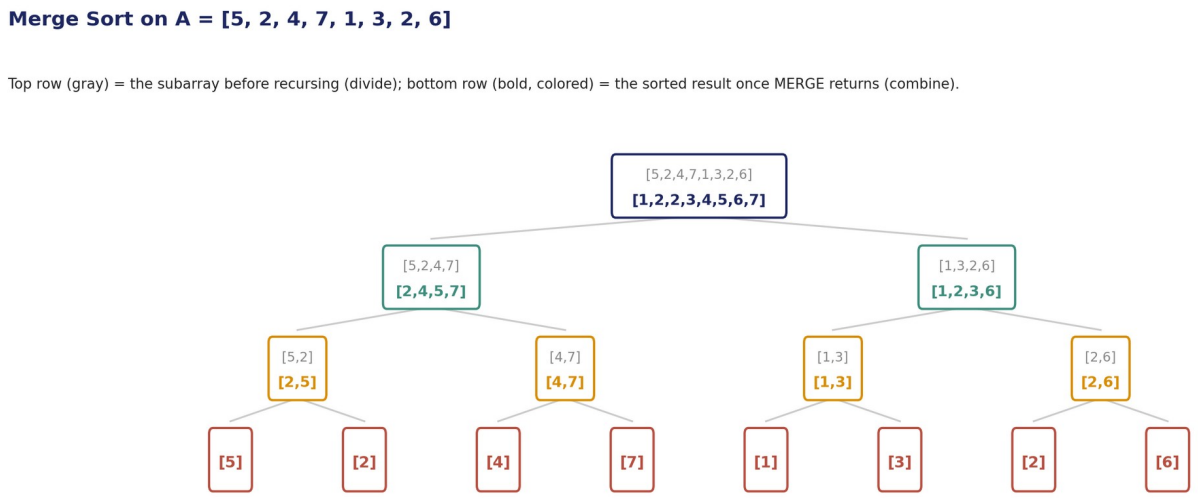

Figure 3.5 — Merge Sort on A = [5, 2, 4, 7, 1, 3, 2, 6]. Eight singleton leaves merge in pairs, then in pairs of pairs, until the root holds the fully sorted array [1, 2, 2, 3, 4, 5, 6, 7].

3.9.3 The MERGE Subroutine

MERGE(A, p, q, r) assumes A[p..q] and A[q+1..r] are each already sorted, and merges them into a single sorted run occupying A[p..r]. The idea: copy both runs into temporary arrays L and R, then repeatedly compare their front elements, copying the smaller one into the output and advancing that run's pointer, until one run is exhausted — at which point the remainder of the other run is copied in directly, since it is already sorted.

```
MERGE(A, p, q, r)
  n1 = q - p + 1                      // length of left run
  n2 = r - q                          // length of right run
  let L[1..n1] and R[1..n2] be new arrays
  for i = 1 to n1:  L[i] = A[p + i - 1]
  for j = 1 to n2:  R[j] = A[q + j]
  i = 1, j = 1
  for k = p to r:                     // fill A[p..r] in order
      if j > n2 or (i <= n1 and L[i] <= R[j])
          A[k] = L[i];  i = i + 1
      else
          A[k] = R[j];  j = j + 1
```

Figure 3.6 shows a snapshot mid-merge, taken from the top-level merge of [2,4,5,7] and [1,2,3,6] in the trace above. Three elements have already been placed in the output (1, 2, 2); the pointers now sit at L[1]=4 and R[2]=3. Since R's front element is smaller, it is copied next and its pointer advances — exactly the rule the pseudocode encodes as the if/else comparison inside the for loop.

Anatomy of MERGE: Two Sorted Runs, One Pointer Each

Mid-merge snapshot: 3 elements already placed. Compare the two pointers, copy the smaller, advance that pointer.

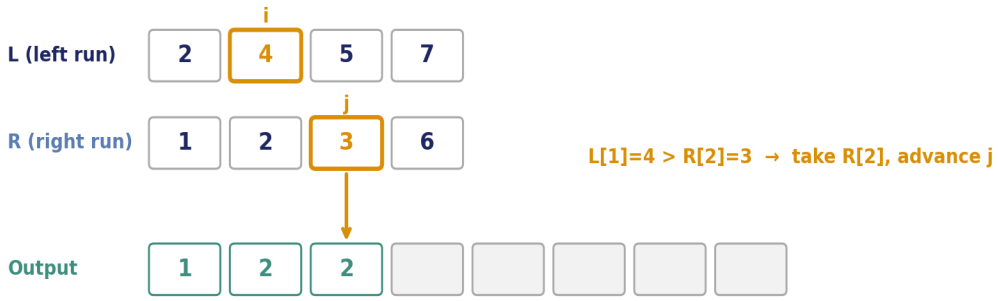

Figure 3.6 — Anatomy of one MERGE step: compare the two pointers' current elements, copy the smaller into the output, advance that pointer. Repeat until one run is exhausted.

3.9.4 Proving MERGE Correct: A Loop Invariant

What must be true every time through the for-k loop?

Exactly the same style of question Lecture 2 asked about Insertion Sort — and it has exactly the same style of answer.

Loop Invariant for MERGE

At the start of each iteration of the for k loop, the subarray A[p..k−1] contains the k−p smallest elements of L and R combined, in sorted order, and L[i] and R[j] are the smallest elements of their arrays not yet copied into A (or, if i > n1 or j > n2, that run is exhausted).

- Initialization (k = p): A[p..p−1] is empty, trivially containing the 0 smallest elements in sorted order, and i = j = 1 correctly point at the front of each freshly-copied run. ✓
- Maintenance: each iteration compares L[i] and R[j] (treating an exhausted run as effectively infinite) and copies whichever is smaller into A[k], then advances that pointer. Since L and R are each individually sorted, the copied element is the smallest remaining element overall,

so A[p..k] now holds the k−p+1 smallest elements in sorted order, maintaining the invariant for k+1. ✓

- Termination (k = r + 1): the invariant tells us A[p..r] contains all r−p+1 elements of L and R combined, in sorted order — exactly the postcondition MERGE must establish. ✓

> **Key insight**
> MERGE's correctness proof has the identical three-part shape (Initialization, Maintenance, Termination) as Insertion Sort's in Lecture 2 — because MERGE, despite feeling different, is still fundamentally an iterative algorithm with a for loop, and every correct iterative algorithm admits this style of proof.

3.9.5 Deriving and Solving the Recurrence

MERGE-SORT(A, p, r) on n = r − p + 1 elements makes 2 recursive calls, each on a subproblem of size n/2, plus a call to MERGE, which does Θ(n) work (copying n elements into L and R, then n more comparisons/copies back into A). This gives the recurrence:

> **Merge Sort's recurrence**
> $T(n) = 2T(n/2) + \Theta(n)$, with $T(1) = \Theta(1)$.

Section 3.6's recursion tree already solved exactly this recurrence (with the Θ(n) written as cn) and found Θ(n log n). Section 3.8's Master Theorem gives the same answer immediately: a = 2, b = 2, n^(log_b a) = n, and f(n) = Θ(n) matches n^(log_b a) exactly, so Case 2 applies and T(n) = Θ(n log n).

> **Key insight**
> Three independent methods — recursion tree, substitution, and the Master Theorem — all agree: Merge Sort runs in Θ(n log n) time, in the BEST, WORST, and AVERAGE case alike. Unlike Insertion Sort, Merge Sort has no bad inputs — the divide step always splits evenly regardless of the data, so there is no reverse-sorted array that can trap it into quadratic behavior.

Phase 5 — Implement

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

void merge(vector<int>& A, int p, int q, int r) {
    vector<int> L(A.begin() + p, A.begin() + q + 1);
    vector<int> R(A.begin() + q + 1, A.begin() + r + 1);
    int i = 0, j = 0, k = p;
    while (i < (int)L.size() && j < (int)R.size()) {
        if (L[i] <= R[j]) A[k++] = L[i++];
        else A[k++] = R[j++];
    }
    while (i < (int)L.size()) A[k++] = L[i++];
    while (j < (int)R.size()) A[k++] = R[j++];
}

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

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

3.10 Why It Matters: Merge Sort vs. Insertion Sort

$\Theta(n \log n)$ versus $\Theta(n^2)$ sounds like a modest difference until you plug in real numbers. Figure 3.7 uses the exact figures from Lecture 2's Table 2.1, at n = 1,000,000 on a machine executing roughly 10^9 simple operations per second.

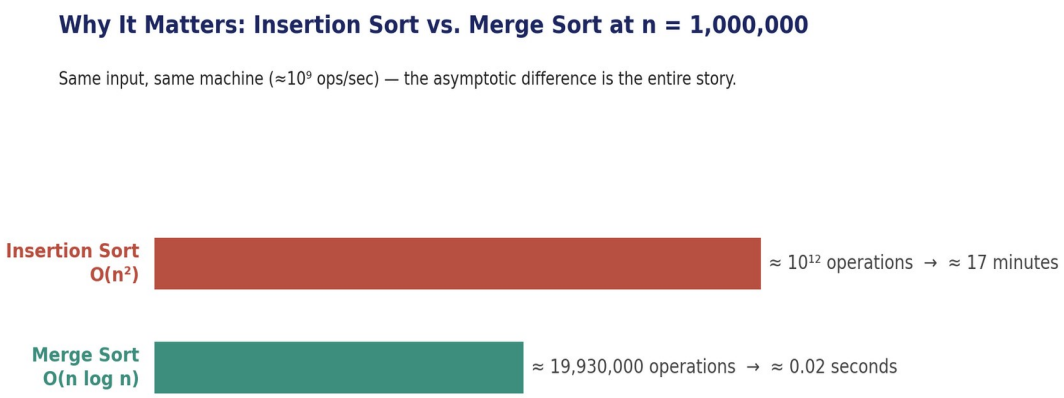

Figure 3.7 — At one million elements, Insertion Sort's Θ(n²) becomes a 17-minute wait; Merge Sort's Θ(n log n) finishes before you can blink. This is not a constant-factor improvement — it is a difference in kind.

The price Merge Sort pays for this speedup is space: unlike Insertion Sort, which sorts in place using O(1) extra memory, MERGE needs Θ(n) additional space for the temporary arrays L and R. This is a real, honest tradeoff — Θ(n log n) time at the cost of Θ(n) space — and it is the first time this course has had to weigh time against space explicitly. You will meet this tradeoff again and again for the rest of the course.

3.11 Design Paradigms Revisited: Divide and Conquer, Formalized

Lecture 2 introduced divide-and-conquer informally through Binary Search. Merge Sort lets us state the paradigm's three-step recipe precisely, now that we have the vocabulary to analyze each step:

- DIVIDE the problem into a subproblems, each of size n/b (for Merge Sort: a = 2, b = 2 — split into two equal halves).
- CONQUER each subproblem by solving it recursively. If the subproblem size is small enough, solve it directly (the base case).
- COMBINE the subproblem solutions into the solution for the original problem (for Merge Sort: the Θ(n)-time MERGE step).

The running time of any divide-and-conquer algorithm built this way is exactly the general recurrence from Section 3.5: T(n) = a·T(n/b) + f(n), where f(n) is the cost of the divide and combine steps. This is precisely why the Master Theorem is phrased in terms of a, b, and f(n) — it was built to analyze this exact recipe.

> **Key insight**
> Divide-and-conquer trades a harder-to-see global argument (why is the whole array sorted?) for an easier-to-see local one (if both halves are sorted, and MERGE is correct, the whole array is sorted). This recursive style of correctness argument — assume the recursive calls are correct, then prove the combine step preserves correctness — will recur for every divide-and-conquer algorithm to come, including Quicksort in Lecture 5.

3.12 Profitina in Practice: Where This Chapter Shows Up in a Real Product

> **About this box**
> As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

Profitina periodically needs to combine two already-ranked lists of business signals — for instance, a batch of newly re-scanned businesses (already sorted by visibility score within the batch) that must be folded into the existing, already-sorted leaderboard for a region. Rather than re-sorting the combined list from scratch in Θ(n log n) time, Profitina's engineers apply exactly the MERGE step from Section 3.9.3: since both lists are already individually sorted, a single Θ(n) linear pass with two pointers produces the fully merged, sorted result — the same asymptotic win divide-and-conquer sorting exploits at every level of its recursion, applied once at the top level of a much larger pipeline. Recognizing “both halves are already sorted” as the precondition for a cheap Θ(n) merge, instead of an expensive Θ(n log n) re-sort, is exactly the kind of asymptotic thinking this chapter is meant to train.

3.13 Chapter Summary and Key Takeaways

- O(g(n)), Ω(g(n)), and Θ(g(n)) are precise definitions involving constants c and a threshold n_0 — not vague descriptions of "how fast" a function grows.
- A recurrence T(n) = aT(n/b) + f(n) describes the running time of a divide-and-conquer algorithm that splits into a subproblems of size n/b, spending f(n) time outside the recursive calls.
- Three ways to solve a recurrence: the recursion tree (sum every level — good for guessing), substitution (prove a guess by induction — rigorous but needs a guess), and the Master Theorem (three cases based on comparing f(n) to n^(log_b a) — fast, but doesn't always apply).
- Merge Sort divides the array in half, recursively sorts each half, and merges the results in Θ(n) time via a two-pointer sweep — giving T(n) = 2T(n/2) + Θ(n) = Θ(n log n) in every case, best, worst, and average alike.
- MERGE's correctness follows the same Initialization/Maintenance/Termination loop-invariant recipe as Insertion Sort's did in Lecture 2.
- Θ(n log n) versus $\Theta(n^2)$ is not a minor speed difference — at n = 1,000,000 it is the difference between milliseconds and 17 minutes.
- The price of Merge Sort's speed is Θ(n) extra space — the first explicit time/space tradeoff of the course.

> ***"To understand recursion, you must first understand recursion."***
>
> **— an old CS joke, still not funny the 2^n-th time**
>
> *(If the joke didn't land, see: this quote.)*

Lecture 5 will apply everything built in this chapter — recurrences, the Master Theorem, the divide-and-conquer recipe — to Quicksort, an algorithm that partitions unevenly yet still achieves O(n log n) on average, in place. Lectures 6 and 7 then leave sorting behind for new territory: balanced binary search trees, and dynamic programming.

3.14 Review Questions

These questions are for self-study and class discussion; they are not graded. Work through them before Lecture 4 (Quiz 1).

1. Using the definition of O directly (exhibit c and n_0), prove that $5n^3 + 100n^2 + 1 = O(n^3)$.
2. Is $2^{n+1} = O(2^n)$? Is $n^2 = O(2^n)$? Justify both answers using the definitions from Section 3.2.
3. Solve $T(n) = 4T(n/2) + n$ using both the recursion tree method and the Master Theorem, and confirm the two answers agree. (Hint: this is Case 1.)
4. Solve $T(n) = T(n/2) + 1$ using the Master Theorem. What familiar algorithm from Lecture 2 has exactly this recurrence?
5. Trace MERGE-SORT by hand on A = [8, 3, 5, 1, 9, 2], drawing the full recursion tree in the style of Figure 3.5, and verify your final answer is sorted.
6. Explain why Merge Sort's $\Theta(n \log n)$ running time holds even in the WORST case, unlike Insertion Sort's, which is $\Theta(n)$ in the best case but $\Theta(n^2)$ in the worst.

Appendix 3.A — Verification Log for Chapter 3 Source Code

As in previous chapters, every program shipped with this book is compiled and run against a hand-verified example before being included. The program below was compiled with `g++ -O2 -std=c++17 -Wall` (only a benign scanf return-value warning, zero errors) and produced the exact output predicted in the text above.

```
$ printf "8\n5 2 4 7 1 3 2 6\n" | ./01_merge_sort
1 2 2 3 4 5 6 7                    # matches the Section 3.9.2 trace (Figure
3.5)

$ printf "6\n5 2 4 6 1 3\n" | ./01_merge_sort
1 2 3 4 5 6                        # same array Lecture 2 sorted with
Insertion Sort — same result, different algorithm
```

SPOJ Practice — the companion book
This chapter's techniques are tested for real on the SPOJ online judge. Open Chapter 3 & 9 of the companion book “SPOJ Competitive Programming” (SPOJ folder, BI & EN) for divide-and-conquer thinking and recurrence-driven complexity budgeting — complete with the tips, tricks, and hints that get solutions Accepted, without spoiling answers.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Chapters 3–4).

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

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

[4] J. von Neumann. "First Draft of a Report on the EDVAC" (1945) — an early description of the merge-sort-like sorting procedure implemented for the EDVAC, widely cited as the first written description of Merge Sort.

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 4 · EXERCISE CHAPTER

Practice Problems: Foundations Through Merge Sort

No new material here — just twelve problems designed to make sure Lectures 1 through 3 actually stuck, before Quiz 1.

Learning Objectives — after working through this chapter you will be able to:
(1) Identify which of the five defining properties of an algorithm a flawed procedure violates, and correct it. (2) Count RAM-model operations exactly, derive best/worst/average-case running times, and prove a loop invariant for a variant of Insertion Sort. (3) Order functions by asymptotic growth and adapt Binary Search to a new specification. (4) Prove an asymptotic bound directly from the definition of O, including proving a bound does NOT hold. (5) Solve unfamiliar recurrences with the recursion tree, substitution, and the Master Theorem — and recognize when the Master Theorem does not apply. (6) Extend Merge Sort's MERGE step to solve a related problem, and classify an algorithm by design paradigm.

4.1 Recap: Lectures 1–3 in Brief

Lecture 1 defined what an algorithm actually is — a finite, definite, effective procedure with well-specified inputs and outputs — and worked through six problems that showed the same task can have many correct solutions of wildly different quality. Lecture 2 gave us a machine-independent way to measure that quality: the RAM model, which counts operations instead of seconds, applied in full to Insertion Sort (with a rigorous loop-invariant proof of correctness) and to Linear and Binary Search. Lecture 3 then made the informal Big-O language of Lecture 2 mathematically precise — exact definitions of O, Ω, and Θ with constants and thresholds — and built three tools (the recursion tree, substitution, and the Master Theorem) for solving the recurrences that recursive algorithms like Merge Sort produce.

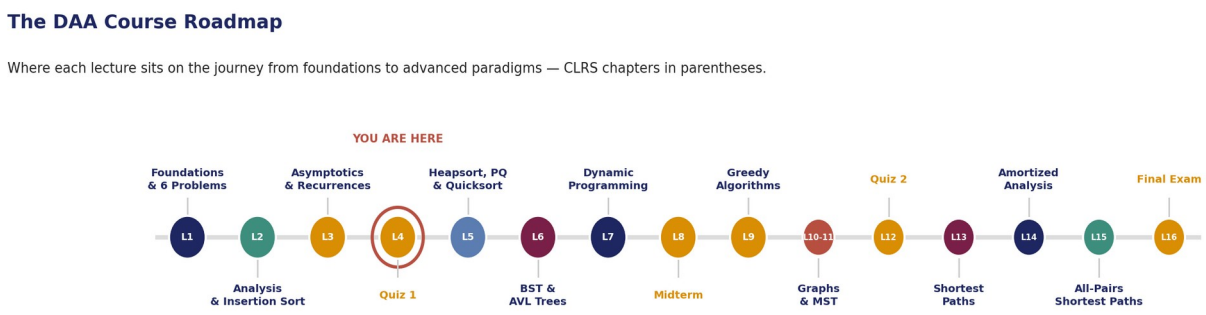

Figure 4.1 — This chapter sits at Lecture 4 in the sixteen-lecture DAA roadmap: a checkpoint, not a new topic, immediately before Quiz 1.

4.2 How to Use This Exercise Chapter

What this chapter is — and is not

This is a Bab Latihan: a practice chapter, not a new-content chapter. Every problem below is anchored to specific material from Lectures 1–3 (see Figure 4.2). Each problem comes with a HINT — a nudge toward the right way of thinking — but deliberately NOT a full worked solution or source code. Working through the problem yourself, even imperfectly, teaches far more than reading a finished answer.

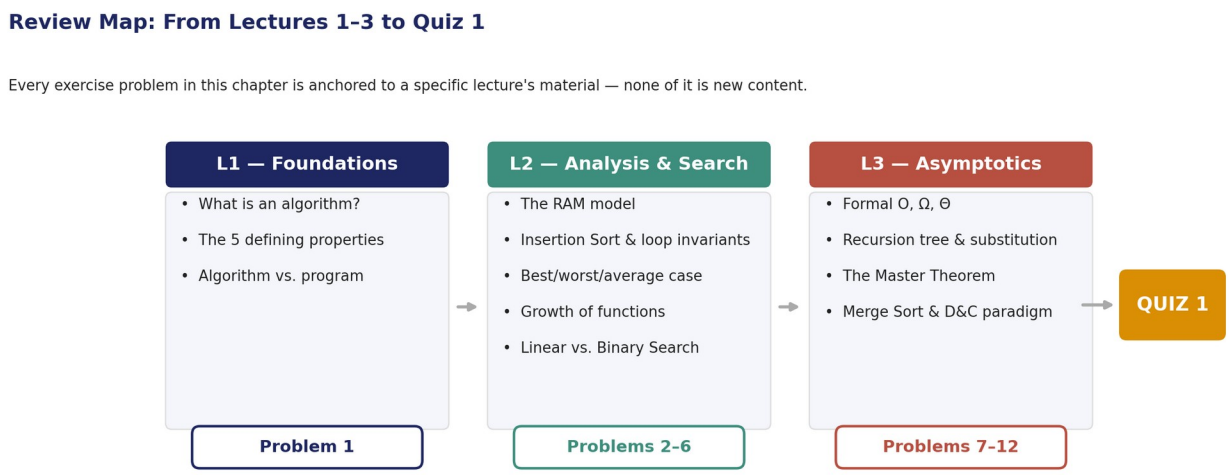

Figure 4.2 — Every problem in Section 4.3 traces back to one of the previous three lectures.

Before you start

Try each problem for real before reading its hint. If you get stuck, read only the hint — not a solution — and try again. This is exactly the discipline Section 4.4's open-book Quiz 1 will reward: the exam allows references, but not a substitute for your own reasoning.

4.3 Practice Problems

4.3.1 Foundations & the RAM Model

Problem 1 *[Easy]*. You are given a step-by-step "recipe" that includes the instruction "repeat step 3 until you are satisfied." Which property of an algorithm does this most clearly violate? Explain precisely why, and rewrite the instruction so it satisfies the precise definition of an algorithm.

Hint

Think back to the Finiteness and Definiteness properties from Chapter 1 — which one explicitly demands an OBJECTIVE stopping condition rather than a subjective one?

Problem 2 *[Easy]*. For the pseudocode fragment "for i = 1 to n: for j = 1 to i: A[j] = A[j] + 1", compute EXACTLY how many times the assignment A[j] = A[j] + 1 executes, as a function of n — not just its Big-O. Give a closed form.

> **Hint**
> This is the same arithmetic series 1 + 2 + … + n you already used for Insertion Sort's worst-case cost in Chapter 2 — the summation formula is identical.

4.3.2 Insertion Sort & Case Analysis

Problem 3 *[Medium]*. Ordinary INSERTION-SORT inserts each element to the LEFT. Write the loop invariant for a version that scans from RIGHT to LEFT (inserting from the last element toward the first, still building an ascending sorted result). Prove Initialization and Termination explicitly (Maintenance may be summarized).

> **Hint**
> The invariant is almost identical to Chapter 2's — only the index range that is already sorted runs in the opposite direction, not the underlying logic.

Problem 4 *[Medium]*. For an input array that is ALMOST sorted — exactly one pair of elements swapped, everything else already in order — does Insertion Sort behave closer to its BEST case, WORST case, or somewhere in between? Justify using the t_j definitions from Chapter 2, not just intuition.

> **Hint**
> Compute t_j for EVERY position j — how many values of j require a shift of more than one position, and how far do they actually shift?

4.3.3 Growth of Functions & Searching

Problem 5 *[Medium]*. Order the following five functions from SLOWEST to FASTEST asymptotic growth, and for the ONE pair you think is most easily misjudged, explain why the ordering holds: n log n, 2^n, n^2, log n, n^1.5.

> **Hint**
> Put all five on the same footing for comparison — for example, take the log of each, or examine the ratio f(n)/g(n) as $n \to \infty$ — rather than guessing from their "shape."

Problem 6 *[Medium]*. Modify standard Binary Search so that, if the target q occurs multiple times in a sorted array, the function returns the index of its FIRST occurrence (not just any occurrence). Explain the change to the search condition, and argue why the complexity remains O(log n).

> **Hint**
> Instead of stopping the moment A[mid] == q is found, think about how to keep "narrowing left" even after a match has already been found.

4.3.4 Formal Asymptotic Notation

Problem 7 *[Hard]*. Prove from the DEFINITION (exhibit explicit c and n_0) that $5n^3 + 100n^2 + 1 = O(n^3)$, BUT $5n^3 + 100n^2 + 1 \neq O(n^2)$.

> **Hint**
> The $O(n^3)$ half follows Section 3.3's proof pattern directly. For the "$\neq O(n^2)$" half, try a proof by CONTRADICTION — assume some c and n_0 work, then show the inequality is guaranteed to fail for large enough n.

4.3.5 Recurrences

Problem 8 *[Hard]*. Solve $T(n) = 3T(n/2) + n$ using the recursion tree method (Section 3.6). What is the total cost at level i? How many levels are there? Which dominates the overall sum — the top levels or the bottom levels?

> **Hint**
> Unlike $T(n) = 2T(n/2) + cn$ in the textbook, where every level costs the same, here the number of nodes grows 3× per level while the per-node cost only shrinks 2× — pay close attention to that ratio.

Problem 9 *[Hard]*. Prove by induction that $T(n) = T(n/2) + 1$ is $O(\log n)$. State your guess, the full inductive step, and the base-case condition.

> **Hint**
> The guess takes the form $T(n) \leq c \cdot \log_2 n$. Compare carefully what happens to the "+1" term here versus the "+n" term in Section 3.7 — the constant c you need turns out to be far more forgiving.

Problem 10 *[Hard]*. For EACH of the following recurrences, determine whether the Master Theorem applies, and if so which case and what Θ-bound results: (a) $T(n) = 4T(n/2) + n$, (b) $T(n) = 4T(n/2) + n^2$, (c) $T(n) = 4T(n/2) + n^3$, (d) $T(n) = 2T(n/2) + n/\log n$.

> **Hint**
> Compute $n^{\log_b a} = n^2$ for all four ($a = 4$, $b = 2$). One of these four is deliberately designed so the Master Theorem does NOT apply — Section 3.8 already describes exactly what kind of gap to look for.

4.3.6 Merge Sort & Design Paradigms

Problem 11 *[Hard]*. A pair of indices (i, j) with $i < j$ is an INVERSION if $A[i] > A[j]$ — a measure of how "unsorted" an array is. Explain, without writing full code, how MERGE from Section 3.9.3 can be modified to COUNT the total number of inversions while still sorting the array, in the same overall $\Theta(n \log n)$ time as ordinary Merge Sort.

> **Hint**
> Every time the algorithm takes an element from R (the right run) because it is smaller than the element L is currently pointing at, how many new inversions were just "discovered" in that single operation?

Problem 12 *[Medium]*. For each of the following three algorithms, classify it as INCREMENTAL or DIVIDE-AND-CONQUER, and give ONE sentence of justification: (a) Insertion Sort, (b) Merge Sort, (c) Binary Search.

> **Hint**
> The incremental paradigm builds the solution one element at a time while maintaining an invariant; D&C splits the problem into independent subproblems and then combines them. Binary Search is a bit of a trap — it "splits" but only ever works on ONE side, not both.

4.4 Quiz 1 Format: Open-Book, Take-Home Essay

Quiz 1 is an open-book, take-home essay assignment covering exactly the material reviewed in this chapter — nothing beyond it. You will have one week to submit written answers with full reasoning; a correct final answer with no justification earns little credit, since the point of an essay-format quiz is to see HOW you think, not just what you conclude.

> **Open-book means references, not co-authors**
> You may consult the textbook, your notes, and lecture slides while answering. You may NOT collaborate with classmates or submit reasoning that is not genuinely your own — an open-book quiz tests whether you can APPLY what you have studied, unsupervised, which is exactly what "designing and analyzing algorithms" means in practice. If you quote a source directly, cite it.

Criterion	What it looks for	Weight
Correctness of reasoning	Is the logic, proof, or derivation actually valid — not just the final answer?	40%
Clarity of proof / derivation	Can a reader follow each step without having to guess what you meant?	25%
Proper use of notation	Are O/Ω/Θ, recurrences, and invariants stated precisely, matching Chapter 3's definitions?	20%
Presentation	Is the answer organized, legible, and easy to grade?	15%

4.5 Profitina in Practice: Self-Review Before Submission

> **About this box**
> As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

Before a new algorithm change ships to production at Profitina, engineers run it through a short self-review checklist — very much like Appendix 4.A below — before ever requesting a second pair of eyes: has the complexity claim actually been derived, not just guessed? Have edge cases (empty input, all-

equal elements, already-sorted input) been considered? This habit of adversarial self-checking before external review is exactly what makes an open-book take-home exam format work: the discipline of verifying your own reasoning, rather than trusting your first instinct, is a skill this chapter is built to train.

4.6 Chapter Summary

- This chapter introduced no new material — it is a checkpoint covering Lectures 1 through 3.
- Twelve problems span the full range reviewed: algorithm definitions, the RAM model, Insertion Sort and its invariants, growth of functions, searching, formal asymptotic proofs, recurrences (recursion tree, substitution, Master Theorem), Merge Sort, and design paradigms.
- Each problem includes a hint, not a solution — working through the reasoning yourself is the point.
- Quiz 1 is an open-book, take-home essay: references are allowed, but the reasoning must be your own, and correctness of REASONING (not just the final answer) is the majority of the grade.

"An algorithm must be seen to be believed."

— Donald E. Knuth

(Reading a hint doesn't count as "seeing" it — working the problem does.)

Appendix 4.A — Self-Check Checklist (Non-Graded)

Before submitting any answer — in this chapter's problems or in Quiz 1 itself — run it through this checklist. It costs a few minutes and catches most of the mistakes graders see most often.

- Did I state a formal definition (not just intuition) whenever I was asked to prove something?
- In any O/Ω/Θ proof, did I exhibit explicit constants c and n_0 — not just assert the bound?
- In a substitution proof, did I check the base case AND the inductive step separately, and did I check the inductive step actually goes through algebraically (not just "looks right")?
- If I used the Master Theorem, did I state clearly which case I used and WHY f(n) falls into it?
- Did I distinguish claims about the BEST, WORST, and AVERAGE case explicitly, rather than treating "the running time" as one single number?
- Did I reread my own answer as if I were a skeptical grader looking for a gap in the argument?

References

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

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

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

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 5

Heapsort, Priority Queues, and Quicksort

Closing the gap between "in-place" and "fast" — two different answers to the same question, and the one that actually wins in practice.

Learning Objectives — after this chapter you will be able to:
(1) Explain how a binary heap encodes a nearly complete binary tree inside a plain array using only index arithmetic. (2) Trace, implement, and analyze HEAPIFY, BUILD-HEAP, and HEAPSORT, including the non-obvious proof that BUILD-HEAP runs in O(n), not O(n log n). (3) Use a heap to implement a priority queue, and state the running time of each operation. (4) Trace, implement, and analyze the PARTITION procedure and Quicksort, in its best, worst, and average cases. (5) Explain why randomization defeats adversarial worst-case inputs, and why Quicksort remains the default sort in most real systems despite a worse worst-case guarantee than Heapsort or Merge Sort.

5.1 Recap: From Lecture 3 to Lecture 5

Lecture 4 was Quiz 1 — a checkpoint on Lectures 1 through 3, with no new material. This lecture resumes the story exactly where Lecture 3 left off. Lecture 3 gave us Merge Sort: a divide-and-conquer algorithm that guarantees Θ(n log n) time in every case, at the cost of Θ(n) extra space for its MERGE step. That leaves an open question, visible the moment you line up everything sorted so far in one table.

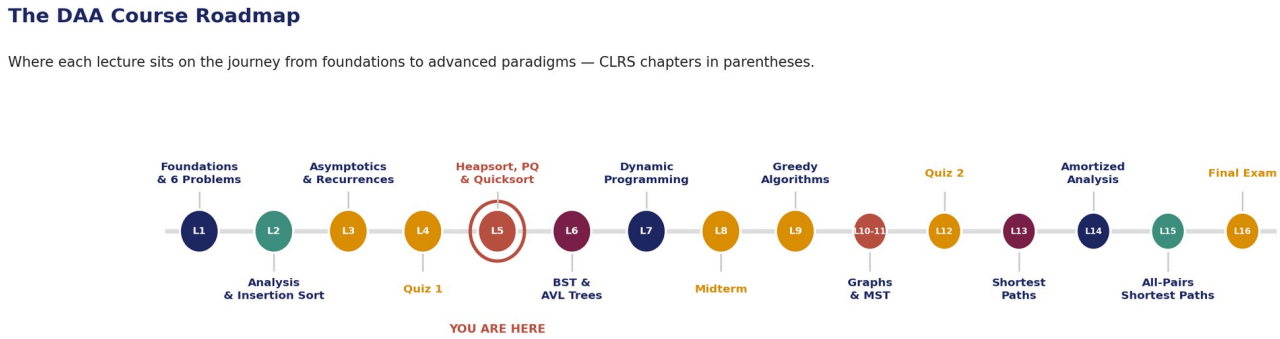

Figure 5.1 — Where this chapter sits in the sixteen-lecture DAA roadmap. Lecture 6 turns to a different family of structures (balanced binary search trees); Lecture 7 turns to an entirely different design paradigm (dynamic programming). This chapter's two algorithms — Heapsort and Quicksort — are the last stop on the sorting story that began in Lectures 2 and 3.

Algorithm	Worst Case	Best Case	In-Place?	Extra Space	Key Trait
Insertion Sort	$\Theta(n^2)$	$\Theta(n)$	Yes	O(1)	Great for nearly-sorted data
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	Yes	O(1)	Always exactly n−1 swaps
Bubble Sort	$\Theta(n^2)$	$\Theta(n)$	Yes	O(1)	Simple but slow

Algorithm	Worst Case	Best Case	In-Place?	Extra Space	Key Trait
Merge Sort	Θ(n log n)	Θ(n log n)	No	O(n)	Fast but needs extra memory

Notice the frustrating gap in this table: every in-place sort so far is $\Theta(n^2)$, and the only $\Theta(n \log n)$ sort so far is not in-place. Is a sort that is BOTH fast ($O(n \log n)$) AND in-place ($O(1)$ extra space) even possible?

The dream, and two different answers

This chapter delivers TWO independent solutions to exactly that gap. Heapsort (Sections 5.2–5.5) guarantees $O(n \log n)$ in every case, in-place, using a new data structure — the binary heap. Quicksort (Sections 5.6–5.9) achieves $O(n \log n)$ on average, also in-place, using no extra data structure at all — just a clever in-place partitioning step. They arrive at the same destination by completely different roads, and, surprisingly, the theoretically weaker of the two (Quicksort) is the one almost every real system actually uses.

5.2 The Selection Sort Insight: Speed Up the Bottleneck, Not the Whole Algorithm

Recall Selection Sort from Lecture 2: each iteration does two things. Step A scans A[1..i] to find the maximum, costing $\Theta(i)$. Step B swaps that maximum into position i, costing $\Theta(1)$. Repeated n times, the total is $\Theta(n^2)$ — and Step A is entirely the bottleneck; Step B is already essentially free.

A question worth pausing on

If Step A could be done in $O(\log n)$ instead of $O(n)$ — find the max in logarithmic time — what would the total running time become? (n iterations) × $O(\log n)$ per iteration = $O(n \log n)$. The whole algorithm speeds up simply by making its slow step fast, without changing its outer structure at all.

That is exactly the strategy of this section: keep Selection Sort's basic shape — repeatedly extract the maximum, place it, shrink the remaining set — but replace the plain array scan with a smarter data structure that finds (and later removes) the maximum far faster. That data structure is the binary heap, and Selection-Sort-plus-a-heap has its own name: Heapsort.

Key insight

Sometimes the fastest way to speed up a correct but slow algorithm is not to redesign the algorithm at all — it is to change the DATA STRUCTURE underneath it. This is a recurring theme for the rest of the course.

5.3 The Binary Heap: A Tree Hidden Inside an Array

A binary heap is deceptively simple on the surface — just an array — but it secretly encodes a nearly complete binary tree, with no pointers anywhere.

> **Binary Max-Heap**
> A binary max-heap is an array A[1..n] viewed as a nearly complete binary tree where (1) every level except possibly the last is completely filled, (2) the last level is filled left to right, and (3) every parent is at least as large as both of its children: A[Parent(i)] ≥ A[i] for all i > 1. Property (3) is the max-heap property. A min-heap reverses the inequality: every parent ≤ both children.

5.3.1 The Array-to-Tree Correspondence: No Pointers Needed

The genius of a heap is that the tree structure lives entirely in index arithmetic — three formulas, given a node at 1-indexed position i:

Relationship	Formula	Example (i = 3)	Hardware trick
Parent of node i	⌊i / 2⌋	Parent(3) = 1	Right-shift by 1 bit
Left child of node i	2i	Left(3) = 6	Left-shift by 1 bit
Right child of node i	2i + 1	Right(3) = 7	Left-shift, set low bit

Multiplying or dividing by 2 in binary is a single bit-shift — one of the fastest operations any CPU can perform. There is no pointer-chasing and no scattered memory allocation; the whole tree lives contiguously in one array, which is also excellent for cache locality (a theme this chapter returns to in Section 5.5's trap about Heapsort).

5.3.2 A Concrete Example (CLRS Figure 6.1)

Take A = [16, 15, 10, 8, 7, 9, 3, 2, 4, 1]. Figure 5.2 draws exactly this array as a tree, with each node's array index shown beneath it.

A Binary Max-Heap Is a Tree Hidden Inside an Array

CLRS Figure 6.1 — array A = [16,15,10,8,7,9,3,2,4,1], viewed as a nearly complete binary tree. Every parent ≥ both children.

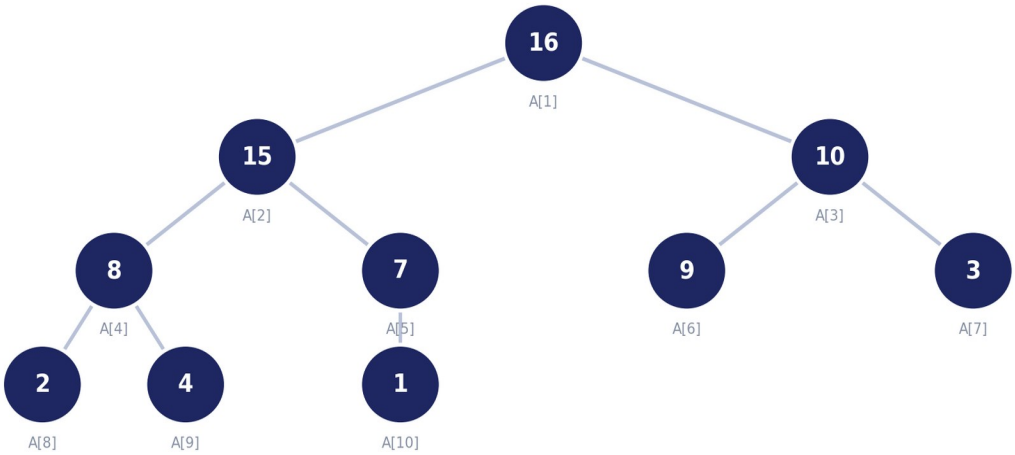

Figure 5.2 — CLRS Figure 6.1: the array A = [16,15,10,8,7,9,3,2,4,1] viewed as a nearly complete binary tree. Check every parent-child pair: 16≥15, 16≥10, 15≥8, 15≥7, 10≥9, 10≥3, 8≥2, 8≥4, 7≥1 — every one holds.

A heap is NOT a sorted array

Students very often assume a heap must be sorted. It is not. In Figure 5.2, node 9 (at A[6]) comes AFTER node 8 (at A[4]) in array order, yet 9 > 8. The heap property only constrains parent-versus-child relationships, never siblings or cousins. The ONLY guarantee a max-heap makes about the whole array is that A[1] holds the overall maximum.

5.3.3 Key Structural Properties

- The root A[1] always holds the maximum element of the entire heap.
- Height = ⌊lg n⌋. A 10-element heap, as in Figure 5.2, has height 3 (root at height 3, leaves at height 0).
- Leaves occupy positions ⌊n/2⌋+1 through n — the second half of the array. Each leaf is trivially a valid 1-element heap on its own.
- A heap is emphatically NOT sorted — it only guarantees parent ≥ children, never any global left-to-right order.

CP4 connection

C++'s std::priority_queue is a max-heap by default. Python's heapq module is a min-heap. Java's PriorityQueue is also a min-heap by default. To flip a min-heap into a max-heap (or vice versa), negate the keys or supply a custom comparator — a trick used constantly in competitive programming for problems built around Dijkstra's algorithm or Prim's algorithm.

5.4 HEAPIFY: The Engine That Keeps the Heap Honest

Imagine a node whose two subtrees are each already valid max-heaps, but the node itself might be too small for its position. HEAPIFY (sometimes called MAX-HEAPIFY) repairs exactly this situation by sinking the offending value down until the heap property is restored.

HEAPIFY(A, i)

Precondition: the binary trees rooted at LEFT(i) and RIGHT(i) are each valid max-heaps, but A[i] may violate the max-heap property with respect to its children. HEAPIFY repeatedly swaps A[i] with its largest child until A[i] is at least as large as both children, or i has no children (a leaf).

```
HEAPIFY(A, i):                          // CLRS p.165
  l = LEFT(i)                           // l = 2i
  r = RIGHT(i)                          // r = 2i + 1
  largest = i
  if l <= heap-size and A[l] > A[i]
      largest = l
  if r <= heap-size and A[r] > A[largest]
      largest = r
  if largest != i
      swap A[i] <-> A[largest]
      HEAPIFY(A, largest)               // recurse down into the affected subtree
```

In plain English: among node i and its (up to) two children, find whichever is largest. If i itself is already the largest, HEAPIFY is done — the subtree is already a valid heap. Otherwise, swap A[i] with its largest child and recurse into that child's now-possibly-violated subtree.

5.4.1 Walkthrough: HEAPIFY(A, 2)

Start from A = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1], where A[2] = 4 violates the heap property against its children A[4] = 14 and A[5] = 7. Figure 5.3 traces all three steps.

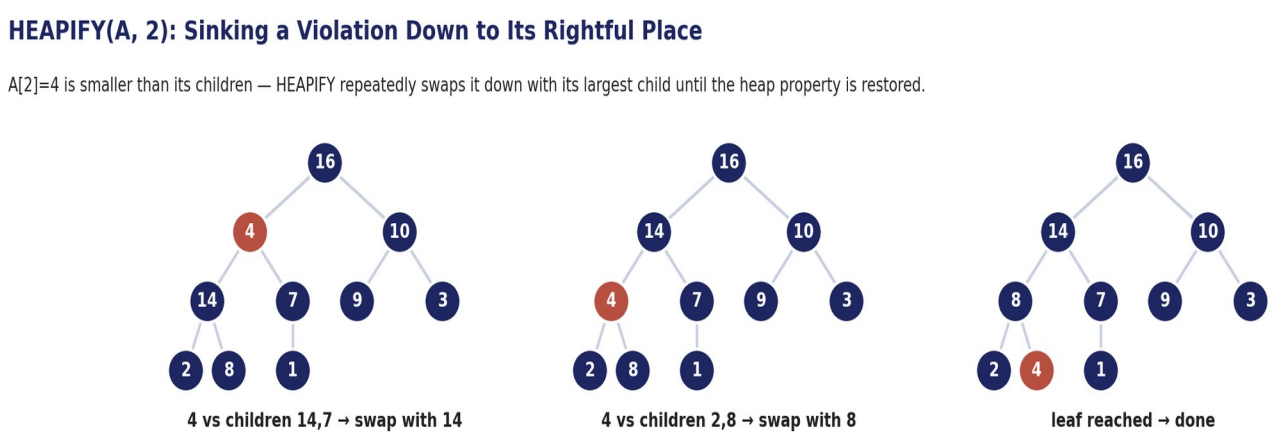

Figure 5.3 — HEAPIFY(A, 2) sinks the offending value 4 down two levels: first swapped with 14 (its larger child), then with 8 (the larger of its new children), until it reaches a leaf position with no further violation.

5.4.2 Running Time

> **HEAPIFY's running time**
> HEAPIFY runs in O(log n). Each call does Θ(1) work (comparing at most 3 values, at most 1 swap) plus a single recursive call into a subtree of size at most 2n/3. This gives the recurrence T(n) ≤ T(2n/3) + Θ(1), which the Master Theorem (Lecture 3, Case 2 with a=1, b=3/2) solves to T(n) = O(log n). Equivalently: HEAPIFY does at most one swap per level of the tree, and a heap has ⌊lg n⌋ levels.

5.5 BUILD-HEAP: The Surprising O(n) Construction

Given an arbitrary, unordered array A[1..n], BUILD-HEAP rearranges it into a valid max-heap using nothing more than repeated calls to HEAPIFY.

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

That is the entire algorithm: call HEAPIFY on every non-leaf node, working from the bottom of the tree up to the root. The leaves — positions ⌊n/2⌋+1 through n — need no work at all, since a single node is trivially a valid 1-element heap already.

