Design and Analysis of Algorithms (EF234405)

Lecture 1 — Foundations, Framework, and First Problems

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*, Chapter 1

Lecturer: Irfan Subakti — `yifana@gmail.com`

Department of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya

Today we build the method we will use for the rest of the semester. Book: Chapter 1, and Cormen et al., Introduction to Algorithms, 4th ed., Chapter 1.

Agenda

1. What an algorithm is, and how it differs from a program and from a data structure.

2. The two goals: **correctness** (proved, never merely tested) and **efficiency** (measured against a budget).

3. The **five design paradigms** and the **five-phase framework**: Understand, Abstract, Design, Analyze, Implement.

4. Six real judge problems, from the statement to accepted C++: *Columns*, *Journey*, *Unique Numbers*, *Minimum and Maximum Numbers*, *Interesting Sequence*, *Recursive Function 1*.

5. The judges, the books and the road map — and your workbench: **VS Code and a C++17 compiler** on Windows, macOS or Linux.

Items 1 to 3 take the first half of the meeting, item 4 the second half; item 5 is the homework that makes Chapter 2 possible. Book: Chapter 1, Sections 1.1–1.19.

Where We Begin

This course trains **computational thinking**, which means four things, practiced in every chapter.

Model the problem algorithmically — exactly what comes in, exactly what must go out, exactly which inputs are possible.

Design an algorithm — choose one of a handful of strategies and lay out the steps.

Analyze correctness and complexity — *prove* the steps are right for every allowed input, then compute the time and memory they need.

Implement it cleanly in C++. Notice the order: **the code is the last step, not the first.**

The book assumes nothing: every symbol is introduced in words before it is used, and every formula is derived, not announced. Book: Section 1.1.

What an Algorithm Is

DEFINITION. An **algorithm** is a finite, step-by-step computational procedure that transforms an *input* into an *output*.

The word comes from al-Khwarizmi, the ninth-century Persian scholar. A recipe is the best first analogy: for *nasi goreng*, the rice, egg and shallots are the **input**, the ordered steps are the procedure, and the plate of fried rice is the **output**.

Knuth's five properties: **(1)** a clearly defined input; **(2)** a clearly defined output; **(3) definiteness** — "add 3 grams of salt", never "salt to taste".

(4) Finiteness — it stops on *every* valid input. The Collatz rule ("halve if even, else $3n+1$") is not known to stop, so it is not an algorithm for anything. **(5) Effectiveness** — every step can actually be carried out.

EXAMPLE. "Remember the first number; for each remaining number, if it is bigger, remember it instead; at the end, announce what you remember." Input, output, definiteness, finiteness (n steps), effectiveness (one comparison) — all five hold. It is an algorithm, and our first C++ program.

Book: Section 1.2.1.

Program and Data Structure

A **program** is an algorithm expressed in a programming language. The algorithm is the *idea*; the program is its *realization*. You can be wrong in two places — the idea or the typing — and only a proof tells you which. And the algorithm lasts: C++ will be replaced one day, binary search will not.

A **data structure** organizes data so that it can be accessed and modified efficiently. Recipe and kitchen: same recipe, better equipment, faster dish. Table 1.1 is the map of the territory.

Table 1.1 The data structures used in this book, and the single thing each one is best at.

Structure	C++ name	What it is good at
Array / dynamic array	`int a[]`, `vector`	Reading or writing the i-th item instantly
Linked list	`list`	Inserting and removing in the middle cheaply
Stack	`stack`	Last in, first out — undo, recursion, bracket matching
Queue	`queue`	First in, first out — scheduling, breadth-first search
Hash map	`unordered_map`	Looking up by a key, on average in constant time
Balanced search tree	`map`, `set`	The same, but keeping the keys in sorted order
Heap / priority queue	`priority_queue`	Repeatedly pulling out the smallest or largest item
Graph	adjacency list	Modeling relationships, roads, links, dependencies

Do not memorize Table 1.1 now; each entry arrives when a problem forces you to want it. Book: Sections 1.2.2–1.2.3 and Table 1.1.

The Goals We Are Aiming At

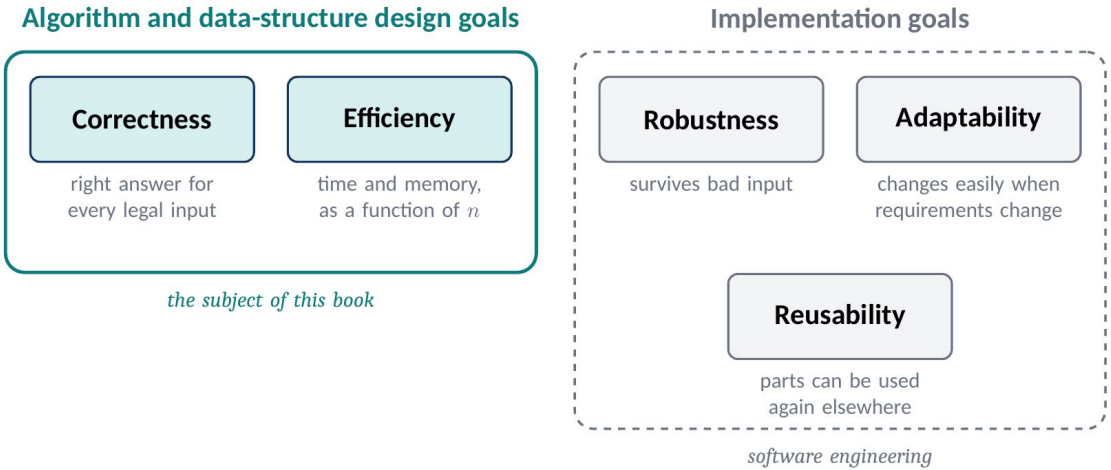


Figure 1.1 The goals of data structure and algorithm design (correctness, efficiency) set beside the goals of implementation (robustness, adaptability, reusability).

Figure 1.1: correctness and efficiency (left) belong to algorithm design and are the subject of this course; robustness, adaptability and reusability (right) belong to software engineering. A fast, reusable, robust and wrong algorithm is worthless. Book: Section 1.2.4 and Figure 1.1.

Goal One — Correctness

> **DEFINITION.** An algorithm is **correct** if, for *every* input allowed by the problem statement, it halts and produces the output the problem asks for.

With $1 \le n \le 1000$ and $1 \le a_i \le 1000$ there are 1000^{1000} inputs of size 1000 alone. Testing can show the presence of bugs, never their absence. Only a proof can.

Direct proof — a lower bound, then a construction that attains it. **Proof by contradiction** — assume a wrong answer and derive the impossible (*Journey*, Section 1.7).

Loop invariant — true before the loop, preserved by every pass, useful at the end. **Induction** — true at the bottom, and each rung leads to the next (*Interesting Sequence*, Section 1.10).

All four techniques appear in this chapter. Book: Section 1.3.1.