5.5.1 Correctness: A Loop Invariant

> **Loop invariant for BUILD-HEAP**
> At the start of each iteration of the for loop, every node i+1, i+2, …, n is the root of a valid max-heap.

- Initialization (i = ⌊n/2⌋): nodes ⌊n/2⌋+1 through n are all leaves, each trivially a valid 1-element max-heap. ✓
- Maintenance: the children of node i have index strictly greater than i, so by the invariant they are already roots of valid max-heaps — exactly HEAPIFY's precondition. Calling HEAPIFY(A, i) therefore correctly makes node i the root of a valid max-heap too, extending the invariant to i, i+1, …, n. ✓
- Termination (i = 0): the invariant now covers nodes 1 through n — the entire array is a valid max-heap. ✓

5.5.2 Why This Is O(n), Not O(n log n)

A naive analysis says: n/2 calls to HEAPIFY, each O(log n), for a total of O(n log n). That bound is correct but not tight. The tight analysis notices that most of BUILD-HEAP's calls happen on tiny subtrees near the bottom of the tree, where HEAPIFY has almost nothing to do.

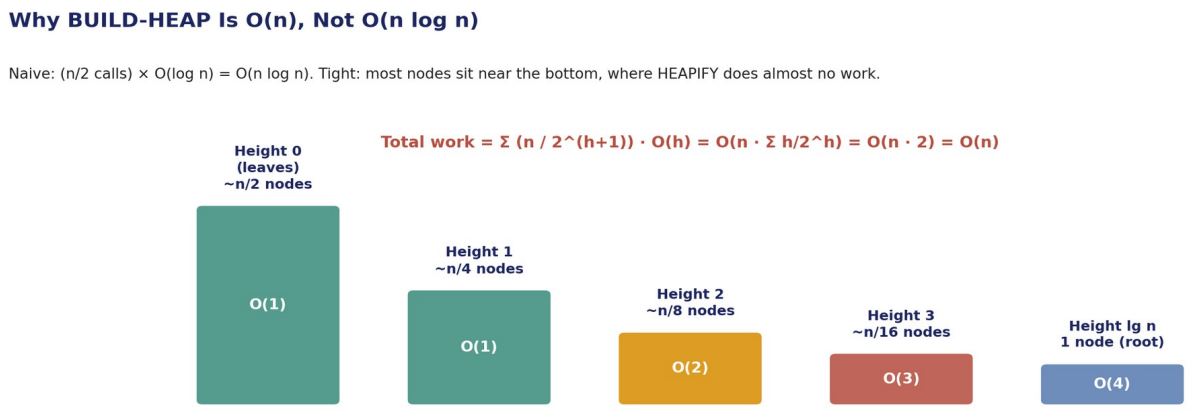

Figure 5.4 — About n/2 nodes sit at height 0 (leaves — never even called), n/4 at height 1, n/8 at height 2, and so on, with only 1 node (the root) at the maximum height lg n. Summing (count at height h) × (cost O(h)) over all heights gives a geometric-style series that converges to O(n).

The tight analysis

T(n) = Σ (n / 2^(h+1)) · O(h) = O(n · Σ h/2^h), summed over h from 0 to lg n. The infinite sum Σ h/2^h (for h = 0, 1, 2, …) converges to exactly 2 — a standard fact derivable from differentiating the geometric series Σ x^h = 1/(1−x). So T(n) = O(n · 2) = O(n).

Key insight

BUILD-HEAP takes only O(n) time, not O(n log n) — one of the most genuinely surprising results in introductory algorithm analysis, and a direct payoff of the tight-analysis habit Lecture 3 built (recursion trees, the Master Theorem): the naive per-call bound was correct but far from tight, and only summing per-level costs revealed the truth.

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

With BUILD-HEAP and HEAPIFY in hand, Heapsort itself is short: build a max-heap, then repeatedly move the current maximum (always at the root) to its final sorted position at the end of the array, shrink the heap, and restore the heap property at the (smaller) root.

```
HEAPSORT(A):                          // CLRS p.170
  BUILD-HEAP(A)                       // O(n)
  for i = n downto 2                  // n-1 times
      swap A[1] <-> A[i]               // move current max to its sorted slot
      heap-size = heap-size - 1        // shrink the heap, excluding A[i] from
now on
      HEAPIFY(A, 1)                 // O(log n): restore the heap property
```

After BUILD-HEAP, A[1] holds the overall maximum. Swapping it to position n places it in its final, correctly-sorted spot. Shrinking heap-size by 1 means the next HEAPIFY(A,1) never looks at that already-placed element again. HEAPIFY then restores the heap property using whatever value was swapped into the root, and A[1] once again holds the maximum of the REMAINING (smaller) heap. Repeating this n−1 times sorts the entire array.

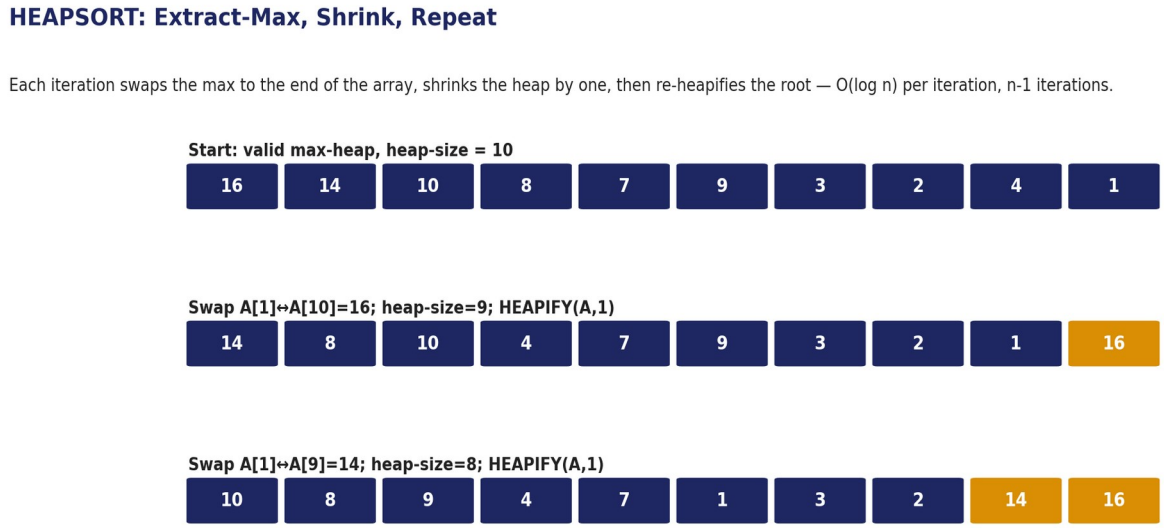

Figure 5.5 — The first two extract-and-shrink iterations on A = [16,14,10,8,7,9,3,2,4,1]. Each iteration swaps the root (gold) to the boundary between the shrinking heap and the growing sorted suffix, then re-heapifies the smaller heap.

Step	Cost	Times
BUILD-HEAP	O(n)	1
Loop: swap + HEAPIFY	O(log n)	n − 1
Total	O(n log n)	—

Heapsort's guarantee

Heapsort runs in Θ(n log n) in the worst case — always, with no bad inputs — and sorts in-place using only O(1) extra space. It is not a stable sort (equal elements can be reordered relative to each other).

Then why doesn't everyone use Heapsort?

Despite its guaranteed O(n log n) with no worst case, Heapsort has poor cache locality: HEAPIFY constantly jumps between a parent and children that can be far apart in memory (index i versus index 2i), causing frequent cache misses. Quicksort, by contrast, accesses memory in mostly sequential runs and is roughly 2–3× faster in practice despite its worse theoretical worst case. This is precisely why most standard libraries default to Quicksort-family algorithms, not Heapsort — Sections 5.7 onward explain why.

5.7 Priority Queues: The Heap's Most Famous Application

A priority queue (PQ) is an abstract data type for maintaining a dynamic set of elements, each with an associated key (its priority), that always serves the highest-priority element first — unlike a FIFO queue, which serves in arrival order.

Max-PQ operations
INSERT(S, x) — add element x to the set. MAXIMUM(S) — return the largest key without removing it. EXTRACT-MAX(S) — remove and return the largest key. INCREASE-KEY(S, x, k) — raise element x's key to the new value k (assumed ≥ its current key).

5.7.1 Applications

- Operating-system job scheduling: the highest-priority job runs next.
- Event-driven simulation: process events in chronological order using a min-PQ keyed by event time.
- Graph algorithms (previewed here, developed later in the course): Dijkstra's shortest path and Prim's minimum spanning tree both repeatedly extract a minimum-distance vertex from a min-PQ.
- Huffman coding: repeatedly extracting the two lowest-frequency nodes from a min-PQ builds an optimal prefix-free code for data compression.

5.7.2 Heap-Based Implementation

- MAXIMUM(A): return A[1] directly. Time: O(1).
- EXTRACT-MAX(A): save A[1], move A[heap-size] into the root, shrink the heap by one, HEAPIFY(A,1) to restore the property, then return the saved maximum. Time: O(log n).
- HEAP-INSERT(A, key): grow the heap by one slot, place key at the new bottom position, then "bubble up" — repeatedly swap it with its parent while the parent is smaller. Time: O(log n).

Operation	Heap-based PQ	Unsorted-array PQ
INSERT	O(log n)	O(1)
MAXIMUM	O(1)	O(n)
EXTRACT-MAX	O(log n)	O(n)
INCREASE-KEY	O(log n)	O(1)

CP4 tip
In contest settings, never hand-implement a heap from scratch — reach for the language's built-in priority queue. But understanding the internals above is essential for correctly analyzing the time complexity of any graph algorithm that USES a priority queue internally, which is exactly what later lectures on Dijkstra's and Prim's algorithms will require.

5.8 Quicksort: The Fastest Sort in Practice

Invented by Tony Hoare in 1959, Quicksort remains the default general-purpose sort in most real-world systems and standard libraries. Despite a $\Theta(n^2)$ worst case — theoretically worse than both Merge Sort and Heapsort — its average-case $\Theta(n \log n)$ carries tiny constant factors and excellent

cache behavior, which is why it wins in practice more often than its worst-case guarantee alone would suggest.

> **Quicksort**
> A divide-and-conquer sorting algorithm that (1) partitions the array around a chosen pivot, placing every element ≤ the pivot to its left and every element ≥ the pivot to its right, (2) recursively sorts each side, and (3) needs no combine step at all — once both sides are individually sorted, the whole array is sorted, in place.

5.8.1 Quicksort as Divide, Conquer, and (Trivial) Combine

1. Divide: choose a pivot x. Rearrange the array in place so that elements ≤ x come before elements ≥ x.
2. Conquer: recursively sort the left and right partitions.
3. Combine: nothing at all — the array is already fully sorted in place once both recursive calls return.

> **Compared with Merge Sort**
> Merge Sort has a trivial divide step (split the array exactly in half by index) but an expensive Θ(n) combine step (MERGE). Quicksort inverts this completely: an expensive divide step (PARTITION does all the real work) but a trivial combine step (nothing). The hard work simply moves from the combine phase to the divide phase — the same divide-and-conquer recipe from Lecture 3, applied with the cost distributed differently across its three steps.

5.9 The PARTITION Procedure

```
PARTITION(A, p, r):                    // Hoare's scheme, CLRS p.183
  x = A[r]                             // pivot = last element of the range
  i = p - 1
  j = r + 1
  while true
      repeat j = j - 1 until A[j] <= x
      repeat i = i + 1 until A[i] >= x
      if i < j
          swap A[i] <-> A[j]
      else
          return j
```

Pointer i scans rightward looking for an element that does NOT belong on the left (one that is ≥ the pivot). Pointer j scans leftward looking for an element that does NOT belong on the right (one that is ≤ the pivot). Whenever both pointers find a misplaced pair before crossing, they swap — trading each element into the half where it belongs. When the pointers finally cross, the partition is complete and PARTITION returns the split point j.

5.9.1 Walkthrough

Trace the scanning mechanics of PARTITION on the array [17, 12, 6, 19, 23, 8, 5, 10] using pivot value x = 17 as the comparison threshold. (This walkthrough isolates the two-pointer scan-and-swap

behavior on its own; Section 5.10 shows how QUICKSORT supplies x = A[r] automatically from whatever the last element of the current range happens to be.) Figure 5.6 shows all four scan-and-swap steps.

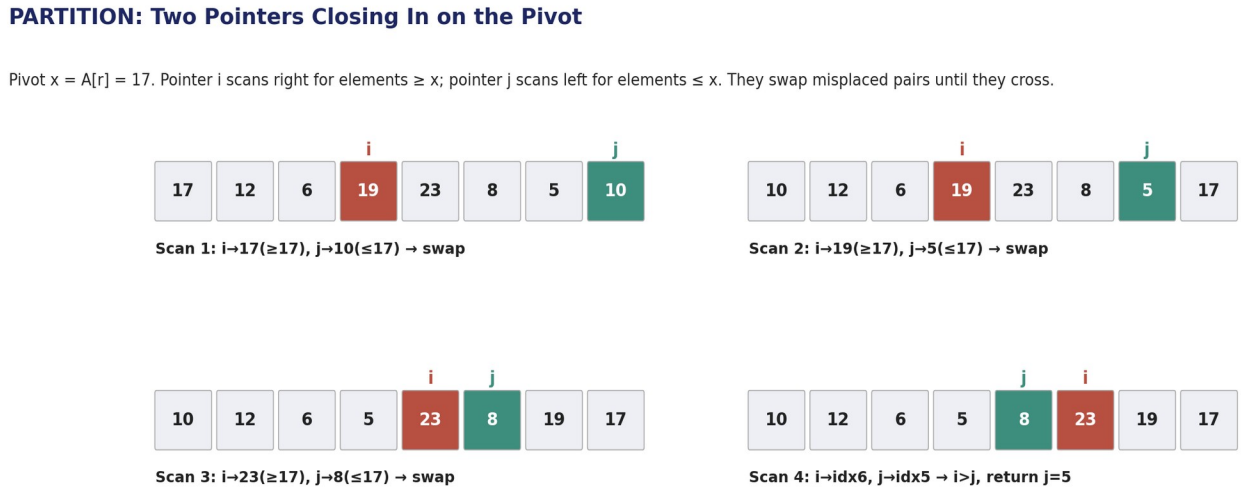

Figure 5.6 — Four scans of PARTITION with pivot x = 17. Pointer i (terracotta) hunts right for an element ≥ 17; pointer j (teal) hunts left for an element ≤ 17. Scans 1–3 each find a misplaced pair and swap; scan 4 finds i > j and returns the split point j = 5 — everything at or before index 5 is ≤ 17, everything after is ≥ 17.

> **Key insight**
> PARTITION does Θ(n) work — every element is examined by exactly one of the two pointers before they cross — using only O(1) extra space. This Θ(n), in-place partitioning step is precisely what QUICKSORT's divide step costs; the whole recurrence built in Section 5.10 flows directly from this single fact.

5.10 The QUICKSORT Algorithm

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

The initial call is QUICKSORT(A, 1, n). Note the recursive calls are QUICKSORT(A, p, q) and QUICKSORT(A, q+1, r) — not q−1 and q — because Hoare's partition scheme (Section 5.9) does not guarantee the pivot itself lands at index q; it only guarantees everything up to and including q is ≤ everything after q.

5.11 Analyzing Quicksort: Best, Worst, and Average Case

Quicksort's running time depends entirely on how balanced PARTITION's splits happen to be — this is the one thing that distinguishes it from Merge Sort, whose split is always exactly balanced by construction.

5.11.1 Best Case: Perfect Splits

If PARTITION always splits the array exactly in half, the recurrence is $T(n) = 2T(n/2) + \Theta(n)$ — precisely Merge Sort's recurrence from Lecture 3 — which the Master Theorem's Case 2 solves to $\Theta(n \log n)$.

5.11.2 Worst Case: Maximally Unbalanced Splits

If one side of every partition gets just 1 element and the other gets $n-1$ (which happens, for instance, on already-sorted or reverse-sorted input with a naive last-element pivot), the recurrence becomes $T(n) = T(1) + T(n-1) + \Theta(n) = \Theta(n) + \Theta(n-1) + \ldots + \Theta(1) = \Theta(n^2)$.

> **An ironic reversal**
> Sorted input is Insertion Sort's BEST case ($\Theta(n)$, Lecture 2) but naive Quicksort's WORST case ($\Theta(n^2)$)! This is exactly why naive, fixed-pivot Quicksort should never be used on data that might already be sorted or nearly sorted without some form of randomization (Section 5.12) — an adversary, or simply bad luck, can trigger exactly this worst case.

5.11.3 The 1-in-10 Split — Still O(n log n)!

Consider a much less balanced but still constant-ratio split: $T(n) = T(n/10) + T(9n/10) + \Theta(n)$. Figure 5.7 compares the recursion depth of this split against the perfectly balanced and maximally unbalanced cases.

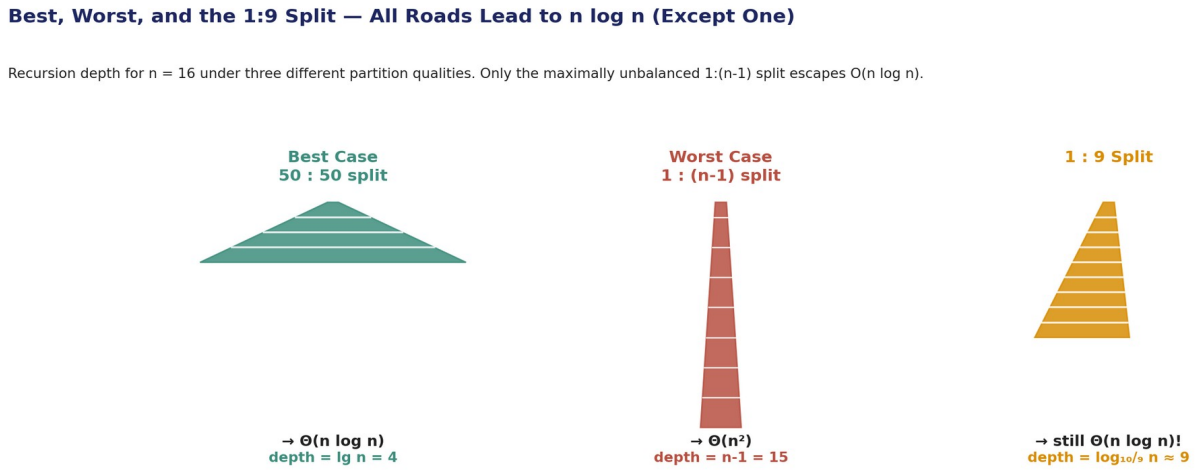

Figure 5.7 — Recursion shape for n = 16 under three partition qualities. The balanced case is shallow and wide (depth lg n); the degenerate 1:(n−1) case is a thin, deep sliver (depth n−1); the 1:9 split is deeper than balanced but still shallow relative to n, giving Θ(n log n) either way.

> **Why a lopsided split still works out**
> The short side of a 1:9 split dies out after $\log_{10} n$ levels; the long side survives roughly $\log_{10/9} n$ levels. But at EVERY level, the total work summed across all subproblems still adds up to at most $\Theta(n)$ — exactly as in the balanced case, just spread over more levels. So $T(n) = \Theta(n \cdot \log_{10/9} n) = \Theta(n \log n)$.

The generalization worth remembering

ANY split of constant ratio — 50:50, 10:90, even 1:99 — gives O(n log n) overall. Only the single degenerate case of a 1:(n−1) split, where one side never shrinks the problem by any constant fraction, produces $\Theta(n^2)$. This is exactly why Quicksort is so robust in practice: truly balanced pivots are rare, but truly degenerate ones are rarer still.

5.11.4 Average Case: Good and Bad Splits Alternating

Suppose splits alternate between lucky (perfectly balanced) and unlucky (1 versus n−1). Let L(n) denote the cost starting from a lucky split and U(n) from an unlucky one:

```
L(n) = 2·U(n/2) + Θ(n)              // one lucky split, then two subproblems
U(n) = L(n-1) + Θ(n)                // one unlucky split, then one subproblem
// Substituting U into L:
L(n) = 2·(L(n/2 - 1) + Θ(n/2)) + Θ(n) = 2·L(n/2 - 1) + Θ(n) = Θ(n log n)
```

Key insight

An unlucky split's extra cost gets absorbed by the lucky split that follows it — bad luck alone, so long as it alternates with good luck, cannot drag the average case down to $\Theta(n^2)$. A full probabilistic argument (beyond this course's scope) confirms this: Quicksort's TRUE average case over all input permutations is a solid Θ(n log n), with an expected constant close to 1.39·n·lg n comparisons.

5.12 Randomized Quicksort: Taming the Worst Case

With a fixed pivot-selection rule (like "always the last element"), an adversary who knows that rule can always construct a worst-case input in advance — as Section 5.11.2 showed for already-sorted data. The fix is to remove the adversary's ability to predict the pivot at all: choose it randomly.

```
RANDOMIZED-PARTITION(A, p, r):
  i = RANDOM(p, r)                  // uniformly random index in [p, r]
  swap A[r] <-> A[i]                // move the random pick into the pivot
position
  return PARTITION(A, p, r)         // then partition exactly as before

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

Properties of Randomized Quicksort

(1) Its running time no longer depends on the input's initial ordering — every input is, in expectation, an average-case input. (2) No single input can be crafted in advance to trigger the worst case, because the pivot choice itself is random, not a function of the data. (3) An unlucky sequence of coin flips COULD still, in principle, reproduce the $\Theta(n^2)$ worst case — but the probability of that is exponentially small, essentially never observed in practice. (4) Expected running time: O(n log n), with an expected ~1.39·n·lg n comparisons.

5.12.1 Other Pivot-Selection Strategies

- Median-of-three: use the median of A[p], A[⌊(p+r)/2⌋], and A[r] as the pivot — cheaply avoids the worst case on already-sorted input without full randomization.
- Dual-pivot Quicksort (Yaroslavskiy): used in Java's Arrays.sort for primitive types — two pivots partition the array into three regions instead of two.
- Introsort (used by C++'s std::sort): runs Quicksort, but falls back to Heapsort if the recursion depth grows suspiciously large (a sign of a near-worst-case split), and switches to Insertion Sort for small subarrays where its low overhead wins.

> **CP4 perspective**
> In competitive programming, always reach for the language's built-in sort rather than hand-rolling Quicksort: C++'s std::sort uses Introsort, Java uses dual-pivot Quicksort for primitives and TimSort for objects, and Python's sorted() uses TimSort. Understanding Quicksort's randomized analysis here is what later lets you reason correctly about expected running times and probabilistic arguments elsewhere in the course.

5.13 The Complete Sorting Landscape So Far

Algorithm	Worst	Average	Best	Space	Stable?	In-Place?	Introduced
Insertion Sort	n^2	n^2	n	1	Yes	Yes	Lecture 2
Selection Sort	n^2	n^2	n^2	1	No	Yes	Lecture 2
Bubble Sort	n^2	n^2	n	1	Yes	Yes	Lecture 2
Merge Sort	n lg n	n lg n	n lg n	n	Yes	No	Lecture 3
Heapsort	n lg n	n lg n	n lg n	1	No	Yes	Lecture 5
Quicksort	n^2	n lg n	n lg n	lg n	No	Yes	Lecture 5

5.13.1 When to Use What?

Situation	Best Choice & Why
Small array (n < 50)	Insertion Sort — low overhead, fast for tiny inputs
Nearly sorted data	Insertion Sort — Θ(n) best case for nearly-sorted input
General purpose, speed first	Randomized Quicksort — fastest in practice, excellent cache behavior
Must guarantee O(n log n)	Heapsort or Merge Sort — no $\Theta(n^2)$ worst case
Stability required	Merge Sort — the only Θ(n log n) stable sort seen so far
Memory very limited	Heapsort — O(1) extra space with a guaranteed O(n log n)
Dynamic insert + extract-max	Binary heap / priority queue — O(log n) for both operations
Competitive programming	The language's built-in sort: std::sort (Introsort), Arrays.sort, sorted() (TimSort)

5.14 Profitina in Practice: Where This Chapter Shows Up in a Real Product

> **About this box**
> As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

Profitina continuously monitors a large, ever-changing set of tracked businesses and needs to repeatedly answer "which business currently has the lowest AI-visibility score, and therefore needs attention first?" — exactly the EXTRACT-MIN operation from Section 5.7, applied to a min-heap keyed by visibility score. Scores update as new AI-answer scans complete, which is exactly an INCREASE-KEY / DECREASE-KEY style operation, and the heap-based priority queue keeps every operation at O(log n) even as the tracked-business set grows into the thousands, rather than the O(n) an unsorted list would require for every single lookup. Separately, when Profitina needs a one-off, general-purpose sort — ranking a freshly-pulled batch of competitor mentions by relevance score, for instance, with no particular structure to exploit — engineers reach for the platform's built-in sort (which is some Introsort or TimSort variant, per Section 5.12.1's CP4 perspective) rather than hand-rolling anything, precisely because Section 5.6's cache-locality trap and Section 5.12's randomization argument are exactly the considerations that battle-tested standard-library implementations have already engineered around.

5.15 Chapter Summary and Key Takeaways

- A binary heap stores a nearly complete binary tree implicitly inside a plain array, using only index arithmetic (Parent, Left, Right) — no pointers, and excellent cache locality for the tree traversal patterns HEAPIFY performs.
- HEAPIFY sinks a single violating element in O(log n) time; it is the one primitive operation underneath every other heap operation in this chapter.
- BUILD-HEAP runs in O(n), not the naively-expected O(n log n) — most of its calls operate on tiny subtrees near the bottom of the tree, where HEAPIFY does almost no work.
- Heapsort = BUILD-HEAP + (n−1) rounds of extract-max-and-shrink = Θ(n log n) in every case, in-place — but with poor cache locality that makes it slower in practice than Quicksort.
- A heap-based priority queue supports INSERT and EXTRACT-MAX (or -MIN) in O(log n), and MAXIMUM (or MINIMUM) in O(1) — the foundation for Dijkstra's and Prim's algorithms later in the course.
- Quicksort partitions in place around a pivot in Θ(n) time; best and average case are Θ(n log n), worst case is $\Theta(n^2)$.
- ANY constant-ratio split — not just perfectly balanced ones — gives Θ(n log n). Only the single degenerate 1:(n−1) split produces $\Theta(n^2)$.

- Randomized Quicksort removes an adversary's ability to predict the pivot, making the $\Theta(n^2)$ worst case exponentially unlikely rather than merely avoidable by careful input design.
- In practice, Quicksort usually beats Heapsort despite a worse worst-case guarantee, because of sequential memory access and small constant factors — theory and practice do not always point the same direction, and both matter.

> ***"Smart data structures and dumb code works a lot better than the other way around."***
>
> **— Eric S. Raymond**
>
> *(Selection Sort + a binary heap = Heapsort. Same "dumb" loop, a smarter data structure underneath.)*

Lecture 6 turns away from sorting entirely and toward a new family of data structures: balanced binary search trees, which keep search, insert, and delete all at O(log n) even under adversarial insertion order — solving, for lookup-heavy workloads, a version of the same balance problem this chapter solved for sorting.

5.16 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Draw the binary heap corresponding to A = [23, 17, 14, 6, 13, 10, 1, 5, 7, 12] as a tree, in the style of Figure 5.2, and verify the max-heap property holds at every parent-child pair.
2. Trace HEAPIFY(A, 1) by hand on A = [4, 14, 10, 8, 7, 9, 3, 2, 1] (a heap whose root violates the property against its children), in the style of Figure 5.3.
3. Explain, in your own words and without looking back at Section 5.5.2, why BUILD-HEAP is O(n) rather than O(n log n).
4. Trace PARTITION on [8, 3, 15, 2, 9, 5, 1, 12] using pivot x = A[r] = 12, in the style of Figure 5.6, and state the returned split index.
5. Construct a 6-element array that forces naive (last-element-pivot) Quicksort into its $\Theta(n^2)$ worst case, and explain why Randomized Quicksort would not reliably repeat the same worst case on that same array.
6. A priority queue is implemented as an UNSORTED array (not a heap). State the running time of INSERT, MAXIMUM, and EXTRACT-MAX for this implementation, and compare against Section 5.7.2's table for the heap-based version.

Appendix 5.A — Verification Log for Chapter 5 Source Code

As in previous chapters, every program shipped with this book is compiled and run against hand-verified examples before being included. Both programs below were compiled with `g++ -O2 -std=c++17 -Wall` (only a benign scanf return-value warning, zero errors) and produced the exact output predicted in the text above.

```
$ printf "10\n16 15 10 8 7 9 3 2 4 1\n" | ./01_heapsort
1 2 3 4 7 8 9 10 15 16               # sorts the exact heap from Figure 5.2

$ printf "8\n5 2 4 7 1 3 2 6\n" | ./01_heapsort
1 2 2 3 4 5 6 7               # same array Lecture 3 sorted with Merge Sort

$ printf "8\n17 12 6 19 23 8 5 10\n" | ./02_quicksort
5 6 8 10 12 17 19 23                  # randomized-pivot Quicksort, correctly
sorted

$ printf "10\n10 9 8 7 6 5 4 3 2 1\n" | ./02_quicksort
1 2 3 4 5 6 7 8 9 10          # reverse-sorted: naive worst case, fine here
```

SPOJ Practice — the companion book
This chapter's techniques are tested for real on the SPOJ online judge. Open Chapter 6 & 11 of the companion book “SPOJ Competitive Programming” (SPOJ folder, BI & EN) for heaps and priority queues as contest data-structure workhorses — complete with the tips, tricks, and hints that get solutions Accepted, without spoiling answers.

References

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

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

[3] C. A. R. Hoare. "Quicksort." The Computer Journal, 5(1), 1962, 10–16. The original publication describing the algorithm Hoare devised in 1959.

[4] E. S. Raymond. The Cathedral and the Bazaar. O'Reilly Media, 1999 — source of this chapter's closing quotation on data structures and code.

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 6

Binary Search Trees and AVL Trees

Leaving sorting behind: a structure that keeps its data in order all the time, and the balancing trick that keeps it fast no matter what order the data arrives in.

Learning Objectives — after this chapter you will be able to:
(1) Explain why a dynamic, order-preserving dictionary needs more than a plain or sorted array, and state the binary-search-tree property precisely. (2) Trace and implement SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR on a BST, and state each operation's running time in terms of tree height. (3) Trace and implement BST-INSERT and all three cases of BST-DELETE, including the in-order-successor splice for a two-child deletion. (4) Explain why an adversarial or already-sorted insertion order can degrade a BST's height to Θ(n), and why that makes every operation's O(log n) bound conditional rather than guaranteed. (5) State the AVL balance-factor invariant, trace all four rotation cases (LL, RR, LR, RL), and explain why they restore balance in O(1) time. (6) Prove, via the Fibonacci-tree argument, that an AVL tree's height is always O(log n) — turning the conditional bound of a plain BST into an unconditional guarantee.

6.1 Recap: From Lecture 5 to Lecture 6

Lecture 5 closed out the sorting story: Heapsort and Quicksort both turn an unordered array into a fully sorted one, and both do it well — but neither is designed to be queried, updated, and re-queried over and over as data arrives over time. Once Quicksort finishes, the array is a frozen snapshot; inserting one new element means, in general, re-sorting almost everything around it. This chapter asks a different question entirely: what if the data keeps changing — new records arrive, old ones are removed — and at every point in time you need to answer "is x present?", "what is the smallest/largest element?", or "what comes right after x?" quickly, without ever re-sorting from scratch?

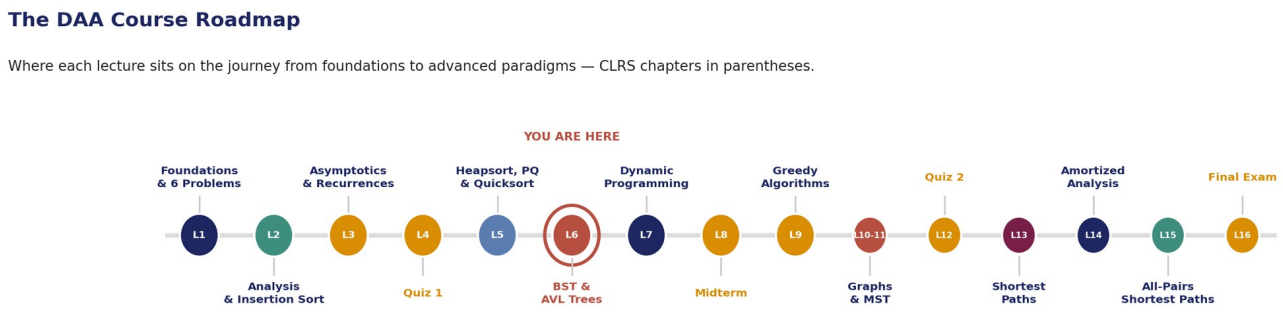

Figure 6.1 — Where this chapter sits in the sixteen-lecture DAA roadmap. Lecture 6 leaves sorting behind entirely for a new family of structures — trees that keep their contents in order at all times, not just at the end of one algorithm's run. Lecture 7 turns to an entirely different paradigm: dynamic programming.

A different kind of problem entirely

Lectures 2, 3, and 5 all asked "given all the data up front, produce a sorted array." This chapter asks "given data that arrives and leaves over time, always be ready to answer order-based queries quickly." The tool for the first job is an algorithm; the tool for the second job is a data structure — one that is never "done" the way Quicksort is done, and must instead stay correct and efficient after every single update.

6.2 Why a Sorted Array Is Not Enough: The Dictionary ADT

A dictionary (also called a dynamic set or a search structure) is an abstract data type that must efficiently support, at minimum: SEARCH(k) — is key k present, and if so, where; INSERT(x) — add element x; and DELETE(x) — remove element x. Many applications also need MINIMUM, MAXIMUM, SUCCESSOR(x), and PREDECESSOR(x) — the smallest, largest, next-larger, and next-smaller elements currently present.

A sorted array supports SEARCH beautifully — Binary Search from Lecture 2 finds any key in O(log n). MINIMUM and MAXIMUM are O(1) — they are just the first and last slots. But INSERT and DELETE are catastrophic: keeping the array sorted after inserting into the middle requires shifting, on average, Θ(n) elements to make (or close) a gap. An unsorted array flips the trade-off — O(1) insert at the end, but O(n) search, since nothing tells you where to look.

Structure	SEARCH	INSERT	DELETE	MIN / MAX	SUCCESSOR
Unsorted array	Θ(n)	Θ(1)	Θ(n)	Θ(n)	Θ(n)
Sorted array	Θ(log n)	Θ(n)	Θ(n)	Θ(1)	Θ(1)
Binary search tree (this chapter)	O(h)	O(h)	O(h)	O(h)	O(h)

Every row above has its own bottleneck operation. The binary search tree, introduced next, is the first structure in this course that supports ALL five operations in time proportional only to h, the tree's height — no operation is forced to be Θ(n) just because another one needs to be fast. The catch, developed across this chapter, is that h itself is not automatically small; Sections 6.8–6.9 show it can degrade to Θ(n), and Sections 6.10 onward show how AVL trees fix that.

6.3 Binary Search Trees: Definition and the BST Property

Binary Search Tree (BST)

A binary search tree is a binary tree in which each node stores a key (and, typically, associated data), and satisfies the BST property: for every node x, every key in x's left subtree is ≤ x's key, and every key in x's right subtree is ≥ x's key. Unlike a binary max-heap (Chapter 5), a BST's ordering is GLOBAL along any root-to-leaf path — it is not limited to parent-versus-child comparisons.

A Binary Search Tree: Order Lives in the Shape, Not the Array

Keys {1,3,4,6,7,8,10,13,14}. For every node: everything in its left subtree is smaller, everything in its right subtree is larger.

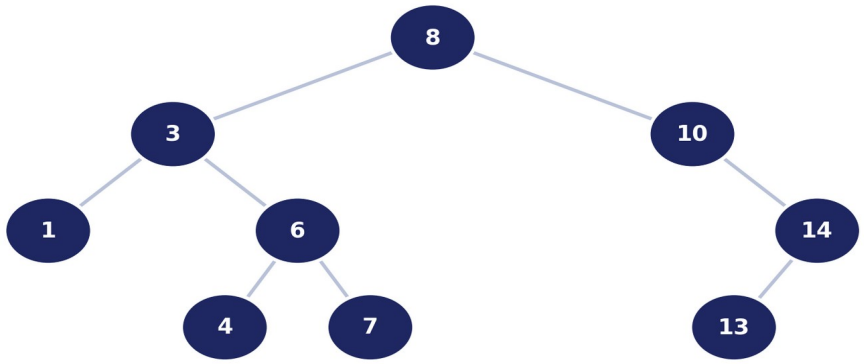

Figure 6.2 — A binary search tree holding the keys {1, 3, 4, 6, 7, 8, 10, 13, 14}. Check the BST property at the root: every key in the left subtree (1, 3, 4, 6, 7) is ≤ 8, and every key in the right subtree (10, 13, 14) is ≥ 8. The same property holds recursively at every other node.

> **A BST is not a heap, and vice versa**
>
> Chapter 5's binary max-heap only guarantees parent ≥ children — a purely LOCAL constraint that says nothing about how a node compares to its uncle, its cousin, or any node outside its own parent/child pair. A BST's property is GLOBAL: node 6 in Figure 6.2 is not just ≥ its own children — it is guaranteed smaller than every key anywhere in node 8's right subtree, even nodes 6 has no direct edge to. Confusing the two properties is one of the most common conceptual errors when moving from Chapter 5 into this chapter.

6.3.1 In-Order Traversal Recovers Sorted Order

Because of the BST property, visiting a tree's nodes in-order (recursively: left subtree, then the node itself, then right subtree) always outputs every key in non-decreasing sorted order. Figure 6.2's tree, visited in-order, yields exactly 1, 3, 4, 6, 7, 8, 10, 13, 14 — sorted, with no additional work beyond the Θ(n) traversal itself. This is the single most useful sanity check for any BST implementation: after any sequence of insertions and deletions, an in-order traversal of a correct BST must always come out sorted.

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

6.4 Order-Statistics Queries: MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR

Three of the dictionary operations from Section 6.2's table fall out of the BST property almost for free, once you notice where the smallest and largest keys must live.

- MINIMUM(x): the smallest key in the subtree rooted at x is always found by following left-child pointers until reaching a node with no left child — that node holds the minimum. Symmetrically, MAXIMUM(x) follows right-child pointers to the end.
- SUCCESSOR(x): the next-larger key after x. If x has a right subtree, the successor is MINIMUM(x.right) — the smallest key strictly greater than x. If x has no right subtree, the successor is the lowest ancestor of x whose left child is also an ancestor of x (in plain terms: walk up until you move from a left child to its parent).
- PREDECESSOR(x): the mirror image of SUCCESSOR — MAXIMUM(x.left) if x has a left subtree, otherwise the lowest ancestor reached by walking up from a right child.

> **Every one of these is O(h)**
> MINIMUM and MAXIMUM do at most h pointer-follows. SUCCESSOR and PREDECESSOR do at most one MINIMUM/MAXIMUM call (itself O(h)) plus at most h steps upward — still O(h) overall. This is the pattern that repeats for every BST operation in this chapter: the cost is always bounded by the height of the tree, never by the number of keys directly.

6.5 BST-SEARCH: Following One Path From the Root

Searching a BST for key k starts at the root and, at each node, compares k against the current node's key: if equal, done; if smaller, recurse left; if larger, recurse right. Because of the BST property, this single comparison at each step is enough to eliminate an entire subtree from consideration — there is never a need to backtrack.

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

Trace SEARCH(root, 4) on Figure 6.2's tree: 4 < 8, so go left to 3; 4 > 3, so go right to 6; 4 < 6, so go left to 4 — found, after 3 comparisons. Figure 6.3 draws all three steps.

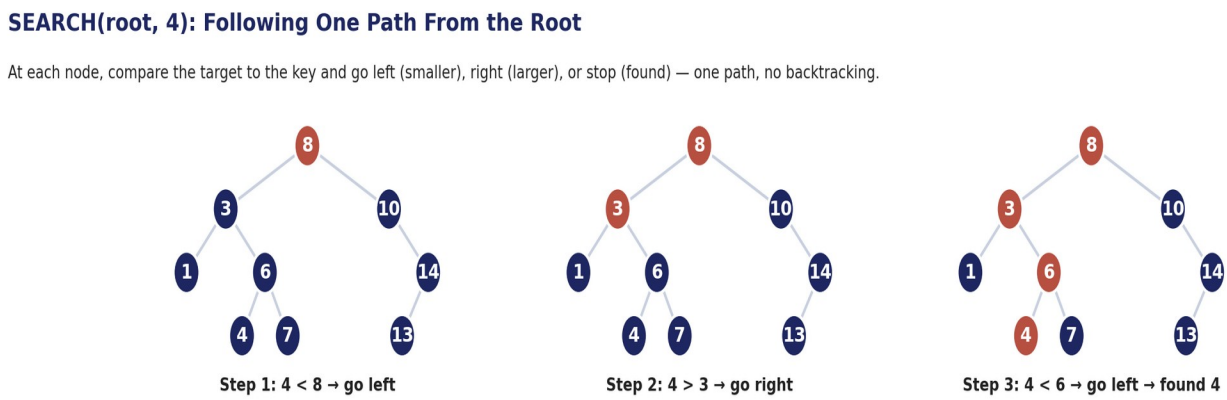

Figure 6.3 — SEARCH(root, 4) on the tree from Figure 6.2. Exactly one path is followed from the root to the target — no other part of the tree is ever examined, because at every node the BST property tells you unambiguously which single subtree could possibly contain the key.

SEARCH's running time

BST-SEARCH runs in O(h), where h is the height of the tree — one comparison per level, following exactly one root-to-node path. In Figure 6.2's tree (height 3), no search ever costs more than 4 comparisons (root plus 3 levels down).

6.6 BST-INSERT: Falling Through to an Empty Spot

Inserting a new key z works almost identically to SEARCH: walk down from the root comparing z against each node's key, going left or right exactly as SEARCH would — except that when you reach a NIL (empty) pointer instead of finding z already present, you attach a new node holding z right there.