Stating a Problem Precisely

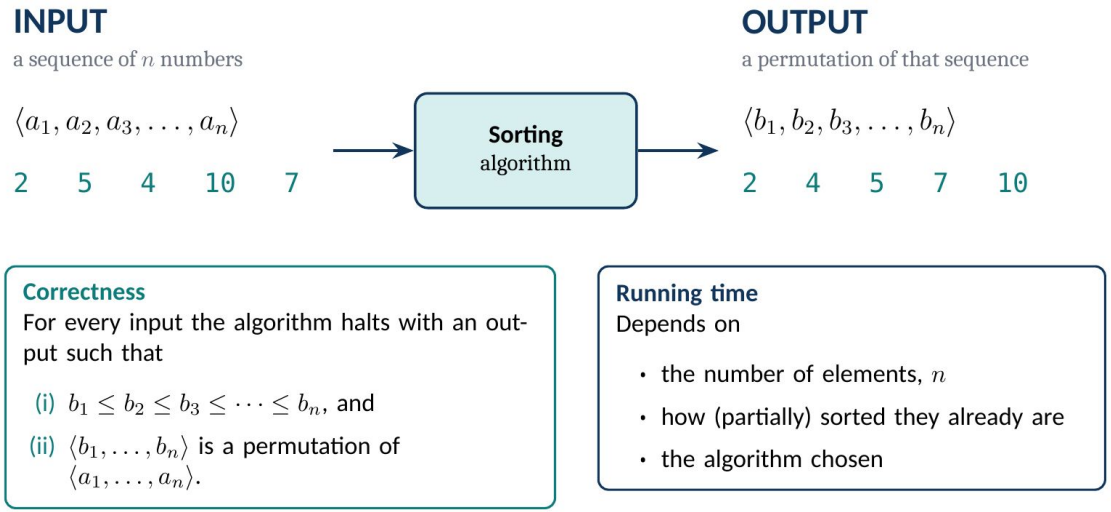


Figure 1.2 The sorting problem: a sequence goes in, a permutation of that sequence comes out, and correctness means two separate conditions must both hold.

*Figure 1.2 is Phase 1 applied to the most-studied problem in computer science; notice that it insists on **two** conditions, not one. Book: Section 1.3.2 and Figure 1.2.*

The Sorting Problem

> **THE SORTING PROBLEM.** *Input:* a sequence of n numbers $a_1, a_2, \ldots, a_n$. *Output:* a sequence $b_1, b_2, \ldots, b_n$ such that (i) $b_1 \le b_2 \le \cdots \le b_n$, and (ii) $b_1, b_2, \ldots, b_n$ is a permutation of $a_1, a_2, \ldots, a_n$.

n is how many numbers there are; the subscript in a_1 is a label, not a multiplication; a **permutation** is a rearrangement — nothing added, nothing thrown away.

Input $[2, 5, 4, 10, 7]$. Output $[2, 4, 5, 7, 10]$ is **correct**: in order, and exactly the given numbers.

Output $[1, 2, 3, 4, 5]$ is perfectly sorted and **wrong**: 1 and 3 were never there, 7 and 10 have vanished. A checker that asks only "is it sorted?" accepts it.

Output $[2, 5, 4, 10, 7]$ satisfies (ii) and fails (i). Also wrong. The problem statement, not the test, defines correctness.

Book: Section 1.3.2.

Insertion Sort in C++

How everybody sorts a hand of cards: pick up the next card and move it left past every bigger card, then drop it into the gap. Listing 1.1 is the whole of it.

Listing 1.1 Insertion sort in C++, complete and runnable, sorting the sequence from Figure 1.2.

```cpp
#include <cstdio>

void insertionSort(int a[], int n) {
    for (int j = 1; j < n; j++) {          // a[0..j-1] is already sorted
        int key = a[j];                    // the card we are about to place
        int i = j - 1;
        while (i >= 0 && a[i] > key) {     // slide bigger cards one place right
            a[i + 1] = a[i];
            i = i - 1;
        }
        a[i + 1] = key;                    // the hole left behind is where key belongs
    }
}

int main() {
    int a[] = {2, 5, 4, 10, 7};
    int n = 5;
    insertionSort(a, n);
    for (int i = 0; i < n; i++) printf("%d%c", a[i], i + 1 == n ? '\n' : ' ');
    return 0;
}
```

```
2 4 5 7 10
```

The output matches the right-hand side of Figure 1.2. One correct answer proves nothing, so we prove it next. Book: Section 1.3.3 and Listing 1.1.

The Loop Invariant and Its Proof

DEFINITION. A **loop invariant** is a statement about the program's variables that is (a) true before the loop runs — *initialization*; (b) still true after each pass if it was true before it — *maintenance*; and (c) useful when the loop stops — *termination*.

LOOP INVARIANT (insertion sort). At the start of each pass with counter j, the sub-array $a[0\ldots j-1]$ contains exactly the elements that originally occupied positions 0 through $j-1$, and they are in sorted order.

Initialization. $j=1$: $a[0\ldots 0]$ is one element — trivially sorted, and nothing has moved.

Maintenance. Shifting loses nothing (the overwritten value was already copied right, or it is the saved `key`), and `key` lands after every element $\le$ it and before every element $>$ it, so $a[0\ldots j]$ is sorted.

Termination. $j=n$: $a[0\ldots n-1]$ holds exactly the original elements, in order — conditions (ii) and (i). ▰

The numbers $2,5,4,10,7$ *appear nowhere in the proof: it covers every input of every size. The idea goes back to Floyd and Hoare; Gries made it a discipline. Book: Section 1.3.3.*

The Invariant, Made Visible

Listing 1.2 A tracing version of insertion sort that prints the loop invariant after every pass.

```cpp
#include <cstdio>

int a[] = {2, 5, 4, 10, 7};
const int n = 5;

void show(const char* label, int j) {
    printf("%s", label);
    for (int i = 0; i < n; i++) {
        if (i == j) printf(" |");          // the bar marks the sorted/unsorted boundary
        printf(" %d", a[i]);
    }
    printf("\n");
}

int main() {
    show("start        ", 1);
    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;
        printf("after j = %d: ", j);
        for (int t = 0; t < n; t++) {
            if (t == j + 1) printf(" |");
            printf(" %d", a[t]);
        }
        printf("\n");
    }
```

```
    return 0;
}
```

```
start         2 | 5 4 10 7
after j = 1:  2 5 | 4 10 7
after j = 2:  2 4 5 | 10 7
after j = 3:  2 4 5 10 | 7
after j = 4:  2 4 5 7 10
```

Listing 1.2 draws a bar between the sorted part and the untouched part. Read left of each bar: sorted every time, always the original occupants. Book: Section 1.3.3 and Listing 1.2.

Goal Two — Efficiency and the Budget

Bubble sort needs about $n^2 = 10^{12}$ operations for $n = 10^6$ (an hour); merge sort about $n \log_2 n \approx 2 \times 10^7$ (twenty milliseconds). Both are correct.

> **KEY INSIGHT — THE OPERATION BUDGET.** A modern computer performs roughly 10^8 to 10^9 simple operations per second. More than about 10^8 operations on the largest allowed input will very probably exceed the time limit.

Table 1.2 Turning a constraint into a design decision. The figures in the middle columns were computed by a short C++ program; the last column applies the 10^8 budget to them.

Largest n	n	$n \log_2 n$	n^2	What you can afford
10^3	10^3	$\approx 10^4$	10^6	Even $O(n^3)$ is fine
10^5	10^5	$\approx 1.7 \times 10^6$	10^{10}	$O(n \log n)$ yes, $O(n^2)$ no
10^6	10^6	$\approx 2.0 \times 10^7$	10^{12}	$O(n \log n)$ yes, $O(n^2)$ hopeless
10^8	10^8	$\approx 2.7 \times 10^9$	10^{16}	Only $O(n)$, and be careful
10^{18}	—	—	—	Only $O(\log n)$ or $O(1)$

Look at the last row of Table 1.2: at $n = 10^{18}$ you cannot even *read* n items, so the solution must exploit structure. Correctness without efficiency is *Time Limit Exceeded*; efficiency without correctness is *Wrong Answer*. You need both.

Big-O is defined rigorously in Chapter 3; for now $O(f(n))$ says how time grows with the input. Book: Sections 1.3.4–1.3.6 and Tables 1.2–1.3.

Paradigm 1 — Incremental

The move. Process the input one element at a time, keeping a running state; when the input runs out, the state *is* the answer. Listing 1.3 is the simplest case.