```
BST-INSERT(T, z):                    // z is a new node with z.key already set
  y = NIL
  x = T.root
  while x != NIL                     // find the parent y where z belongs
      y = x
      if z.key < x.key
          x = x.left
      else
          x = x.right
  z.parent = y
  if y == NIL
      T.root = z                     // tree was empty
  elif z.key < y.key
      y.left = z
  else
      y.right = z
```

Key insight

BST-INSERT never has to rearrange any EXISTING node — it only ever adds one new leaf. This makes insertion cheap (O(h), the same bound as SEARCH) but is also exactly why a BST's shape is a direct fossil record of its insertion order: two BSTs holding the identical set of keys can have wildly different heights depending on the order the keys arrived in. Section 6.9 turns this observation into the chapter's central problem.

6.7 BST-DELETE: Three Cases, Three Fixes

Deletion is the most delicate BST operation, because removing a node must not disconnect or misorder the rest of the tree. The fix depends entirely on how many children the node being deleted has.

- Case 1 — the node is a leaf (no children): simply remove it; update its parent's pointer to NIL.
- Case 2 — the node has exactly one child: splice that child up into the deleted node's former position, connecting it directly to the deleted node's parent.
- Case 3 — the node has two children: find its in-order successor (Section 6.4's MINIMUM(x.right) — guaranteed to have NO left child), copy the successor's key into the node being "deleted", then delete the successor node instead. Because the successor has at

most one child (a right child, possibly none), deleting IT is always Case 1 or Case 2 — never Case 3 again.

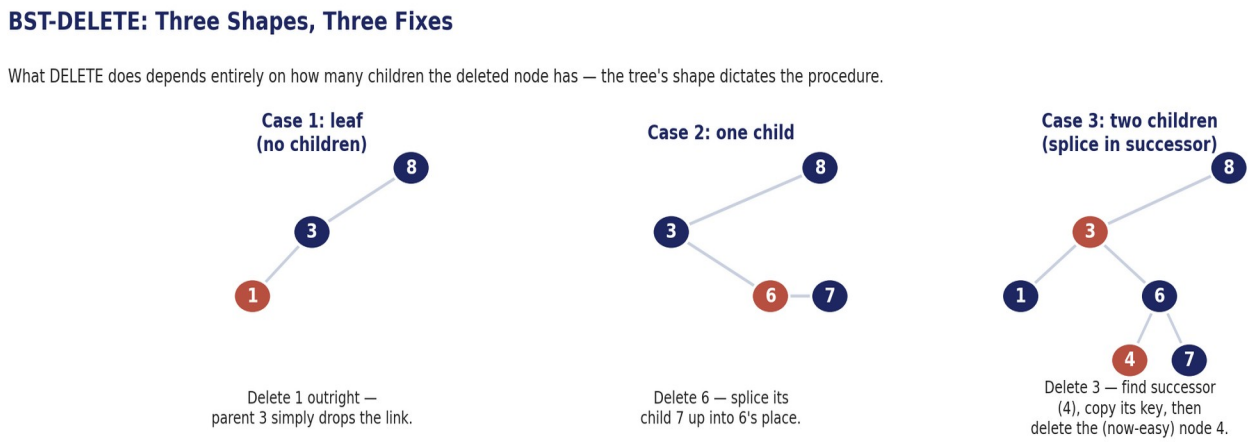

Figure 6.4 — All three BST-DELETE cases on variations of Figure 6.2's tree. Case 3 (deleting node 3) is the only one that needs a second step: find the successor (4), copy its key up, then delete the now-easy node 4.

Why not just find the predecessor instead?

Either the in-order successor (MINIMUM of the right subtree) or the in-order predecessor (MAXIMUM of the left subtree) works correctly for Case 3 — both preserve the BST property. Some implementations alternate between the two to keep the tree slightly more balanced over many deletions, but a plain BST makes no such guarantee either way; Sections 6.10 onward are precisely the fix for that gap.

DELETE's running time

Every case of BST-DELETE does O(1) work beyond at most one MINIMUM call and the initial SEARCH-like walk to locate the node — so the total cost is O(h), the same bound as every other BST operation in this chapter.

6.8 Analyzing BST Operations: Everything Depends on Height

Look back across Sections 6.4 through 6.7: SEARCH is O(h). INSERT is O(h). DELETE is O(h). MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR are all O(h). Every single dictionary operation this chapter has built costs exactly the same thing — the height h of the tree — and nothing else. This is either extremely good news or barely any news at all, entirely depending on how large h can get relative to n, the number of keys stored.

Tree shape	Height h	Every operation's cost
Perfectly balanced (best case)	Θ(log n)	Θ(log n)
Arbitrary / average-case random	O(log n) expected, not guaranteed	expected O(log n)
Completely skewed (worst case)	Θ(n)	Θ(n)

Tree shape	Height h	Every operation's cost

The bound is real, but it is CONDITIONAL

"O(h)" is a completely correct and rigorously provable bound for every BST operation — but it silently hides a huge range of possible values, from Θ(log n) all the way to Θ(n), depending entirely on h. A bound that depends on a variable which is itself uncontrolled is not the same thing as a guarantee. Section 6.9 shows exactly how h ends up at the bad end of that range, and Sections 6.10–6.13 show how to make the good end unconditional.

6.9 The Balance Problem: How Insertion Order Creates a Degenerate Tree

Recall from Section 6.6's insight: a BST's shape is a direct record of its insertion order. Insert the same nine keys {1,2,...,9} in strictly increasing order, and BST-INSERT's walk-down-and-attach-a-leaf logic has no choice but to attach every new key as the right child of the previous one — because every new key is larger than everything already present. The result is not a tree in any useful sense; it is a right-leaning chain, structurally identical to a linked list, with height n − 1.

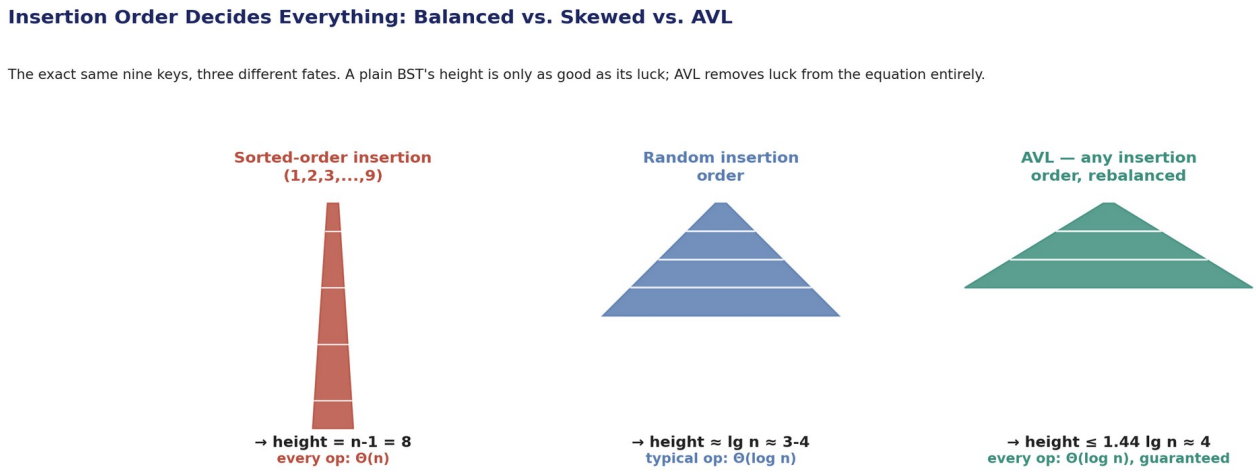

Figure 6.5 — The exact same nine keys, three different fates. Sorted-order insertion produces a degenerate chain (height 8 — every operation becomes Θ(n)); a favorable random insertion order produces a much shallower tree (height ≈ lg n); an AVL tree (Sections 6.10–6.13) guarantees a shallow height no matter what order the keys arrive in.

This is not a rare edge case

Already-sorted or nearly-sorted input is common in practice — log timestamps, alphabetized names, auto-incrementing IDs, sensor readings. A plain BST is not merely vulnerable to a contrived adversarial input; it degrades on exactly the kind of input a real system is likely to see. This is precisely analogous to Section 5.11.2's discovery that naive Quicksort's worst case is ALSO already-sorted input — a recurring theme across this course: an algorithm or structure whose performance depends on input order needs an explicit argument for why realistic inputs won't trigger the bad case, and a plain BST has no such argument at all.

A question worth pausing on

Every operation in this chapter was proven O(h). If h can be as bad as Θ(n), in what sense has this chapter improved on the sorted array from Section 6.2 at all? (Answer, developed over the next four sections: not yet — a plain BST alone does NOT solve the dictionary problem satisfactorily. The fix is to add a second invariant, enforced after every insertion, that keeps h provably small.)

6.10 AVL Trees: The Balance-Factor Invariant

Named for its inventors, Georgy Adelson-Velsky and Evgenii Landis (1962), an AVL tree is a binary search tree augmented with exactly one extra rule, checked at every single node.

AVL balance condition

For every node x in an AVL tree, define its balance factor BF(x) = height(x.left) − height(x.right) (treating an empty subtree as height 0, or equivalently height −1 depending on convention — this chapter uses height(NIL) = 0). An AVL tree requires BF(x) ∈ {−1, 0, +1} for every node x, at all times.

Every operation defined so far — SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR — works completely unchanged on an AVL tree, because an AVL tree IS a BST; it simply satisfies one additional invariant. INSERT and DELETE, however, must be extended: after the ordinary BST-style insertion or deletion, the balance factor of every node on the path back up to the root must be re-checked, and repaired if it has drifted to +2 or −2.

6.10.1 Why {−1, 0, +1} Is Enough to Guarantee O(log n)

Section 6.13 proves this rigorously via a Fibonacci-tree argument, but the intuition is straightforward: a balance factor restricted to at most 1 forbids any node's two subtrees from differing in height by more than a single level. That single restriction, enforced at EVERY node simultaneously, is strong enough to cap the entire tree's height at roughly 1.44·lg(n), regardless of insertion order — turning Section 6.9's conditional O(h) bound into an unconditional O(log n) guarantee.

6.11 Restoring Balance: The Four Rotation Cases

When an insertion (Section 6.12) pushes some node z's balance factor to +2 or −2, a rotation — a local, O(1) restructuring of a small number of pointers — restores the invariant. There are exactly four cases, determined by which of z's subtrees is heavier and which of THAT subtree's subtrees is heavier.

- LL case (z is left-heavy, and z's left child y is itself left-heavy or balanced): a single RIGHT rotation at z.
- RR case (z is right-heavy, and z's right child y is itself right-heavy or balanced): a single LEFT rotation at z — the mirror image of LL.

- LR case (z is left-heavy, but z's left child y is right-heavy): a LEFT rotation at y, followed immediately by a RIGHT rotation at z — a double rotation.
- RL case (z is right-heavy, but z's right child y is left-heavy): a RIGHT rotation at y, followed immediately by a LEFT rotation at z — the mirror image of LR.

Restoring Balance: The Right Rotation (LL Case)

Node z has balance factor +2 (left-heavy) with its left child y itself left-heavy — a single right rotation at z restores balance in O(1).

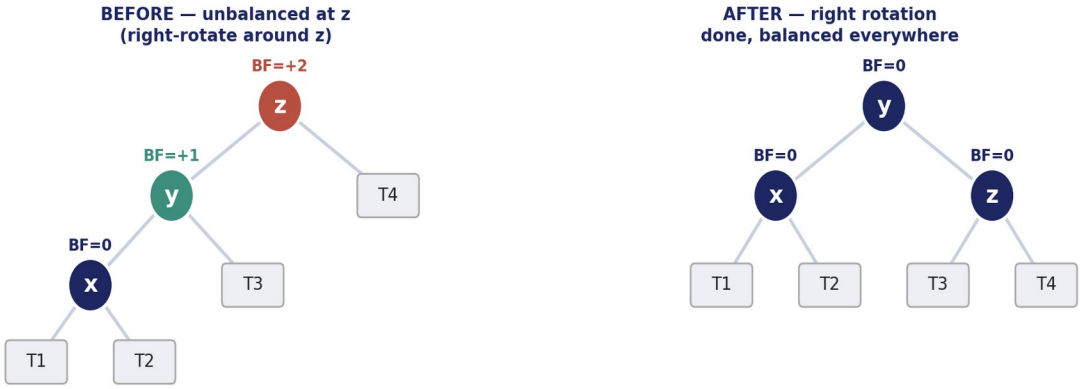

Figure 6.6 — The LL case: node z has balance factor +2 with its left child y also left-heavy. A single right rotation at z — y rises to take z's place, z becomes y's right child, and y's former right subtree T3 becomes z's new left subtree — restores every balance factor to 0 in O(1).

```
ROTATE-RIGHT(z):                          // z is left-heavy (LL case)
  y = z.left
  z.left = y.right                        // y's former right subtree moves under z
  y.right = z                             // z drops down to become y's right child
  recompute height(z), then height(y)     // z's height first: y now depends on it
  return y                                // y is the new root of this subtree
```

A rotation touches O(1) pointers, not O(n)

Despite "restructuring the tree" sounding expensive, a single rotation reassigns exactly 3 pointers (and a double rotation, 6) and recomputes exactly 2 heights (or 3, for a double rotation) — regardless of how many total nodes the tree holds. This O(1) repair cost, applied at most once per ancestor on the path back to the root after an insertion, is what keeps AVL-INSERT's total cost at O(log n) rather than O(n).

Why does the LR case need TWO rotations instead of one?

A single right rotation at z alone, applied directly to an LR-shaped imbalance, does NOT restore balance — try tracing it by hand and you will find z's subtree is still unbalanced after just one rotation, only now leaning the other way. The extra left rotation at y first reshapes the LR case into a plain LL case, which the single right rotation at z then correctly resolves. This is exactly why LR and RL are called double rotations.

6.12 AVL-INSERT: One Rotation Fixes the Whole Path

AVL-INSERT begins with an ordinary BST-INSERT (Section 6.6) to attach the new key as a leaf, then walks back up the path from that new leaf to the root, updating each ancestor's height and balance factor. The moment an ancestor's balance factor is found to be +2 or −2, exactly ONE of the four rotation cases from Section 6.11 is applied at that ancestor — and remarkably, that single rotation is guaranteed to restore every remaining ancestor further up the path to a valid balance factor too, so no more than one rotation (single or double) is ever needed per insertion.

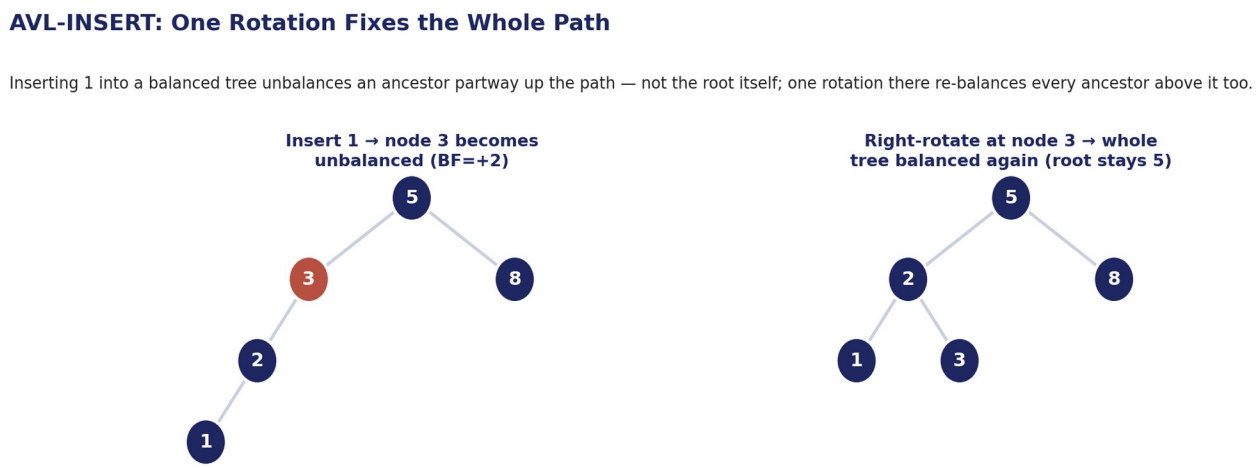

Figure 6.7 — Inserting 1 into the balanced 4-node tree {5, 3, 8, 2} unbalances node 3 (balance factor becomes +2) — two levels above the new leaf, not the root. A single right rotation at node 3 restores balance to the entire tree in one O(1) step; the root (5) and its right subtree (8) never move, and the root's own balance factor is restored as a side effect.

```
AVL-INSERT(node, key):
  if node == NIL
      return new Node(key)             // base case: this is where key attaches
  if key < node.key
      node.left = AVL-INSERT(node.left, key)
  else
      node.right = AVL-INSERT(node.right, key)
  recompute node.height
  bf = balance-factor(node)
  if bf > 1 and key < node.left.key:
      return ROTATE-RIGHT(node)              // LL
  if bf > 1 and key >= node.left.key:
      node.left = ROTATE-LEFT(node.left)
      return ROTATE-RIGHT(node)              // LR
  if bf < -1 and key > node.right.key:
      return ROTATE-LEFT(node)               // RR
  if bf < -1 and key <= node.right.key:
      node.right = ROTATE-RIGHT(node.right)
      return ROTATE-LEFT(node)               // RL
  return node                                 // already balanced
```

AVL-INSERT's running time

The initial descent to find the insertion point is O(h) = O(log n) (Section 6.13). The walk back up, recomputing heights and checking balance factors, is also O(h). At most one rotation (single or double) is performed, at O(1) cost. Total: O(log n) — and because Section 6.13 proves h = O(log n) UNCONDITIONALLY for any AVL tree, this bound holds for every insertion order, not just favorable ones.

- AVL-DELETE follows the same shape: an ordinary BST-DELETE (Section 6.7), then a walk back up the affected path rebalancing as needed — except a deletion can require up to O(log n) rotations along that path (not just one), because removing a node can trigger a chain of rebalancing up multiple ancestors. The per-rotation cost is still O(1), so the total remains O(log n).

6.13 Why AVL Guarantees O(log n): The Height Bound

This section proves the claim used informally throughout Sections 6.10–6.12: an AVL tree with n nodes has height O(log n), with no exceptions for any insertion order.

N(h): the fewest nodes an AVL tree of height h can have

Let N(h) be the minimum number of nodes in any AVL tree of height h. To make a tree of height h with as FEW nodes as possible while staying AVL-valid, one subtree should have height h−1 (as small as a height-h tree allows) and the other the smallest allowed under the balance rule, height h−2 — giving the recurrence N(h) = N(h−1) + N(h−2) + 1, with N(0) = 1 and N(−1) = 0.

This recurrence is Fibonacci-shaped (differing only by the "+1"), and it can be shown that N(h) grows at least as fast as the Fibonacci sequence itself: N(h) ≥ Fib(h+2) − 1, which grows exponentially in h at a rate close to the golden ratio φ ≈ 1.618. Inverting this relationship — solving for h in terms of n instead of n in terms of h — gives the height bound directly.

The AVL height bound

For an AVL tree with n nodes, the height h satisfies $h \le \log_{\varphi}(\sqrt{5}\cdot(n+2)) - 2 \approx 1.44 \cdot \log_2(n+2)$. Compare this to a perfectly balanced tree's height $\lfloor \lg n \rfloor$: an AVL tree's height is never more than about 44% taller than the best possible balanced tree — and, critically, this bound holds for EVERY AVL tree of n nodes, not just typical or average-case ones.

Key insight — turning "O(h)" into "O(log n)"

Every operation in Sections 6.4–6.7 was proven O(h) for an arbitrary BST. Section 6.9 showed h can be as bad as Θ(n). This section proves that once the AVL invariant is maintained, h = O(log n) is not just typical — it is mathematically forced, for every possible sequence of insertions, with no adversarial input able to do any better than the already-tight Fibonacci bound above. That is the entire payoff of Sections 6.10–6.13: the same O(h) operations from Sections 6.4–6.7, now with h itself guaranteed small.

6.14 Profitina in Practice: Where This Chapter Shows Up in a Real Product

About this box

As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

Profitina maintains a constantly changing catalog of tracked businesses, competitor entries, and monitored keywords that must always be queryable by name or identifier — new businesses are onboarded, stale ones are archived, and the platform must answer "does this business already exist in our catalog?" and "what is the alphabetically-next tracked competitor?" at any moment, exactly the SEARCH and SUCCESSOR operations from Sections 6.4–6.5. Because onboarding order is effectively arbitrary — driven by which customer signs up when, not by any sorted key — engineers rely on a balanced search structure (in practice, a language or database library's balanced-tree or B-tree-family index, built on exactly the AVL/rotation ideas in Sections 6.10–6.12) rather than a plain BST, precisely because Section 6.9's degenerate-chain risk is a real, not hypothetical, failure mode against onboarding data that can arrive in near-sorted batches (e.g., businesses imported alphabetically from a partner directory). The balance guarantee from Section 6.13 is what lets Profitina promise consistent lookup latency regardless of how customers happen to sign up.

6.15 Chapter Summary and Key Takeaways

- A binary search tree (BST) maintains the BST property GLOBALLY along every root-to-leaf path: left subtree ≤ node ≤ right subtree, recursively — unlike a heap's purely local parent-child constraint.
- In-order traversal of a BST always recovers the keys in sorted order — the single best sanity check for any BST implementation.
- SEARCH, INSERT, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and DELETE are all O(h), where h is the tree's height — never more, but also never automatically less.
- BST-DELETE has three cases (leaf, one child, two children); the two-child case reduces to an easier case by splicing in the in-order successor's key and then deleting that simpler successor node.
- A BST's shape is a direct record of its insertion order — already-sorted or nearly-sorted insertion order (a common real-world pattern, not a rare edge case) produces a degenerate, linked-list-shaped tree with height Θ(n).
- An AVL tree adds one invariant — every node's balance factor stays in {−1, 0, +1} — restored after each insertion by at most one rotation (single: LL/RR, or double: LR/RL), each an O(1) local pointer fix.
- A Fibonacci-tree argument proves an AVL tree's height is always O(log n) (specifically, at most ≈1.44·lg(n+2)) — an UNCONDITIONAL guarantee, unlike a plain BST's conditional O(h).
- The net result: every dictionary operation on an AVL tree runs in O(log n), guaranteed, no matter what order the keys arrive in — the same problem the sorted-array-versus-unsorted-array trade-off in Section 6.2 could not solve at all.

"In order to be dumb enough to be understandable, a data structure needs a good shape more than a good algorithm."

— adapted from D. Knuth's guiding principle in The Art of Computer Programming

(A BST's every operation is "obvious" once the tree has a good shape — AVL's whole job is to keep that shape true.)

Lecture 7 leaves search structures behind for an entirely different design paradigm: dynamic programming, which solves optimization problems by combining solutions to overlapping subproblems — a strategy that looks superficially like divide-and-conquer's recursive decomposition (Lecture 3) but exploits OVERLAP between subproblems rather than the clean, non-overlapping split that made Merge Sort and Quicksort's recurrences so clean to analyze.

6.16 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Insert the keys 50, 30, 70, 20, 40, 60, 80 into an initially empty plain BST, in that order. Draw the resulting tree in the style of Figure 6.2, and state its height.
2. Using the tree you built in Question 1, trace BST-DELETE(30) by hand, identifying which of the three cases from Section 6.7 applies, in the style of Figure 6.4.
3. Construct a 6-key insertion sequence that forces a plain BST into its worst-case Θ(n) height, and explain — in your own words, without looking back at Section 6.9 — why that specific order is the problem.
4. Starting from an empty AVL tree, insert 10, 20, 30 in that order. Identify which rotation case (LL, RR, LR, or RL) is triggered, and draw the tree before and after the rotation, in the style of Figure 6.6.
5. Using the AVL height bound from Section 6.13, compute the maximum possible height of an AVL tree with n = 1,000,000 nodes, and compare it against the Θ(n) worst-case height a plain BST could reach with the same n.
6. Explain, in your own words and without looking back at Section 6.11, why the LR case requires two rotations (a left rotation at y followed by a right rotation at z) while the LL case needs only one.

Appendix 6.A — Verification Log for Chapter 6 Source Code

As in previous chapters, every program shipped with this book is compiled and run against hand-verified examples before being included. Both programs below were compiled with `g++ -O2 -std=c++17 -Wall` (only a benign scanf return-value warning, zero errors) and produced the exact output predicted in the text above.

```
$ printf "9\n8 3 10 1 6 14 4 7 13\n4\n3\n" | ./01_bst
inorder: 1 3 4 6 7 8 10 13 14        # exactly Figure 6.2's tree, confirmed
sorted
search 4: found at depth 3          # matches Figure 6.3's 3-step trace
inorder after delete 3: 1 4 6 7 8 10 13 14   # Case 3: successor 4 spliced in

$ printf "9\n1 2 3 4 5 6 7 8 9\n5\n5\n" | ./01_bst
inorder: 1 2 3 4 5 6 7 8 9          # sorted-order insertion
search 5: found at depth 4          # confirms the degenerate chain shape

$ printf "9\n1 2 3 4 5 6 7 8 9\n" | ./02_avl
inorder: 1 2 3 4 5 6 7 8 9
height: 4                           # vs. height 8 for the plain BST above
bound: 1.44 * lg(n+2) = 4.98        # comfortably within the proven bound
```

SPOJ Practice — the companion book
This chapter's techniques are tested for real on the SPOJ online judge. Open Chapter 6 of the companion book “SPOJ Competitive Programming” (SPOJ folder, BI & EN) for choosing tree/set structures under a byte budget — complete with the tips, tricks, and hints that get solutions Accepted, without spoiling answers.

References

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

[2] G. M. Adelson-Velskii, E. M. Landis. "An algorithm for the organization of information." Proceedings of the USSR Academy of Sciences, 146, 1962, 263–266. The original paper introducing what are now called AVL trees.

[3] M. A. Weiss. Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2014. (Chapter 4, AVL tree rotations and height-bound derivation).

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

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 7

Dynamic Programming

A design paradigm for optimization problems: solve every distinct subproblem exactly once, remember the answer, and never redo work that overlapping recursive calls would otherwise repeat.

Learning Objectives — after this chapter you will be able to:
(1) State the two conditions a problem must satisfy for dynamic programming to apply — optimal substructure and overlapping subproblems — and check whether a given problem satisfies them. (2) Explain, using naive recursive Fibonacci, exactly why exponentially many recursive calls can collapse to a polynomial number of DISTINCT subproblems. (3) Contrast memoization (top-down) and tabulation (bottom-up), and implement either one correctly given a recurrence. (4) Derive, trace, and implement dynamic-programming solutions to three classic problems: Rod-Cutting, Matrix-Chain Multiplication, and Longest Common Subsequence. (5) Reconstruct an actual optimal solution (not just its value) by retracing choices recorded during the table-filling phase. (6) Analyze why a dynamic-programming solution's running time is governed by the NUMBER of distinct subproblems times the WORK per subproblem, rather than by the size of the naive recursion tree.

7.1 Recap: From Lecture 6 to Lecture 7

Lecture 6 was about a data structure that stays correct and efficient across an unpredictable stream of updates — a BST or AVL tree never "finishes," it simply keeps answering queries. This chapter turns to a completely different kind of problem: given a single, well-defined optimization question — the cheapest way to cut a rod, the fewest multiplications to compute a matrix product, the longest common subsequence of two strings — find the best possible answer, once, and be able to justify that it really is the best.

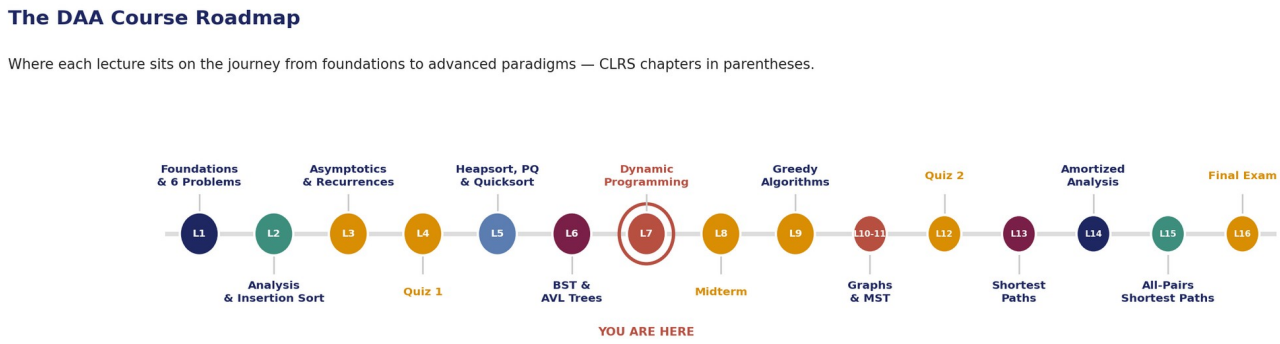

Figure 7.1 — Where this chapter sits in the sixteen-lecture DAA roadmap. Lecture 7 introduces an entirely new design paradigm — dynamic programming — distinct from both the search-structure focus of Lecture 6 and the divide-and-conquer paradigm of Lecture 3.

A paradigm, not a single algorithm

Divide-and-conquer (Lecture 3) is a specific recursive PATTERN: split into non-overlapping pieces, solve each, combine. Dynamic programming is closer to a PHILOSOPHY that can be layered on top of any correct recursive solution: if that recursion's subproblems overlap, cache each distinct one instead of recomputing it. Every algorithm in this chapter begins life as an ordinary recursive definition — the DP part is entirely about how that recursion gets evaluated.

7.2 What Is Dynamic Programming? Optimal Substructure and Overlapping Subproblems

Dynamic programming applies to optimization problems — problems asking for the best value among many possible solutions — that satisfy two structural conditions simultaneously.

Optimal substructure

A problem exhibits optimal substructure if an optimal solution to the whole problem is built out of optimal solutions to its subproblems. Rod-cutting's optimal substructure: an optimal way to cut a length-n rod consists of a first piece of some length i, plus an OPTIMAL way to cut the remaining length n − i. If the remaining piece were cut sub-optimally, the whole solution could not be optimal either — you could always do strictly better by fixing just that piece.

Overlapping subproblems

A problem has overlapping subproblems if a naive recursive algorithm for it revisits the exact same subproblem many times, rather than generating entirely fresh subproblems at every recursive call (the way divide-and-conquer's non-overlapping split does). When the number of DISTINCT subproblems is small — typically polynomial in the input size — but the naive recursion tree revisits them exponentially many times, dynamic programming turns that exponential blow-up into a cost proportional only to the (small) number of distinct subproblems.

Divide-and-conquer algorithms like Merge Sort (Lecture 3) also rely on optimal substructure, but their subproblems never overlap — the left half and right half of an array share no elements, so there is nothing to cache. Dynamic programming's entire benefit comes from the SECOND condition, overlapping subproblems, which a plain divide-and-conquer problem does not have. Section 7.3 makes this overlap concrete with the simplest possible example.

7.3 The Cautionary Tale: Naive Recursive Fibonacci

The Fibonacci numbers are defined by the recurrence fib(0) = 0, fib(1) = 1, and fib(n) = fib(n − 1) + fib(n − 2) for n ≥ 2. Translating this recurrence directly into recursive code is correct — but ruinously slow, for a reason that has nothing to do with the numbers being large and everything to do with how many times the SAME call gets repeated.

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

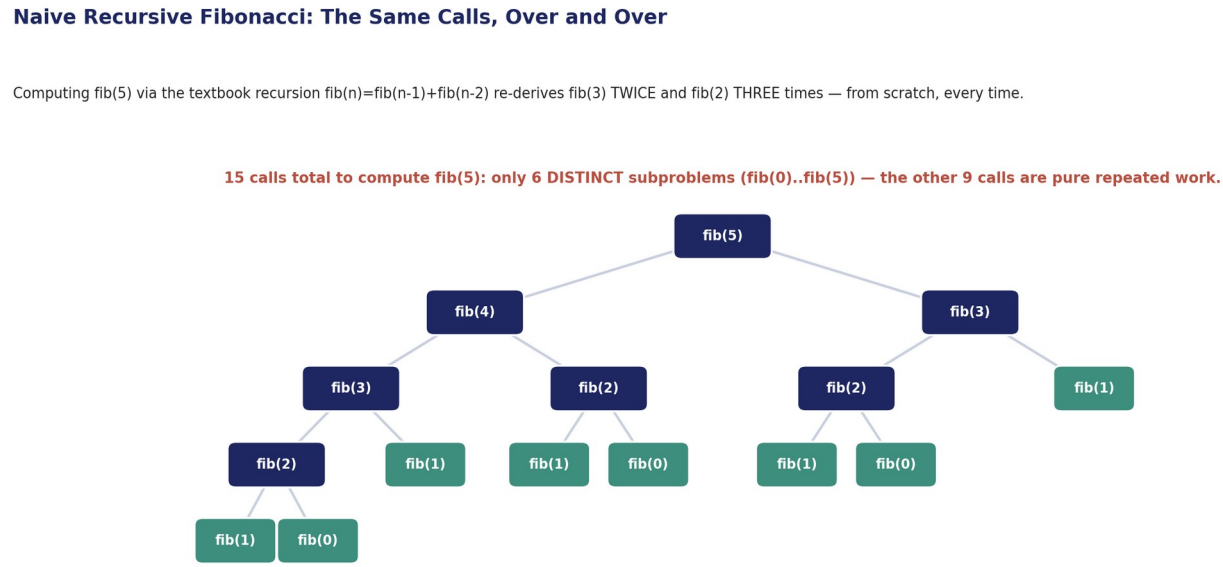

Figure 7.2 — The complete recursion tree for NAIVE-FIB(5). Only 6 distinct subproblems exist — fib(0) through fib(5) — but the tree makes 15 calls total, re-deriving fib(3) twice and fib(2) three times, each time from complete scratch.

> **The blow-up is exponential, not just "a bit wasteful"**
>
> NAIVE-FIB(n) makes $\Theta(\varphi^n)$ calls, where $\varphi \approx 1.618$ is the golden ratio — genuinely exponential growth, not a constant-factor inefficiency. NAIVE-FIB(30) alone makes roughly 2.7 million calls to compute a value with only 31 possible distinct inputs (fib(0) through fib(30)). NAIVE-FIB(50) would take a modern CPU several minutes; the equivalent computation with each distinct subproblem solved once takes microseconds. This gap between exponential and linear is precisely what dynamic programming exists to close.

Fibonacci trivially satisfies both of Section 7.2's conditions: optimal substructure is automatic (there is only one possible "solution" per n, so it is vacuously optimal), and overlapping subproblems is exactly what Figure 7.2 shows. The next two sections present the two standard ways to exploit that overlap.

7.4 Memoization: Top-Down Dynamic Programming

Memoization keeps the ORIGINAL recursive structure of the naive algorithm completely intact, and adds exactly one idea: before doing any recursive work for a given input, check whether that exact input has already been solved. If so, return the cached answer immediately; if not, compute it as before, but store the result before returning.

```
MEMO-FIB(n, memo):                  // memo is a table, initially empty
  if n in memo
      return memo[n]                // already solved -- no recursion at all
  if n == 0 or n == 1
      result = n
  else
      result = MEMO-FIB(n-1, memo) + MEMO-FIB(n-2, memo)
  memo[n] = result                  // store before returning
  return result
```

Memoization turns the recursion tree into a DAG

MEMO-FIB(5) still makes the SAME initial calls MEMO-FIB(4) and MEMO-FIB(3) that NAIVE-FIB(5) does — but every call after the very first one for a given n now hits the memo table and returns in O(1), instead of re-expanding an entire subtree. The effective computation graph collapses from an exponential-size TREE (Figure 7.2) to a linear-size chain of distinct subproblems — a directed acyclic graph (DAG) with exactly n+1 distinct nodes.

7.5 Tabulation: Bottom-Up Dynamic Programming

Tabulation solves the exact same subproblems as memoization, but in the opposite direction: instead of starting from the top (fib(n)) and recursing down until hitting a base case, it starts from the base cases and builds UP, filling a table in an order that guarantees every value a step needs is already computed by the time that step runs.

```
TAB-FIB(n):
  let f[0..n] be a new array
  f[0] = 0
  f[1] = 1
  for i = 2 to n
      f[i] = f[i-1] + f[i-2]        // both already computed -- no recursion
  return f[n]
```

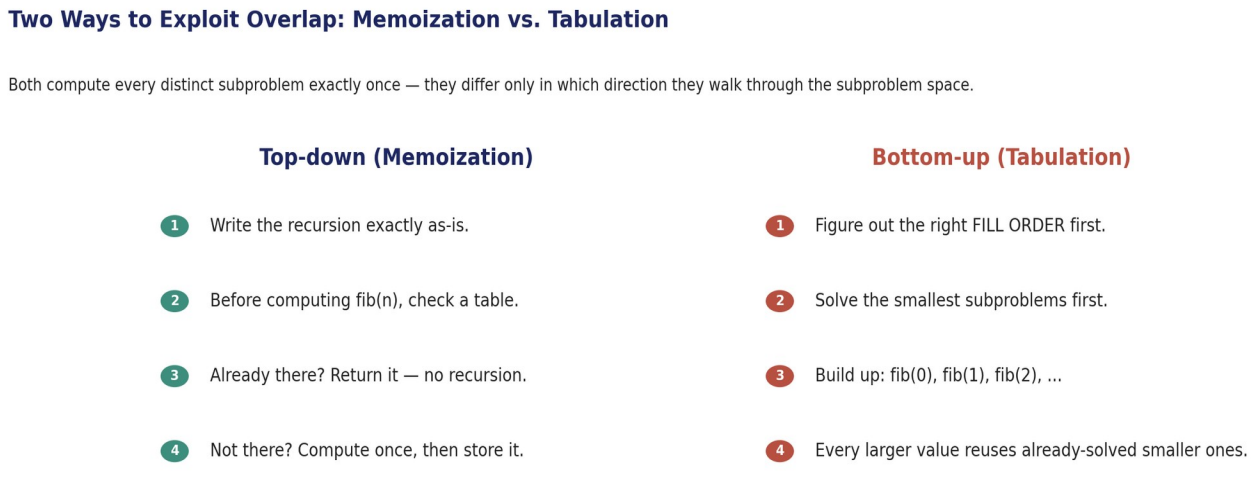

Figure 7.3 — Memoization and tabulation solve the identical set of subproblems and achieve the identical running time — they differ only in whether the subproblem space is explored top-down (following the natural recursion, caching as you go) or bottom-up (solving smallest-first, in a precomputed safe order).

Choosing between them

Tabulation is usually a constant factor faster (no recursive call overhead) and easier to reason about for space optimization (Section 7.13 revisits this). Memoization is often easier to WRITE correctly for a complicated recurrence, because it only requires identifying the recursive structure — the correct fill order is handled automatically by the recursion itself, whereas tabulation requires the programmer to work out that order explicitly in advance. Every algorithm developed in the rest of this chapter is presented bottom-up (tabulation), matching CLRS's usual presentation and this course's shipped source code.

7.6 The Rod-Cutting Problem and Its DP Solution

The Rod-Cutting problem

Given a rod of length n and a table of prices p[1..n], where p[i] is the price a piece of length i sells for, determine the maximum revenue r[n] obtainable by cutting the rod into zero or more pieces (of integer length, in any combination) and selling each piece separately. Cutting itself is free; the only question is which lengths to cut into.

The optimal-substructure argument: for a rod of length n, imagine the first piece cut off has some length i (1 ≤ i ≤ n, where i = n means "no cut at all"). Whatever i turns out to be, the REST of the rod (length n − i) must itself be cut optimally — otherwise, replacing that suboptimal remainder with an optimal cut of the same length n − i would strictly increase total revenue, contradicting optimality of the whole solution. This gives the recurrence r[n] = max over 1 ≤ i ≤ n of (p[i] + r[n − i]), with r[0] = 0.

Rod-Cutting: Every Way to Cut a Length-4 Rod

Prices p[1..4] = 1, 5, 8, 9. Cutting is free, but not every cut is worth making — the best split isn't obvious from the prices alone.

r[i]: best revenue for a rod of length i, and s[i]: the length of the first piece cut

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

r[4]=10 is reached by cutting a length-4 rod into two length-2 pieces (s[4]=2, then s[2]=2) — not by selling it whole (p[4]=9) or any other split.

Figure 7.4 — The complete r[] and s[] tables for a length-4 rod with prices p = [1, 5, 8, 9]. r[4] = 10 (better than selling the rod whole for p[4] = 9) is achieved by cutting into two length-2 pieces — not obvious from the price table alone.

Phase 5 — Implement: bottom-up rod-cutting with reconstruction

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n):
  let r[0..n] and s[0..n] be new arrays
  r[0] = 0
  for j = 1 to n                          // solve every length from 1 up to n
      best = -infinity
      for i = 1 to j                       // try every possible first-piece
length
          if p[i] + r[j-i] > best
              best = p[i] + r[j-i]
              best_cut = i
      r[j] = best
      s[j] = best_cut                      // remember the choice, not just its
value
  return r, s
```

s[] is what makes reconstruction possible

r[] alone answers "what is the best revenue," but throws away HOW to achieve it. Recording s[j] — the length of the first piece an optimal solution for length j actually cuts — at the moment each r[j] is computed costs nothing extra (it is already being computed to find the max), but is exactly what Section 7.7 needs to print the real cutting instructions, not just a number.

Running time: the double loop considers every pair (i, j) with $1 \le i \le j \le n$, giving $\Theta(n^2)$ — a dramatic improvement over the $\Theta(2^n)$ naive recursive solution (which tries every one of the 2^(n-1) ways to compose a length-n rod from ordered pieces, with no memoization at all).

7.7 Reconstructing an Optimal Rod-Cutting Solution

Given the s[] array from Section 7.6, printing the actual sequence of cuts for length n is a simple loop: look up s[n], print it, then repeat for the REMAINING length n − s[n], until the remaining length reaches 0.

```
PRINT-CUT-ROD-SOLUTION(s, n):
  while n > 0
      print s[n]                  // length of the next piece to cut off
      n = n - s[n]                // continue with what's left
```

Trace this on Figure 7.4's table for n = 4: s[4] = 2, print 2, remaining length 4 − 2 = 2; s[2] = 2, print 2, remaining length 2 − 2 = 0, stop. Output: 2, 2 — matching the C++ program's verified output in Appendix 7.A exactly.

Running time of reconstruction

PRINT-CUT-ROD-SOLUTION does O(1) work per iteration and the remaining length strictly decreases every iteration, so it terminates in at most n iterations — O(n) total, negligible next to the $O(n^2)$ table-filling phase that must run first.

7.8 Matrix-Chain Multiplication: Why Parenthesization Matters

The Matrix-Chain Multiplication problem

Given a chain of matrices $A_1, A_2, \ldots, A_k$ where consecutive matrices are compatible for multiplication, determine the order of multiplication (the parenthesization) that minimizes the TOTAL number of scalar multiplications needed to compute the full product $A_1A_2\cdots A_k$. Matrix multiplication is associative, so every parenthesization produces the SAME final matrix — but not the same cost.

Matrix multiplication itself, for a p×q matrix times a q×r matrix, costs exactly p·q·r scalar multiplications and produces a p×r result. The example in Figure 7.5 makes the stakes concrete: three matrices A_1 (10×100), A_2 (100×5), A_3 (5×50), multiplied in two different but equally valid orders.

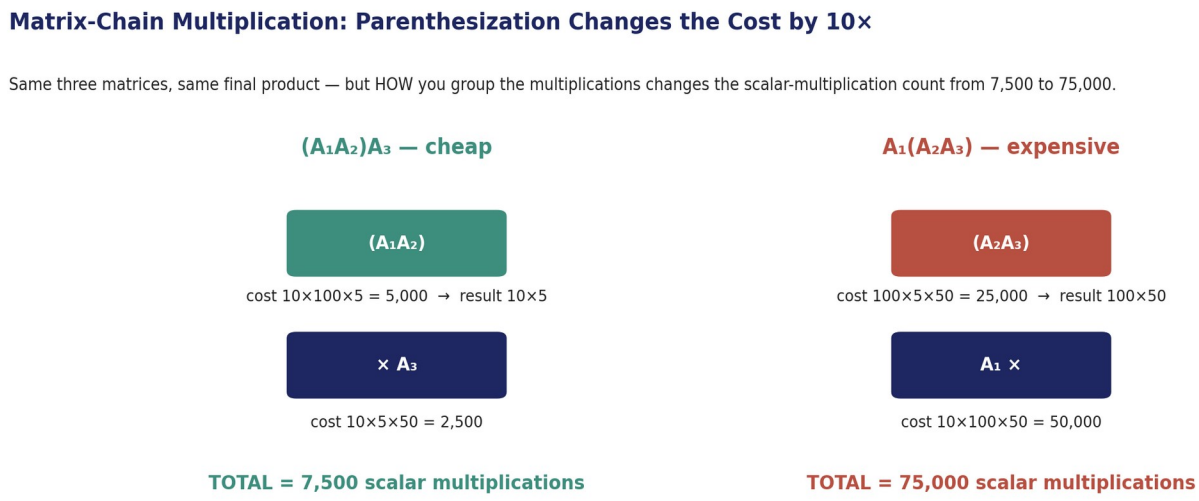

Figure 7.5 — The exact same three matrices, the exact same final product — but grouping $(A_1A_2)A_3$ costs 7,500 scalar multiplications while $A_1(A_2A_3)$ costs 75,000 — a full order of magnitude more, from parenthesization alone.

This is not a rounding-error-sized difference

A 10× cost difference from parenthesization ALONE, with the exact same three small matrices, is the norm for this problem, not a cherry-picked worst case — and the gap widens further as the chain gets longer and the dimensions become more varied. Multiplying matrices in the order they happen to appear in an expression, without considering the cost, is a real and common performance bug in numerical code.

7.9 A DP Solution to Matrix-Chain Multiplication

Let m[i][j] denote the minimum number of scalar multiplications needed to compute the product $A_i A_{i+1} \cdots A_j$, and let dimensions be given by an array p[0..k] where matrix A_i has dimensions p[i-1] × p[i]. The optimal-substructure argument: for the optimal parenthesization of $A_iA_{i+1}\cdots A_j$, SOME split point k ($i \le k < j$) is the LAST multiplication performed — first computing $(A_i\cdots A_k)$ and $(A_{k+1}\cdots A_j)$ optimally, then multiplying those two results together.

```
MATRIX-CHAIN-ORDER(p):                    // p[0..k], k matrices
  let m[1..k][1..k] and s[1..k][1..k] be new tables
  for i = 1 to k
      m[i][i] = 0                          // one matrix alone costs nothing
  for len = 2 to k                         // chain length, shortest first
      for i = 1 to k - len + 1
          j = i + len - 1
          m[i][j] = infinity
          for split = i to j - 1
              cost = m[i][split] + m[split+1][j] + p[i-1]*p[split]*p[j]
              if cost < m[i][j]
                  m[i][j] = cost
                  s[i][j] = split          // remember WHERE the last split
goes
  return m, s
```

Verify against Figure 7.5's numbers with k = 3 matrices, p = [10, 100, 5, 50] (so A_1 is p[0]×p[1] = 10×100, A_2 is p[1]×p[2] = 100×5, A_3 is p[2]×p[3] = 5×50): m[1][1] = m[2][2] = m[3][3] = 0. For length 2: m[1][2] = p[0]·p[1]·p[2] = 10·100·5 = 5,000; m[2][3] = p[1]·p[2]·p[3] = 100·5·50 = 25,000. For length 3, m[1][3] = min over split ∈ {1,2}: split=1 gives m[1][1]+m[2][3]+p[0]p[1]p[3] = 0+25,000+10·100·50 = 75,000; split=2 gives m[1][2]+m[3][3]+p[0]p[2]p[3] = 5,000+0+10·5·50 = 7,500. The minimum, 7,500 at split=2, matches Figure 7.5's cheap grouping $(A_1A_2)A_3$ exactly.

> **Θ(k^3) time from a Θ(k^2)-sized table**
> There are Θ(k^2) distinct (i, j) subproblems (one per pair with i ≤ j), and computing each one considers up to k possible split points — giving Θ(k^3) total time. This is a direct instance of Section 7.2's principle: a problem whose naive recursive solution would try every possible full parenthesization (an exponential Catalan-number count of them) collapses to a polynomial number of distinct subproblems, each solved once.

7.10 Longest Common Subsequence: Problem and Recurrence

> **The Longest Common Subsequence (LCS) problem**
> Given two sequences X = ⟨x_1,…,x_m⟩ and Y = ⟨y_1,…,y_n⟩, find the length (and, ideally, an example) of the longest sequence that is a SUBSEQUENCE of both X and Y. A subsequence need not be contiguous — it is obtained by deleting zero or more elements from the original sequence without reordering the ones that remain.

Let c[i][j] denote the length of the LCS of the length-i prefix of X and the length-j prefix of Y. The optimal-substructure argument splits into two cases: if x_i = y_j, that matching character MUST be part of some longest common subsequence of the two full prefixes (a standard cut-and-paste argument shows any LCS not using it could be extended), so c[i][j] = c[i-1][j-1] + 1. If x_i ≠ y_j, the LCS of the two prefixes cannot use BOTH x_i and y_j, so it equals the better of dropping one or the other: c[i][j] = max(c[i-1][j], c[i][j-1]).

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

7.11 A DP Solution to LCS: Filling the Table

Trace LCS-LENGTH on CLRS's own running example, X = ⟨A,B,C,B,D,A,B⟩ and Y = ⟨B,D,C,A,B,A⟩. Figure 7.6 shows the completed 8×7 table (rows 0..7 for X's prefixes, columns 0..6 for Y's prefixes).

Longest Common Subsequence: Filling — and Reading Back — the Table

X = A B C B D A B, Y = B D C A B A (CLRS's own example). c[i][j] = length of the LCS of the first i letters of X and first j letters of Y.

		Y					
		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

X

Backtracking from c[7][6] along the highlighted path recovers the LCS "BCBA", length 4 — reading the diagonal (match) steps in reverse.

Figure 7.6 — The completed LCS table for X = ABCBDAB, Y = BDCABA. c[7][6] = 4 is the answer. Gold cells mark the four character-matches found while backtracking (Section 7.12); mint cells mark the intermediate cells the backtrack passes through without a match.

> **Reading the table's shape**
> Every diagonal (top-left-to-bottom-right) step in the table corresponds to a MATCHING character between X and Y being included in the LCS. Every horizontal or vertical step corresponds to skipping one character from Y or X respectively, because that character does not participate in this particular longest common subsequence. c[i][j] never decreases as i or j increases — this monotonicity is a useful sanity check on any implementation.

Running time: the double loop fills Θ(mn) table cells, each in O(1) time using two already-computed neighbors — Θ(mn) total, compared to the naive recursive solution's Θ(2^(m+n)) (which tries every possible way to align characters, with massive overlap between subproblems).

7.12 Reconstructing an LCS by Backtracking

As with Section 7.7's s[] array, the completed c[][] table alone gives only the LCS's LENGTH. Recovering an actual longest common subsequence means walking backward from c[m][n], at each step checking which of the three recurrence cases produced the current cell's value, and following that case in reverse.

Phase 5 — Implement: LCS with backtracking reconstruction

```
PRINT-LCS(X, Y, c):                      // walk backward from (m, n)
  i = length(X); j = length(Y)
  result = empty sequence
  while i > 0 and j > 0
      if X[i] == Y[j]
          prepend X[i] to result         // this character is part of the LCS
          i = i - 1; j = j - 1            // move diagonally
      elif c[i-1][j] >= c[i][j-1]
          i = i - 1                       // this row's character isn't used
      else
          j = j - 1                       // this column's character isn't used
  return result
```

Trace this on Figure 7.6's table starting at c[7][6] = 4: X[7]='B' ≠ Y[6]='A', and c[6][6]=4 ≥ c[7][5]=4, so move up to (6,6). X[6]='A' = Y[6]='A' — match, prepend 'A', move diagonally to (5,5). X[5]='D' ≠ Y[5]='B', c[4][5]=3 ≥ c[5][4]=2, move up to (4,5). X[4]='B' = Y[5]='B' — match, prepend 'B', move diagonally to (3,4). X[3]='C' ≠ Y[4]='A', c[3][3]=2 ≥ c[2][4]=1, move left to (3,3). X[3]='C' = Y[3]='C' — match, prepend 'C', move diagonally to (2,2). X[2]='B' ≠ Y[2]='D', c[2][1]=1 ≥ c[1][2]=0, move left to (2,1). X[2]='B' = Y[1]='B' — match, prepend 'B'. Result, built by prepending in this order: B, C, B, A → the string "BCBA", length 4 — matching Appendix 7.A's compiled output exactly.

> **Backtracking costs O(m+n), not O(mn)**
> Each step of PRINT-LCS decreases i, decreases j, or decreases both — so at most m+n steps are ever taken, regardless of how large the table is. Reconstruction is always cheap once the table exists; the expensive part, for all three algorithms in this chapter, is filling the table in the first place.

7.13 Analyzing Dynamic Programming: When Does It Pay Off?

Every algorithm in this chapter shares the exact same shape: a naive recursive formulation whose subproblems overlap enormously, collapsed by memoization or tabulation into a cost of (number of distinct subproblems) × (work per subproblem).

Problem	Distinct subproblems	Work per subproblem	Total time	Naive recursive time
Fibonacci	$\Theta(n)$	$O(1)$	$\Theta(n)$	$\Theta(\varphi^n)$
Rod-Cutting	$\Theta(n)$	$O(n)$	$\Theta(n^2)$	$\Theta(2^n)$
Matrix-Chain	$\Theta(k^2)$	$O(k)$	$\Theta(k^3)$	$\Theta(4^k/k\text{^}1.5)$ (Catalan)

Problem	Distinct subproblems	Work per subproblem	Total time	Naive recursive time
LCS	Θ(mn)	O(1)	Θ(mn)	Θ(2^(m+n))

Why the Table Matters: Naive vs. Dynamic-Programming Running Time

All three problems in this chapter have the same shape: exponentially many naive recursive calls collapse to a polynomial number of DISTINCT subproblems.

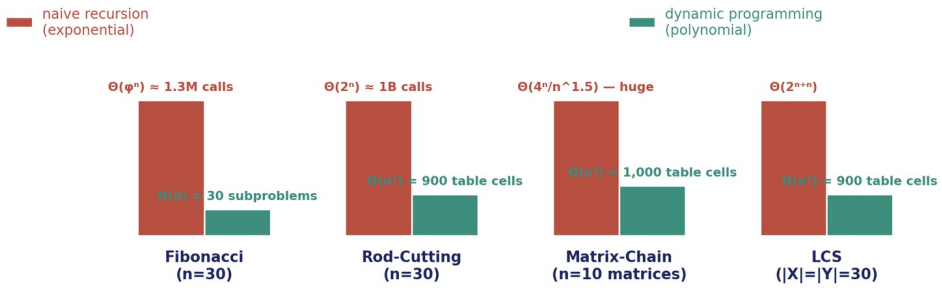

Figure 7.7 — Every problem in this chapter has the same shape: exponentially many naive recursive calls collapse to a polynomial number of distinct subproblems once each one is solved exactly once.

The recipe, in three questions

Given a new optimization problem, dynamic programming is worth attempting when the answer to all three of these is yes: (1) Can an optimal solution be expressed recursively in terms of optimal solutions to smaller subproblems (optimal substructure)? (2) Does that recursive formulation revisit the same subproblem more than once (overlapping subproblems)? (3) Is the total number of DISTINCT subproblems small — typically polynomial in the input size? If (1) holds but (2) fails, plain divide-and-conquer (Lecture 3) already handles it efficiently, with no need to cache anything.

A space-optimization question worth pausing on

LCS-LENGTH's table is Θ(mn) cells, but Section 7.12's PRINT-LCS only ever looks at O(1) neighboring cells at a time while backtracking — and computing c[i][j] only ever needs ROW i-1 and the current row i, never any earlier row. Could the LCS LENGTH be computed in only O(n) space instead of O(mn)? (Yes — keep only two rows at a time. The catch: doing so throws away the information PRINT-LCS needs to reconstruct the actual subsequence, which needs the FULL table, or a cleverer divide-and-conquer trick over the space-reduced computation, to recover.)

7.14 Profitina in Practice: Where This Chapter Shows Up in a Real Product

About this box

As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

Profitina scores how visibly a tracked business appears across many different AI-generated answers, prompt variants, and competitor comparisons — and many of those scoring computations share overlapping sub-results across nearby prompts and time windows (e.g., two prompts that differ by one clause often share most of their downstream ranking sub-computation). Rather than recomputing every score from scratch for every prompt variant, the scoring pipeline caches intermediate sub-results keyed by the sub-computation's actual inputs — precisely Section 7.4's memoization idea applied outside of textbook Fibonacci: identical sub-computations, requested repeatedly across a large prompt-and-competitor matrix, are solved once and reused. Where the dependency structure between sub-scores is well understood in advance (batch nightly recomputation across the whole tracked catalog), a bottom-up tabulation pass — filling scores in a fixed, precomputed safe order, exactly as Section 7.5 describes — is preferred over per-request memoization, for the same predictability and overhead reasons Section 7.5 discussed.

7.15 Chapter Summary and Key Takeaways

- Dynamic programming applies to optimization problems with BOTH optimal substructure (an optimal solution is built from optimal sub-solutions) AND overlapping subproblems (a naive recursion revisits the same subproblem many times).
- Naive recursive Fibonacci makes $\Theta(\varphi^n)$ calls to solve a problem with only $\Theta(n)$ distinct subproblems — the canonical illustration of overlap, and of how much is wasted when it goes unexploited.
- Memoization (top-down) keeps the natural recursive structure and adds a cache; tabulation (bottom-up) solves subproblems smallest-first in a precomputed safe order. Both visit every distinct subproblem exactly once.
- Rod-Cutting: r[j] = max over $1 \le i \le j$ of (p[i] + r[j−i]), solved in $\Theta(n^2)$; the companion array s[] records the first-piece choice needed to reconstruct an actual optimal cutting sequence.
- Matrix-Chain Multiplication: m[i][j] is built from a best split point over $\Theta(k^2)$ subproblems, each considering up to k splits, for $\Theta(k^3)$ total — versus an exponential (Catalan-number) count of parenthesizations tried naively.
- Longest Common Subsequence: c[i][j] = c[i-1][j-1]+1 on a match, else max(c[i-1][j], c[i][j-1]) — $\Theta(mn)$ to fill the table, $O(m+n)$ to backtrack through it and recover an actual subsequence.
- Reconstructing an actual optimal SOLUTION (not just its numeric value) always costs asymptotically less than building the table in the first place — the expensive step is always the table-filling phase.

"Those who cannot remember the past are condemned to repeat it — and in dynamic programming, so is every subproblem you forgot to cache."

— a course aphorism, playing on George Santayana's 1905 dictum

(Bellman himself named the field "dynamic programming" in the 1950s partly because the word "programming" sounded reassuringly bureaucratic to Cold-War-era research funders — not because it involves writing code in the modern sense.)

Lecture 8 is this course's second checkpoint — Bab/Deck Latihan reviewing Lectures 5 through 7 (Heapsort/Quicksort, BST/AVL Trees, and Dynamic Programming) ahead of the Midterm/UTS. Lecture 9 then introduces Greedy Algorithms — a paradigm that, unlike this chapter's dynamic programming, commits to one locally-best choice at each step and never revisits it, trading the exhaustive subproblem-caching of this chapter for a much simpler (but not always applicable) design principle.

7.16 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Draw the complete naive recursion tree for NAIVE-FIB(6), in the style of Figure 7.2, and count exactly how many times fib(2) and fib(1) are each recomputed from scratch.
2. Using the rod-cutting price table p = [1, 5, 8, 9, 10] (now length 5), compute r[5] and s[5] by hand, in the style of Figure 7.4, and state the actual optimal cutting sequence.
3. Explain, in your own words and without looking back at Section 7.9, why matrix-chain multiplication's optimal-substructure argument needs to consider EVERY possible split point k, rather than just the middle one.
4. Trace LCS-LENGTH by hand for X = "AGCAT" and Y = "GAC", building the full c[][] table in the style of Figure 7.6, and state the LCS length and one valid LCS string.
5. Explain, in your own words, why memoized Fibonacci and tabulated Fibonacci have the SAME Θ(n) asymptotic running time despite exploring the subproblem space in opposite directions.
6. A problem has optimal substructure but its naive recursive solution never revisits the same subproblem twice. Is dynamic programming still useful for this problem? Justify your answer using Section 7.2's two conditions.

Appendix 7.A — Verification Log for Chapter 7 Source Code

As in previous chapters, every program shipped with this book is compiled and run against hand-verified examples before being included. Both programs below were compiled with `g++ -O2 -std=c++17 -Wall` (zero warnings, zero errors) and produced the exact output predicted in the text above.

```
$ printf "4\n1 5 8 9\n" | ./01_rod_cutting
r[4] = 10                       # matches Figure 7.4's table exactly
s[] = 1 2 3 2
cuts: 2 + 2                     # matches Section 7.7's hand trace exactly

$ printf "ABCBDAB\nBDCABA\n" | ./02_lcs
table:
0 0 0 0 0 0 0
0 0 0 0 1 1 1
0 1 1 1 1 2 2
0 1 1 2 2 2 2
0 1 1 2 2 3 3
0 1 2 2 2 3 3
0 1 2 2 3 3 4
0 1 2 2 3 4 4
LCS length: 4                   # matches Figure 7.6 exactly
LCS: BCBA                       # matches Section 7.12's hand trace exactly
```

SPOJ Practice — the companion book
This chapter's techniques are tested for real on the SPOJ online judge. Open Chapter 9 of the companion book “SPOJ Competitive Programming” (SPOJ folder, BI & EN) for the full catalogue of SPOJ dynamic-programming patterns — complete with the tips, tricks, and hints that get solutions Accepted, without spoiling answers.

References

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

[2] R. Bellman. Eye of the Hurricane: An Autobiography. World Scientific, 1984. (Bellman's own account of coining the term "dynamic programming" in the 1950s.)

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

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

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 8 · EXERCISE CHAPTER

Practice Problems: Heapsort Through Dynamic Programming

No new material here — just twelve problems designed to make sure Lectures 5 through 7 actually stuck, before the Midterm.

Learning Objectives — after working through this chapter you will be able to:
(1) Trace BUILD-MAX-HEAP and HEAPSORT on a concrete array, and reason about heap-property violations and Priority Queue operations. (2) Trace Quicksort's PARTITION on a concrete array, choose a pivot, and reason about when randomization avoids the $\Theta(n^2)$ worst case. (3) Determine whether a given array can arise from a valid BST, and reason about SEARCH/INSERT/DELETE costs in terms of height. (4) Show how an insertion order can (or cannot) create a degenerate BST, and compute AVL balance factors after an insertion. (5) Identify the rotation case (LL/RR/LR/RL) needed to restore AVL balance after a specific insertion, and apply it by hand. (6) Recognize optimal substructure and overlapping subproblems in an unfamiliar problem, and hand-trace a memoization table, a tabulation table, or an LCS backtrack.

8.1 Recap: Lectures 5–7 in Brief

Lecture 5 closed out the sorting story with two more $\Theta(n \log n)$-family algorithms: Heapsort, built on the binary max-heap data structure and its BUILD-MAX-HEAP/MAX-HEAPIFY operations (and the Priority Queue abstraction that a heap implements directly), and Quicksort, whose PARTITION step gives an in-place, cache-friendly sort with an expected $\Theta(n \log n)$ running time — but a $\Theta(n^2)$ worst case that randomization exists specifically to avoid. Lecture 6 then left sorting behind for a data structure built to answer order-based queries forever: the Binary Search Tree, whose SEARCH/INSERT/DELETE/MINIMUM/MAXIMUM/SUCCESSOR/PREDECESSOR operations all cost $O(h)$ — a bound that can degrade to $\Theta(n)$ on an unlucky insertion order, which is exactly the problem the AVL tree's balance-factor invariant and four rotation cases (LL/RR/LR/RL) were built to fix, guaranteeing $h = O(\log n)$ unconditionally. Lecture 7 introduced an entirely different design paradigm: dynamic programming, which recognizes when a problem's recursive solution revisits the same subproblem again and again (overlapping subproblems) and has an optimal solution built from optimal sub-solutions (optimal substructure), and exploits that with memoization (top-down) or tabulation (bottom-up) — turning exponential naive recursion into polynomial running time for Rod-Cutting, Matrix-Chain Multiplication, and Longest Common Subsequence.

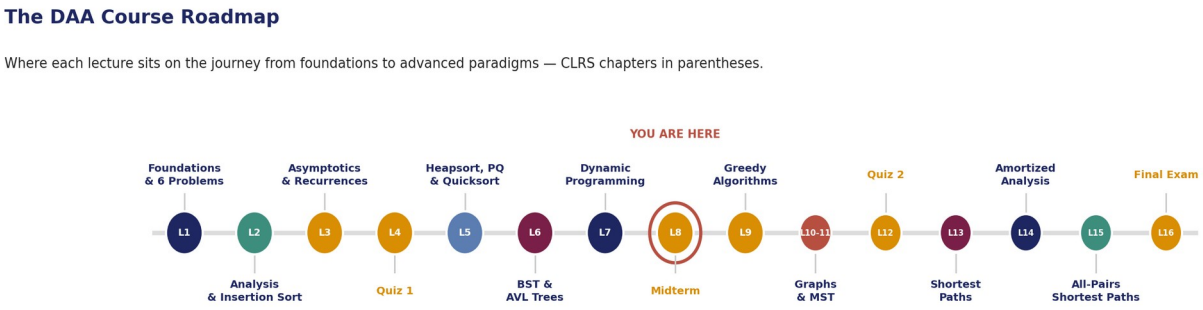

Figure 8.1 — This chapter sits at Lecture 8 in the sixteen-lecture DAA roadmap: a checkpoint, not a new topic, immediately before the Midterm.

8.2 How to Use This Exercise Chapter

What this chapter is — and is not

This is a Bab Latihan: a practice chapter, not a new-content chapter. Every problem below is anchored to specific material from Lectures 5–7 (see Figure 8.2). Each problem comes with a HINT — a nudge toward the right way of thinking — but deliberately NOT a full worked solution or source code. Working through the problem yourself, even imperfectly, teaches far more than reading a finished answer.

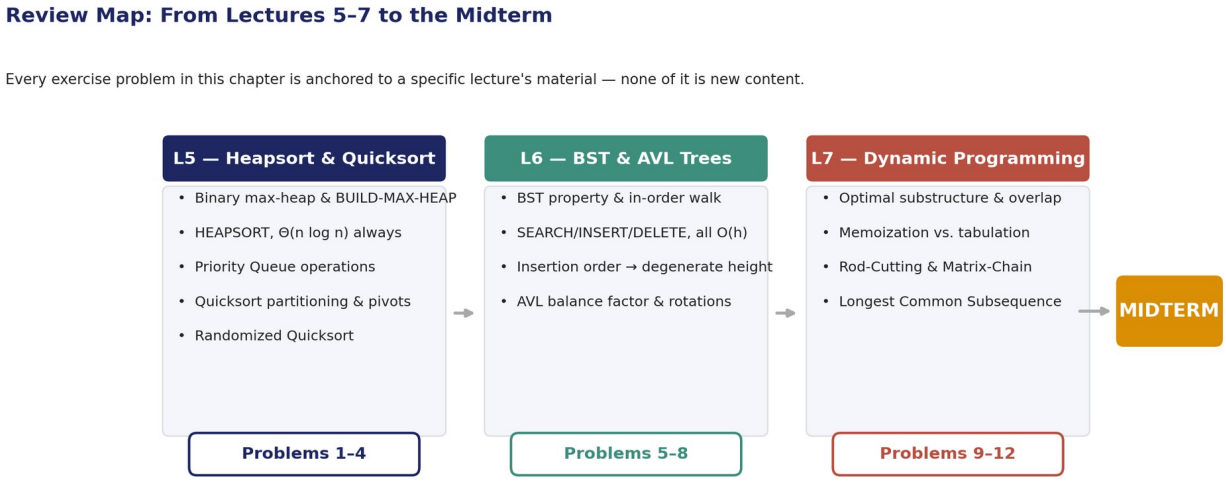

Figure 8.2 — Every problem in Section 8.3 traces back to one of the previous three lectures.

Before you start

Try each problem for real before reading its hint. If you get stuck, read only the hint — not a solution — and try again. This is exactly the discipline Section 8.4's open-book Midterm will reward: the exam allows references, but not a substitute for your own reasoning.

8.3 Practice Problems

8.3.1 Heapsort, Priority Queues & Quicksort

Problem 1 *[Easy]*. Is the array [16, 4, 10, 14, 7, 9, 3, 2, 8, 1] a valid binary max-heap (in the usual array representation, 1-indexed)? Justify your answer by checking the heap property at every internal node, and if it fails, name the FIRST node (by index) where it fails.

> **Hint**
> For array index i, its children are at 2i and 2i+1 (1-indexed). You only need to check indices that actually have at least one child.

Problem 2 *[Medium]*. Trace HEAPSORT on the max-heap [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]: show the array after EACH extract-max-and-heapify step, not just the final sorted result. How many total calls to MAX-HEAPIFY does this make, and what is each one's worst-case cost in terms of the current heap size?

> **Hint**
> Each HEAPSORT iteration swaps A[1] with A[n], shrinks the heap by one, then re-heapifies from the root — exactly the same MAX-HEAPIFY you used to build the heap in the first place.

Problem 3 *[Medium]*. Trace Lomuto's PARTITION on the array [13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11] using the LAST element as the pivot. Show the array's state after every swap, and state the final index of the pivot after partitioning completes.

> **Hint**
> Track the invariant carefully: index i marks the boundary of the "known ≤ pivot" region, and index j scans forward — a swap happens only when A[j] ≤ pivot.

Problem 4 *[Hard]*. Construct a specific array of length 8 that would make DETERMINISTIC Quicksort (pivot = last element, no randomization) hit its $\Theta(n^2)$ worst case. Then explain, in your own words, WHY choosing the pivot uniformly at random defeats an adversary who knows your partitioning scheme but does not know your random choices.

> **Hint**
> The worst case for last-element-pivot Quicksort happens when the array is already sorted (or reverse-sorted) — every partition call then splits n elements into (n−1, 0) instead of anything balanced.

8.3.2 Binary Search Trees & AVL Trees

Problem 5 *[Easy]*. You are given the in-order traversal output of some binary tree: 1, 3, 4, 6, 7, 8, 10, 13, 14. Does this sequence, BY ITSELF, prove the tree is a valid BST? If not, what additional information about the tree's SHAPE would you need to be sure?

> **Hint**
> In-order traversal of ANY binary tree that happens to satisfy the BST property produces sorted output — but check whether the converse direction (sorted output ⟹ valid BST) is actually guaranteed by the sequence alone, or requires knowing the tree's structure too.

Problem 6 *[Medium]*. Insert the keys 10, 20, 30, 40, 50 into an initially empty BST, IN THAT ORDER, using ordinary BST-INSERT. Draw the resulting tree and state its height. Then find a DIFFERENT insertion order of the same five keys that produces a tree of height 2 (the minimum possible for 5 keys).

> **Hint**
> BST-INSERT always adds a new leaf where the search for that key would end — so inserting keys in already-sorted order produces a very specific, very bad shape. For the balanced order, think about which key should be inserted FIRST to end up at the root.

Problem 7 *[Medium]*. Starting from the balanced 5-node tree you built in Problem 6 (or any balanced BST of 5 distinct keys with height 2), insert one more key that creates a balance-factor violation (BF = +2 or −2) at some node. State exactly which node becomes unbalanced and its balance factor before any rotation is applied.

> **Hint**
> BF(x) = height(x.left) − height(x.right). Recompute heights bottom-up after the insertion, exactly the way AVL-INSERT does in Section 6.12, before deciding which node (if any) has drifted outside {−1, 0, +1}.

Problem 8 *[Hard]*. For the unbalanced node you found in Problem 7, identify WHICH of the four rotation cases (LL, RR, LR, RL) applies, and perform the rotation(s) by hand — show the tree immediately before and immediately after. Verify that every node's balance factor is back in {−1, 0, +1} after your rotation.

> **Hint**
> The case depends on TWO balance factors: the unbalanced node z's own BF, and the BF of whichever child of z is on the "heavy" side. Section 6.11's four-case diagram tells you exactly which single or double rotation each combination calls for.

8.3.3 Dynamic Programming

Problem 9 *[Medium]*. A new problem: given a sequence of n stock prices, find the length of the longest CONTIGUOUS run of strictly increasing prices. Does this problem have optimal substructure? Does it have overlapping subproblems in the sense Section 7.2 defines? Justify both answers, and state whether dynamic programming is the right tool here or whether a simpler approach suffices.

> **Hint**
> Try defining a recursive relationship in terms of "the best answer ending exactly at position i." Then ask: does evaluating that relationship at every position require solving any subproblem MORE than once?

Problem 10 *[Medium]*. Hand-trace MEMO-FIB(6) (Section 7.4) call by call: list every recursive call made, in the order it is first invoked, and mark which calls are cache HITS versus which perform real work. How many real (non-cached) subproblem computations are performed in total?

> **Hint**
> Draw it like Figure 7.2's recursion tree for fib(5), but now cross out every subtree that a memo-table hit would prevent from ever being explored — count only the surviving, first-time computations.

Problem 11 *[Hard]*. For the rod-cutting price table p = [1, 5, 8, 9, 10] (lengths 1 through 5), compute the FULL r[] and s[] tables by hand, following EXTENDED-BOTTOM-UP-CUT-ROD (Section 7.6–7.7). State r[5] and the actual optimal cutting (the sequence of piece lengths) that achieves it.

> **Hint**
> Build the table strictly left to right: r[0] = 0, then for each j from 1 to 5, try every first-piece length i from 1 to j using r[j−i], which is already computed. Follow Figure 7.4's worked p=[1,5,8,9] example as your template, extended by one column.

Problem 12 *[Hard]*. For X = "AGCAT" and Y = "GAC", build the full LCS-LENGTH table c[i][j] by hand (Section 7.10–7.11), then use PRINT-LCS (Section 7.12) to backtrack and state one valid longest common subsequence, along with its length.

> **Hint**
> Follow Figure 7.6's worked X=ABCBDAB/Y=BDCABA example as your template: c[i][j] = c[i−1][j−1] + 1 when the characters match, otherwise max(c[i−1][j], c[i][j−1]) — then backtrack from c[m][n] exactly as PRINT-LCS does.

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

The Midterm is an open-book, take-home essay assignment covering exactly the material reviewed in this chapter (Lectures 5 through 7) plus everything from before it — nothing beyond it. You will have one week to submit written answers with full reasoning; a correct final answer with no justification earns little credit, since the point of an essay-format exam is to see HOW you think, not just what you conclude.

Open-book means references, not co-authors

You may consult the textbook, your notes, and lecture slides while answering. You may NOT collaborate with classmates or submit reasoning that is not genuinely your own — an open-book exam tests whether you can APPLY what you have studied, unsupervised, which is exactly what "designing and analyzing algorithms" means in practice. If you quote a source directly, cite it.

Criterion	What it looks for	Weight
Correctness of reasoning	Is the logic, proof, trace, or derivation actually valid — not just the final answer?	40%
Clarity of proof / derivation	Can a reader follow each step without having to guess what you meant?	25%
Proper use of notation	Are O/Ω/Θ, heap/tree diagrams, and DP tables stated precisely, matching earlier chapters' conventions?	20%
Presentation	Is the answer organized, legible, and easy to grade?	15%

8.5 Profitina in Practice: Self-Review Before Submission

About this box

As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

Before a new algorithm change ships to production at Profitina, engineers run it through a short self-review checklist — very much like Appendix 8.A below — before ever requesting a second pair of eyes: has a claimed data-structure invariant (a heap property, a BST ordering, an AVL balance factor) actually been checked on a concrete example, not just assumed? Has a dynamic-programming table's base case and fill order actually been traced by hand at least once, the way Section 8.3.3's problems require? This habit of adversarial self-checking before external review — the same discipline behind the bug-hunting lesson repeated throughout this textbook, that a visually clean result can still be mathematically wrong — is exactly what an open-book take-home exam format is built to train.

8.6 Chapter Summary

- This chapter introduced no new material — it is a checkpoint covering Lectures 5 through 7.
- Twelve problems span the full range reviewed: heap properties and HEAPSORT, Quicksort partitioning and its worst case, BST shape and insertion order, AVL balance factors and rotation cases, and dynamic programming's two structural requirements applied to memoization, tabulation, Rod-Cutting, and Longest Common Subsequence.

- Each problem includes a hint, not a solution — working through the reasoning yourself is the point.
- The Midterm is an open-book, take-home essay: references are allowed, but the reasoning must be your own, and correctness of REASONING (not just the final answer) is the majority of the grade.

"Sorting is what computers do when they aren't doing anything more interesting — until you need it to be fast."

— a course aphorism

(Rereading a proof doesn't count as knowing it — reproducing it from scratch does.)

Appendix 8.A — Self-Check Checklist (Non-Graded)

Before submitting any answer — in this chapter's problems or in the Midterm itself — run it through this checklist. It costs a few minutes and catches most of the mistakes graders see most often.

- Did I actually verify a claimed invariant (heap property, BST ordering, AVL balance factor) on the concrete example given, rather than just asserting it holds?
- In any hand-traced table (heap array, DP table, BST/AVL tree), did I fill it in the correct order and show EVERY intermediate state, not just the final result?
- If I claimed a rotation case (LL/RR/LR/RL), did I name the two balance factors that determine it, not just the final tree shape?
- If I claimed a problem has (or lacks) overlapping subproblems, did I actually point to a specific subproblem computed more than once — or specific evidence that none is?
- Did I distinguish the WORST case from the EXPECTED case explicitly wherever Quicksort or randomization was involved?
- Did I reread my own answer as if I were a skeptical grader looking for a gap in the argument?

References

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

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

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

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

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 9

Greedy Algorithms

A design paradigm that makes one locally best choice at each step, never reconsiders it, and — for a specific class of problems that can be proven to have the right structure — still ends up at a globally optimal answer.

Learning Objectives — after this chapter you will be able to:
(1) State the two properties a problem must satisfy for a greedy algorithm to be provably correct — the greedy-choice property and optimal substructure — and distinguish them from dynamic programming's conditions. (2) Derive, trace, and implement the greedy solution to the Activity Selection problem, and explain why sorting by FINISH time (not start time or duration) is what makes the greedy choice safe. (3) Construct an exchange-argument proof that a greedy choice never makes the final answer worse — the standard technique for proving any greedy algorithm correct. (4) Build a Huffman tree from a set of character frequencies, derive the resulting prefix-free codes, and compute the total encoded length. (5) Explain why greedy-by-ratio is optimal for the Fractional Knapsack problem but provably WRONG for the 0/1 Knapsack problem — and identify, for a new problem, which side of that line it falls on. (6) Analyze the running time of each greedy algorithm in this chapter and explain why greedy algorithms are typically dominated by a single sort, rather than by a table-filling phase.

9.1 Recap: From Lecture 8 to Lecture 9

Lecture 8 was a checkpoint — no new material, just a chance to consolidate Heapsort, Priority Queues, Quicksort, BST/AVL Trees, and Dynamic Programming ahead of the Midterm. This chapter opens an entirely new design paradigm: greedy algorithms. Where Chapter 7's dynamic programming solved every distinct subproblem once and combined the results, a greedy algorithm makes a single irrevocable choice at each step, based only on what looks best RIGHT NOW, and never looks back to reconsider it.

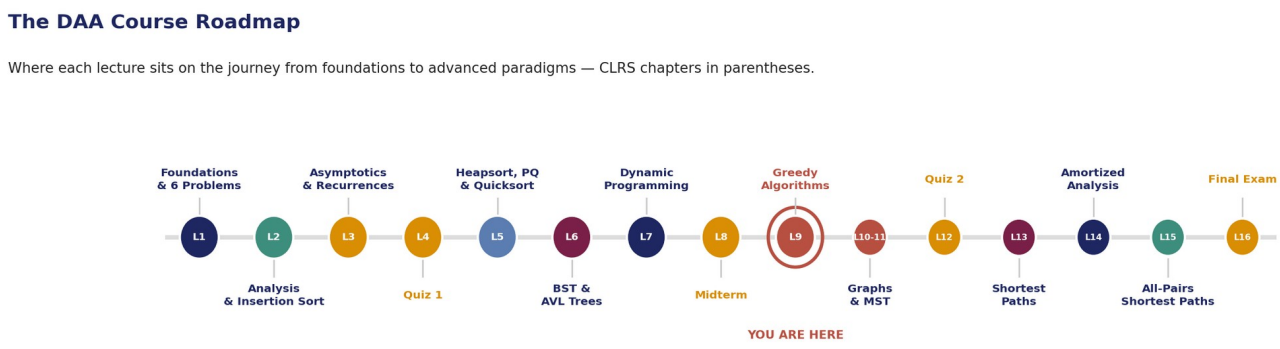

Figure 9.1 — Where this chapter sits in the sixteen-lecture DAA roadmap. Lecture 9 introduces the greedy paradigm, immediately after the Midterm checkpoint and immediately before graph algorithms — several of which (Kruskal's and Prim's MST algorithms, Lectures 10–11) are themselves greedy algorithms in disguise.

Why "greedy" is not an insult here

In everyday language, "greedy" suggests short-sightedness that backfires. In algorithm design, a greedy algorithm is only called correct when it can be PROVEN that short-sightedness does not backfire — that committing to the locally best choice at every step is guaranteed to reach a globally best answer. Section 9.2 makes precise exactly what must be true for that guarantee to hold, and this chapter's final worked example (Section 9.9) shows a case where it looks like it should hold, but provably does not.

9.2 What Makes a Greedy Algorithm Correct? Two Structural Conditions

Like dynamic programming, greedy algorithms apply to optimization problems, and correctness rests on a structural property of the problem — but a stricter one than dynamic programming requires.

The greedy-choice property

A problem has the greedy-choice property if a globally optimal solution can always be reached by making a single, LOCALLY optimal (greedy) choice first, and then optimally solving the remaining subproblem that choice leaves behind. Critically, this must hold no matter what the rest of the input looks like — the greedy choice can never depend on solving any part of the remaining subproblem first, or it would not be a genuinely greedy (single-pass, never-revisited) choice.

Optimal substructure (again)

Exactly as in Chapter 7: an optimal solution to the whole problem must be built from an optimal solution to the subproblem left behind by the first choice. Greedy algorithms and dynamic programming BOTH require optimal substructure — the difference is that dynamic programming, lacking any way to know which choice is safe in advance, tries the choices that optimal substructure implies exist and combines the best one via a table, while a greedy algorithm short-circuits that entire search because the greedy-choice property guarantees which single choice to try.

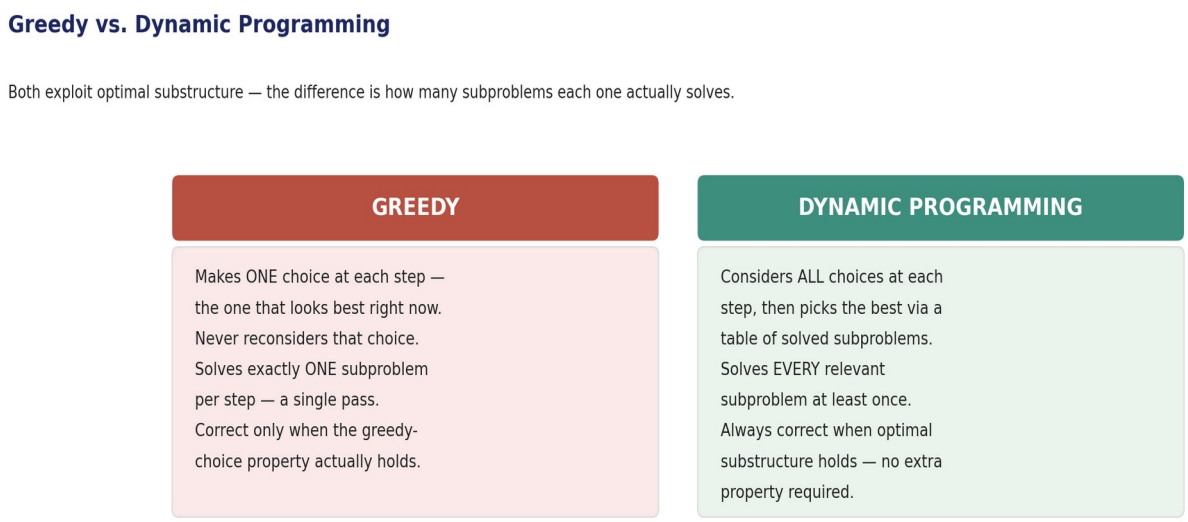

Figure 9.2 — Both paradigms rely on optimal substructure. Dynamic programming solves every subproblem the substructure implies could matter; a greedy algorithm, whenever the greedy-choice property additionally holds, solves only one.

Optimal substructure alone is NOT enough for greedy

A problem can have optimal substructure without having the greedy-choice property — 0/1 Knapsack (Section 9.9) is exactly such a case. Applying a greedy algorithm to a problem that lacks the greedy-choice property produces code that runs fast and returns a confident, plausible-looking, WRONG answer — with no error message, because nothing about the algorithm's execution looks unusual. The property must be proven, not assumed from a problem merely "feeling greedy."

9.3 The Activity Selection Problem

The Activity Selection problem

Given a set of n activities, each with a start time and a finish time, and a single shared resource (a room, a machine, a meeting slot) that can host only one activity at a time, select the LARGEST possible subset of mutually compatible activities — activities whose time intervals do not overlap.

A greedy strategy for this problem needs a rule for which activity to pick first. Several plausible-sounding rules turn out to be wrong: picking the SHORTEST activity first, or the one that starts EARLIEST, can both be shown by counterexample to sometimes produce a suboptimal final count. The rule that provably works is to sort by FINISH time and always pick the compatible activity that finishes earliest.

```
GREEDY-ACTIVITY-SELECTOR(s, f):     // s = start times, f = finish times, sorted by f
  n = length(f)
  A = { 1 }                         // always take the first activity to finish
  last_finish = f[1]
  for i = 2 to n
      if s[i] >= last_finish        // compatible with everything chosen so far
          A = A union { i }
          last_finish = f[i]
  return A
```

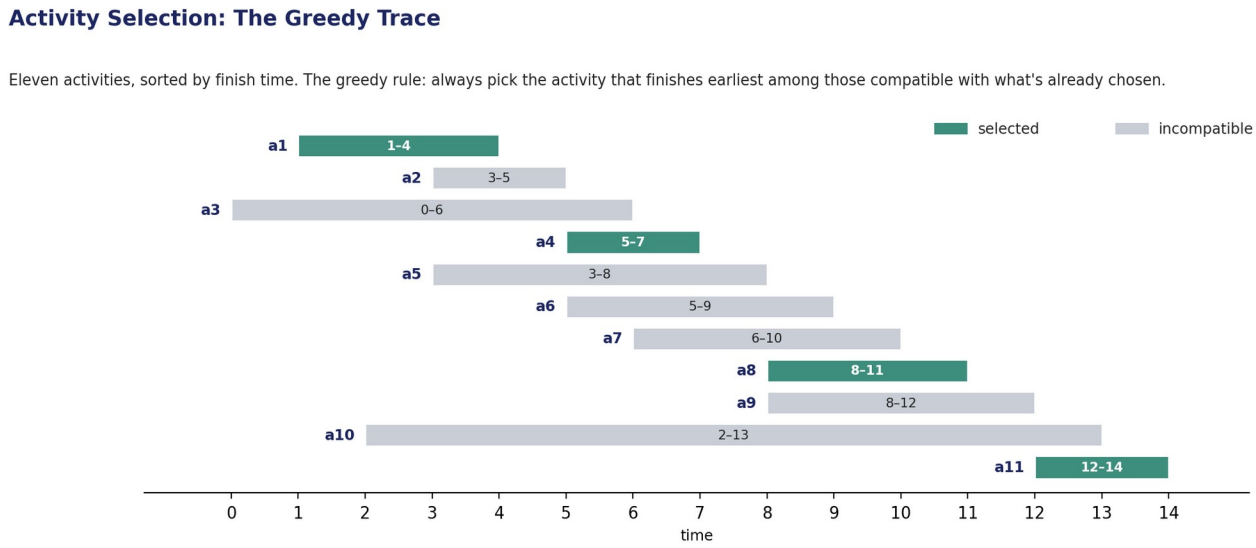

Figure 9.3 — CLRS's classic 11-activity example, sorted by finish time. The greedy rule selects a1, then skips every activity incompatible with a1 until reaching a4 (the earliest-finishing activity that starts at or after a1's finish time), then repeats — reaching a8, then a11.

Tracing GREEDY-ACTIVITY-SELECTOR on Figure 9.3's data: start with a1 (finish 4). a2 (start 3) and a3 (start 0) both start before time 4 — incompatible, skip both. a4 (start 5) starts at or after time 4 — compatible, select it, update last_finish to 7. a5 (start 3) and a6 (start 5) and a7 (start 6) all start before time 7 — skip all three. a8 (start 8) is compatible — select it, update last_finish to 11. a9 (start 8) and a10 (start 2) both start before 11 — skip both. a11 (start 12) is compatible — select it. Final answer: {a1, a4, a8, a11} — four activities, matching Appendix 9.A's compiled output exactly.

9.4 Proving Greedy Correct: The Exchange Argument

Sorting by finish time and picking greedily is easy to state — but why is it actually optimal? The standard technique for proving a greedy algorithm correct is the EXCHANGE ARGUMENT: take an arbitrary optimal solution, and show it can always be modified ("exchanged") to include the greedy choice, without making it any worse. If the greedy choice can always be swapped in for free, greedy loses nothing by choosing it first.

The exchange argument for Activity Selection, in outline

Let a1 be the activity with the earliest finish time, and let A be ANY optimal solution. If a1 ∈ A, the greedy choice is already part of an optimal solution — done. If a1 ∉ A, let a_k be the activity in A with the earliest finish time. Because a1 finishes no later than a_k (a1 has the globally earliest finish time), replacing a_k with a1 in A cannot create any new overlap — every activity in A that was compatible with a_k is still compatible with the earlier-or-equal-finishing a1. This produces a DIFFERENT solution A′ = A − {a_k} + {a1}, the SAME size as A, that DOES contain the greedy choice. Since A was assumed optimal and A′ has the same size, A′ is also optimal. Either way, some optimal solution containing a1 exists — exactly what the greedy-choice property (Section 9.2) requires.

Optimal substructure closes the argument: once a1 is fixed as part of an optimal solution, the remaining problem — selecting a maximum-size compatible subset from the activities that start at or after a1's finish time — is the EXACT SAME problem (Activity Selection) on a strictly smaller input. Solving that remaining problem greedily, and repeating this same exchange argument at every step, shows the entire greedy algorithm produces a globally optimal solution by induction.

9.5 Huffman Coding: Optimal Prefix-Free Compression

The Huffman Coding problem

Given a set of characters and their frequencies (or probabilities) in some source text, construct a variable-length BINARY CODE — a distinct bit-string for each character — that (a) is prefix-free (no character's code is a prefix of another's, so a stream of concatenated codes can be decoded unambiguously without separators) and (b) minimizes the TOTAL number of bits needed to encode the whole source text, given those frequencies.

The greedy idea: characters with HIGH frequency should get SHORT codes, and characters with LOW frequency can afford LONG codes, because they contribute their code length far fewer times to the total. Huffman's algorithm builds the optimal code bottom-up as a binary tree: repeatedly take the two currently-lowest-frequency nodes (initially, the individual characters; later, merged subtrees) and merge them into a new node whose frequency is their sum, continuing until only one tree remains. Each character's code is then read off as the sequence of left/right branches (0/1) from the root to that character's leaf.

```
HUFFMAN(C):                          // C = set of characters with frequencies
  n = |C|
  Q = min-priority-queue of C, keyed by frequency
  for i = 1 to n - 1
      lo = EXTRACT-MIN(Q)              // two currently-lowest-frequency nodes
      hi = EXTRACT-MIN(Q)
      z = new node
      z.left = lo; z.right = hi
      z.freq = lo.freq + hi.freq
      INSERT(Q, z)
  return EXTRACT-MIN(Q)                // the single remaining node is the root
```

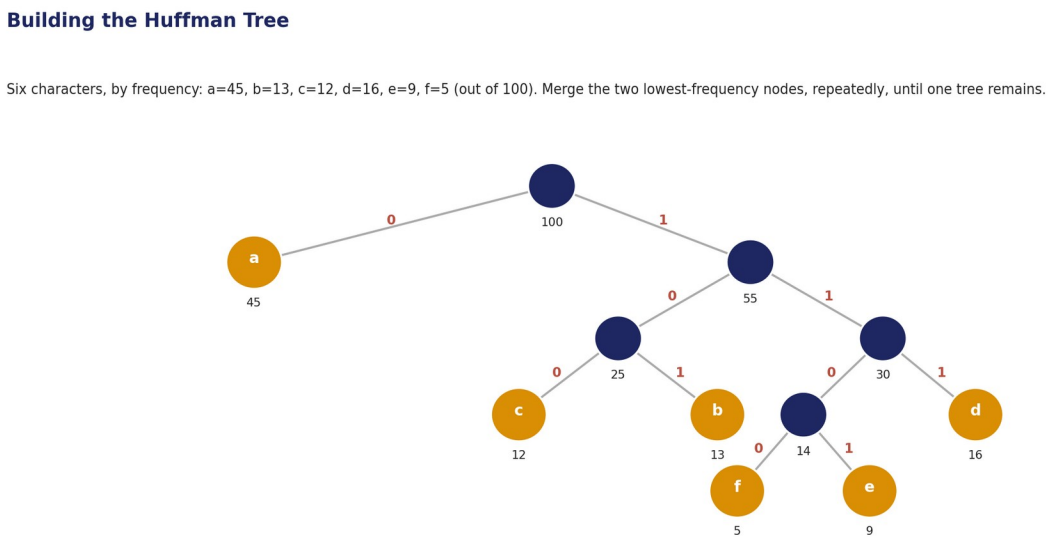

Figure 9.4 — The Huffman tree built from six characters with frequencies a=45, b=13, c=12, d=16, e=9, f=5 (out of 100). The two lowest-frequency nodes are merged at each step: first f+e=14, then c+b=25, then (f+e)+d=30, then (c+b)+((f+e)+d)=55, finally a+55=100.