Listing 1.3 The incremental paradigm at its simplest: one running variable, one pass.

```cpp
#include <cstdio>

int main() {
    int a[] = {3, 9, 2, 9, 4, 1};
    int n = 6;
    int best = a[0];                        // the maximum of the one-element prefix a[0..0]
    for (int i = 1; i < n; i++)
        if (a[i] > best) best = a[i];       // extend the prefix by one element
    printf("maximum = %d\n", best);
    return 0;
}
```

```
maximum = 9
```

Invariant: *`best` holds the largest value among* $a[0 \ldots i-1]$. **Brute force** is the incremental process that tries every possibility — often the right first thought, because it is obviously correct.

Columns and Journey are exactly this shape of loop. Book: Section 1.4.1 and Listing 1.3.

Paradigm 2 — Greedy

The move. At each step take the choice that looks best *right now*, and never reconsider.

Walk uphill in fog and you reach *a* summit, not necessarily the highest one. Greed is seductive and frequently wrong.

> **THE GREEDY-CHOICE PROPERTY.** Taking the locally best option never destroys the possibility of a globally best solution. With it, greedy algorithms are short and fast; without it, they are short, fast and wrong.

Proving the property is not optional (Chapter 9). Today it is proved three times: *Columns*, *Journey*, and the digits of *Minimum and Maximum Numbers*. Kruskal's minimum spanning tree is the classic example still to come.

Book: Section 1.4.2.

Paradigm 3 — Divide and Conquer

The move. Split into smaller sub-problems *of the same kind*, solve them recursively, combine the answers. Listing 1.4 shows all three steps.

Listing 1.4 Merge sort: split in half, sort each half, merge. The divide-and-conquer paradigm in twenty lines.

```cpp
#include <cstdio>

int tmp[64];

void mergeSort(int a[], int lo, int hi) {   // sorts a[lo..hi]
    if (lo >= hi) return;                   // a run of one element is already sorted
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, lo, mid);
    mergeSort(a, mid + 1, hi);
    int i = lo, j = mid + 1, k = lo;
    while (i <= mid && j <= hi) tmp[k++] = (a[i] <= a[j]) ? a[i++] : a[j++];
    while (i <= mid) tmp[k++] = a[i++];
    while (j <= hi)  tmp[k++] = a[j++];
    for (int t = lo; t <= hi; t++) a[t] = tmp[t];
}

int main() {
    int a[] = {2, 5, 4, 10, 7, 1, 8, 3};
    int n = 8;
    mergeSort(a, 0, n - 1);
    for (int i = 0; i < n; i++) printf("%d%c", a[i], i + 1 == n ? '\n' : ' ');
    return 0;
}
```

```
1 2 3 4 5 7 8 10
```

Divide: compute `mid`. Conquer: two recursive calls. Combine: the merging loops. Chapter 3 computes its running time from a recurrence; Chapter 7 develops the paradigm. Book: Section 1.4.3 and Listing 1.4.

Paradigm 4 — Dynamic Programming

The move. Sub-problems **overlap**, so solve each once, store it, look it up afterward — **memoization**. $F(0)=0$, $F(1)=1$, $F(n)=F(n-1)+F(n-2)$: the naive recursion recomputes the same values exponentially often.

Listing 1.5 Naive recursion against memoized recursion for Fibonacci numbers. The call counter makes the difference impossible to argue with.

```cpp
#include <cstdio>
#include <map>
using namespace std;

long long calls = 0;                          // how many times fibNaive is entered

long long fibNaive(int n) {
    calls++;
    if (n <= 1) return n;
    return fibNaive(n - 1) + fibNaive(n - 2);
}

map<int, long long> memo;

long long fibMemo(int n) {
    if (n <= 1) return n;
    if (memo.count(n)) return memo[n];        // already computed once: reuse it
    return memo[n] = fibMemo(n - 1) + fibMemo(n - 2);
}

int main() {
    calls = 0;
    long long v1 = fibNaive(30);              // finish the call BEFORE reading `calls`
    long long c1 = calls;
    printf("fibNaive(30) = %lld, calls made = %lld\n", v1, c1);

    long long v2 = fibMemo(30);
    int stored = (int)memo.size();
```

```
    printf("fibMemo(30)  = %lld, map entries = %d\n", v2, stored);
    printf("fibMemo(90)  = %lld\n", fibMemo(90));
    return 0;
}
```

```
fibNaive(30) = 832040, calls made = 2692537
fibMemo(30)  = 832040, map entries = 29
fibMemo(90)  = 2880067194370816120
```