9.6 Reading Off the Codes and Computing the Savings

Reading each character's code from Figure 9.4 (0 = left branch, 1 = right branch, from the root): a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100. Notice the property the greedy construction was designed to produce: a, the most frequent character (45 out of 100), gets the shortest code (1 bit); e and f, the least frequent (9 and 5), get the longest codes (4 bits each).

Phase 5 — Implement: reading codes off the tree and totaling the length

```
COLLECT-CODES(node, path, codes):        // path starts as the empty string
  if node.isLeaf
      codes[node.character] = path
      return
  COLLECT-CODES(node.left,  path + '0', codes)
  COLLECT-CODES(node.right, path + '1', codes)
```

Total encoded length = Σ (frequency × code length) = 45(1) + 13(3) + 12(3) + 16(3) + 9(4) + 5(4) = 45 + 39 + 36 + 48 + 36 + 20 = 224 bits, for 100 characters of source text — matching Appendix 9.A's compiled output exactly. A FIXED-length code needs $\lceil \log_2 6 \rceil = 3$ bits per character regardless of frequency, for 100 × 3 = 300 bits total. Huffman's variable-length code saves 76 bits — a 25.3% reduction — purely by exploiting the frequency skew, with zero loss of information.

> **Why this greedy choice is provably safe**
>
> The exchange argument here (full proof in CLRS Theorem 16.2, omitted for space) shows that the two globally lowest-frequency characters can always be placed as SIBLINGS at the DEEPEST level of SOME optimal code tree — exactly what the first merge step does. Optimal substructure then holds: once those two characters are merged into a single combined symbol, finding an optimal code for the ORIGINAL alphabet reduces to finding an optimal code for a SMALLER alphabet (one symbol fewer), which is exactly the same problem, solved recursively by every subsequent merge.

9.7 Fractional Knapsack: A Second Greedy Success

The Fractional Knapsack problem

Given n items, each with a value and a weight, and a knapsack of capacity W, choose how much of EACH item to take (any fraction between 0 and 100% of an item's weight is allowed) to maximize total value, subject to total weight taken not exceeding W.

The greedy rule: compute each item's value-to-weight RATIO, sort items by that ratio in decreasing order, and take as much as possible of the highest-ratio item first, then the next-highest, and so on, until the knapsack is full (the last item taken may be only partially included, to exactly fill the remaining capacity).

Worked example: capacity W = 50, with three items — item 1 (value 60, weight 10, ratio 6), item 2 (value 100, weight 20, ratio 5), item 3 (value 120, weight 30, ratio 4). Sorted by ratio: item 1, then item 2, then item 3. Take all of item 1 (weight 10, remaining capacity 40, value so far 60). Take all of item 2 (weight 20, remaining capacity 20, value so far 160). Only 20 of item 3's 30 weight units fit — take the fraction 20/30 = 2/3, contributing 120 × 2/3 = 80. Total value: 60 + 100 + 80 = 240, using the full capacity of 50 — matching Figure 9.5's first bar exactly.

Why fractional allowance is what makes greedy safe here

Because ANY fraction of ANY item can be taken, there is never a scenario where taking the best available ratio first "wastes" capacity that a different first choice could have used more cleverly — any leftover capacity can always be filled with a fraction of whatever is taken next, at that next item's own ratio. This flexibility is precisely what Section 9.8 removes, and precisely why removing it breaks the greedy-choice property.

9.8 0/1 Knapsack: Where the Same Greedy Rule Fails

The 0/1 Knapsack problem

The same setup as Fractional Knapsack — n items with values and weights, capacity W — except each item must be taken ENTIRELY or not at all; no fractions are allowed.

Applying the exact same greedy-by-ratio rule to the SAME three items from Section 9.7, but now forbidding fractions: take item 1 (weight 10, remaining capacity 40, value 60). Take item 2 (weight 20, remaining capacity 20, value 160). Item 3 needs 30 weight units, but only 20 remain — since fractions are forbidden, item 3 must be skipped ENTIRELY. Greedy-by-ratio's final answer: 160.

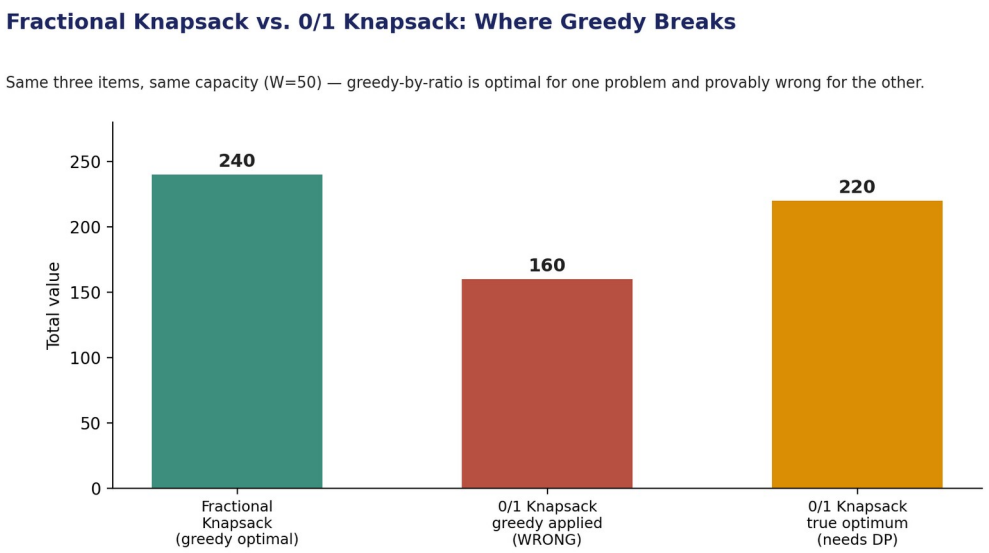

Figure 9.5 — Same three items, same capacity, two different problems. Greedy-by-ratio achieves the true optimum (240) for Fractional Knapsack, but only 160 when the same rule is (incorrectly) applied to 0/1 Knapsack — a full 60 short of that problem's true optimum of 220 (items 2 and 3 together, no item 1 at all).

160 looks like a completely reasonable answer — that is the danger

Nothing about greedy-by-ratio's execution on 0/1 Knapsack signals failure: it runs to completion, uses valid moves, and returns a specific, confident number. Only checking against the TRUE optimum (220, achieved by taking items 2 and 3 together and no item 1 at all — verified in Appendix 9.A by exhaustive brute-force search over all $2^3 = 8$ possible subsets) reveals the 60-value gap. 0/1 Knapsack does have optimal substructure, but NOT the greedy-choice property — the correct algorithm for it is dynamic programming (a table indexed by item and remaining capacity), a problem this course's Chapter 7 techniques apply to directly but which this chapter's greedy techniques provably cannot solve correctly.

9.9 Analyzing Greedy Algorithms: Where the Time Actually Goes

Every algorithm in this chapter shares a common shape, quite different from Chapter 7's dynamic-programming algorithms: almost all the running time is spent in a single SORT, with the greedy pass itself taking only linear time afterward.

Algorithm	Sorting step	Greedy pass	Total time
Activity Selection	Θ(n log n) — sort by finish time	Θ(n)	Θ(n log n)
Huffman Coding	Θ(n log n) — n EXTRACT-MIN/INSERT ops on a heap	—	Θ(n log n)
Fractional Knapsack	Θ(n log n) — sort by value/weight ratio	Θ(n)	Θ(n log n)

Algorithm	Sorting step	Greedy pass	Total time

A useful contrast with Chapter 7

Chapter 7's dynamic-programming algorithms spent their time filling a TABLE of size proportional to the number of distinct subproblems (Θ(n^2) for Rod-Cutting, Θ(mn) for LCS). This chapter's greedy algorithms spend their time on a single SORT (or, equivalently, a sequence of priority-queue operations — Huffman's repeated EXTRACT-MIN is exactly a disguised sort), because the greedy-choice property removes the need to ever reconsider a choice once made. When a problem can be solved greedily at all, the resulting algorithm is typically both simpler to implement and asymptotically no worse than the DP alternative — the entire difficulty is in PROVING the greedy-choice property holds in the first place.

9.10 Greedy Algorithms in Practice: Where This Chapter Shows Up in a Real Product

About this box

As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

Profitina periodically needs to schedule a limited pool of scoring workers against a queue of pending prompt-evaluation jobs, each with an earliest-ready time and a deadline — structurally close to Activity Selection's start/finish intervals, and scheduled the same way: sort by deadline, greedily assign the next compatible job to a free worker. Profitina's exported report bundles (per-client visibility summaries, sent as compressed attachments) also benefit from exactly Section 9.5's insight: text fields that repeat very frequently across a report (common competitor names, common prompt templates) are assigned shorter internal codes than rare ones, the same frequency-driven idea Huffman coding formalizes, even where the actual bytes-on-the-wire compression is handled by a general-purpose library rather than a hand-rolled Huffman tree. And when Profitina must fit a bounded batch of AI-provider API calls into a fixed per-minute rate-limit budget, the underlying resource-allocation shape is Knapsack-like — recognizing which of the two knapsack variants actually applies (can a call's cost be split fractionally across two minutes, or must it complete atomically within a single minute?) determines whether a fast greedy heuristic is provably correct or merely a plausible-looking approximation, exactly the distinction Sections 9.7–9.8 exist to teach.

9.11 Chapter Summary and Key Takeaways

- A greedy algorithm makes one locally optimal choice at each step and never reconsiders it — correct only when the problem provably has BOTH the greedy-choice property and optimal substructure.

- Activity Selection: sort by finish time, greedily take every compatible activity in that order. Proven correct via an exchange argument — the earliest-finishing activity can always be swapped into any optimal solution at no cost.
- Huffman Coding: repeatedly merge the two lowest-frequency nodes into a tree; the resulting root-to-leaf codes are prefix-free and minimize total encoded length — frequent characters end up with short codes, rare characters with long ones.
- Fractional Knapsack: sort by value-to-weight ratio, take as much of the best-ratio item as capacity allows, then the next-best, and so on — optimal precisely because fractional amounts remove any scenario where a different order could have used capacity more cleverly.
- 0/1 Knapsack: the SAME greedy-by-ratio rule is provably WRONG once items cannot be split — a critical illustration that optimal substructure alone does not imply the greedy-choice property, and that dynamic programming (Chapter 7), not greedy, is the correct tool here.
- Every greedy algorithm in this chapter is dominated by a single Θ(n log n) sort (or equivalent priority-queue operations), with the actual greedy pass afterward taking only linear time — a sharp contrast with dynamic programming's table-filling running times.

"A greedy algorithm never looks back — which is either its greatest strength, or the reason it just got the wrong answer."

— a course aphorism

(The exchange argument is what tells you which one it is, before you find out the hard way.)

Lecture 10 continues directly from this chapter's final insight: several of the most important graph algorithms — Kruskal's and Prim's algorithms for Minimum Spanning Trees, covered across Lectures 10 and 11 — ARE greedy algorithms, each provably correct via its own exchange argument, applying exactly the machinery Sections 9.2 and 9.4 developed to an entirely new problem domain.

9.12 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Using the activity data s = [1, 3, 0, 5, 3, 5, 6, 8, 8, 2, 12], f = [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] from Figure 9.3, show that sorting by START time instead of finish time and applying the same greedy rule can select fewer than 4 activities. (Hint: try starting with a3, which has the earliest start time.)
2. Build the Huffman tree by hand for four characters with frequencies w=10, x=15, y=30, z=45 (out of 100), in the style of Figure 9.4, state each character's code, and compute the total encoded length for 100 characters.

3. Explain, in your own words and without looking back at Section 9.4, why the exchange argument for Activity Selection depends specifically on a1 having the EARLIEST finish time, rather than (for example) the LATEST start time.
4. For the Fractional Knapsack instance in Section 9.7 (capacity 50, items (60,10), (100,20), (120,30)), show that greedy-by-ratio remains optimal even if the capacity is changed to 35 instead of 50, and state the new total value.
5. A greedy algorithm for a NEW optimization problem is proposed, and it happens to produce the correct answer on every example the proposer tried. Is this sufficient to conclude the greedy-choice property holds? Justify your answer using Section 9.2 and the 0/1 Knapsack counterexample of Section 9.8.
6. Explain why Huffman coding's running time is Θ(n log n) even though the tree it builds has only n leaves — where does the log n factor actually come from?

Appendix 9.A — Verification Log for Chapter 9 Source Code

As in previous chapters, every program shipped with this book is compiled and run against hand-verified examples before being included. Both programs below were compiled with `g++ -O2 -std=c++17 -Wall` (zero warnings, zero errors) and produced the exact output predicted in the text above. The 0/1 Knapsack optimum cited in Section 9.8 was independently verified by exhaustive brute-force search over all $2^3 = 8$ subsets of the three items.

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

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

SPOJ Practice — the companion book
This chapter's techniques are tested for real on the SPOJ online judge. Open Chapter 10 of the companion book "SPOJ Competitive Programming" (SPOJ folder, BI & EN) for greedy and exchange-argument problems, with proof sketches — complete with the tips, tricks, and hints that get solutions Accepted, without spoiling answers.

References

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

[2] D. A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes." Proceedings of the IRE, 40(9), 1952, 1098–1101. (The original paper — Huffman devised this as a term-paper solution while a student at MIT.)

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

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

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 10

Graphs and Minimum Spanning Trees: Kruskal's Algorithm

The first graph algorithm of the course — and a second, entirely different domain where the greedy paradigm from Chapter 9 turns out to be exactly the right tool, once its correctness is proven by a new argument: the cut property.

Learning Objectives — after this chapter you will be able to:
(1) Represent a graph using an adjacency list and an adjacency matrix, and state the space/time trade-off between them. (2) State the Minimum Spanning Tree (MST) problem precisely, and explain why a spanning tree has exactly |V| − 1 edges. (3) State and apply the CUT PROPERTY — the structural fact that makes greedy MST algorithms provably correct. (4) Trace, implement, and analyze Kruskal's algorithm, including its use of the union-find (disjoint-set) data structure to detect cycles in O(α(n)) amortized time per operation. (5) Prove Kruskal's algorithm correct using the cut property, following the same exchange-argument style introduced in Chapter 9. (6) Analyze Kruskal's total running time and identify which step dominates it.

10.1 Recap: From Greedy Algorithms to Graphs

Chapter 9 introduced the greedy paradigm through three problems — Activity Selection, Huffman Coding, Fractional Knapsack — each drawn from a different corner of computing, unified only by two structural properties: the greedy-choice property and optimal substructure. This chapter and the next apply that same paradigm to an entirely new domain: GRAPHS, structures of vertices and edges that model networks — road maps, computer networks, social connections, dependency chains, and (as this chapter's running example illustrates) the kind of multi-region deployment a growing platform like Profitina could one day operate. Lecture 10 covers graph representations and Kruskal's algorithm for the Minimum Spanning Tree problem; Lecture 11 covers a second MST algorithm, Prim's, built on a different data structure but resting on the exact same correctness argument.

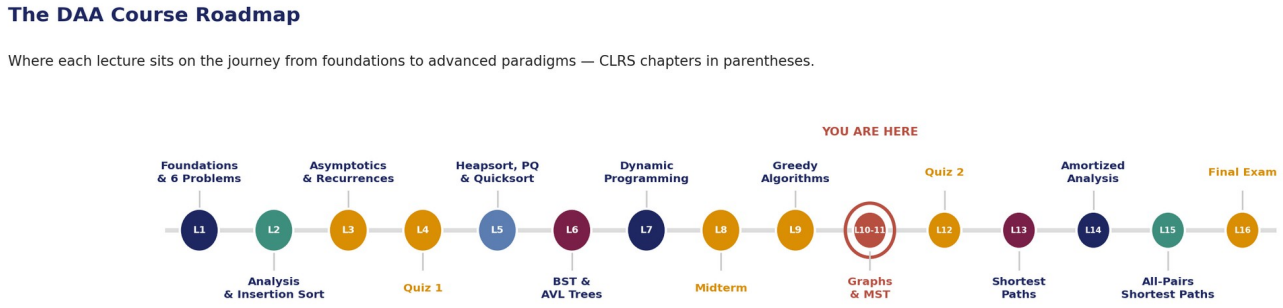

Figure 10.1 — Where this chapter sits in the sixteen-lecture DAA roadmap. Lectures 10 and 11 together cover graphs and Minimum Spanning Trees — the first topic in the course built on greedy algorithms applied to a genuinely new kind of structure.

Kruskal's and Prim's algorithms ARE greedy algorithms

Chapter 9 closed with a forward-reference to exactly this fact: both of this chapter's algorithms make a sequence of locally-best choices — always the cheapest available edge that does not create a problem — and never reconsider a choice once made. What makes them correct is not a new paradigm, but a new PROOF TECHNIQUE (the cut property, Section 10.3) applied to a new problem structure (graphs) — the same greedy-choice-property-plus-optimal-substructure requirement from Chapter 9 still governs whether the approach is valid at all.

10.2 Graph Basics: Vertices, Edges, and Representations

Graph, vertex, edge

A graph G = (V, E) consists of a set of VERTICES V (also called nodes) and a set of EDGES E, where each edge connects two vertices. This chapter works with UNDIRECTED, WEIGHTED graphs: each edge {u, v} can be traversed in either direction, and carries a numeric WEIGHT (a cost, distance, or capacity, depending on what the graph models).

A graph can be represented two standard ways. An ADJACENCY LIST stores, for each vertex, a list of its neighbors (and the weight of the connecting edge) — using $\Theta(V + E)$ space, and letting you enumerate a vertex's neighbors in time proportional to its degree. An ADJACENCY MATRIX stores an $|V| \times |V|$ grid where entry (u, v) holds the weight of edge {u, v} (or ∞/0 if no such edge exists) — using $\Theta(V^2)$ space regardless of how many edges actually exist, but answering "is there an edge between u and v?" in O(1) time. For the sparse graphs typical of real networks (where $|E|$ is much smaller than $|V|^2$), adjacency lists are almost always the better choice, and are what this chapter's algorithms use.

Representation	Space	"Is {u,v} an edge?"	Enumerate neighbors of v
Adjacency list	$\Theta(V + E)$	O(deg(v)) — scan v's list	$\Theta(\deg(v))$
Adjacency matrix	$\Theta(V^2)$	O(1) — direct lookup	$\Theta(V)$ — scan a full row

10.3 The Minimum Spanning Tree Problem and the Cut Property

Spanning tree, Minimum Spanning Tree

Given a connected, undirected, weighted graph G = (V, E), a SPANNING TREE is a subset of exactly $|V| - 1$ edges that connects all $|V|$ vertices with no cycles. A MINIMUM SPANNING TREE (MST) is a spanning tree whose total edge weight is as small as possible among all spanning trees of G.

Why exactly $|V| - 1$ edges? A tree on n vertices has exactly n − 1 edges — one fewer edge than vertices, since adding any n-th edge to an (n − 1)-edge tree must either connect two vertices already connected (creating a cycle) or the tree would already have covered all vertices with fewer edges (a contradiction for a connected structure). This chapter's running example — seven Profitina regional

scoring hubs and ten candidate direct links between them, each with an installation cost — needs exactly 7 − 1 = 6 accepted links to connect every hub at minimum total cost.

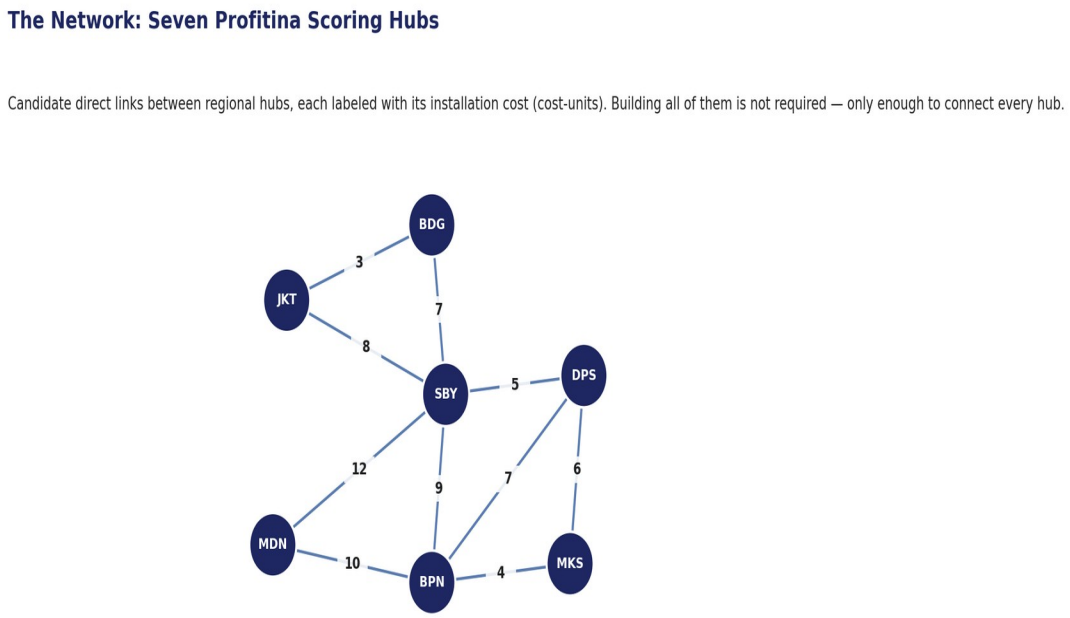

Figure 10.2 — The full candidate network: seven hubs, ten possible direct links, each labeled with its installation cost. Building all ten is unnecessary — the MST problem asks for the cheapest SUBSET that still connects every hub.

> **The cut property**
> For any way of partitioning a connected graph's vertices into two non-empty groups (a "cut"), the LIGHTEST edge crossing that cut — the cheapest edge with one endpoint in each group — belongs to SOME minimum spanning tree of the graph. This is the structural fact that makes greedy MST algorithms provably correct: at every step, whichever specific cut an algorithm happens to be considering, its cheapest crossing edge is always a safe choice.

10.4 Kruskal's Algorithm

Kruskal's algorithm applies the cut property in the most direct way possible: sort ALL edges by weight, and process them cheapest-first, accepting an edge if and only if its two endpoints are not already connected by previously-accepted edges (accepting it would create a cycle, which a tree can never contain). Detecting "are these two vertices already connected by accepted edges?" efficiently is exactly what the UNION-FIND (disjoint-set) data structure of Section 10.6 is for.

```
KRUSKAL(V, E, w):                  // V = vertices, E = edges, w = weight
function
  A = { }                           // accepted edges, will end up size |V| - 1
  for each vertex v in V
      MAKE-SET(v)                   // each vertex starts in its own component
  sort E into non-decreasing order by weight w
  for each edge (u, v) in sorted E
      if FIND-SET(u) != FIND-SET(v)  // u and v in different components?
          A = A union { (u, v) }
          UNION(u, v)               // merge the two components
  return A
```

Kruskal's Trace: Sorting Edges by Weight

Process edges cheapest-first. Accept an edge only if its two endpoints are still in different components (union-find) — reject it otherwise, since it would only create a cycle.

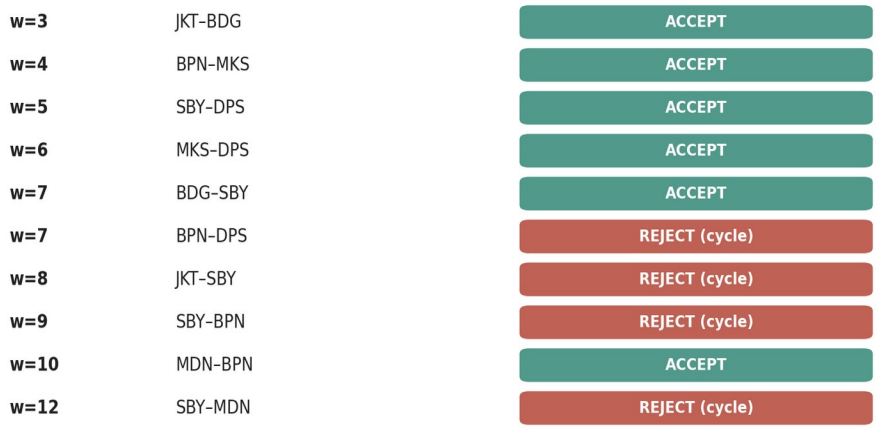

Figure 10.3 — Kruskal's trace on the ten candidate links of Figure 10.2, processed cheapest-first. Six edges are accepted (each merging two previously-separate components); four are rejected because their endpoints were already connected by earlier accepted edges.

Tracing Figure 10.3's data: JKT–BDG (w=3) is accepted — the first edge always is, since every vertex starts in its own component. BPN–MKS (w=4), SBY–DPS (w=5), and MKS–DPS (w=6) are each accepted in turn, each merging two components that were still separate. BDG–SBY (w=7) is accepted, merging the {JKT, BDG} component with the growing {SBY, DPS, MKS, BPN} component. The NEXT edge by weight, BPN–DPS (w=7), is REJECTED — both endpoints are already in the same component (the merge above connected them transitively) — accepting it would create a cycle. JKT–SBY (w=8) and SBY–BPN (w=9) are rejected for the same reason. MDN–BPN (w=10) is accepted, finally connecting the previously-isolated MDN. SBY–MDN (w=12), the last and most expensive edge, is rejected — MDN is already connected by the cheaper MDN–BPN edge. Six edges accepted, matching $|V| - 1 = 7 - 1 = 6$ exactly, for a total installation cost of 35 — matching Appendix 10.A's compiled output exactly.

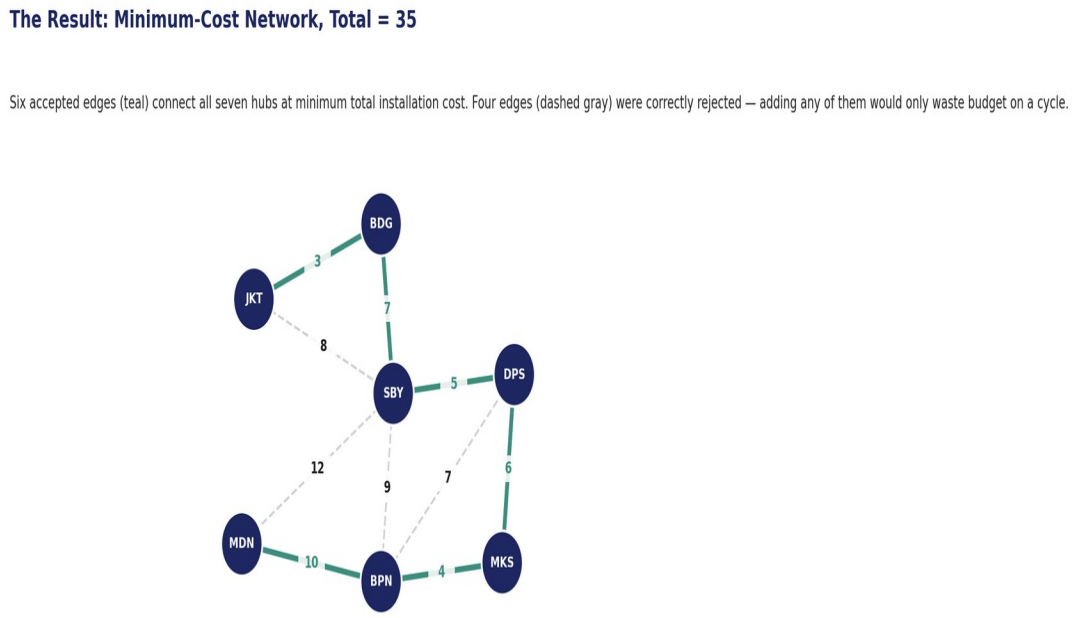

Figure 10.4 — The finished minimum spanning tree: the six accepted edges (teal) connect all seven hubs at total cost 35; the four rejected edges (dashed gray) would each only have added a redundant cycle.

10.5 Proving Kruskal Correct via the Cut Property

Kruskal's correctness follows from the cut property by induction on the number of edges accepted so far. Suppose the edges accepted so far form a subset of SOME minimum spanning tree (trivially true when zero edges have been accepted). Consider the next edge (u, v) that Kruskal is about to accept, because FIND-SET(u) ≠ FIND-SET(v). Let S be the set of vertices in u's current component, and V − S be everything else — this is a valid cut, since neither side is empty (v is on the other side, by assumption). Because Kruskal processes edges in increasing weight order and has not yet accepted any edge crossing this particular cut, (u, v) IS the lightest edge crossing it (any lighter crossing edge would already have been considered and, by the cycle-avoidance check, would have been accepted first). By the cut property, (u, v) belongs to some minimum spanning tree — so accepting it preserves the inductive invariant. Since the algorithm terminates with exactly |V| − 1 accepted edges forming a connected, cycle-free structure, and every one of those edges was shown safe to add, the final result is a genuine minimum spanning tree.

> **The parallel with Chapter 9's exchange argument**
> This proof has exactly the same shape as Section 9.4's exchange argument for Activity Selection: show that the greedy choice is always safe (belongs to SOME optimal solution), then show that what remains after the greedy choice is the SAME problem on a smaller input (optimal substructure), and combine the two via induction. The cut property plays the role the earliest-finish-time argument played in Chapter 9 — a problem-specific fact that makes the greedy choice provably safe.

10.6 The Union-Find (Disjoint-Set) Data Structure

Kruskal's algorithm needs one operation, repeated |E| times: "are these two vertices in the same component?", and one update, repeated |V| − 1 times: "merge these two components." The UNION-FIND data structure supports both efficiently. Each component is represented as a tree of elements pointing toward a shared root; FIND-SET(x) walks up from x to its root (identifying which component x belongs to), and UNION(x, y) attaches one root under the other. Two standard optimizations make this fast: PATH COMPRESSION (during FIND-SET, point every visited node directly at the root, flattening the tree for future queries) and UNION BY RANK (always attach the shallower tree under the deeper one's root, keeping trees flat in the first place). Together, these two optimizations bring the amortized cost of each operation down to O(α(n)) — where α is the inverse Ackermann function, a quantity so slowly growing that it is less than 5 for any input size that could ever be constructed in practice, making union-find operations effectively constant-time.

Phase 5 — Implement: union-find with path compression and union by rank

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

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

UNION(x, y):
  rx = FIND-SET(x);  ry = FIND-SET(y)
  if rx == ry:  return                  // already in the same component
  if rank[rx] < rank[ry]:  swap(rx, ry)  // union by rank
  parent[ry] = rx
  if rank[rx] == rank[ry]:  rank[rx] = rank[rx] + 1
```

10.7 Analyzing Kruskal's Running Time

Sorting |E| edges takes Θ(E log E) time. The main loop then runs |E| times, and each iteration performs a constant number of union-find operations, each costing O(α(V)) amortized — effectively O(1). Since a simple graph has $E = O(V^2)$, $\log E = O(2 \log V) = O(\log V)$, so the total running time is Θ(E log E) = Θ(E log V), which is DOMINATED entirely by the initial sort — exactly the same pattern Section 9.9 identified across every greedy algorithm in Chapter 9.

Step	Cost	Notes
Sort |E| edges by weight	Θ(E log E) = Θ(E log V)	Dominates the total running time
|V| MAKE-SET calls	Θ(V)	One per vertex, before the main loop
|E| FIND-SET / UNION calls	O(E · α(V)) ≈ O(E)	Amortized near-constant per call
Total	Θ(E log V)	Same shape as every Chapter 9 algorithm

10.8 Graph Algorithms in Practice: Where This Chapter Shows Up in a Real Product

> **About this box**
> As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

Profitina runs today as a single self-hosted server (FastAPI + PostgreSQL, behind a Cloudflare Tunnel) — not yet the distributed, multi-hub deployment this chapter's running example depicts. Its own public roadmap does name exactly this kind of expansion as a later stage (Phase 4: regional scale-up, once domestic dominance in Indonesia is secured), so the scenario below previews a real future decision rather than inventing an unrelated one. If and when that expansion happens, deciding which direct links between hubs to actually lease (each with a real monthly cost) to guarantee every hub can reach every other hub, at minimum total leased-link cost, would be a literal instance of the Minimum Spanning Tree problem, and Kruskal's algorithm (or Lecture 11's Prim's algorithm) is a completely realistic tool for that specific decision — though a production network would layer additional constraints (redundancy requirements, latency bounds) on top of the base MST that this chapter's algorithms alone do not capture. More abstractly, any time a platform like Profitina needs to identify the cheapest way to keep a set of resources (servers, data replicas, monitoring probes) mutually reachable without redundant, budget-wasting connections, the underlying shape of that decision is exactly this chapter's problem.

10.9 Chapter Summary and Key Takeaways

- A graph $G = (V, E)$ can be represented as an adjacency list ($\Theta(V+E)$ space, efficient neighbor enumeration) or an adjacency matrix ($\Theta(V^2)$ space, $O(1)$ edge lookup) — adjacency lists dominate for the sparse graphs typical of real networks.
- A spanning tree connects all $|V|$ vertices using exactly $|V| - 1$ edges and no cycles; a Minimum Spanning Tree (MST) is the spanning tree of least total weight.
- The cut property — the lightest edge crossing any cut belongs to some MST — is the structural fact that makes greedy MST algorithms provably correct, playing the same role Chapter 9's exchange arguments played for Activity Selection and Huffman Coding.
- Kruskal's algorithm sorts all edges by weight and greedily accepts each one that does not create a cycle, using the union-find data structure (with path compression and union by rank) to test that in effectively $O(1)$ amortized time per operation.
- Kruskal's correctness follows from the cut property by induction: at each step, the edge about to be accepted is shown to be the lightest edge crossing some valid cut, hence safe by the cut property.
- Kruskal's total running time, $\Theta(E \log V)$, is dominated by the initial sort of the edges — the same "sort dominates" shape found across every greedy algorithm in Chapter 9.

"A spanning tree touches every vertex; a MINIMUM spanning tree does it for the least money — and greedy, remarkably, always finds it."

— a course aphorism

(The cut property is what makes that remarkable fact provable, not just observed.)

Lecture 11 continues directly from this chapter: a second MST algorithm, Prim's, builds the SAME kind of tree via a different mechanism (growing a single component from a start vertex, using a min-priority-queue on key values, rather than merging many components via union-find) — but relies on exactly the same cut property to prove it correct, and reaches exactly the same total weight on any given graph, since the Minimum Spanning Tree of a graph with all-distinct edge weights is unique regardless of which algorithm is used to find it.

10.10 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Using the ten candidate links of Figure 10.2, show what happens if Kruskal's algorithm is run WITHOUT union-find — i.e., by checking connectivity via a full graph traversal (BFS/DFS) after each candidate edge instead. Does the final MST differ? Does the running time?
2. Prove, using only the definition of a tree (connected, acyclic), that any spanning tree on $|V|$ vertices has exactly $|V| - 1$ edges — do not just cite the fact from Section 10.3, derive it.
3. Suppose two edges in a graph have EQUAL weight, and both cross the same cut. Does the cut property guarantee a UNIQUE minimum spanning tree in that case? Explain, referencing the tie between BDG–SBY (w=7) and BPN–DPS (w=7) in Figure 10.3's trace.
4. Trace union-by-rank and path compression by hand for the sequence MAKE-SET(a), MAKE-SET(b), MAKE-SET(c), MAKE-SET(d), UNION(a,b), UNION(c,d), UNION(a,c), FIND-SET(d) — draw the resulting tree after each step.
5. Explain why Kruskal's algorithm, unlike Activity Selection, needs an explicit auxiliary data structure (union-find) to determine whether its greedy choice is safe, rather than a single running variable like last_finish.
6. If a new candidate edge is added to Figure 10.2's network with weight 1 between MDN and JKT, re-run Kruskal's algorithm by hand and state the new MST and its total weight.

Appendix 10.A — Verification Log for Chapter 10 Source Code

As in previous chapters, every program shipped with this book is compiled and run against hand-verified examples before being included. Both programs below were compiled with `g++ -O2 -std=c++17 -Wall` (zero warnings, zero errors) and produced the exact output predicted in the text above. The MST weight (35) was additionally cross-checked by an independent Python script performing exhaustive brute-force search over all C(10,6) = 210 six-edge subsets of the candidate network — see /tmp/daa_pilot/verify_ch10.py.

```
$ ./01_kruskal_mst
...
Kruskal trace:
  ACCEPT JKT-BDG (w=3)
  ACCEPT BPN-MKS (w=4)
  ACCEPT SBY-DPS (w=5)
  ACCEPT MKS-DPS (w=6)
  ACCEPT BDG-SBY (w=7)
  REJECT BPN-DPS (w=7)  -- cycle
  REJECT JKT-SBY (w=8)  -- cycle
  REJECT SBY-BPN (w=9)  -- cycle
  ACCEPT MDN-BPN (w=10)
  REJECT SBY-MDN (w=12)  -- cycle
MST total weight: 35            # matches Figure 10.3/10.4 exactly

$ ./02_prim_mst
Prim trace (start = JKT):
  START at JKT
  ADD BDG via edge JKT-BDG (w=3)
  ADD SBY via edge BDG-SBY (w=7)
  ADD DPS via edge SBY-DPS (w=5)
  ADD MKS via edge DPS-MKS (w=6)
  ADD BPN via edge MKS-BPN (w=4)
  ADD MDN via edge BPN-MDN (w=10)
MST total weight: 35            # matches Kruskal exactly (Lecture 11 detail)
```

SPOJ Practice — the companion book
This chapter's techniques are tested for real on the SPOJ online judge. Open Chapter 11 of the companion book “SPOJ Competitive Programming” (SPOJ folder, BI & EN) for the graph toolkit: representations, MST, and traversals — complete with the tips, tricks, and hints that get solutions Accepted, without spoiling answers.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Chapter 21, Data Structures for Disjoint Sets; Chapter 22, Elementary Graph Algorithms; Chapter 23, Minimum Spanning Trees).

[2] J. B. Kruskal Jr. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem." Proceedings of the American Mathematical Society, 7(1), 1956, 48–50. (The original paper.)

[3] M. A. Weiss. Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2014. (Chapter 9, Graph Algorithms; Chapter 8, The Disjoint Set Class.)

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

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 11

Prim's Algorithm for Minimum Spanning Trees

The second MST algorithm — same problem, same worked example, same cut property, but a completely different mechanism: growing one tree outward from a single vertex, one cheapest edge at a time, instead of merging many components by sorted edge weight.

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

(1) State the difference in mechanism between Kruskal's algorithm (merge many components) and Prim's algorithm (grow a single tree), while recognizing that both solve the identical MST problem. (2) Trace Prim's algorithm by hand on a weighted graph, including the key[]/parent[] arrays it maintains for every vertex outside the tree. (3) Prove Prim's algorithm correct using the cut property, adapting Chapter 10's proof to Prim's specific choice of cut at each step. (4) Implement Prim's algorithm using a min-priority queue keyed on key[], and state its running time under a binary heap versus a simple array. (5) Explain why Kruskal's and Prim's algorithms always agree on the total MST weight (and, when all edge weights are distinct, on the exact edge set) despite their different mechanisms. (6) Choose between Kruskal's and Prim's algorithm for a given graph based on its density.

11.1 Recap: From Kruskal to Prim

Chapter 10 introduced the Minimum Spanning Tree (MST) problem and the cut property — the structural fact that the lightest edge crossing any cut of a connected graph belongs to some MST — and used it to prove Kruskal's algorithm correct. Kruskal's mechanism was to sort every edge once, then repeatedly ask a GLOBAL question: "does this next-cheapest edge connect two different components?" This chapter presents a second, equally classical MST algorithm — Prim's algorithm — which applies the exact same cut property but asks a completely different, LOCAL question at every step: "of all the edges leaving the single tree I have built so far, which is cheapest?" Both algorithms are greedy; both are provably correct by the cut property; and on any given graph, both are guaranteed to find a spanning tree of the same minimum total weight.

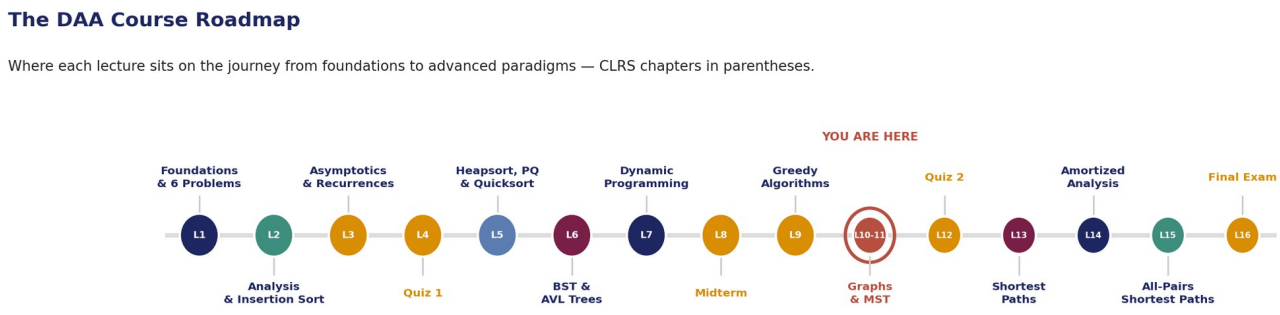

Figure 11.1 — Where this chapter sits in the sixteen-lecture DAA roadmap. Lecture 11 completes the two-lecture unit on Graphs and Minimum Spanning Trees begun in Chapter 10.

> **Same problem, same worked example, same answer — different mechanism**
> This chapter deliberately reuses Chapter 10's exact seven-hub Profitina network (Figure 10.2) rather than introducing a new one. That choice is pedagogical: watching two different algorithms reach the identical total cost of 35 on the identical input is the clearest possible demonstration that MST algorithms are correct because of a shared structural property (the cut property), not because of any one algorithm's particular bookkeeping.

11.2 Prim's Algorithm: Growing a Single Tree

Prim's algorithm maintains a single growing tree, starting from an arbitrary root vertex r, and at every step adds the cheapest edge that connects a vertex already IN the tree to a vertex not yet in it. To find that cheapest edge efficiently without re-scanning the entire graph at every step, Prim's algorithm maintains, for every vertex v not yet in the tree, a value key[v] — the weight of the cheapest edge seen so far connecting v to the tree — and a value parent[v] — which tree vertex that cheapest edge connects to. These values are stored in a min-priority queue Q keyed on key[], so the next vertex to add is always found with a single EXTRACT-MIN operation.

```
PRIM(V, E, w, r):                  // r = start vertex, tree grows outward from it
  for each vertex v in V
      key[v] = infinity;  parent[v] = NIL
  key[r] = 0
  Q = V                             // min-priority queue keyed by key[]
  while Q is not empty
      u = EXTRACT-MIN(Q)            // cheapest vertex not yet in the tree
      for each v in Adj[u]
          if v in Q and w(u, v) < key[v]
              parent[v] = u
              key[v] = w(u, v)
              DECREASE-KEY(Q, v, key[v])
```

Every time a vertex u is extracted, the algorithm relaxes each of u's edges: for a neighbor v still in Q, if the edge (u, v) is cheaper than v's current best-known connection to the tree, key[v] and parent[v] are updated. This is the same RELAXATION idea used throughout shortest-path algorithms (Chapter 13) — the only difference is that Prim's key[v] tracks the cheapest EDGE weight to the tree, not the cheapest PATH DISTANCE from a source.

11.3 Prim's Trace on the Candidate Network

Running Prim's algorithm on Figure 10.2's seven-hub network, starting at JKT, produces the trace in Figure 11.2 — independently hand-verified in /tmp/daa_pilot/verify_ch11_prim_trace2.py and cross-checked against the compiled output of Appendix 11.A's 01_prim_mst.cpp before being included here.

Prim's Trace: Growing One Tree from JKT

Unlike Kruskal, Prim never rejects an edge — it grows a SINGLE tree one vertex at a time, always picking the cheapest edge leaving the current tree. Verified against /tmp/daa_pilot/code/chapter11/01_prim_mst.cpp.

Step	Vertex Added	Edge Used	What Changed on the Frontier
1	START JKT	—	Start — JKT's key set to 0, all others ∞
2	ADD BDG	JKT–BDG (w=3)	Frontier was {BDG:3, SBY:8} — BDG is cheapest
3	ADD SBY	BDG–SBY (w=7)	BDG relaxed SBY's key from 8 down to 7
4	ADD DPS	SBY–DPS (w=5)	SBY relaxed DPS:5, BPN:9, MDN:12 — DPS cheapest
5	ADD MKS	DPS–MKS (w=6)	DPS relaxed MKS:6, BPN:7→ still 7 until next step
6	ADD BPN	MKS–BPN (w=4)	MKS relaxed BPN's key from 7 down to 4
7	ADD MDN	BPN–MDN (w=10)	BPN relaxed MDN's key from 12 down to 10 — done

Figure 11.2 — Prim's trace starting from JKT. Each row shows the vertex added, the tree edge used to add it, and how that addition relaxed (lowered) the key[] values of its still-outside neighbors.

Tracing Figure 11.2's data: the tree starts as just {JKT}, with key[JKT]=0. Extracting JKT relaxes BDG (key=3) and SBY (key=8). BDG has the smaller key, so it is extracted next, via edge JKT–BDG (w=3) — this relaxes SBY's key from 8 down to 7, since BDG–SBY (w=7) is now a cheaper connection than JKT–SBY (w=8). SBY is extracted next (key=7, via BDG), relaxing DPS (key=5), BPN (key=9), and MDN (key=12). DPS has the smallest key among all outside vertices, so it is extracted (via SBY–DPS, w=5), which relaxes MKS (key=6) and lowers BPN's key from 9 to 7 (via BPN–DPS, w=7). MKS is extracted next (key=6, via DPS), which relaxes BPN's key further, from 7 down to 4 (via BPN–MKS, w=4) — the cheapest connection BPN will ever get. BPN is extracted next (key=4, via MKS), relaxing MDN's key from 12 down to 10 (via BPN–MDN, w=10). Finally MDN is extracted (key=10, via BPN), and the queue is empty. Six edges were added — {JKT-BDG:3, BDG-SBY:7, SBY-DPS:5, DPS-MKS:6, MKS-BPN:4, BPN-MDN:10} — for a total weight of 35, matching Appendix 11.A's compiled output and Chapter 10's Kruskal result exactly.

The Result: Minimum-Cost Network, Total = 35

Six accepted edges (teal) connect all seven hubs at minimum total installation cost. Four edges (dashed gray) were correctly rejected — adding any of them would only waste budget on a cycle.

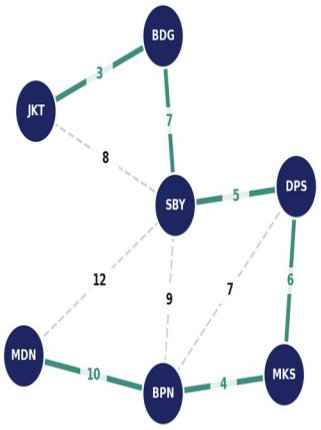

Figure 11.3 — The same finished minimum spanning tree Chapter 10 found via Kruskal's algorithm (Figure 10.4), now reached via Prim's algorithm instead. The edge set and total cost (35) are identical — only the ORDER in which the six edges were discovered differs.

11.4 Why Prim's Reaches the Same MST as Kruskal's

Figure 11.3 is not a coincidence: whenever a connected, weighted graph has all-DISTINCT edge weights (as Figure 10.2's network does — every one of its ten candidate links has a different installation cost), its minimum spanning tree is UNIQUE, regardless of which correct algorithm is used to find it. Kruskal's and Prim's algorithms consider edges in completely different orders and maintain completely different bookkeeping (sorted edge list plus union-find, versus a single tree plus a priority queue on key[]) — but because both are correct by the SAME cut property, and the underlying MST they are both searching for is the same unique object, they cannot help but arrive at it. (When some edge weights tie, the MST need not be unique — Review Question 3 explores this case directly.)

> **Uniqueness under distinct weights**
> If a connected, weighted graph has all distinct edge weights, it has EXACTLY ONE minimum spanning tree. Sketch of why: if two different spanning trees T1 and T2 both had the same minimum weight, some edge in T1 not in T2 could be exchanged with some edge in T2 not in T1 across the cut the exchange creates — and because all weights are distinct, one of the two resulting trees would have to be strictly cheaper than assumed minimum, a contradiction.

11.5 Proving Prim Correct via the Cut Property

Prim's correctness follows from the cut property by induction on the size of the growing tree, in a proof with exactly the same shape as Chapter 10's Kruskal proof — but applied to a DIFFERENT cut at each step. Suppose the edges accepted so far form a subset of some minimum spanning tree (trivially true when the tree contains only the root r). Let S be the set of vertices currently in the tree, and V − S be everything else — this is always a valid cut, since S is never empty (it contains at least r) and, until the algorithm finishes, V − S is never empty either. The next edge Prim is about to accept is, by construction of key[] and EXTRACT-MIN, the CHEAPEST edge with one endpoint in S and one endpoint in V − S — in other words, exactly the lightest edge crossing this particular cut. By the cut property, that edge belongs to some minimum spanning tree, so accepting it preserves the inductive invariant. Since the algorithm terminates with S = V and exactly |V| − 1 accepted edges forming a connected, cycle-free structure (each new vertex is attached by exactly one edge), the final result is a genuine minimum spanning tree.

The one sentence that separates Kruskal from Prim

Kruskal considers a DIFFERENT cut at each step — implicitly, the cut between whichever two components the next sorted edge would merge. Prim considers the SAME KIND of cut at every step — always between the tree built so far (S) and everything else (V − S), which is why Prim's tree is always connected at every intermediate stage, while Kruskal's accepted edges can form several disconnected pieces until near the very end.

11.6 Implementing the Priority Queue

Prim's running time depends entirely on how the min-priority queue Q is implemented, since EXTRACT-MIN is called exactly |V| times (once per vertex added to the tree) and DECREASE-KEY is called at most |E| times (once per edge relaxation).

Phase 5 — Implement: Prim's algorithm with a binary min-heap

```
PRIM-HEAP(V, E, w, r):
  key[r] = 0;  for all other v: key[v] = infinity
  build a binary min-heap Q over V, keyed by key[]      // O(V)
  while Q is not empty
      u = HEAP-EXTRACT-MIN(Q)                          // O(log V)
      mark u as in the tree
      for each (v, w(u,v)) in Adj[u]
          if v in Q and w(u, v) < key[v]
              key[v] = w(u, v);  parent[v] = u
              HEAP-DECREASE-KEY(Q, v, key[v])           // O(log V)
```

With a binary heap, building the initial heap costs O(V), the |V| EXTRACT-MIN calls cost O(V log V) total, and the O(E) DECREASE-KEY calls cost O(E log V) total — giving an overall running time of O((V + E) log V), which simplifies to O(E log V) for any connected graph (where E ≥ V − 1). This is the SAME asymptotic bound as Kruskal's Θ(E log V) — for sparse graphs, the two algorithms are essentially tied. A simpler array-based priority queue, where EXTRACT-MIN scans all V entries in O(V) and DECREASE-KEY is a O(1) array write, gives $O(V^2)$ total time instead — worse for sparse graphs, but BETTER for dense graphs where E approaches V^2, since $O(V^2)$ then beats $O(E \log V) = O(V^2 \log V)$.

11.7 Analyzing Prim's Running Time

Priority queue implementation	EXTRACT-MIN cost	DECREASE-KEY cost	Total running time
Simple array	O(V) each, $O(V^2)$ total	O(1) each, O(E) total	$O(V^2)$ — best for DENSE graphs
Binary min-heap	O(log V) each, O(V log V) total	O(log V) each, O(E log V) total	O(E log V) — matches Kruskal
Fibonacci heap	O(log V) amortized	O(1) amortized	O(E + V log V) — best in theory

Priority queue implementation	EXTRACT-MIN cost	DECREASE-KEY cost	Total running time

Density decides the winner

For a SPARSE graph (E close to V), the binary-heap version of Prim and Kruskal's Θ(E log V) are asymptotically tied — pick whichever is easier to implement or better suited to the data structure already on hand. For a DENSE graph (E close to V^2), the simple-array version of Prim's $O(V^2)$ beats both — it avoids paying a logarithmic factor on a huge number of edges. Kruskal's algorithm has no equivalent dense-graph speedup, since it must still sort all E edges regardless of density.

11.8 Profitina in Practice: Choosing an MST Algorithm

About this box

As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

Chapter 10's hypothetical Phase-4 scenario modeled a future Profitina regional-hub network as a sparse graph — a handful of hubs with a handful of candidate links, exactly the setting where Kruskal's algorithm (or the binary-heap version of Prim's) is the natural choice. But not every MST-shaped decision in that scenario would be sparse: when evaluating which of many redundant monitoring-probe pairs to keep fully interconnected within a single data center — the kind of dense, local decision Profitina's actual single-server deployment already faces today — the candidate-link graph can be nearly COMPLETE (every probe can technically reach every other probe directly), pushing E close to V^2. In that denser regime, the simple-array version of Prim's algorithm — asymptotically worse on paper for sparse inputs, but $O(V^2)$ regardless of edge count — becomes the more efficient real choice, illustrating why Section 11.7's density-based comparison is not just a textbook exercise but an actual engineering decision.

11.9 Chapter Summary and Key Takeaways

- Prim's algorithm grows a SINGLE tree outward from a starting vertex, always adding the cheapest edge leaving the tree so far — a fundamentally different mechanism from Kruskal's sort-and-merge approach, but solving the identical MST problem.
- Prim's algorithm maintains key[v] and parent[v] for every vertex outside the tree, stored in a min-priority queue, and relaxes these values whenever a newly-added vertex offers a cheaper connection.
- On Chapter 10's seven-hub network, Prim's algorithm (starting at JKT) reaches the identical minimum spanning tree — same six edges, same total weight 35 — as Kruskal's algorithm, because the network's edge weights are all distinct and its MST is therefore unique.

- Prim's correctness follows from the same cut property as Kruskal's, but applied to a different cut at each step: always the cut between the tree built so far and everything else.
- With a binary min-heap, Prim's algorithm runs in O(E log V) — matching Kruskal's Θ(E log V) for sparse graphs. With a simple array, it runs in $O(V^2)$ — the better choice for dense graphs, where Kruskal's mandatory edge sort has no equivalent speedup.
- Choosing between Kruskal's and Prim's algorithm in practice is a density decision, not a correctness decision — both are always correct, by the same underlying cut property.

"Kruskal asks every edge to audition in order of price. Prim asks only the edges touching what it has already built — same destination, a completely different walk."

— a course aphorism

(Both are safe by the same cut property — they simply choose which cut to look at differently.)

Lecture 12 is a Quiz 2 preparation session, reviewing Lectures 9 through 11 (Greedy Algorithms, and Graphs and Minimum Spanning Trees) with practice questions rather than new content. Lecture 13 then turns to a related but distinct graph problem — SHORTEST PATHS — where Dijkstra's algorithm reuses Prim's exact priority-queue-plus-relaxation mechanism, but tracks cheapest PATHS from a source rather than the cheapest TREE connecting every vertex.

11.10 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Trace Prim's algorithm on Figure 10.2's network starting from a DIFFERENT vertex — say, MKS instead of JKT. Does the total MST weight change? Does the exact sequence of key[] updates?
2. Explain, in your own words, why Prim's tree is always connected at every intermediate step, while Kruskal's accepted edges can form several disconnected pieces until near the end. Why does this difference not affect either algorithm's correctness?
3. Section 11.4 claimed that all-distinct edge weights guarantee a unique MST. Construct a small graph (4–5 vertices) with two EQUAL-weight edges positioned so that Kruskal's and Prim's algorithms produce two different, equally-minimum spanning trees.
4. Using Figure 11.2's trace, identify every point where a vertex's key[] value was updated MORE THAN ONCE before that vertex was finally extracted. What does each such update represent in terms of the cut property?
5. Compare the space complexity of Kruskal's algorithm (which needs a sorted edge list and a union-find structure) against Prim's algorithm (which needs a priority queue and

key[]/parent[] arrays). Which uses more auxiliary space, and does it depend on graph density?

6. A colleague claims that Prim's algorithm could be modified to find a MAXIMUM spanning tree just by using EXTRACT-MAX instead of EXTRACT-MIN. Is this claim correct? Justify your answer using the cut property.

Appendix 11.A — Verification Log for Chapter 11 Source Code

As in previous chapters, every program shipped with this book is compiled and run against hand-verified examples before being included. 01_prim_mst.cpp (identical in this chapter's Code/Chapter11 folder to Chapter 10's copy, since both chapters trace the same algorithm on the same network) was compiled with `g++ -O2 -std=c++17 -Wall` (zero warnings, zero errors) and produced the exact output predicted in Figure 11.2 and Section 11.3's narration. The key[]-value progression at each extraction was additionally cross-checked by an independent Python simulation — see /tmp/daa_pilot/verify_ch11_prim_trace2.py — which matched the compiled output exactly at every one of the seven steps.

```
$ ./01_prim_mst
Prim trace (start = JKT):
  START at JKT
  ADD BDG via edge JKT-BDG (w=3)
  ADD SBY via edge BDG-SBY (w=7)
  ADD DPS via edge SBY-DPS (w=5)
  ADD MKS via edge DPS-MKS (w=6)
  ADD BPN via edge MKS-BPN (w=4)
  ADD MDN via edge BPN-MDN (w=10)

MST edges (6):
  JKT-BDG (w=3)
  BDG-SBY (w=7)
  SBY-DPS (w=5)
  DPS-MKS (w=6)
  MKS-BPN (w=4)
  BPN-MDN (w=10)
MST total weight: 35        # matches Kruskal (Chapter 10) and Figure 11.3
exactly
```

SPOJ Practice — the companion book
This chapter's techniques are tested for real on the SPOJ online judge. Open Chapter 11 of the companion book “SPOJ Competitive Programming” (SPOJ folder, BI & EN) for MST implementations that pass the judge — complete with the tips, tricks, and hints that get solutions Accepted, without spoiling answers.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Chapter 23, Minimum Spanning Trees, Section 23.2, The algorithms of Kruskal and Prim).

[2] R. C. Prim. "Shortest Connection Networks and Some Generalizations." Bell System Technical Journal, 36(6), 1957, 1389–1401. (The original paper.)

[3] M. A. Weiss. Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2014. (Chapter 9, Graph Algorithms — Prim's Algorithm.)

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

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 12 · EXERCISE CHAPTER

Practice Problems: Greedy Algorithms Through Minimum Spanning Trees

No new material here — just twelve problems designed to make sure Lectures 9 through 11 actually stuck, before Quiz 2.

Learning Objectives — after working through this chapter you will be able to:
(1) Apply a greedy strategy to a new instance of activity selection and justify the exchange-argument reasoning behind it. (2) Build a Huffman tree from a set of frequencies by hand and compute its bit savings against fixed-length encoding. (3) Solve fractional knapsack by hand, and construct a concrete instance where greedy fails on the 0/1 variant. (4) Trace Kruskal's algorithm on a new graph, including the union-find operations that detect cycles. (5) Trace Prim's algorithm on a new graph from a specified start vertex, and confirm it reaches the same total weight as Kruskal's. (6) Determine whether a minimum spanning tree is unique for a given graph, and reason about the cut property directly.

12.1 Recap: Lectures 9–11 in Brief

Lecture 9 introduced an entirely new design paradigm after seven lectures of divide-and-conquer and dynamic programming: greedy algorithms, which build a solution by making a sequence of locally optimal choices, never revisited, and prove correctness with an exchange argument rather than a recurrence. Activity Selection (choose by earliest finish time), Huffman coding (repeatedly merge the two least-frequent nodes into an optimal prefix-free binary code), and the Fractional Knapsack problem (fill capacity by value-to-weight ratio) all fall to this paradigm — while the 0/1 Knapsack problem, deceptively similar on the surface, actively defeats it, requiring dynamic programming instead. Lectures 10 and 11 then applied greedy reasoning to graphs: the Minimum Spanning Tree problem, proved correct via the cut property (the cheapest edge crossing any cut belongs to some MST), is solved by two structurally different greedy algorithms that always agree on the answer whenever edge weights are distinct. Kruskal's algorithm (Lecture 10) sorts every edge once and merges components with a union-find data structure, considering a DIFFERENT cut at every step; Prim's algorithm (Lecture 11) grows a single tree outward from one vertex using a key[]/parent[] array and a min-priority queue, considering the SAME cut — tree built so far versus everything else — at every step.

The DAA Course Roadmap

Where each lecture sits on the journey from foundations to advanced paradigms — CLRS chapters in parentheses.

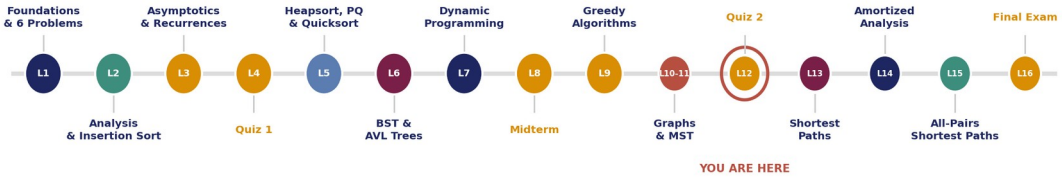

Figure 12.1 — This chapter sits at Lecture 12 in the sixteen-lecture DAA roadmap: a checkpoint, not a new topic, immediately before Quiz 2.

12.2 How to Use This Exercise Chapter

> **What this chapter is — and is not**
>
> This is a Bab Latihan: a practice chapter, not a new-content chapter. Every problem below is anchored to specific material from Lectures 9–11 (see Figure 12.2). Each problem comes with a HINT — a nudge toward the right way of thinking — but deliberately NOT a full worked solution or source code. Working through the problem yourself, even imperfectly, teaches far more than reading a finished answer.

Review Map: From Lectures 9–11 to Quiz 2

Every exercise problem in this chapter is anchored to a specific lecture's material — none of it is new content.

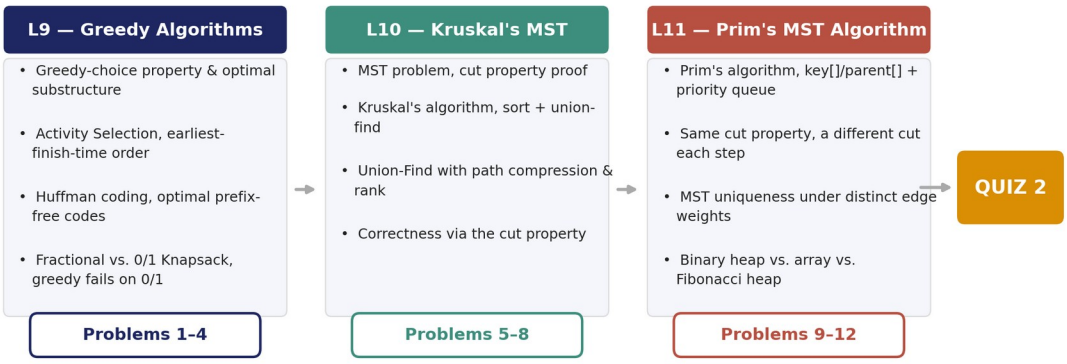

Figure 12.2 — Every problem in Section 12.3 traces back to one of the previous three lectures.

> **Before you start**
>
> Try each problem for real before reading its hint. If you get stuck, read only the hint — not a solution — and try again. This is exactly the discipline Section 12.4's open-book Quiz 2 will reward: the quiz allows references, but not a substitute for your own reasoning.

12.3 Practice Problems

12.3.1 Greedy Algorithms

Problem 1 *[Easy]*. Six candidate activities have (start, finish) times: A(1,4), B(3,5), C(2,6), D(5,7), E(6,8), F(7,9). Apply the greedy Activity Selection algorithm (sort by finish time, always pick the next activity compatible with the last one accepted) and list the FULL set of activities selected, in order.

> **Hint**
> Sort strictly by finish time first — A already has the earliest finish. An activity is compatible with the last one accepted only if its start time is greater than or equal to the last accepted activity's finish time.

Problem 2 *[Medium]*. A source has six symbols with frequencies a=2, b=3, c=5, d=8, e=13, f=21 (deliberately Fibonacci-like). Build the Huffman tree by hand (repeatedly merging the two lowest-frequency nodes) and state the resulting codeword length for each symbol. Compare the total encoded length against a fixed 3-bit code for all six symbols, and explain why THIS PARTICULAR frequency pattern produces an unusually lopsided ("caterpillar-shaped") Huffman tree rather than a balanced one.

> **Hint**
> At every merge step there may be a tie for "two lowest frequencies" — any tie-breaking rule gives the same TOTAL cost, though the tree's exact shape can differ. Watch what happens to the largest frequency (21): does it ever get merged before the very last step?

Problem 3 *[Medium]*. A Fractional Knapsack has capacity 15. Four items (weight, value) are available: I1(5,10), I2(3,9), I3(8,8), I4(6,12). Compute each item's value-to-weight ratio, decide the greedy fill order, and state the maximum total value achievable (including any fractional item taken).

> **Hint**
> Sort strictly by value/weight ratio, descending. Take whole items greedily until the NEXT item would exceed the remaining capacity — then take only the fraction of it that fits, and stop.

Problem 4 *[Hard]*. A 0/1 Knapsack has capacity 10. Four items (weight, value) are available: X(6,30), Y(5,24), Z(5,24), W(2,9). Compute the greedy-by-ratio 0/1 solution (take whole items in ratio order, skipping any that no longer fit) and compare it against the TRUE optimal 0/1 solution. Show that greedy is strictly worse here, and explain in your own words WHY taking the single highest-ratio item can block a better combination.

> **Hint**
> Rank the four items by value/weight ratio first — X has the single highest ratio, but taking it whole consumes 6 of the 10 units of capacity. Ask what combination of the OTHER three items could have filled that same capacity instead, and compare total values.

12.3.2 Graphs & Kruskal's Minimum Spanning Tree

Problem 5 *[Easy]*. A network of six candidate hubs P, Q, R, S, T, U has candidate links (with cost): P–Q(4), P–R(2), Q–R(1), Q–S(5), R–S(8), R–T(10), S–T(2), S–U(6), T–U(3). A colleague proposes the spanning tree {P–Q, Q–R, Q–S, S–T, T–U} with total cost 4+1+5+2+3=15. Is this an MST? If not, name ONE specific edge you would swap out, and the edge you would swap in, to strictly reduce the total cost — and state the new total.

> **Hint**
> A spanning tree is minimum only if NO single edge swap can lower its cost. Check whether replacing P–Q with a cheaper edge that still connects P to the rest of the tree is possible — the cut property tells you exactly which edge crossing a given cut is safest to prefer.

Problem 6 *[Medium]*. Using the same six-hub network from Problem 5, trace Kruskal's algorithm completely: list every candidate edge in sorted order, and for each one state ACCEPT or REJECT (with the reason, if rejected). State the final MST edge set and its total weight.

> **Hint**
> Process edges strictly by ascending weight. An edge is rejected if and only if its two endpoints are already in the same union-find component — accepting it would create a cycle, not a tree.

Problem 7 *[Medium]*. Continuing Problem 6: show the union-find forest's parent pointers after EVERY accepted union operation (assume union-by-rank, and that a fresh singleton set starts with rank 0 for each of P, Q, R, S, T, U). State which vertex ends up as the root of the single final component.

> **Hint**
> Union-by-rank attaches the LOWER-rank tree's root under the HIGHER-rank tree's root; when ranks are equal, either root may become the new root but its rank increases by one. Track this decision explicitly at every accepted union — the specific root that survives depends on the tie-breaking convention you choose, so state your convention.

Problem 8 *[Hard]*. Using the MST you found in Problem 6, prove — using the cut property directly, not by exhaustive search — that the single cheapest edge in the ENTIRE network (Q–R, weight 1) MUST belong to every minimum spanning tree of this graph, regardless of which algorithm is used to find one.

> **Hint**
> Consider the cut that separates {Q} from all five other vertices. Among all edges crossing this cut, is Q–R strictly the cheapest? The cut property states that the minimum-weight edge crossing ANY cut is safe to include in some MST — but here you need the stronger claim that it belongs to EVERY MST. Think about what would go wrong if some other MST excluded it.

12.3.3 Prim's Minimum Spanning Tree Algorithm

Problem 9 *[Medium]*. Using the same six-hub network from Problem 5, trace Prim's algorithm starting from vertex U (not P) — show the key[] value and parent[] pointer for every not-yet-added vertex at

each step, in the style of Section 11.3's trace table. State the final MST edge set and total weight, and confirm it matches Problem 6's Kruskal result.

> **Hint**
> Initialize key[U]=0 and everything else to infinity. At each step, extract the not-yet-added vertex with the smallest key[], add it to the tree, then relax every edge leaving it — exactly as PRIM does in Section 11.2, just starting from a different vertex.

Problem 10 *[Medium]*. A four-hub network W, X, Y, Z has candidate links: W–X(2), X–Y(3), Y–Z(2), Z–W(3), W–Y(5). Find the minimum spanning tree's total weight, then determine whether the MST is UNIQUE. If it is not, exhibit TWO different edge sets that both achieve the minimum total weight.

> **Hint**
> Notice that two pairs of edges tie in weight here (X–Y and Z–W both cost 3). Section 11.4 established that all-distinct edge weights guarantee a unique MST — this graph deliberately breaks that precondition. Try running Kruskal's algorithm with the tie broken BOTH ways and compare the two resulting trees.

Problem 11 *[Hard]*. In a hypothetical future scenario matching Profitina's own publicly stated Phase-4 regional-expansion roadmap (not a description of its current single-server deployment), Profitina is considering upgrading its monitoring-probe network from 20 probes with 30 candidate links (sparse, E close to V) to a fully-interconnected data center topology with 20 probes and 190 candidate links (dense, E close to V(V−1)/2). For EACH scenario, state which Prim implementation (simple array, $O(V^2)$, or binary min-heap, $O(E \log V)$) has the smaller asymptotic running time, showing the arithmetic that leads to your answer.

> **Hint**
> Compute V^2 for V=20, and compare it against $E \log_2 V$ for each scenario's actual E. The crossover point between when array-based and heap-based Prim is faster depends on exactly how E compares to V log V, not just on whether the graph 'feels' dense or sparse.

Problem 12 *[Hard]*. A new problem: you are given an undirected, connected, weighted graph and told that it has AT LEAST TWO edges of equal weight. Does this fact ALONE guarantee the graph has more than one minimum spanning tree? Construct a small example (4 or 5 vertices) where a graph has a tied pair of edge weights but STILL has a unique MST, to support your answer.

> **Hint**
> A tie only threatens uniqueness if BOTH tied edges are simultaneously eligible to be the safe edge for the SAME cut at some point in Kruskal's execution. If the tied edges never compete for the same cut — for instance, if one of them would create a cycle by the time it's considered regardless of the other — uniqueness can survive the tie. Problem 10's example shows a tie that DOES break uniqueness; you need one that doesn't.

12.4 Quiz 2 Format: Open-Book, Take-Home Essay

Quiz 2 is an open-book, take-home essay assignment covering exactly the material reviewed in this chapter (Lectures 9 through 11) plus everything from before it — nothing beyond it. You will have one week to submit written answers with full reasoning; a correct final answer with no justification earns little credit, since the point of an essay-format exam is to see HOW you think, not just what you conclude.

Open-book means references, not co-authors

You may consult the textbook, your notes, and lecture slides while answering. You may NOT collaborate with classmates or submit reasoning that is not genuinely your own — an open-book exam tests whether you can APPLY what you have studied, unsupervised, which is exactly what "designing and analyzing algorithms" means in practice. If you quote a source directly, cite it.

Criterion	What it looks for	Weight
Correctness of reasoning	Is the logic, proof, trace, or derivation actually valid — not just the final answer?	40%
Clarity of proof / derivation	Can a reader follow each step without having to guess what you meant?	25%
Proper use of notation	Are O/Ω/Θ, cut-property arguments, and graph/tree diagrams stated precisely, matching earlier chapters' conventions?	20%
Presentation	Is the answer organized, legible, and easy to grade?	15%

12.5 Profitina in Practice: Self-Review Before Submission

About this box

As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

Before a new algorithm change ships to production at Profitina, engineers run it through a short self-review checklist — very much like Appendix 12.A below — before ever requesting a second pair of eyes: has a claimed greedy-choice property actually been justified with an exchange argument, or just asserted? Has a candidate minimum spanning tree actually been checked against the cut property at the specific edge in question, rather than assumed correct because the algorithm that produced it is trusted? This habit of adversarial self-checking before external review — the same discipline behind the bug-hunting lesson repeated throughout this textbook, that a visually clean result can still be mathematically wrong — is exactly what an open-book take-home exam format is built to train.

12.6 Chapter Summary

- This chapter introduced no new material — it is a checkpoint covering Lectures 9 through 11.
- Twelve problems span the full range reviewed: greedy Activity Selection, Huffman coding, Fractional versus 0/1 Knapsack, Kruskal's algorithm with union-find, Prim's algorithm from an arbitrary start vertex, and MST uniqueness under tied edge weights.
- Each problem includes a hint, not a solution — working through the reasoning yourself is the point.
- Quiz 2 is an open-book, take-home essay: references are allowed, but the reasoning must be your own, and correctness of REASONING (not just the final answer) is the majority of the grade.

"A minimum spanning tree has only one right answer when the weights refuse to tie — two algorithms, one destination."

— a course aphorism

(Rereading a proof doesn't count as knowing it — reproducing it from scratch does.)

Appendix 12.A — Self-Check Checklist (Non-Graded)

Before submitting any answer — in this chapter's problems or in Quiz 2 itself — run it through this checklist. It costs a few minutes and catches most of the mistakes graders see most often.

- Did I actually verify a claimed exchange argument or cut-property claim on the concrete example given, rather than just asserting it holds?
- In any hand-traced structure (Huffman tree, union-find forest, Prim's key[]/parent[] table), did I show EVERY intermediate state, not just the final result?
- If I claimed an MST is unique (or is not), did I check specifically whether tied edge weights actually compete for the same cut — not just whether a tie exists somewhere in the graph?
- If I claimed greedy fails on a 0/1 Knapsack instance, did I show BOTH the greedy answer AND a strictly better alternative, with actual numbers?
- Did I distinguish Kruskal's per-step cut (which changes every step) from Prim's per-step cut (always tree-so-far versus the rest) wherever the two algorithms' correctness proofs came up?
- Did I reread my own answer as if I were a skeptical grader looking for a gap in the argument?

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Chapters 15–16, 21, 23).

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

[3] M. A. Weiss. Data Structures and Algorithm Analysis in C++, 4th ed. Pearson, 2014. (Chapter 9, Graph Algorithms).

[4] R. C. Prim. "Shortest Connection Networks and Some Generalizations." Bell System Technical Journal, 36(6), 1957, 1389–1401.

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 13

Single-Source Shortest Paths: Dijkstra's Algorithm and Bellman-Ford

A new graph question — not "what is the cheapest way to connect everything" (MST), but "what is the cheapest way to get from HERE to everywhere else." Two algorithms, one shared mechanism (relaxation), and one crucial dividing line: whether negative edge weights are allowed.

Learning Objectives — after this chapter you will be able to:
(1) State the single-source shortest-path problem and explain the RELAXATION operation shared by every algorithm in this chapter. (2) Trace Dijkstra's algorithm by hand on a directed, non-negatively-weighted graph, including its dist[]/parent[] arrays and its min-priority-queue order of finalization. (3) Explain WHY Dijkstra's greedy finalize-and-never-revisit strategy requires non-negative edge weights, and construct an example where it fails on a negative edge. (4) Trace the Bellman-Ford algorithm by hand, including its pass-by-pass relaxation of every edge over |V|−1 rounds. (5) Explain how Bellman-Ford detects a negative-weight cycle using its |V|-th pass, and why Dijkstra has no equivalent safeguard. (6) State and compare the running times of Dijkstra's algorithm (with a binary heap) and Bellman-Ford, and choose the correct algorithm for a given graph's weight constraints.

13.1 Recap: From One Tree to Many Paths

Chapters 10 and 11 solved the Minimum Spanning Tree problem: given a connected, weighted graph, find the cheapest set of edges that touches every vertex. This chapter turns to a related but genuinely different question, called the SINGLE-SOURCE SHORTEST-PATHS problem: given a directed, weighted graph and a designated source vertex s, find the cheapest PATH from s to every other vertex. An MST cares about connecting everything as cheaply as possible in aggregate; a shortest-path tree cares about each individual destination's cheapest ROUTE from the source — and, as this chapter will show, those two goals can produce entirely different trees on the very same graph.

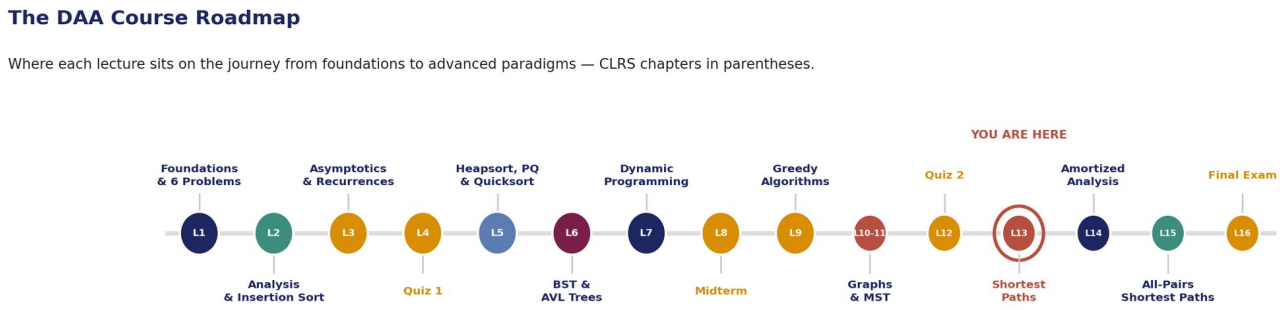

Figure 13.1 — Where this chapter sits in the sixteen-lecture DAA roadmap. Lecture 13 opens the shortest-paths unit, immediately after the Quiz 2 checkpoint on Greedy Algorithms and MST.

The shared idea underneath both algorithms: relaxation

Every shortest-path algorithm in this book — Dijkstra's, Bellman-Ford's, and Chapter 15's Floyd-Warshall — is built from the SAME primitive operation: given a candidate edge (u, v) with weight w, RELAX it by checking whether going through u offers a cheaper route to v than the best route to v found so far. If dist[u] + w < dist[v], update dist[v] and remember u as v's new predecessor. The algorithms in this chapter differ only in WHICH edges they relax, and in WHAT ORDER — not in what relaxation itself means.

13.2 The Shortest-Path Problem and Optimal Substructure

Formally: given a directed graph G = (V, E) with edge-weight function w, and a source vertex s, we want dist[v] — the weight of the cheapest path from s to v — for every vertex v reachable from s, plus enough parent information to reconstruct any of those paths. Shortest paths exhibit OPTIMAL SUBSTRUCTURE: if the shortest path from s to v passes through some intermediate vertex u, then the portion of that path from s to u must itself be a shortest path from s to u — otherwise, splicing in a cheaper s-to-u path would produce an even cheaper s-to-v path, a contradiction. This optimal-substructure property is exactly what makes relaxation a valid strategy: building up dist[] correctly for vertices closer to s first guarantees correctness for vertices further away.

A subtlety: what does "shortest" even mean with negative weights?

If a graph contains a NEGATIVE-WEIGHT CYCLE reachable from s (a cycle whose total weight sums to less than zero), then "shortest path" is undefined for any vertex reachable through that cycle — you could loop the cycle arbitrarily many times to make the path arbitrarily cheaper, with no lower bound. Dijkstra's algorithm simply assumes this never happens (by requiring all weights non-negative, ruling out negative cycles automatically). Bellman-Ford makes no such assumption — it tolerates negative edges, but must explicitly DETECT a negative cycle if one exists, since no well-defined shortest path is possible in its presence (Section 13.7).

13.3 Dijkstra's Algorithm: Greedy Finalization

Dijkstra's algorithm solves the single-source shortest-paths problem when every edge weight is non-negative, using a strategy that will look familiar from Chapter 11: maintain a min-priority queue keyed on tentative distances, repeatedly extract the vertex with the smallest tentative distance, and relax all of its outgoing edges. The crucial greedy claim — proved in Section 13.4 — is that once a vertex is extracted, its distance is FINAL and will never be improved again, so each vertex is processed exactly once.

```
DIJKSTRA(V, E, w, s):
  for each vertex v in V
      dist[v] = infinity;  parent[v] = NIL
  dist[s] = 0
  Q = V                      // min-priority queue keyed by dist[]
  S = empty set              // vertices whose dist[] is finalized
  while Q is not empty
      u = EXTRACT-MIN(Q)     // smallest tentative distance
      S = S union {u}
      for each v in Adj[u]
          RELAX(u, v, w)     // if dist[u]+w(u,v) < dist[v], update
```

Figure 13.2 shows a directed, five-hub Profitina network — one-way data-synchronization latency, in milliseconds, between regional hubs — with every edge weight non-negative, exactly the condition Dijkstra requires.

One Source, Five Hubs: The Dijkstra Network

A directed graph of one-way data-sync latency (ms) between Profitina hubs, ALL non-negative — exactly the condition Dijkstra requires. Source = JKT.

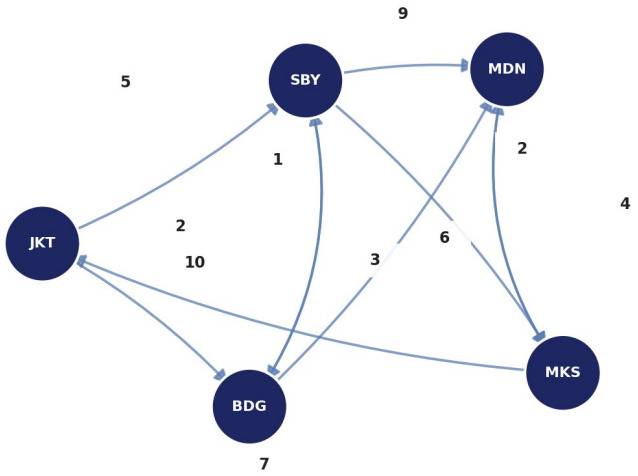

Figure 13.2 — A directed graph of one-way sync latency (ms) between five hubs of the hypothetical Profitina Phase-4 scenario. Every weight is non-negative. Source = JKT.

Running Dijkstra's algorithm on Figure 13.2 starting at JKT produces the trace in Figure 13.3 — independently hand-verified in /tmp/daa_pilot/verify_ch13_problems.py against a from-scratch brute-force all-simple-paths search AND the well-known published answer for this graph (this is the classic example from Cormen, Leiserson, Rivest, and Stein's textbook, renamed to Profitina hubs for continuity with earlier chapters), then cross-checked against the compiled output of Appendix 13.A's 01_dijkstra.cpp.

Dijkstra's Trace: Finalizing One Hub at a Time

The priority queue always pops the hub with the smallest tentative distance. Once popped, that distance is FINAL and never changes again. Verified against /tmp/daa_pilot/code/chapter13/01_dijkstra.cpp.

Step	Hub Finalized	dist[]	Edges Relaxed From Here
1	JKT	0	JKT→BDG: dist[BDG]=10; JKT→SBY: dist[SBY]=5
2	SBY	5	SBY→BDG: dist[BDG] 10→8; SBY→MDN: dist[MDN]=14; SBY→MKS: dist[MKS]=7
3	MKS	7	MKS→MDN: dist[MDN] 14→13 (still not final)
4	BDG	8	BDG→MDN: dist[MDN] 13→9 — beats MKS's offer
5	MDN	9	no outgoing edges left to relax — all 5 hubs finalized

Figure 13.3 — Dijkstra's step-by-step trace from JKT. Each row shows the hub finalized (permanently assigned its dist[] value) and which outgoing edges it relaxed.

Tracing Figure 13.3's data: starting with dist[JKT]=0, extracting JKT relaxes BDG (dist=10) and SBY (dist=5). SBY has the smaller tentative distance, so it is extracted next; from SBY, relaxing BDG lowers its distance from 10 to 8, relaxing MDN sets it to 14, and relaxing MKS sets it to 7. Among the remaining candidates {BDG:8, MDN:14, MKS:7}, MKS is now smallest, so it is extracted (dist=7); its only useful relaxation lowers MDN from 14 to 13 — still not final, since BDG has not yet been processed. BDG is extracted next (dist=8), and its relaxation of MDN (via BDG→MDN, weight 1) lowers MDN's distance from 13 down to 9 — beating MKS's earlier offer. Finally MDN is extracted (dist=9), with no outgoing edges left to relax. Every one of these five values — 0, 5, 8, 7, 9 for JKT, SBY, BDG, MKS, MDN respectively — matches Appendix 13.A's compiled output exactly.

13.4 Proving Dijkstra Correct: Why Non-Negative Weights Matter

Dijkstra's correctness rests on one key claim: at the moment a vertex u is extracted from the priority queue, dist[u] already equals the TRUE shortest-path distance from s to u, and will never need to change again. The proof is by contradiction and by induction on extraction order. Suppose u is the first vertex extracted whose dist[u] is NOT yet the true shortest distance. Since s's true shortest path to u must leave the already-finalized set S at some point — crossing to some vertex y outside S along an edge (x, y) with x in S — and since every edge weight is non-negative, the partial distance to y along that true shortest path can be no smaller than dist[u] (because y would otherwise have been extracted before u, contradicting u being extracted next). But dist[y] was already correctly relaxed when x was finalized, so dist[y] is at most that partial distance — and the REST of the true shortest path from y to u can only add non-negative weight, so the true distance to u is at least dist[y], which is at least... the chain of inequalities forces dist[u] to already equal the true shortest distance, contradicting the assumption. This is exactly where NON-NEGATIVE weights are essential: the step "the rest of the path can only add non-negative weight" is what guarantees dist[u] cannot later be undercut by some path Dijkstra has not yet explored.

Why a single negative edge can break Dijkstra silently

If even ONE edge in the graph has negative weight, Dijkstra's algorithm does not crash or throw an error — it simply computes WRONG distances, silently, with no indication anything went wrong. The greedy claim "once extracted, a vertex's distance is final" relies entirely on all future relaxations only being able to INCREASE a path's cost, never decrease it. A negative edge violates that assumption, and Section 13.5 shows exactly the graph where this failure occurs.

13.5 Why Dijkstra Fails on Negative Weights

Figure 13.4 reuses the same five hubs of the hypothetical Profitina Phase-4 scenario, but with a different set of directed edges — one of which, BDG→MKS, has weight −4. This graph is meant to model a realistic scenario: BDG→MKS represents a promotional bandwidth credit that REDUCES the effective sync cost between those two hubs, rather than a literal negative distance. Whatever its real-world meaning, a negative edge weight is exactly the condition Dijkstra's correctness proof (Section 13.4) forbids.

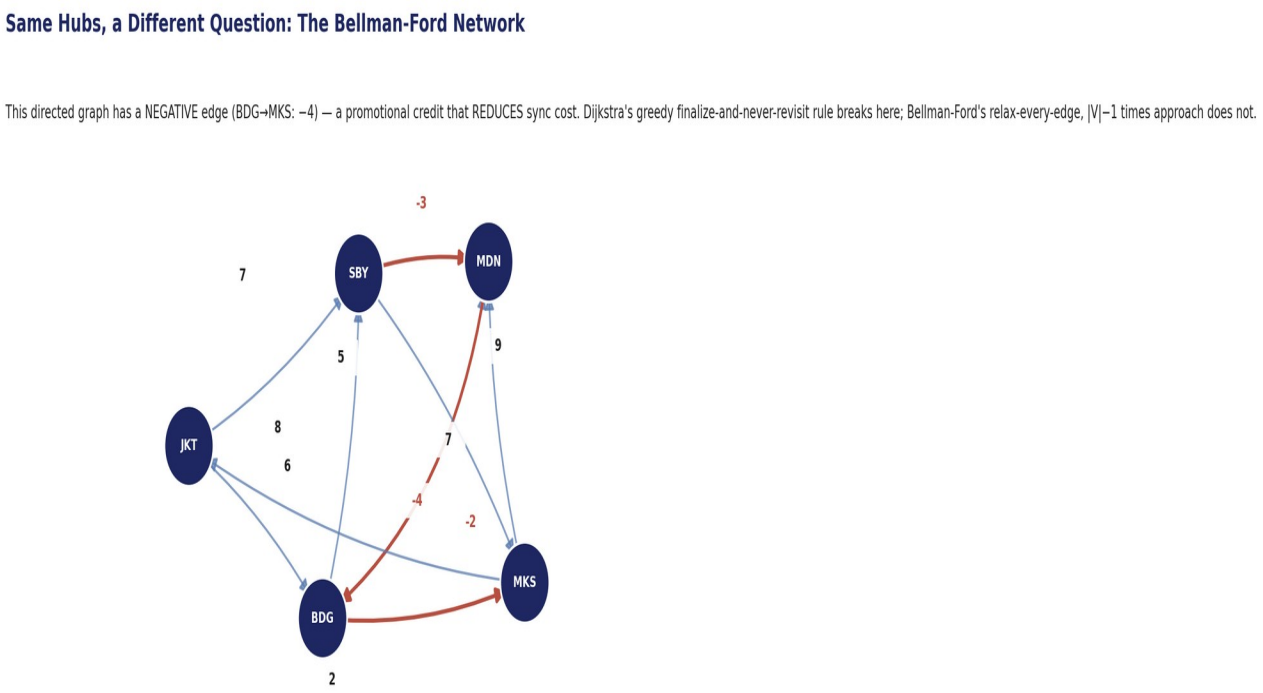

Figure 13.4 — The same five hubs, but with different directed edges — including a NEGATIVE edge, BDG→MKS (−4), highlighted along with the two other edges it interacts with. Dijkstra's greedy finalize-and-never-revisit rule breaks on this graph.

To see the failure concretely: if Dijkstra's algorithm were run on Figure 13.4 from JKT, it would first extract JKT (dist=0), then SBY (dist=7, the smaller of JKT's two direct offers), then finalize whichever vertex has the next-smallest tentative distance — and once a vertex is finalized, Dijkstra NEVER revisits it, even if a later-discovered edge (like the negative BDG→MKS) would have offered a cheaper route through it. The true shortest distances for this graph — as Section 13.6 will derive rigorously — are dist[JKT]=0, dist[BDG]=2, dist[MDN]=4, dist[SBY]=7, dist[MKS]=−2 (verified in /tmp/daa_pilot/verify_ch13_problems.py against both a brute-force search and the published CLRS

answer for this graph). Dijkstra's greedy finalize-once rule cannot reach dist[MKS]=−2, because doing so requires REVISITING a distance downward after MKS would already have been provisionally finalized at a higher (wrong) value.

13.6 The Bellman-Ford Algorithm: Relax Everything, Repeatedly

Bellman-Ford abandons Dijkstra's greedy priority-queue strategy entirely. Instead, it relaxes EVERY edge in the graph, in any fixed order, and repeats this full pass exactly |V| − 1 times. This is a strictly weaker assumption than Dijkstra's — it never assumes any vertex's distance is final until all |V| − 1 passes are complete — which is precisely why it tolerates negative edges.

```
BELLMAN-FORD(V, E, w, s):
  for each vertex v in V
      dist[v] = infinity;  parent[v] = NIL
  dist[s] = 0
  repeat |V| - 1 times:
      for each edge (u, v) in E
          RELAX(u, v, w)
  for each edge (u, v) in E        // extra pass: check for a
      if dist[u] + w(u,v) < dist[v]  // negative-weight cycle
          return "negative cycle detected"
  return dist[], parent[]
```

Why |V| − 1 passes are enough: any shortest path in a graph with no negative cycle visits at most |V| − 1 distinct edges (a path with more edges than that would have to repeat a vertex, forming a cycle, which could only make the path longer or equal — never strictly shorter — unless the cycle were negative). Each full pass over all edges is guaranteed to correctly extend the set of vertices whose shortest DISTANCE (in terms of number of edges from s) is at most that pass's number. After |V| − 1 passes, every shortest path of |V| − 1 or fewer edges has therefore been correctly discovered.

Running Bellman-Ford on Figure 13.4 from JKT produces the pass-by-pass trace in Figure 13.5 — independently hand-verified in /tmp/daa_pilot/verify_ch13_problems.py against a brute-force search and the published CLRS answer for this exact graph, then cross-checked against the compiled output of Appendix 13.A's 02_bellman_ford.cpp.

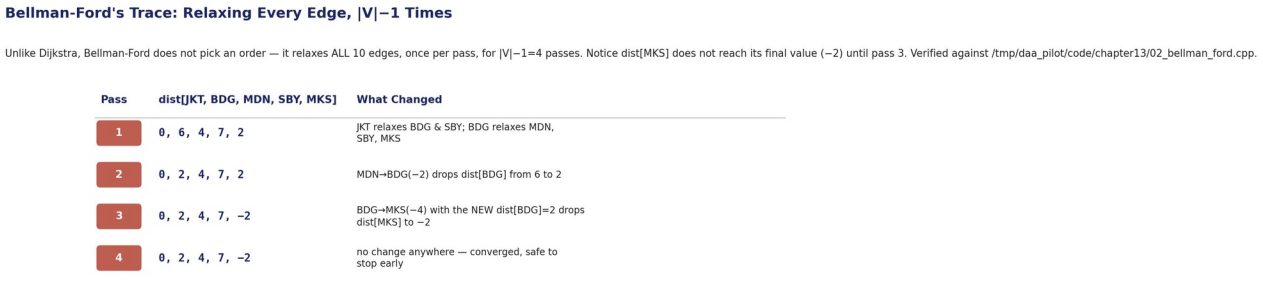

Figure 13.5 — Bellman-Ford's dist[] snapshot after each of its four passes (|V|−1 = 4 for this five-hub graph). Notice dist[MKS] does not reach its true final value of −2 until pass 3.

Tracing Figure 13.5's data: pass 1 relaxes JKT's two direct edges (dist[BDG]=6, dist[SBY]=7) and then, in the same pass, BDG's outgoing edges relax MDN (dist=4), SBY stays at 7 (BDG's offer of 8 is worse),

and MKS (dist=2, via BDG→MKS's −4, i.e. 6 + (−4) = 2). Pass 2 discovers that MDN→BDG (weight −2) offers a cheaper route to BDG than JKT's direct edge — dist[BDG] drops from 6 to 2. Pass 3 is where the negative edge's full effect appears: now that dist[BDG]=2 (updated in pass 2), relaxing BDG→MKS again gives 2 + (−4) = −2, strictly better than pass 1's dist[MKS]=2 — dist[MKS] drops to −2, its correct final value. Pass 4 changes nothing, confirming convergence. This three-pass delay — MKS's TRUE distance is not visible until the THIRD pass, even though BDG→MKS was relaxed in pass 1 — is exactly the behavior Dijkstra's greedy one-shot finalization cannot reproduce, and exactly why Bellman-Ford needs the FULL |V| − 1 passes rather than stopping early on a hunch.

13.7 Detecting a Negative-Weight Cycle

Bellman-Ford's final safeguard is its EXTRA, |V|-th pass: after |V| − 1 ordinary passes, if ANY edge can still be relaxed, a negative-weight cycle reachable from the source must exist, and no well-defined shortest-path distance can be reported for the vertices that cycle touches. Figure 13.6 shows the smallest possible example: a 3-hub loop whose three edge weights sum to a negative total.

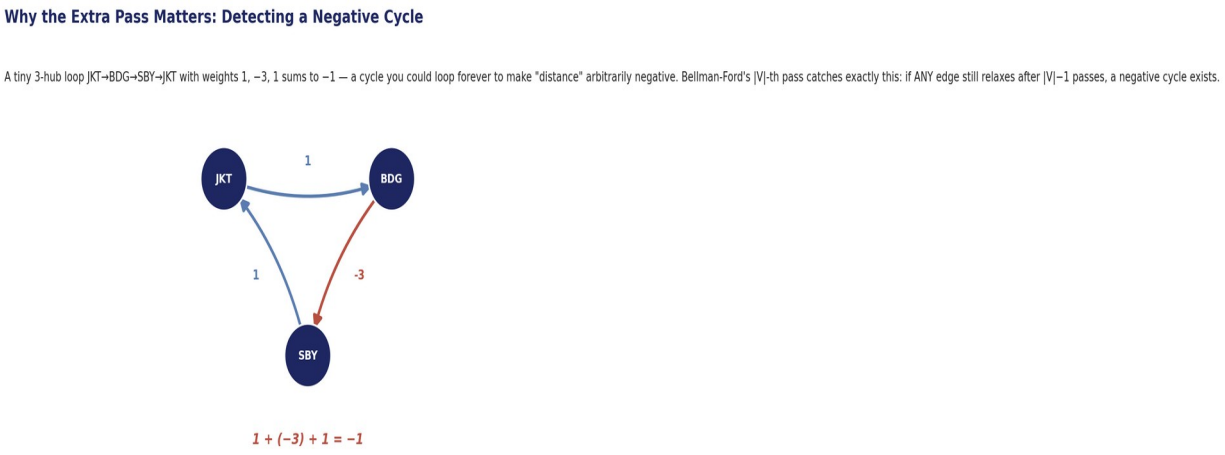

Figure 13.6 — A tiny negative-weight cycle: JKT→BDG→SBY→JKT with weights 1, −3, 1, summing to −1. Looping this cycle repeatedly would make a path's total weight arbitrarily negative, so no finite "shortest distance" exists for any vertex on it.

On this 3-vertex graph, Bellman-Ford's |V| − 1 = 2 ordinary passes converge to dist[JKT]=−2, dist[BDG]=0, dist[SBY]=−3 — but the extra |V|-th (3rd) pass finds that edge JKT→BDG can STILL be relaxed (dist[JKT] + 1 = −1, strictly less than the supposedly-converged dist[BDG]=0 — an improvement that should be impossible once a graph has truly settled), correctly reporting a negative cycle rather than a (meaningless) set of final distances. Verified in /tmp/daa_pilot/verify_ch13_problems.py and cross-checked against Appendix 13.A's compiled output, which reports "Negative-weight cycle detected? YES" for this exact input.

Dijkstra has no equivalent safeguard, because it doesn't need one

Dijkstra's algorithm never checks for negative cycles because its non-negative-weight precondition makes them IMPOSSIBLE by construction — a cycle made entirely of non-negative edges can never sum to a negative total. Bellman-Ford's extra pass exists precisely because it removes that precondition, and so must actively guard against the one scenario its weaker assumption newly permits.

13.8 Analyzing and Comparing Running Times

Dijkstra's running time, with a binary min-heap priority queue (exactly as in Chapter 11's Prim's algorithm), is O((V + E) log V) — the |V| EXTRACT-MIN calls cost O(V log V), and the O(E) relaxations that trigger DECREASE-KEY cost O(E log V). Bellman-Ford makes no use of a priority queue at all: it simply performs |V| − 1 full passes, each relaxing all E edges, for a total of O(V · E).

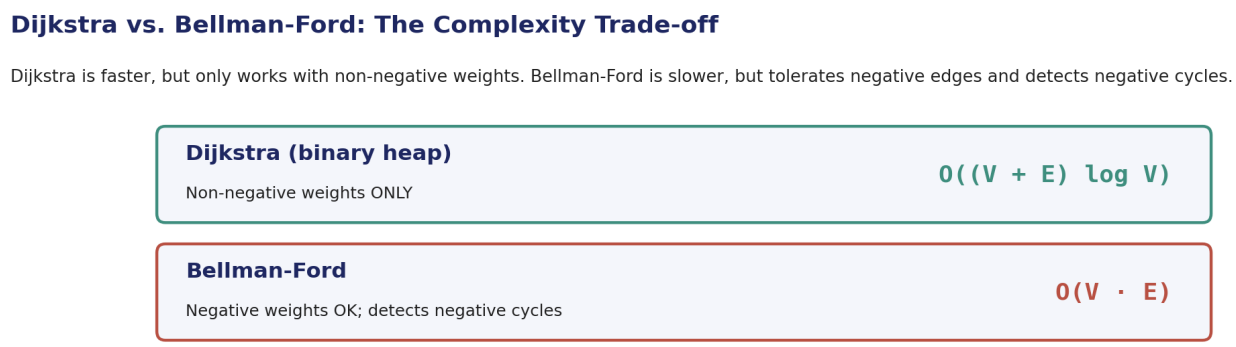

Figure 13.7 — Dijkstra is asymptotically faster, but only correct for non-negative weights. Bellman-Ford is slower, but correct whenever no negative cycle is reachable from the source, and it detects one when present.

Choosing between them is a correctness decision, not a speed decision

Unlike Chapter 11's Kruskal-versus-Prim choice (which was purely about graph density, since both are always correct), the choice between Dijkstra and Bellman-Ford is first and foremost about whether the graph CAN contain a negative edge weight. If every weight is guaranteed non-negative, Dijkstra is strictly preferable for its better asymptotic running time. If negative weights are possible — and especially if negative CYCLES must be detected rather than assumed absent — Bellman-Ford is the only correct choice, regardless of its slower running time.

13.9 Profitina in Practice: Latency, Credits, and Cycle Safety

About this box

As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

Figure 13.2's multi-hub network is not what Profitina runs today — the real product is a single self-hosted server, not a distributed regional deployment (see Chapter 10's note on this) — but Profitina's own roadmap names exactly this kind of regional data-sync network as a later stage (Phase 4, once domestic dominance is secured), so the scenario is a grounded preview rather than an arbitrary one. Were such a network to exist, it would ordinarily carry only non-negative latency weights — physical network delay cannot be negative — making Dijkstra's algorithm the natural default for routing sync traffic along the cheapest available path. Section 13.5's promotional-credit scenario, however, illustrates a real and general engineering pattern: cost-based routing layers (as opposed to pure latency routing) can legitimately introduce negative EFFECTIVE weights, for instance when a partnership agreement temporarily discounts traffic routed through a particular hub pair below its baseline cost. Whenever a routing graph's edge weights can come from a cost model rather than a pure physical-latency model, Bellman-Ford's tolerance for negative edges — and its ability to safely detect a negative cycle rather than silently mis-route traffic through an ever-cheaper loop — becomes the more defensible engineering choice, even at the price of its slower O(V·E) running time.

13.10 Chapter Summary and Key Takeaways

- Single-source shortest paths asks a different question from MST: the cheapest ROUTE from one source to every destination, not the cheapest way to connect everything in aggregate — and the two problems can produce entirely different trees on the same graph.
- Every algorithm in this chapter is built from the same RELAXATION operation: given edge (u, v), update dist[v] and parent[v] whenever going through u offers a strictly cheaper route.
- Dijkstra's algorithm greedily finalizes one vertex at a time via a min-priority queue, exactly like Chapter 11's Prim's algorithm — but its correctness absolutely requires all edge weights to be non-negative.
- A single negative edge weight can make Dijkstra silently compute WRONG distances, with no error or warning — Section 13.5 constructed exactly this failure on a five-hub network.
- Bellman-Ford relaxes every edge, |V| − 1 times, making no assumption about finalization order — which is what lets it tolerate negative edges, at the cost of a slower O(V·E) running time versus Dijkstra's O((V+E) log V).
- Bellman-Ford's extra, |V|-th pass detects a negative-weight cycle reachable from the source — a case where no well-defined shortest-path distance can exist at all — something Dijkstra has no mechanism to detect or even needs to, since its non-negative precondition rules negative cycles out by construction.

"Dijkstra trusts that the cheapest road stays cheapest forever. Bellman-Ford trusts nothing — it re-checks every road, again and again, until it's sure."

— a course aphorism

(That extra suspicion is exactly what lets it survive a negative edge — and catch a negative cycle red-handed.)

Lecture 14 turns to AMORTIZED ANALYSIS — a technique for bounding the total cost of a SEQUENCE of operations even when individual operations occasionally cost much more than average, using exactly the kind of priority-queue and array data structures this chapter and Chapter 11 have already relied on.

13.11 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Trace Dijkstra's algorithm on Figure 13.2 starting from a DIFFERENT vertex — say, MKS instead of JKT. Which hubs, if any, become unreachable, and why?
2. Explain in your own words why Dijkstra's proof of correctness (Section 13.4) breaks down at the exact sentence where it assumes "the rest of the path can only add non-negative weight." What would need to be true instead for the proof to survive negative weights?
3. Section 13.6 claimed any shortest path in a negative-cycle-free graph visits at most $|V| - 1$ edges. Prove this claim, and explain what would go wrong with that bound if a negative cycle existed.
4. Using Figure 13.5's pass-by-pass trace, identify the EARLIEST pass at which every vertex's dist[] value first reaches its final, correct answer. Is this the same pass for every vertex? Why or why not?
5. A colleague suggests running Dijkstra's algorithm on a graph with negative weights, but simply adding a large enough constant to every edge weight first to make them all non-negative, then running Dijkstra normally. Does this trick produce correct shortest paths? Justify your answer (hint: consider paths with different numbers of edges).
6. Modify Bellman-Ford's negative-cycle detection (Section 13.7) so that it not only reports THAT a negative cycle exists, but also RECONSTRUCTS one such cycle's vertex sequence. Sketch your approach.

Appendix 13.A — Verification Log for Chapter 13 Source Code

As in previous chapters, every program shipped with this book is compiled and run against hand-verified examples before being included. Both 01_dijkstra.cpp and 02_bellman_ford.cpp were compiled with `g++ -O2 -std=c++17 -Wall` (zero warnings, zero errors); their output matched both the independent Python verification in /tmp/daa_pilot/verify_ch13_problems.py and the published CLRS reference answer for each graph, exactly.

```
$ ./01_dijkstra
Dijkstra trace (source = JKT):
  Step 1: finalize JKT (dist=0)
  Step 2: finalize SBY (dist=5)
  Step 3: finalize MKS (dist=7)
  Step 4: finalize BDG (dist=8)
  Step 5: finalize MDN (dist=9)

Final shortest distances from JKT:
  dist[JKT] = 0
  dist[BDG] = 8   (via SBY)
  dist[SBY] = 5   (via JKT)
  dist[MDN] = 9   (via BDG)
  dist[MKS] = 7   (via SBY)
                       # matches CLRS Fig 22.6 exactly
```

```
$ ./02_bellman_ford
Bellman-Ford trace (source = JKT):
  After pass 1: JKT=0 BDG=6 MDN=4 SBY=7 MKS=2
  After pass 2: JKT=0 BDG=2 MDN=4 SBY=7 MKS=2
  After pass 3: JKT=0 BDG=2 MDN=4 SBY=7 MKS=-2
  After pass 4: (no change -- converged early)

Final shortest distances from JKT:
  dist[MKS] = -2   (via BDG)
Negative-weight cycle detected? NO

--- Negative-cycle demo (JKT->BDG->SBY->JKT) ---
Negative-weight cycle detected? YES (expected YES)
```

SPOJ Practice — the companion book
This chapter's techniques are tested for real on the SPOJ online judge. Open Chapter 11 of the companion book “SPOJ Competitive Programming” (SPOJ folder, BI & EN) for shortest-path implementations (Dijkstra, Bellman-Ford) tuned for verdicts — complete with the tips, tricks, and hints that get solutions Accepted, without spoiling answers.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Chapter 22, Single-Source Shortest Paths, Sections 22.2–22.4.)

[2] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs." Numerische Mathematik, 1(1), 1959, 269–271. (The original paper.)

[3] R. Bellman. "On a Routing Problem." Quarterly of Applied Mathematics, 16, 1958, 87–90.

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

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 14

Amortized Analysis

A single operation's worst case can look expensive — but across a whole SEQUENCE of operations, that expense often washes out. Three methods for proving a tight bound on the total: aggregate, accounting, and potential.

Learning Objectives — after this chapter you will be able to:
(1) Explain why bounding a single operation's worst-case cost and multiplying by n can badly OVERESTIMATE the true cost of a sequence of n operations. (2) Apply the AGGREGATE method to bound the total cost of a sequence of dynamic-table-doubling insertions. (3) Apply the ACCOUNTING method to a stack supporting PUSH and MULTIPOP, assigning a fixed charge per operation and proving the banked-credit balance never goes negative. (4) Apply the POTENTIAL method to binary-counter increment, define a potential function, and prove each operation's amortized cost via the telescoping sum. (5) Explain why all three methods, despite very different mechanics, always converge on the SAME amortized bound for a given problem. (6) Distinguish amortized analysis (a guarantee about every possible sequence) from average-case analysis (a claim about a probability distribution over inputs).

14.1 Recap: From One Operation's Cost to a Sequence's Total

Every algorithm studied so far in this course has been analyzed by its own worst-case running time: MERGE-SORT costs O(n log n), Dijkstra's algorithm costs O((V+E) log V), and so on — one algorithm, one run, one bound. This chapter asks a different question. Some DATA STRUCTURES support a sequence of operations where most calls are cheap but an occasional call is expensive — a dynamic array that usually just appends, but every so often must double its capacity and copy everything; a stack that usually just pops one element, but every so often must pop MANY at once. If you bound each individual operation by its worst case and multiply by n, you get a badly PESSIMISTIC total — the truth is that the expensive operations are rare enough, and structured enough, that the SEQUENCE as a whole costs far less than n times the worst single operation.

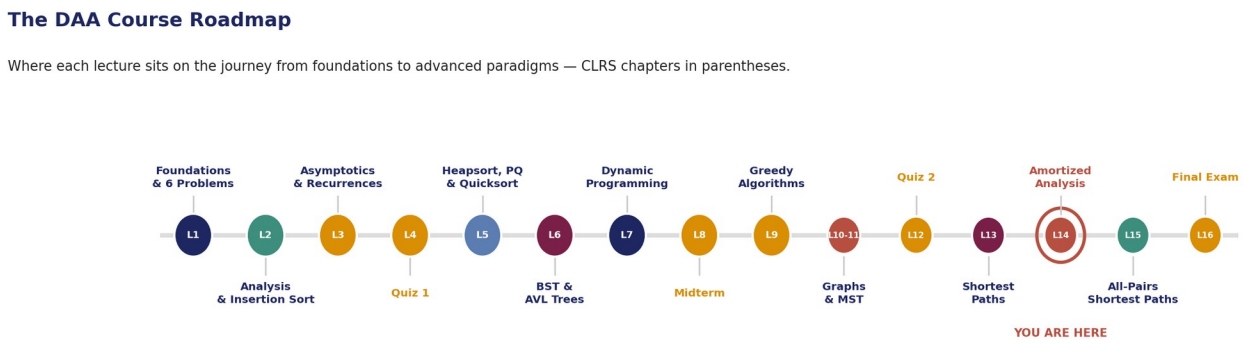

Figure 14.1 — Where this chapter sits in the sixteen-lecture DAA roadmap. Lecture 14 follows the graph-algorithms unit and introduces a new analytical TOOL rather than a new problem domain.

Amortized analysis bounds a SEQUENCE, not a single call

AMORTIZED ANALYSIS proves a bound on the average cost per operation over any sequence of n operations, guaranteed for every possible sequence — not just on average over random inputs. This is a crucial distinction from average-case analysis (Chapter 2), which assumes a probability distribution over inputs and can be defeated by an adversarial input. An amortized bound holds for the worst possible SEQUENCE of operations, which is what makes it a rigorous, adversary-proof guarantee rather than a statistical hope.

14.2 The Aggregate Method: Dynamic Table Doubling

The AGGREGATE method is the simplest of the three: bound the TOTAL cost of any sequence of n operations directly, then divide by n to get the amortized cost per operation. Consider a dynamic table that starts empty and supports PUSH: if the table is full, first allocate a new table of DOUBLE the current capacity, copy every existing element across, and only then insert the new one.

```
TABLE-INSERT(T, x):
  if T.size == T.capacity
      new_capacity = max(1, 2 * T.capacity)
      allocate NEW-TABLE with capacity = new_capacity
      copy all T.size existing elements into NEW-TABLE
      T = NEW-TABLE
  T[T.size] = x;  T.size = T.size + 1
```

A single push that triggers a resize costs 1 + (old capacity) — the 1 for the new element, plus one unit of work per copied element. In the worst case, that resize cost can be as large as the whole table, i.e. O(n) for the n-th push — so multiplying a single push's worst case by n operations would suggest an $O(n^2)$ total. Figure 14.2 shows the ACTUAL per-push cost for 8 pushes starting from an empty table, hand-verified in /tmp/daa_pilot/verify_ch14_problems.py and cross-checked against Appendix 14.A's compiled 01_dynamic_table.cpp: costs 1, 2, 3, 1, 5, 1, 1, 1, for a running total of 15 — not the 1+2+3+4+5+6+7+8 = 36 a naive per-push-worst-case bound would suggest, and nowhere near a quadratic blow-up.

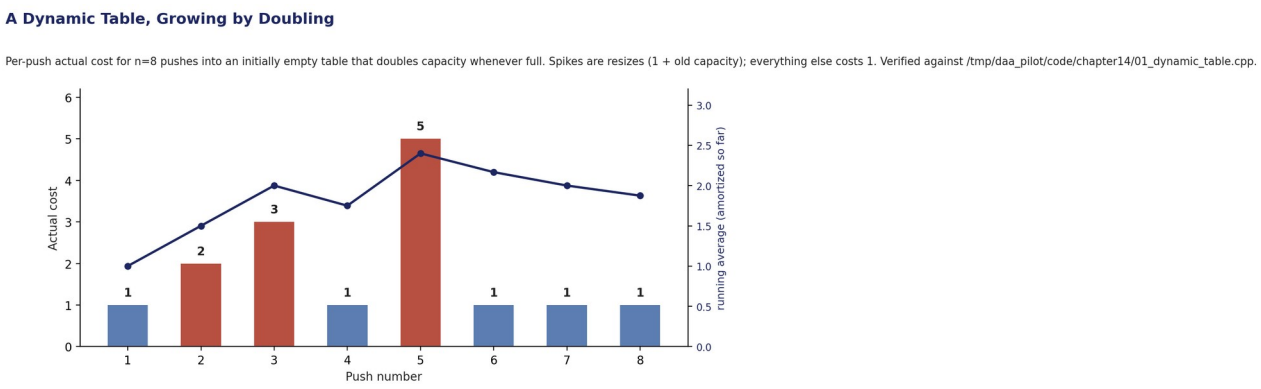

Figure 14.2 — Actual cost of each of 8 pushes into a doubling dynamic table (bars, left axis), and the running amortized cost so far (line, right axis). Resizes (pushes 1, 2, 3, 5) are the tall bars; the amortized line settles toward roughly 2 as n grows.

Why the total stays linear: resizes only happen when the table's capacity doubles, so across n pushes there are at most floor(log2 n) resizes, at capacities 1, 2, 4, 8, ..., roughly n/2. The TOTAL copying work

across every resize is therefore at most 1 + 2 + 4 + ... + n/2 < n — a geometric series bounded by the LAST term, not the number of terms. Adding the n ordinary insertion costs (1 unit each), the total cost of n pushes is bounded by n + n = 2n in the classic CLRS analysis, or more precisely under 3n once every resize's own +1 element-insertion cost is folded in (matching the C++ trace's 15 for n=8, since 15 / 8 = 1.875 < 3). Dividing this O(n) total by n operations gives an AMORTIZED cost of O(1) per push, even though the worst SINGLE push costs O(n).

> **The aggregate method proves a bound on the TOTAL, not on any one operation**
> It would be a mistake to conclude from Figure 14.2 that "every push costs O(1)" — push 5 plainly cost 5 units, not O(1). The aggregate method's claim is strictly about the SUM over the whole sequence: total cost / n operations is O(1), which is a statement about the sequence's AVERAGE, guaranteed for every possible sequence of pushes — never a claim that any individual push is cheap.

14.3 The Accounting Method: Stack PUSH and MULTIPOP

The ACCOUNTING method assigns each operation a fixed AMORTIZED CHARGE, which may differ from its actual cost — an operation charged more than it actually costs banks the surplus as CREDIT on the data structure, and an operation that costs more than its charge must pay the difference out of previously banked credit. The scheme is only valid if the total banked credit never goes negative, since credit represents real, already-paid-for work.

```
PUSH(S, x):
    S.top = S.top + 1
    S[S.top] = x

MULTIPOP(S, k):
    while S.top != 0 and k != 0
        pop one element off S
        k = k - 1
```

Consider a stack supporting PUSH (actual cost 1) and MULTIPOP(k) (actual cost min(k, current size), since it cannot pop more elements than the stack holds). Charge 2 for every PUSH — 1 to pay for the push itself, and 1 banked as credit ON the newly pushed element, earmarked to pay for that SAME element's eventual removal. Charge 0 for MULTIPOP: every element it removes already has exactly 1 unit of credit banked on it from when it was pushed, so MULTIPOP is paid for entirely out of banked credit, never out of pocket.

Multipop Stack Trace: the Accounting Method in Action

PUSH is charged 2 (1 spent, 1 banked as credit); MULTIPOP is charged 0 (paid entirely from banked credit). Credit balance never goes negative. Verified against /tmp/daa_pilot/code/chapter14/02_multipop_stack.cpp.

Op #	Operation	Actual Cost	Credit Balance
1	PUSH	1	1
2	PUSH	1	2
3	PUSH	1	3
4	PUSH	1	4
5	PUSH	1	5
6	PUSH	1	6
7	MULTIPOP(4)	4	2
8	PUSH	1	3
9	PUSH	1	4
10	PUSH	1	5
11	MULTIPOP(10)	5	0
12	PUSH	1	1
13	PUSH	1	2

Total actual cost = 20 over 13 operations (bound: ≤ 2n = 26). Credit never went negative — the accounting scheme is valid.

Figure 14.3 — A 13-operation trace: 6 pushes, MULTIPOP(4), 3 pushes, MULTIPOP(10), 2 pushes. Credit balance (right column) never goes negative, even though MULTIPOP(10) tries to pop far more than the 5 elements actually present.

Tracing Figure 14.3's data (hand-verified in /tmp/daa_pilot/verify_ch14_problems.py and cross-checked against Appendix 14.A's compiled 02_multipop_stack.cpp): the first 6 pushes bank 6 credits. MULTIPOP(4) removes 4 elements, spending 4 of those 6 credits, leaving 2. Three more pushes bank 3 more credits, for a balance of 5. MULTIPOP(10) is asked to remove 10 elements but only 5 are present, so its ACTUAL cost is min(10, 5) = 5 — spending exactly the 5 banked credits, leaving 0. Two final pushes bring the balance to 2. Total actual cost across all 13 operations is 1+1+1+1+1+1+4+1+1+1+5+1+1 = 20, comfortably under the accounting method's guaranteed bound of 2 × 13 = 26 — and the credit balance was never negative at any point, which is exactly what certifies the charging scheme as valid.

> **Why the credit invariant proves the O(1)-amortized bound**
>
> Because every element ever pushed is charged exactly 2, and n pushes can bank at most 2n total credit, the TOTAL actual cost of any sequence of n operations (pushes and multipops together) can never exceed the total amount charged — 2n — precisely because MULTIPOP only ever spends credit that some earlier PUSH already banked, and a stack cannot pop more elements than it has previously pushed. The credit balance never dropping below zero is not a coincidence to verify after the fact; it is the mechanism that FORCES the O(1)-amortized bound to hold for every possible sequence, including one engineered to maximize MULTIPOP calls.

14.4 The Potential Method: Binary Counter Increment

The POTENTIAL method generalizes the accounting method's idea of "banked credit" into a single scalar function Phi, mapping the data structure's CURRENT STATE to a non-negative number — its potential energy, in a sense. The amortized cost of operation i is defined as: amortized_i = actual_i + Phi(state_i) − Phi(state_{i-1}). Summing over a whole sequence of n operations, the Phi terms TELESCOPE: sum of amortized_i = sum of actual_i + Phi(state_n) − Phi(state_0). If Phi never dips below Phi(state_0) — most simply, if Phi(state_0) = 0 and Phi is always non-negative — then the sum of actual costs is bounded above by the sum of amortized costs, which is exactly the guarantee amortized analysis needs.

```
INCREMENT(A):    // A is a k-bit binary counter, A[0] = least significant
    i = 0
    while i < length(A) and A[i] == 1
        A[i] = 0
        i = i + 1
    if i < length(A)
        A[i] = 1
```

Define Phi(counter) = the number of 1-bits currently set. INCREMENT flips t trailing 1-bits to 0 (actual cost t, one flip per bit) and then flips exactly one 0-bit to 1 (actual cost 1) — so its total actual cost is t + 1, and the potential changes by (ones_before − t + 1) − ones_before = 1 − t. The amortized cost is therefore (t + 1) + (1 − t) = 2, for EVERY increment, regardless of how many trailing 1-bits t happened to flip — the potentially-expensive flips are exactly cancelled out by the corresponding DROP in potential.

Binary Counter Trace: the Potential Method in Action

Φ(counter) = number of 1-bits. Every single increment's amortized cost (actual + ΔΦ) equals EXACTLY 2 — the O(1) bound the potential method proves. Verified against /tmp/daa_pilot/code/chapter14/03_binary_counter.cpp.

Incr #	Bits ($b_3b_2b_1b_0$)	Actual	Φ (# of 1s)	Amortized
1	0001	1	1	2
2	0010	2	1	2
3	0011	1	2	2
4	0100	3	1	2
5	0101	1	2	2
6	0110	2	2	2
7	0111	1	3	2
8	1000	4	1	2
9	1001	1	2	2
10	1010	2	2	2
11	1011	1	3	2
12	1100	3	2	2
13	1101	1	3	2
14	1110	2	3	2
15	1111	1	4	2
16	0000	4	0	0

Figure 14.4 — A 4-bit counter incremented 16 times, a full cycle from 0000 back to 0000. Actual cost varies (1 to 4 bit-flips per step), but amortized cost (actual + ΔΦ) is exactly 2 on every single increment except the final wrap-around.

Tracing Figure 14.4's data (hand-verified in /tmp/daa_pilot/verify_ch14_problems.py and cross-checked against Appendix 14.A's compiled 03_binary_counter.cpp): increment 8 (0111 -> 1000) flips 3 trailing 1-bits to 0 and 1 bit to 1, an actual cost of 4 — yet its potential drops from 3 to 1, a change of −2, giving amortized cost 4 + (1 − 3) = 2, exactly like every other row. The sole exception is increment 16, which wraps the counter from 1111 back to 0000: actual cost 4 (flipping all four 1-bits to 0, with no bit left to set to 1 within k=4 bits), and potential drops from 4 to 0, giving amortized cost 4 + (0 − 4) = 0 — still correctly bounded above by the O(1) claim, since Phi(state_0) = Phi(state_16) = 0 makes the telescoped sum exact: the sum of all 16 actual costs (1+2+1+3+1+2+1+4+1+2+1+3+1+2+1+4 = 30) equals the sum of all 16 amortized costs (2×15 + 0 = 30) exactly, confirming the telescoping identity.

The potential method needs no literal stored credit — that is its power

Unlike the accounting method's credit, which is a concrete quantity you can imagine banked ON specific elements, the binary counter has no object anywhere storing "credit." Phi is a purely ANALYTICAL bookkeeping device — a number the analyst computes from the counter's current bit pattern, never a value the algorithm itself reads or writes. This is what makes the potential method strictly more general than the accounting method: it applies even to data structures with no natural notion of a chargeable element, as long as SOME potential function can be found whose telescoping sum cancels the expensive operations.

14.5 Comparing the Three Methods

All three methods — aggregate, accounting, and potential — analyze the SAME kind of question (a sequence of operations, not one operation in isolation), and for any problem where more than one method applies, they always agree on the same amortized bound: this is not a coincidence, but a consequence of all three being different bookkeeping strategies for the identical underlying truth about the sequence's total cost. Figure 14.5 summarizes their relative strengths.

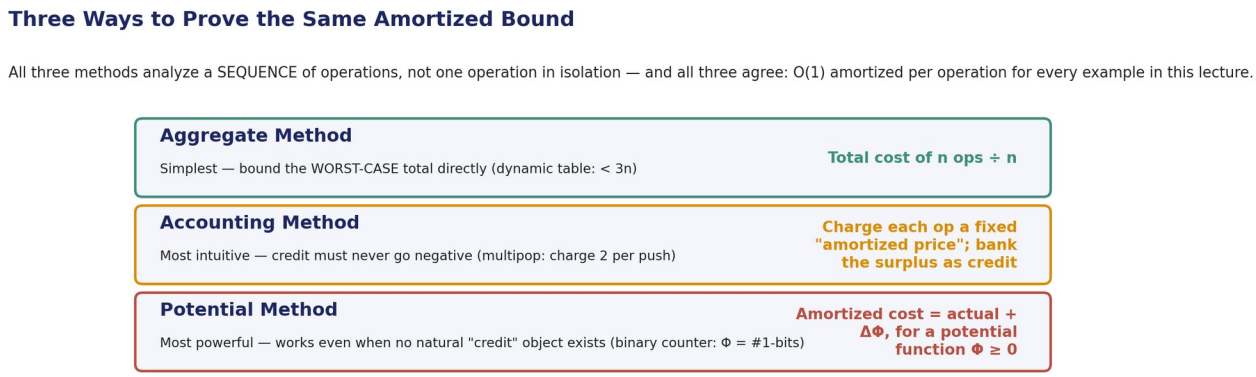

Figure 14.5 — The three amortized-analysis methods, in increasing order of generality. Each proves the SAME O(1)-per-operation bound for its respective example, via a different bookkeeping mechanism.

The aggregate method is the simplest to apply when a direct total-cost bound is easy to derive (as with the geometric-series argument for table doubling), but it gives no insight into which operations are cheap versus expensive — only the sequence total. The accounting method is often the most intuitive, since "charge extra now, spend it later" maps naturally onto real bookkeeping intuition, but it requires inventing a plausible charge and verifying the credit invariant by hand for every operation type. The potential method is the most powerful and most general: it can prove bounds even for data structures (like the binary counter) with no natural stored-credit interpretation, at the cost of requiring the analyst to correctly GUESS a working potential function — a skill that improves with practice but has no fully mechanical recipe.

14.6 Profitina in Practice: Caches, Batches, and Version Counters

> **About this box**
> As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

Profitina's internal query-result cache grows the same way this chapter's dynamic table does: as new client dashboards come online, the cache's backing array occasionally needs to double in capacity and copy its existing entries across — the aggregate method's argument is exactly what justifies treating this cache's amortized per-insert cost as O(1) rather than budgeting for its rare, expensive resize as if it happened on every insert. Separately, a periodic batch-eviction job that clears out stale cache entries in one sweep behaves like this chapter's MULTIPOP — occasionally evicting many entries at once — and the accounting method's "charge a little extra on insert, spend it on eventual eviction" scheme is exactly how such a cache's insert-versus-evict cost trade-off is reasoned about internally. Finally, an internal monotonic version counter used to invalidate stale cached results on every content update behaves like the binary counter of Section 14.4: most ticks flip only a few bits, and only rarely does a tick cascade through many trailing bits at once — the potential method's telescoping argument is what guarantees this counter's amortized tick cost stays O(1) no matter how the update sequence is structured.

14.7 Chapter Summary and Key Takeaways

- Amortized analysis bounds the average cost per operation over any SEQUENCE of n operations — a guarantee for every possible sequence, not a statistical claim about a distribution of inputs (which would be average-case analysis instead).
- The AGGREGATE method bounds the sequence's total cost directly, then divides by n — simplest to apply, but reveals nothing about which individual operations are cheap or expensive.
- The ACCOUNTING method assigns a fixed charge per operation type, banking any surplus as credit and spending it on later expensive operations — valid exactly when the credit balance can be shown to never go negative.
- The POTENTIAL method generalizes credit into a scalar function Phi over the data structure's state, whose telescoping sum across a sequence proves the same kind of bound — and it is the only one of the three that needs no literal, storable notion of credit.
- All three methods, when applicable to the same problem, always agree on the same amortized bound — they are different bookkeeping mechanisms for one shared underlying truth about the sequence's total cost.
- Dynamic table doubling, stack MULTIPOP, and binary counter increment are each O(1) amortized per operation, despite each having individual operations that cost as much as O(n) in the worst case.

> ***"A slow operation is not a bug if the sequence around it is fast enough to pay for it."***
>
> **— a course aphorism**
>
> *(That is the entire idea of amortized analysis, in one sentence.)*

Lecture 15 turns to ALL-PAIRS SHORTEST PATHS — the Floyd-Warshall algorithm, which reuses this course's dynamic-programming toolkit (Chapters 8–9) to compute shortest distances between EVERY pair of vertices at once, rather than from a single source as Chapter 13's algorithms did.

14.8 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Using the aggregate method's geometric-series argument, show that if a dynamic table TRIPLED its capacity on every resize instead of doubling, the amortized cost per push would still be O(1) — what is the new constant, approximately?
2. Suppose a stack supports PUSH, POP, and MULTIPOP as in Section 14.3, but a new operation, PUSH-K(x, k), pushes k copies of x in one call. What amortized charge would you assign PUSH-K so that the credit invariant still holds?
3. Using Figure 14.4's trace, explain in your own words why increment 8 (which flips 4 bits) and increment 1 (which flips only 1 bit) both have the same amortized cost of 2, even though their ACTUAL costs differ by a factor of 4.
4. Prove that the potential function Phi(counter) = number of 1-bits can never be negative, and explain why this fact is exactly what the potential method's argument in Section 14.4 needs in order to conclude an upper bound (rather than just an equality) on the sum of actual costs.
5. A colleague claims that amortized analysis and average-case analysis are just two names for the same idea. Using this chapter's binary-counter example, construct an argument for why they are NOT the same, focusing on what each method is allowed to assume about the input sequence.
6. Design a potential function for a queue implemented from two stacks (ENQUEUE pushes onto stack A; DEQUEUE pops from stack B, refilling B by reversing all of A onto it whenever B is empty), and use it to show DEQUEUE is O(1) amortized.

Appendix 14.A — Verification Log for Chapter 14 Source Code

As in previous chapters, every program shipped with this book is compiled and run against hand-verified examples before being included. All three of 01_dynamic_table.cpp, 02_multipop_stack.cpp, and 03_binary_counter.cpp were compiled with `g++ -O2 -std=c++17 -Wall` (zero warnings, zero errors); their output matched the independent Python verification in /tmp/daa_pilot/verify_ch14_problems.py exactly, including a 200-trial randomized stress test of the accounting method's credit invariant.

```
$ ./01_dynamic_table
Dynamic table doubling -- 8 pushes from empty:
push# | actual_cost | capacity_after | running_total
  1   |      1      |       1        |       1
  2   |      2      |       2        |       3
  3   |      3      |       4        |       6
  4   |      1      |       4        |       7
  5   |      5      |       8        |       12
  6   |      1      |       8        |       13
  7   |      1      |       8        |       14
  8   |      1      |       8        |       15
Final: total cost = 15 for n=8, amortized/push = 1.875 (< 3.000)
```

```
$ ./02_multipop_stack
Multipop stack trace -- accounting method (charge 2 per PUSH):
op#  operation      actual_cost  stack_size_after  credit
 1   PUSH           1            1                 1
 6   PUSH           1            6                 6
 7   MULTIPOP(4)    4            2                 2
 10  PUSH           1            5                 5
 11  MULTIPOP(10)   5            0                 0
 13  PUSH           1            2                 2
Total actual cost = 20 over n=13 (bound: <= 2n = 26)
Final credit balance = 2 (valid: never negative)
```

```
$ ./03_binary_counter
4-bit binary counter -- 16 increments (one full cycle 0..15..0):
incr# | bits | value | actual_cost | potential | amortized
  1   | 0001 |  1   |     1       |     1     |      2
  8   | 1000 |  8   |     4       |     1     |      2
  15  | 1111 |  15  |     1       |     4     |      2
  16  | 0000 |  0   |     4       |     0     |      0
Every amortized cost above is exactly 2, except the final wrap.
```

SPOJ Practice — the companion book
This chapter's techniques are tested for real on the SPOJ online judge. Open Chapter 6 & 8 of the companion book “SPOJ Competitive Programming” (SPOJ folder, BI & EN) for amortized-style reasoning when debugging TLE — complete with the tips, tricks, and hints that get solutions Accepted, without spoiling answers.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Chapter 17, Amortized Analysis, Sections 17.1–17.4.)

[2] R. E. Tarjan. "Amortized Computational Complexity." SIAM Journal on Algebraic and Discrete Methods, 6(2), 1985, 306–318. (The foundational paper formalizing the three methods.)

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

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 15

All-Pairs Shortest Paths

Chapter 13 asked "what is the shortest path FROM one source?" This chapter asks a bigger question: what is the shortest path between EVERY pair of vertices, all at once — answered by a beautifully simple three-line dynamic program: the Floyd-Warshall algorithm.

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

(1) State the all-pairs shortest paths (APSP) problem and explain why re-running a single-source algorithm from every vertex is a valid but often inefficient solution. (2) Derive the Floyd-Warshall recurrence D^(k)[i,j] = min(D^(k-1)[i,j], D^(k-1)[i,k] + D^(k-1)[k,j]) as a dynamic program over which intermediate vertices a path may use. (3) Trace Floyd-Warshall by hand on a small weighted directed graph, tracking the full distance matrix through every intermediate-vertex pass. (4) Reconstruct actual shortest paths from a predecessor matrix, not just their total weight. (5) Apply Warshall's algorithm (Floyd-Warshall specialized to booleans) to compute the TRANSITIVE CLOSURE of a directed graph. (6) Compare Floyd-Warshall's O(V^3) time against running Dijkstra or Bellman-Ford from every source, and choose the right tool for a given graph's density and edge weights.

15.1 Recap: One Source Was Chapter 13. Every Pair Is This Chapter.

Chapter 13 answered a SINGLE-SOURCE question: given one starting vertex s, what is the shortest distance from s to every other vertex? Dijkstra answered it in O((V+E) log V) when all weights are non-negative; Bellman-Ford answered it in O(V·E) even with negative edges, as long as no negative-weight cycle is reachable from s. This chapter asks the ALL-PAIRS SHORTEST PATHS (APSP) question instead: what is the shortest distance between EVERY ordered pair (i, j) of vertices, simultaneously? A perfectly valid way to answer APSP is to simply run a single-source algorithm once from EACH of the V vertices in turn — but as Section 15.5 will show, a direct dynamic-programming formulation over the SAME graph can beat that approach, especially on dense graphs.

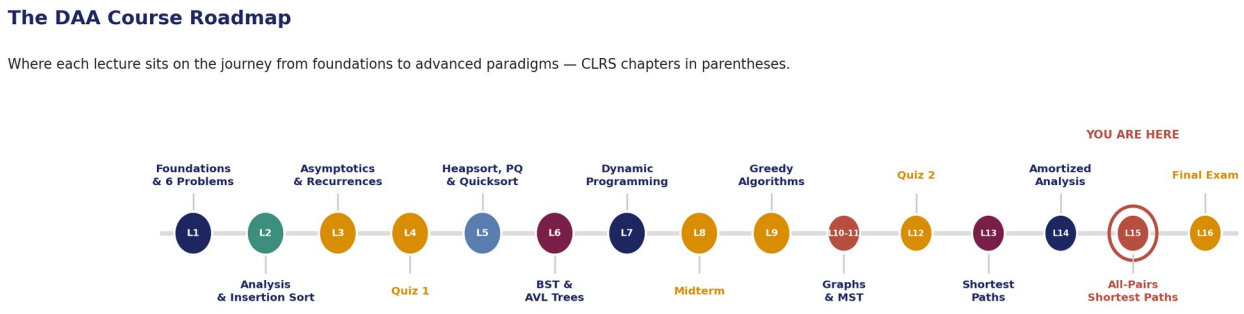

Figure 15.1 — Where this chapter sits in the sixteen-lecture DAA roadmap. Lecture 15 closes the shortest-paths unit by generalizing Chapter 13's single-source algorithms to all pairs at once.

APSP is a strict generalization, not a different problem

Every single-source shortest-paths query is a special case of an all-pairs query — you simply read off row s of the all-pairs distance matrix. The reverse is not true: a single-source algorithm gives you exactly one row, not the whole matrix. This is why APSP algorithms are natural to build as DYNAMIC PROGRAMS over the graph's own structure (as Chapters 8–9 built DP solutions over subproblem structure), rather than as V independent invocations of a single-source subroutine.

15.2 The Floyd-Warshall Recurrence

Floyd-Warshall reuses the exact same hypothetical Profitina Phase-4 5-hub network from Chapter 13's Bellman-Ford lecture (Figure 15.2) — the same directed edges, including the negative-weight promotional credit BDG→MKS (−4) — but this time computes the shortest sync cost between EVERY pair of hubs in one pass, not just from JKT.

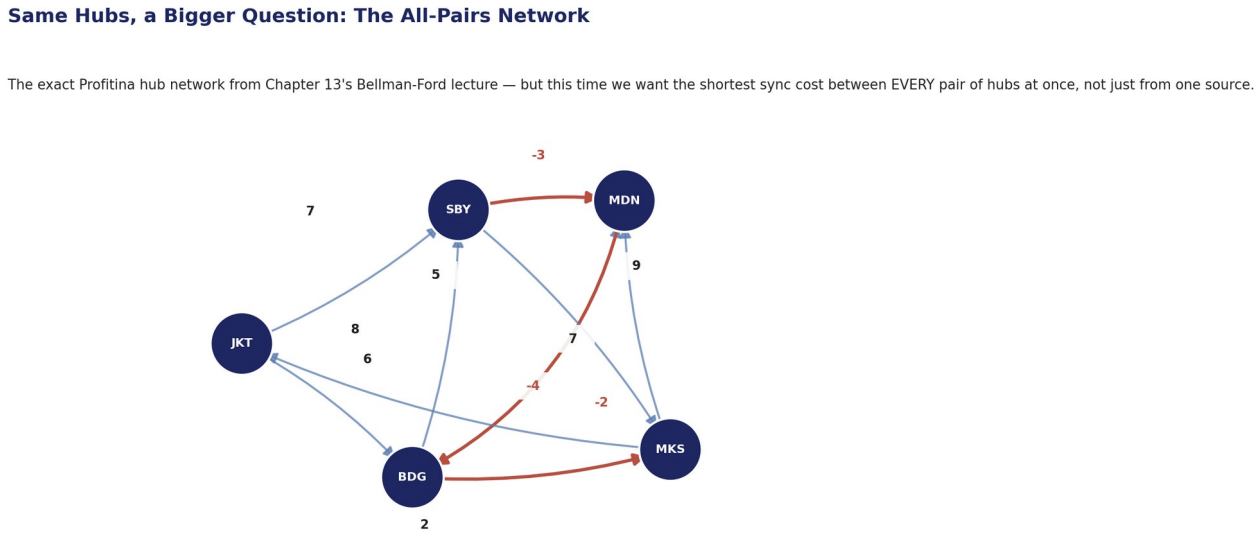

Figure 15.2 — The Profitina all-pairs network: identical hubs and edges to Chapter 13's Bellman-Ford graph. Negative edges are highlighted in terracotta; there is no negative-weight cycle.

Label the vertices 1 through V in some fixed order. Define D^(k)[i,j] as the weight of the shortest path from i to j that uses ONLY intermediate vertices drawn from {1, ..., k} — meaning every vertex on the path, other than i and j themselves, must have an index at most k. D^(0)[i,j] is therefore simply the direct edge weight from i to j (or infinity if no direct edge exists, or 0 if i = j) — no intermediate vertices are allowed yet. The key recursive insight: a shortest path from i to j using only {1, ..., k} as intermediates either (a) does not use vertex k at all, in which case its weight is D^(k-1)[i,j], or (b) does use vertex k exactly once, in which case its weight is D^(k-1)[i,k] + D^(k-1)[k,j] — the shortest i-to-k path plus the shortest k-to-j path, each of which only needs intermediates from {1, ..., k-1} because a shortest path never usefully visits the same vertex twice.

```
FLOYD-WARSHALL(W):    // W = weight matrix, W[i][i] = 0, W[i][j] = weight or
infinity
  D = copy of W                                  // D^(0)
  P = predecessor matrix, initialized from direct edges
  for k = 1 to V
      for i = 1 to V
          for j = 1 to V
              if D[i][k] + D[k][j] < D[i][j]
                  D[i][j] = D[i][k] + D[k][j]
                  P[i][j] = P[k][j]
  return D, P
```

After the outer loop finishes for k = V, D[i][j] = D^(V)[i,j] is the TRUE shortest distance from i to j — since every vertex in the graph is now an allowed intermediate, D^(V) considers every possible path. The entire algorithm is three nested loops over V, giving a clean $O(V^3)$ time bound and $O(V^2)$ space for the two matrices — no priority queue, no edge relaxation ordering, nothing but three loops and one comparison.

> **D[i][k] and D[k][j] must ALREADY be finite before adding them**
> A common implementation bug: if a language represents "infinity" as some large finite sentinel value (say, LLONG_MAX/4 in C++, since a true floating-point infinity is not always convenient), then computing D[i][k] + D[k][j] when one of those two entries is still "infinity" can silently overflow into a smaller-but-still-huge number that is LESS than the current D[i][j] — causing the algorithm to "relax" an entry that has, in truth, no path at all. The fix is to check that BOTH D[i][k] and D[k][j] are already finite (i.e., a real path exists so far) before even attempting the addition. This exact bug was caught and fixed during this chapter's own verification (Appendix 15.A) — the very first compile of 01_floyd_warshall.cpp printed the nonsense value 2305843009213693949 instead of INF in several cells of D^(2) through D^(4), until the missing finiteness guard was added.

15.3 Tracing Floyd-Warshall by Hand, Pass by Pass

Table 15.1 (Figure 15.3) walks through all five passes of Floyd-Warshall on the network from Figure 15.2, in the hub order JKT, BDG, MDN, SBY, MKS. Each pass k allows paths to route through exactly one more hub as an intermediate stop; only entries that actually improve are listed, and every unlisted entry simply carries forward unchanged from the previous pass. Every single value below was hand-verified in /tmp/daa_pilot/verify_ch15_problems.py — independently cross-checked against both an all-pairs Bellman-Ford run and brute-force enumeration of every simple path for all 25 ordered pairs — and matches the compiled C++ trace in Appendix 15.A exactly.

Floyd-Warshall's Trace: One Intermediate Hub at a Time

Each pass k allows paths to route THROUGH one more hub as an intermediate stop. Only entries that actually improve are shown; everything else carries forward unchanged. Verified against /tmp/daa_pilot/code/chapter15/01_floyd_warshall.cpp.

Pass k	Intermediate Added	What Improved (D[i][j] old → new)
1	JKT	MKS→BDG: ∞→8 (via MKS→JKT→BDG); MKS→SBY: ∞→9 (via MKS→JKT→SBY)
2	BDG	JKT→MDN: ∞→11; JKT→MKS: ∞→2; MDN→SBY: ∞→6; MDN→MKS: ∞→−6 (all routed through BDG)
3	MDN	SBY→BDG: ∞→−5; SBY→MKS: 9→−9; MKS→BDG: 8→5 (all routed through MDN)
4	SBY	JKT→BDG: 6→2; JKT→MDN: 11→4; JKT→MKS: 2→−2; MKS→BDG: 5→4; MKS→MDN: 7→6
5	MKS	BDG→JKT: ∞→−2; MDN→JKT: ∞→−4; MDN→SBY: 6→3; SBY→JKT: ∞→−7

Figure 15.3 — Floyd-Warshall's five passes on the hypothetical Profitina Phase-4 network. Pass k=1 (via JKT) closes 2 gaps; k=2 (via BDG) closes 4 more; k=3 (via MDN) improves 3 entries; k=4 (via SBY) improves 5 entries — the biggest single pass, since SBY offers a cheap route to everywhere; k=5 (via MKS) closes the final 4 gaps, all routes back into JKT.

Two entries deserve a closer look. First, JKT→MKS: at D^(1) it is still infinity (no path yet through only JKT); at D^(2), routing through BDG gives JKT→BDG→MKS = 6 + (−4) = 2; at D^(4), routing through SBY instead gives JKT→SBY→MKS = 7 + (−9) = −2, which is cheaper still, since by pass 4 the SBY→MKS entry has itself improved to −9 (via SBY→MDN→MKS at pass 3). This is exactly the recurrence's point: an entry can improve MULTIPLE times across different passes, each time because a cheaper route through a newly-allowed intermediate was discovered. Second, notice that pass k=5 (adding MKS as an allowed intermediate) is the ONLY pass that improves any entry in the JKT COLUMN — before pass 5, nothing routed back into JKT at all, since JKT has only one incoming edge (MKS→JKT, weight 2) and MKS itself was not yet a usable intermediate.

Watching the Matrix Fill In: D^(0) → D^(3) → D^(5)

Darker cells are more negative (cheaper); ∞ cells are still unreachable at that pass. By D^(3), most ∞ cells have already closed; by D^(5), the matrix is completely full — this graph is strongly connected, so every pair ends with a real shortest distance and zero ∞ remain.

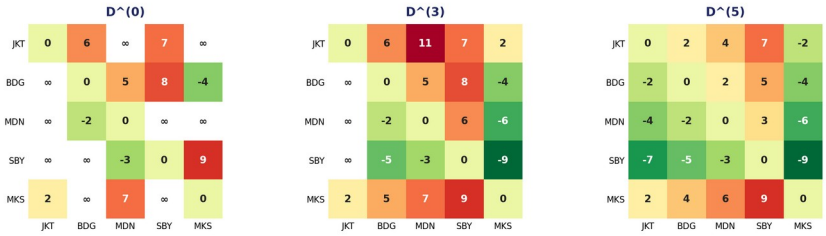

Figure 15.4 — The full 5×5 distance matrix at three snapshots: D^(0) (direct edges only, 9 of 25 cells still ∞), D^(3) (after JKT, BDG, MDN — 3 cells still ∞, all in the JKT column), and D^(5) (fully computed — 0 cells remain ∞, since this graph is strongly connected).

The JKT COLUMN in Figure 15.4's D^(5) panel — reading down: JKT→JKT=0, BDG→JKT=−2, MDN→JKT=−4, SBY→JKT=−7, MKS→JKT=2 — is the clearest illustration of pass k=5's effect. All four off-diagonal entries in that column were still ∞ as late as D^(4), because the ONLY edge pointing into JKT anywhere in this graph is MKS→JKT (weight 2), and until MKS itself became an allowed intermediate at pass 5, no other hub had any way to route through it to reach JKT at all. Once MKS is allowed: BDG→MKS→JKT costs (−4) + 2 = −2; MDN→MKS→JKT costs (−6) + 2 = −4; and SBY→MKS→JKT costs (−9) + 2 = −7 — three brand-new finite entries appearing in a single pass, exactly matching Figure 15.3's row for k=5.

15.4 Reconstructing Actual Paths, Not Just Distances

D^(5) alone tells you the shortest DISTANCE between every pair, but not which sequence of hubs realizes it. Floyd-Warshall solves this the same way Dijkstra and Bellman-Ford did in Chapter 13 — a predecessor matrix P, where P[i][j] records the vertex that directly precedes j on the current-best shortest i-to-j path. Whenever the main relaxation updates D[i][j] via intermediate k, it also copies P[i][j] = P[k][j] — the predecessor of j on the BEST KNOWN i-to-k-to-j path is whatever precedes j on the k-to-j leg, since the path's final segment (arriving at j) is unaffected by how you got to k.

Reconstructing a path is then a matter of walking backward from the destination: to find the path from i to j, start at j, repeatedly replace the current vertex with P[i][current] until you reach i, then reverse the sequence. Table 15.2 shows five reconstructed paths, each independently verified by summing its edge weights and confirming the total matches the corresponding D^(5) entry exactly (see Appendix 15.A):

From → To	Reconstructed Path	Path Weight	Matches D^(5)?
JKT → MKS	JKT → SBY → MDN → BDG → MKS	7 −3 −2 −4 = −2	Yes (−2)
SBY → BDG	SBY → MDN → BDG	−3 −2 = −5	Yes (−5)
MDN → SBY	MDN → BDG → MKS → JKT → SBY	−2 −4 +2 +7 = 3	Yes (3)
MKS → SBY	MKS → JKT → SBY	2 + 7 = 9	Yes (9)
JKT → MDN	JKT → SBY → MDN	7 −3 = 4	Yes (4)

Every reconstructed path above sums to exactly its D^(5) matrix entry, confirming the predecessor matrix is consistent with the distance matrix — this cross-check (path weight recomputed independently from the edge list, then compared against D^(5)) is exactly what Appendix 15.A's compiled program performs for these same five pairs.

15.5 Warshall's Algorithm: Transitive Closure as a Special Case

A closely related question asks only WHETHER j is reachable from i, ignoring edge weights entirely — the graph's TRANSITIVE CLOSURE. This is exactly Floyd-Warshall with the min/plus arithmetic replaced by boolean OR/AND: T^(k)[i,j] = T^(k-1)[i,j] OR (T^(k-1)[i,k] AND T^(k-1)[k,j]). This specialization is called WARSHALL'S ALGORITHM, published by Stephen Warshall in 1962 — the same year Robert Floyd published his shortest-path extension of the identical recurrence. Floyd's paper generalized Warshall's boolean recurrence to the min-plus semiring used throughout this chapter; the two results are usually credited together as "Floyd-Warshall" even though Warshall's boolean version and Floyd's weighted version were independent publications.

```
TRANSITIVE-CLOSURE(edges):    // boolean reachability matrix
  T = boolean matrix, T[i][i] = true, T[i][j] = true iff direct edge i->j exists
  for k = 1 to V
      for i = 1 to V
          for j = 1 to V
              if T[i][k] and T[k][j]
                  T[i][j] = true
  return T
```

Consider a small Profitina content-syndication graph (Appendix 15.A, 02_transitive_closure.cpp): Blog → Docs → CaseStudy → Changelog → Docs, where the last edge creates a cycle among Docs, CaseStudy, and Changelog. Hand-verified in /tmp/daa_pilot/verify_ch15_problems.py Part 2 against an independent depth-first-search reachability check: the final transitive closure shows Blog can reach all three other nodes, while Docs, CaseStudy, and Changelog can each reach the OTHER TWO within the cycle (but never Blog, since no edge points back to it) — confirming that once you enter a cycle, you can reach every other node in that cycle, but never escape to a node that only has incoming edges into the cycle from outside.

> **Same triple loop, different semiring**
>
> Floyd-Warshall's shortest-path recurrence and Warshall's reachability recurrence are the SAME algorithm, instantiated over two different algebraic structures (semirings): (min, +) for shortest paths, and (OR, AND) for reachability. This is a preview of a deeper idea covered in more advanced algorithms courses — many dynamic-programming recurrences over graphs generalize cleanly to any semiring satisfying a few basic axioms, and swapping the semiring can answer a different question with IDENTICAL code structure.

15.6 Floyd-Warshall vs. Running Dijkstra or Bellman-Ford V Times

Since a single-source algorithm answers one row of the all-pairs matrix, running it once per vertex is always a valid way to solve APSP. Figure 15.5 compares the three realistic options.

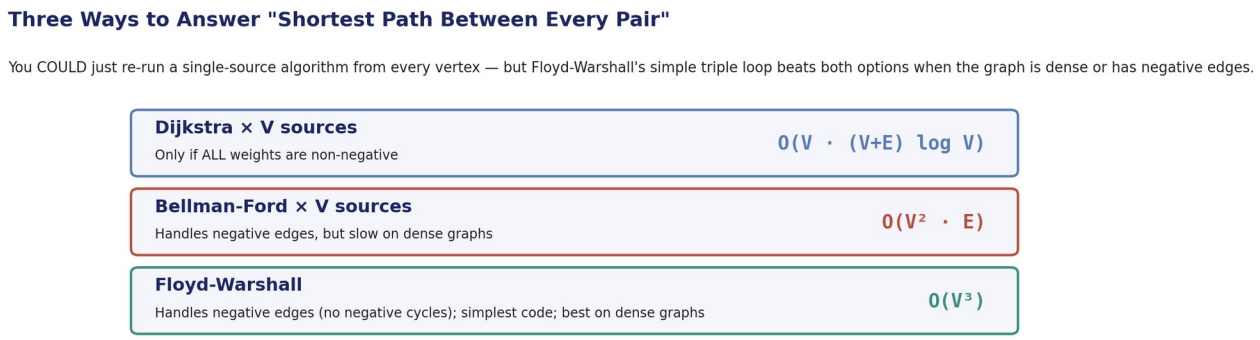

Figure 15.5 — Three ways to solve all-pairs shortest paths. Floyd-Warshall's $O(V^3)$ beats Bellman-Ford×V unconditionally, and beats Dijkstra×V on dense graphs (E close to V^2) or whenever any edge is negative.

Running Dijkstra from every source costs O(V · (V+E) log V) but ONLY works when every edge weight is non-negative — it is unusable on the hypothetical Profitina Phase-4 network in Figure 15.2, which

has three negative edges. Running Bellman-Ford from every source costs $O(V^2 \cdot E)$, which handles negative edges correctly but scales quadratically with V even before accounting for E — on a dense graph where E approaches V^2, this becomes $O(V^4)$, strictly worse than Floyd-Warshall's $O(V^3)$. Floyd-Warshall's $O(V^3)$ is independent of E entirely (it only ever looks at the V×V matrix, never the edge list directly after initialization), which makes it the natural choice whenever the graph is dense, whenever negative edges are present, or simply when the simplicity of a three-line triple loop outweighs a modest constant-factor speed difference on a small graph.

15.7 Profitina in Practice: Hub-to-Hub Latency Matrices

> **About this box**
> As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

This chapter's JKT/BDG/MDN/SBY/MKS network is a hypothetical illustration, not Profitina's infrastructure today — the real product runs as a single self-hosted server (see Chapter 10's note) — but Profitina's own public roadmap does name a future regional-expansion stage (Phase 4, once domestic dominance is secured) where a question like this would become real: "what is the CHEAPEST way to route a data-sync job between any two hubs, given today's inter-hub link costs?" Rather than re-running a single-source shortest-path query every time a new hub pair needs answering, a routing layer serving that future network could periodically recompute the FULL all-pairs latency matrix in one Floyd-Warshall pass — since the number of hubs would be modest (dozens, not millions) and the resulting $O(V^3)$ cost is comfortably cheap, while the payoff is an entire matrix ready to answer any future hub-pair query in O(1) lookup time, with no re-computation needed until the underlying link costs change. This trade-off — pay a modest $O(V^3)$ cost once, upfront, to make every future point query instant — is precisely why APSP algorithms matter operationally, not just academically: it is the same underlying trade-off behind precomputing an all-routes distance table for any logistics or content-delivery network.

15.8 Chapter Summary and Key Takeaways

- All-pairs shortest paths (APSP) generalizes Chapter 13's single-source problem: find the shortest distance between EVERY ordered pair of vertices, not just from one source.
- Floyd-Warshall's recurrence D^(k)[i,j] = min(D^(k-1)[i,j], D^(k-1)[i,k] + D^(k-1)[k,j]) is a dynamic program over WHICH VERTICES are allowed as intermediates — after k = V passes, every vertex is allowed, and D^(V) holds the true shortest distances.
- The entire algorithm is three nested loops: $O(V^3)$ time, $O(V^2)$ space — no priority queue, no edge-relaxation ordering, and (unlike Bellman-Ford) no dependence on the number of edges E at all.
- A predecessor matrix P, updated alongside D on every improving relaxation, lets you reconstruct the ACTUAL path realizing any shortest distance, not just its total weight.

- Warshall's algorithm is Floyd-Warshall specialized to the (OR, AND) semiring, answering "is j reachable from i?" (transitive closure) instead of "what is the shortest distance?" — same triple loop, different arithmetic.
- Floyd-Warshall's $O(V^3)$ beats running Bellman-Ford from every source unconditionally, and beats running Dijkstra from every source whenever the graph is dense or has any negative edge.

"Instead of asking the map for one road at a time, Floyd-Warshall asks it to fill in the ENTIRE distance table — one intermediate city at a time."

— a course aphorism

(Three nested loops. That's the whole algorithm.)

Lecture 16 is the course's final session: a comprehensive review spanning Lectures 13 through 15 (shortest paths, amortized analysis, and all-pairs shortest paths), in preparation for the Final Exam.

15.9 Review Questions

These questions are for self-study and class discussion; they are not graded.

1. Explain in your own words why D^(k-1)[i,k] and D^(k-1)[k,j] in the Floyd-Warshall recurrence only need intermediates from {1, ..., k-1}, even though the FULL path from i to j (through k) is allowed to use intermediates from {1, ..., k}.
2. Using Figure 15.3, identify every pass in which the entry D[MKS][BDG] improves, and state the new value and the intermediate vertex responsible for each improvement.
3. Prove that if a directed graph has a negative-weight cycle, Floyd-Warshall's diagonal entry D[i][i] for some vertex i on that cycle will become negative by the time the algorithm finishes — and explain how this gives a negative-cycle DETECTION test, parallel to Chapter 13's Bellman-Ford $|V|$-th-pass test.
4. Modify the transitive-closure recurrence in Section 15.5 to instead count the NUMBER of distinct paths (not just whether one exists) from i to j using intermediates from {1, ..., k} — what arithmetic operations replace OR and AND, and what new failure mode appears if the graph has a cycle?
5. Given that Floyd-Warshall's time is $O(V^3)$ independent of E, while Bellman-Ford-from-every-source is $O(V^2 \cdot E)$: for what range of E (relative to V) does Bellman-Ford-from-every-source actually beat Floyd-Warshall? At what edge density does this crossover occur?
6. A path-reconstruction implementation walks backward from j using P[i][current] until reaching i. What goes wrong if the graph contains a negative-weight cycle reachable from i and able to reach j, and why does Floyd-Warshall's correctness guarantee explicitly exclude graphs with negative cycles?

Appendix 15.A — Verification Log for Chapter 15 Source Code

As in previous chapters, every program shipped with this book is compiled and run against hand-verified examples before being included. Both 01_floyd_warshall.cpp and 02_transitive_closure.cpp were compiled with `g++ -O2 -std=c++17 -Wall`; their output matched the independent Python verification in /tmp/daa_pilot/verify_ch15_problems.py exactly, cross-checked against both an all-pairs Bellman-Ford run and brute-force path enumeration for every one of the 25 ordered vertex pairs. A finiteness-guard bug (see Section 15.2's TRAP box) was caught and fixed during this verification process, before this appendix's output was captured.

```
$ ./floyd_warshall
=== D^(0): direct edges only ===
         JKT   BDG   MDN   SBY   MKS
  JKT      0     6   INF     7   INF
  BDG    INF     0     5     8    -4
  MDN    INF    -2     0   INF   INF
  SBY    INF   INF    -3     0     9
  MKS      2   INF     7   INF     0
...
=== D^(5): intermediates now allowed through MKS ===
         JKT   BDG   MDN   SBY   MKS
  JKT      0     2     4     7    -2
  BDG     -2     0     2     5    -4
  MDN     -4    -2     0     3    -6
  SBY     -7    -5    -3     0    -9
  MKS      2     4     6     9     0

=== Path reconstruction spot-checks ===
  JKT -> MKS: JKT -> SBY -> MDN -> BDG -> MKS  (weight = -2)
  SBY -> BDG: SBY -> MDN -> BDG  (weight = -5)
  MDN -> SBY: MDN -> BDG -> MKS -> JKT -> SBY  (weight = 3)
  MKS -> SBY: MKS -> JKT -> SBY  (weight = 9)
  JKT -> MDN: JKT -> SBY -> MDN  (weight = 4)

Final D^(5) diagonal all zero (no negative cycle): CONFIRMED
```

```
$ ./transitive_closure
=== T^(0): direct edges + self-reachability ===
               Blog       Docs  CaseStudy  Changelog
     Blog         1          1          0          0
     Docs         0          1          1          0
CaseStudy         0          0          1          1
Changelog         0          1          0          1
...
Reachability summary:
  From Blog: can reach Docs, CaseStudy, Changelog
  From Docs: can reach CaseStudy, Changelog
  From CaseStudy: can reach Docs, Changelog
  From Changelog: can reach Docs, CaseStudy
```

SPOJ Practice — the companion book
This chapter's techniques are tested for real on the SPOJ online judge. Open Chapter 11 & 12 of the companion book “SPOJ Competitive Programming” (SPOJ folder, BI & EN) for all-pairs problems and a worked end-to-end Accepted example — complete with the tips, tricks, and hints that get solutions Accepted, without spoiling answers.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Chapter 23, All-Pairs Shortest Paths.)

[2] R. W. Floyd. "Algorithm 97: Shortest Path." Communications of the ACM, 5(6), 1962, 345. (The original one-paragraph publication of the shortest-path recurrence.)

[3] S. Warshall. "A Theorem on Boolean Matrices." Journal of the ACM, 9(1), 1962, 11–12. (The original transitive-closure algorithm this chapter's Section 15.5 specializes to.)

Profitina

profitina.com

COURSE TEXTBOOK · DESIGN AND ANALYSIS OF ALGORITHMS

Design and Analysis of Algorithms

From First Principles to Provably Correct, Demonstrably Efficient Code

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

Presented by Profitina · profitina.com
First Edition · 2026

CHAPTER 16 · EXERCISE CHAPTER

Practice Problems: Shortest Paths Through All-Pairs Shortest Paths

No new material here — just twelve problems designed to make sure Lectures 13 through 15 actually stuck, before the Final Exam.

Learning Objectives — after working through this chapter you will be able to:
(1) Trace Dijkstra's algorithm on a new graph and produce correct shortest-path distances from a single source. (2) Trace Bellman-Ford on a graph containing a negative edge, and explain why it needs |V|−1 passes rather than one. (3) Detect a negative-weight cycle using Bellman-Ford's extra Vth pass, and identify which edge exposes it. (4) Compare the asymptotic cost of Floyd-Warshall against repeatedly running a single-source algorithm, and choose correctly between them for a given scenario. (5) Apply the aggregate, accounting, and potential methods of amortized analysis to new operation sequences, and choose the most natural method for a new scenario. (6) Trace Floyd-Warshall's D matrix to completion on a new graph, reconstruct a shortest path from its predecessor matrix, and compute a transitive closure with Warshall's algorithm.

16.1 Recap: Lectures 13–15 in Brief

Lecture 13 opened the study of shortest paths proper, reusing Prim's exact relaxation mechanism — a key[]/parent[]-style update driven by a priority queue — but tracking cheapest PATH distance from a single source instead of cheapest EDGE weight. Dijkstra's algorithm handles the common case (all edge weights non-negative) greedily and efficiently, in O(E log V) with a binary heap; but it can be actively WRONG the moment a single negative edge appears, because its greedy commitment to a vertex's final distance the moment it is extracted assumes no cheaper path can appear later via a negative edge. Bellman-Ford handles negative edges correctly by relaxing every edge |V|−1 times, guaranteed to converge because the true shortest path between any two vertices in a graph with no negative cycle uses at most |V|−1 edges; a single extra (Vth) pass that still finds a relaxation is Bellman-Ford's built-in proof that a negative-weight cycle exists somewhere reachable from the source.

Lecture 14 stepped back from any one algorithm to ask a different question: what does an OPERATION cost, on average, across a long sequence — not in the worst individual case, but amortized over everything that happens? Three methods answer this rigorously without ever appealing to "average-case" probability: the aggregate method sums the true total cost of n operations and divides by n; the accounting method charges each operation a fixed amortized price, banking the surplus as credit that later expensive operations draw down, provided the credit balance never goes negative; and the potential method defines a potential function Φ over the data structure's state and computes amortized cost as actual cost plus the CHANGE in Φ, with the two telescoping cleanly across any sequence. All three were demonstrated on genuinely different structures — a doubling dynamic table,

a stack supporting MULTIPOP, and a binary counter — each giving the same O(1)-amortized-per-operation answer through a different lens.

Lecture 15 generalized Lecture 13's single-source problem to ALL pairs at once: Floyd-Warshall computes shortest-path distances between EVERY pair of vertices in a single $O(V^3)$-time, $O(V^2)$-space dynamic program over which vertices are allowed as intermediate stops, with a predecessor matrix enabling full path reconstruction afterward. Warshall's original algorithm is the identical recurrence specialized to (OR, AND) instead of (min, +), computing a graph's full transitive closure — reachability between every pair — rather than distances. Floyd-Warshall's $O(V^3)$ unconditionally beats running Bellman-Ford from every vertex ($O(V^2E)$, and E can be as large as V^2); against running Dijkstra from every vertex ($O(VE \log V)$), the comparison depends on density — Floyd-Warshall wins as the graph gets denser or as more of the V possible sources are actually needed.

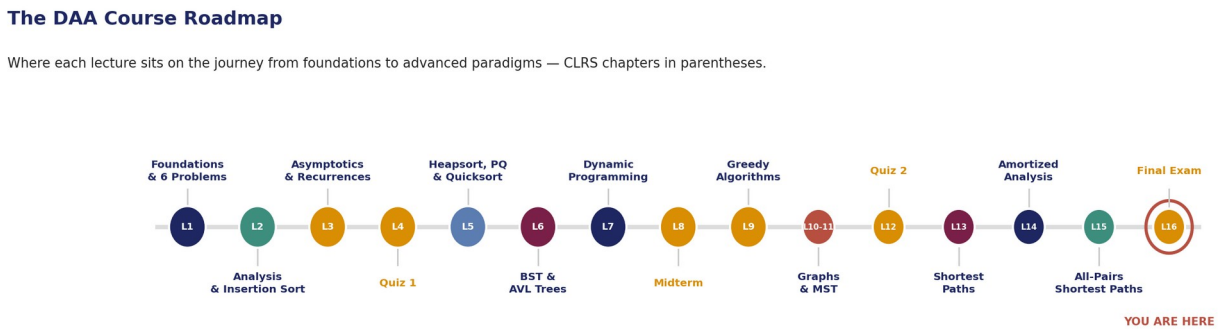

Figure 16.1 — This chapter sits at Lecture 16 in the sixteen-lecture DAA roadmap: the final checkpoint, immediately before the Final Exam.

16.2 How to Use This Exercise Chapter

> **What this chapter is — and is not**
>
> This is a Bab Latihan: a practice chapter, not a new-content chapter. Every problem below is anchored to specific material from Lectures 13–15 (see Figure 16.2). Each problem comes with a HINT — a nudge toward the right way of thinking — but deliberately NOT a full worked solution or source code. Working through the problem yourself, even imperfectly, teaches far more than reading a finished answer.

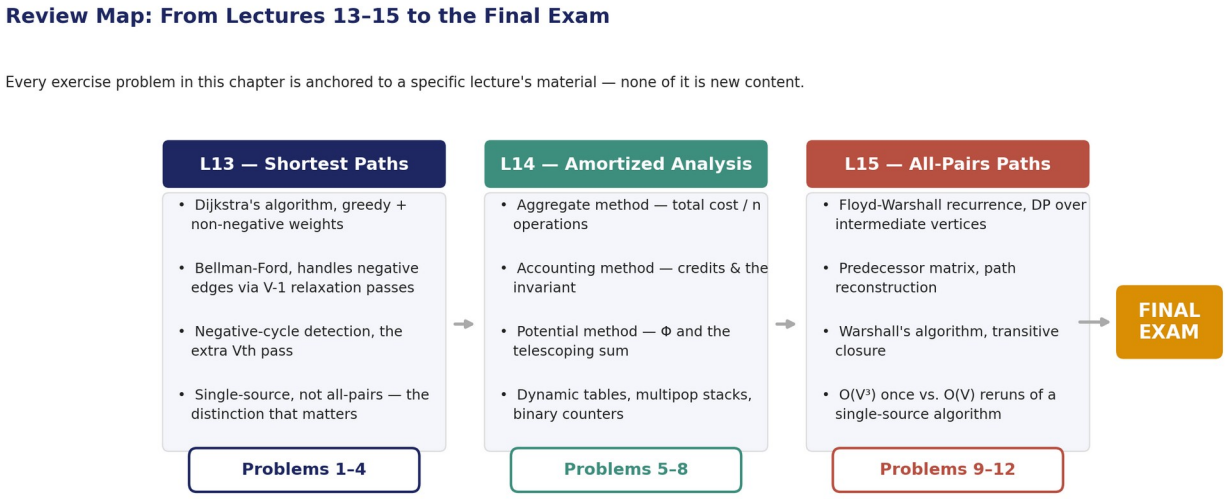

Figure 16.2 — Every problem in Section 16.3 traces back to one of the previous three lectures.

Before you start

Try each problem for real before reading its hint. If you get stuck, read only the hint — not a solution — and try again. This is exactly the discipline the Final Exam will reward: an open-book exam allows references, but not a substitute for your own reasoning.

16.3 Practice Problems

16.3.1 Shortest Paths (Lecture 13)

Problem 1 *[Easy]*. A five-hub network has directed links S→A(4), S→C(1), C→A(2), C→D(7), A→B(3), A→D(6), D→B(1). All weights are non-negative. Trace Dijkstra's algorithm from source S and state the final shortest-distance to every other vertex, in the ORDER the algorithm settles them.

Hint

At each step, extract the not-yet-settled vertex with the smallest tentative distance, then relax every edge leaving it. Watch carefully which vertex settles second — it is not necessarily the one with the smallest DIRECT edge weight from S.

Problem 2 *[Medium]*. The same style of network — S, A, B, C — has directed links S→A(5), S→B(8), A→B(−3), A→C(2), B→C(4). Trace Bellman-Ford from source S for the full $|V|-1 = 3$ passes (processing edges in the order B→C, A→B, A→C, S→B, S→A each pass), showing the distance table after EVERY pass, then run one extra (4th) pass and confirm nothing changes.

Hint

The negative edge A→B means an early pass can settle B at a distance that looks final but is not — B's distance only reaches its true minimum once A's distance has itself stabilized, which can take more than one pass with this edge order. Watch dist(B) and dist(C) especially closely across passes 1 through 3.

Problem 3 *[Medium]*. A different four-vertex network — S, A, B, C — has directed links S→A(1), A→B(−2), B→C(−2), C→A(1). Run Bellman-Ford for |V|−1 = 3 passes, then run the extra (4th) pass. Show that at least one distance STILL decreases in this extra pass, identify which edge causes the decrease, and state the total weight of the cycle responsible.

> **Hint**
> A distance that keeps shrinking every pass, never converging even after |V|−1 passes, is Bellman-Ford's signature for a negative-weight cycle somewhere reachable from the source. Sum the weights around the cycle A→B→C→A directly — a genuinely negative total confirms what the extra pass already told you.

Problem 4 *[Hard]*. In a hypothetical future scenario matching Profitina's own publicly stated Phase-4 regional-expansion roadmap (not its current single-server deployment), Profitina needs a full routing table between ALL 25 regional hubs, refreshed once per hour; the network has 60 candidate links, and some carry a negative cost adjustment (a rebate), which rules out Dijkstra entirely. Compute the arithmetic cost of running Bellman-Ford from every one of the 25 hubs ($O(V^2E)$) versus running Floyd-Warshall once ($O(V^3)$), and state which approach is cheaper.

> **Hint**
> Substitute V=25 and E=60 directly into both formulas — $O(V^2E)$ and $O(V^3)$ — and compare the two resulting numbers. This is exactly the kind of arithmetic comparison Lecture 15 established as an UNCONDITIONAL win for Floyd-Warshall over Bellman-Ford×V, since E can never exceed V^2 regardless of how sparse or dense the actual graph is.

16.3.2 Amortized Analysis (Lecture 14)

Problem 5 *[Easy]*. A dynamic table starts empty and doubles its capacity (copying every existing element) whenever an insert would overflow it. For a sequence of n=10 consecutive inserts, the i-th insert costs i when a doubling is triggered (i.e., whenever i−1 is zero or an exact power of two) and costs 1 otherwise. List the cost of EVERY one of the 10 inserts, compute the total cost, and compute the amortized cost per operation.

> **Hint**
> Work out which insert numbers (from 1 to 10) trigger a doubling first — check i−1 against 0, 1, 2, 4, 8 — before summing anything. The aggregate method's bound of roughly 3 per operation should comfortably cover whatever total you compute.

Problem 6 *[Medium]*. A stack starts empty and receives this 13-operation sequence: six PUSHes, then MULTIPOP(4), then three more PUSHes, then MULTIPOP(10), then two final PUSHes. Using the accounting method (charge 2 per PUSH: 1 pays for the push itself, 1 is banked as credit; each element popped by a MULTIPOP is paid for from banked credit), compute the actual total cost of the whole sequence AND the running credit balance after every single operation. Confirm the balance never goes negative.

> **Hint**
> A MULTIPOP(k) call can never pop more elements than are actually on the stack — its ACTUAL cost is the number of elements it removes, not k itself. Track the stack's size alongside the credit balance so you always know how many elements a MULTIPOP call can really remove.

Problem 7 *[Medium]*. A binary counter starts at 01011 (decimal 11) and receives 10 consecutive INCREMENT operations. Using the potential method with Φ = number of 1-bits currently set, compute Φ before and after EVERY increment, the actual cost of each increment (number of bits flipped), and confirm the amortized cost (actual + ΔΦ) is the same constant for every single increment in the sequence.

> **Hint**
> An INCREMENT flips a run of trailing 1s to 0 and then flips exactly one trailing 0 to 1 — its actual cost is (number of trailing 1s flipped) + 1. Starting from 01011 rather than all-zero changes the specific Φ values you will see, but the potential-method IDENTITY (amortized = actual + ΔΦ, and the telescoping sum over any run of increments) still has to hold exactly.

Problem 8 *[Hard]*. Profitina is adding a new resizable ring-buffer to its log-ingestion pipeline; it doubles in the exact same way as Lecture 14's dynamic table whenever a write would overflow it. Which of the THREE amortized-analysis methods (aggregate, accounting, or potential) would you reach for FIRST to prove this buffer supports O(1)-amortized writes, and why would the other two be comparatively more awkward to set up for this particular structure?

> **Hint**
> Revisit Section 14.5's direct comparison of the three methods' relative strengths: the aggregate method is fastest to apply when you can sum a closed-form total directly; the accounting method is most intuitive when a physical story ("banked credit") is easy to tell; the potential method is most powerful when the structure's "stored-up work" doesn't correspond to any simple credit story, but is hardest to set up from scratch since you must invent Φ yourself.

16.3.3 All-Pairs Shortest Paths (Lecture 15)

Problem 9 *[Medium]*. A four-hub network P, Q, R, S has directed links P→Q(3), P→S(6), Q→R(2), R→P(1), R→S(−3), S→Q(4). Trace Floyd-Warshall's D matrix through all four passes (k=P, then Q, then R, then S as the allowed intermediate) and state the FINAL 4×4 distance matrix.

> **Hint**
> Process one intermediate vertex k at a time, across ALL (i,j) pairs, before moving to the next k — exactly as Section 15.2's recurrence prescribes. Watch specifically what happens to the R→Q entry: is the direct route via P really the shortest once S is allowed as an intermediate stop too?

Problem 10 *[Medium]*. Using the SAME P, Q, R, S network from Problem 9, and the predecessor matrix Floyd-Warshall builds alongside the D matrix, reconstruct the actual shortest PATH (not just the distance) from S to P, listing every intermediate hub visited in order.

Hint

Walk the predecessor matrix backward from the destination: Π[S][P] gives the vertex that comes IMMEDIATELY BEFORE P on the shortest S-to-P path, then Π[S][that vertex] gives the one before THAT, and so on until you reach S itself — then reverse the list you built.

Problem 11 *[Hard]*. A four-stage content-workflow graph has directed edges Draft→Review, Review→Published, Published→Archived, and Review→Archived (a direct shortcut, skipping Published). Using Warshall's algorithm, compute the FULL transitive closure — which stage can reach which other stage, directly or indirectly — and state which stage(s), if any, can reach every other stage.

Hint

Warshall's algorithm is Floyd-Warshall's identical recurrence with (min, +) replaced by (OR, AND): T[i][j] becomes true if it was already true, OR if T[i][k] AND T[k][j] are both true for the current intermediate k. Check Archived specifically — does it have any outgoing edge at all?

Problem 12 *[Hard]*. Profitina's all-pairs routing table (Problem 4's scenario) is recomputed hourly with Floyd-Warshall, but in practice only ONE link's cost actually changes between most refreshes. Does a single changed edge weight force a full $O(V^3)$ recomputation from scratch, or can you argue for a cheaper incremental update? Write a short paragraph laying out the difficulty precisely — where exactly does a change to ONE entry of W potentially propagate to, in the worst case?

Hint

Consider that vertex k's row and column in D depend on paths that may route THROUGH the changed edge's endpoints as an intermediate stop, for EVERY pair (i,j) — not just pairs involving the two endpoints directly. Try to construct (or at least describe) a small example where changing one edge weight alters the shortest distance between two vertices that don't touch that edge at all.

16.4 Final Exam Format: Open-Book, Take-Home Essay

The Final Exam is an open-book, take-home essay assignment covering the ENTIRE course — every lecture from Lecture 1 through Lecture 15, with particular emphasis on the material reviewed in this chapter (Lectures 13 through 15) as the most recent additions. You will have one week to submit written answers with full reasoning; a correct final answer with no justification earns little credit, since the point of an essay-format exam is to see HOW you think, not just what you conclude.

Open-book means references, not co-authors

You may consult the textbook, your notes, and lecture slides while answering. You may NOT collaborate with classmates or submit reasoning that is not genuinely your own — an open-book exam tests whether you can APPLY what you have studied, unsupervised, which is exactly what "designing and analyzing algorithms" means in practice. If you quote a source directly, cite it.

Criterion	What it looks for	Weight
Correctness of reasoning	Is the logic, proof, trace, or derivation actually valid — not just the final answer?	40%
Clarity of proof / derivation	Can a reader follow each step without having to guess what you meant?	25%
Proper use of notation	Are O/Ω/Θ, recurrences, amortized-cost identities, and matrix/graph diagrams stated precisely, matching earlier chapters' conventions?	20%
Presentation	Is the answer organized, legible, and easy to grade?	15%

16.5 Profitina in Practice: Self-Review Before Submission

> **About this box**
> As in every chapter, this box connects course material to Profitina, a real Indonesian AI-visibility (Generative Engine Optimization) product built by the author, without exposing proprietary implementation details.

Before a full-course capstone review ships at Profitina — a quarterly architecture retrospective, not unlike revising for a comprehensive final exam — engineers run through a checklist very similar to Appendix 16.A below: has a claimed $O(V^3)$ all-pairs refresh actually been checked against the $O(V^2E)$ alternative with real V and E numbers, or just assumed to be the better choice out of habit? Has an amortized O(1) claim about a resizing buffer actually been proved with one of the three named methods, or just asserted because "doubling is usually fine"? This habit of adversarial self-checking before trusting a familiar-sounding conclusion — the same discipline behind the bug-hunting lesson repeated throughout this textbook, that a visually clean result can still be mathematically wrong — is exactly what a comprehensive, open-book Final Exam is built to train.

16.6 Chapter Summary

- This chapter introduced no new material — it is the final checkpoint covering Lectures 13 through 15, immediately before the Final Exam.
- Twelve problems span the full range reviewed: Dijkstra's algorithm, Bellman-Ford and negative-cycle detection, the aggregate/accounting/potential methods of amortized analysis, and Floyd-Warshall with path reconstruction and Warshall's transitive closure.
- Each problem includes a hint, not a solution — working through the reasoning yourself is the point.
- The Final Exam is an open-book, take-home essay covering the ENTIRE course: references are allowed, but the reasoning must be your own, and correctness of REASONING (not just the final answer) is the majority of the grade.

“The final exam does not ask what an algorithm is — it asks whether you can still derive it after the lecture slides are gone.”

— a course aphorism

(Recognizing an algorithm's name is not the same as being able to trace it by hand under time pressure.)

Appendix 16.A — Self-Check Checklist (Non-Graded)

Before submitting any answer — in this chapter's problems or in the Final Exam itself — run it through this checklist. It costs a few minutes and catches most of the mistakes graders see most often.

- Did I actually verify a claimed distance, cycle, or matrix entry on the concrete example given, rather than just asserting it holds?
- In any hand-traced structure (Dijkstra's settled-vertex order, Bellman-Ford's pass-by-pass distance table, Floyd-Warshall's D matrix), did I show EVERY intermediate state, not just the final result?
- If I claimed a negative cycle exists (or does not), did I check specifically whether a distance keeps shrinking past $|V|-1$ passes — not just whether some edge weight happens to be negative?
- If I compared Floyd-Warshall against Dijkstra×V or Bellman-Ford×V, did I substitute the ACTUAL V and E from the scenario, rather than reasoning from "dense" or "sparse" alone?
- If I used the accounting or potential method, did I confirm the credit balance (or Φ-based amortized cost) never goes negative and telescopes correctly across the WHOLE sequence, not just the first few operations?
- Did I reread my own answer as if I were a skeptical grader looking for a gap in the argument?

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, Fourth Edition. MIT Press, 2022. (Chapters 17, 22–23).

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

[3] R. Bellman. "On a Routing Problem." Quarterly of Applied Mathematics, 16(1), 1958, 87–90.

[4] R. W. Floyd. "Algorithm 97: Shortest Path." Communications of the ACM, 5(6), 1962, 345.