Listing 1.5: 2,692,537 calls against 29 stored values. Section 1.11 applies the same idea to an input of 10^{18}. Book: Section 1.4.4 and Listing 1.5.

Paradigm 5 — Mathematical

The move. Replace the search by a formula. The **greatest common divisor** $\gcd(a,b)$ is the largest number dividing both. Euclid's identity, Equation (1.1):

$$\gcd(a,b)=\gcd(b,a \bmod b)\,(b>0),\quad \gcd(a,0)=a, \tag{1.1}$$

Why: with $a=qb+r$, every common divisor of a,b divides $r=a-qb$, and every common divisor of b,r divides a. Same divisors, same greatest one. Every two steps halve the larger number, so $O(\log \min(a,b))$ steps suffice.

Listing 1.6 The greatest common divisor computed twice: by exhaustive search, and by Euclid's algorithm.

```cpp
#include <cstdio>

long long gcdSlow(long long a, long long b) {   // test every candidate divisor
    long long g = 1;
    for (long long d = 1; d <= a && d <= b; d++)
        if (a % d == 0 && b % d == 0) g = d;
    return g;
}

long long gcdEuclid(long long a, long long b) { // Euclid: gcd(a,b) = gcd(b, a mod b)
    while (b != 0) { long long r = a % b; a = b; b = r; }
    return a;
}

int main() {
    printf("gcdSlow(1071, 462)   = %lld\n", gcdSlow(1071, 462));
    printf("gcdEuclid(1071, 462) = %lld\n", gcdEuclid(1071, 462));
    printf("gcdEuclid(30, 8)     = %lld\n", gcdEuclid(30, 8));
    printf("gcdEuclid(15, 12)    = %lld\n", gcdEuclid(15, 12));
    return 0;
}
```

```
gcdSlow(1071, 462)   = 21
gcdEuclid(1071, 462) = 21
gcdEuclid(30, 8)     = 2
gcdEuclid(15, 12)    = 3
```

By hand: $1071 \bmod 462 = 147$*;* $462 \bmod 147 = 21$*;* $147 \bmod 21 = 0$*, so* $\gcd = 21$ *in three steps where Listing 1.6's slow search takes* 462*. Keep* 2 *and* 3 *in mind for Section 1.8. Book: Section 1.4.5, Equation (1.1) and Listing 1.6.*

The Five Paradigms in One Table

Table 1.3 The five design paradigms, the single move that characterizes each, and where each appears in this chapter.

Paradigm	The move	Correct when	Seen in this chapter
Incremental / brute force	Extend a partial answer by one element	Almost always; may be too slow	*Columns*, *Journey*
Greedy	Take the locally best option, never revisit	The greedy-choice property holds	*Columns*, *Journey*, *Min/Max Numbers*
Divide and conquer	Split, recurse, combine	Sub-problems are independent	Merge sort (Listing 1.4)
Dynamic programming	Split, recurse, **store**	Sub-problems overlap	*Recursive Function 1*
Mathematical / number-theoretic	Replace the search by a formula	You can prove the formula	*Unique Numbers*, *Interesting Sequence*

When stuck, ask in order: **can I just scan? can I be greedy and prove it? can I split it? do the splits repeat? is there a formula?** One of the five almost always applies — Table 1.3 is the checklist.

Book: Section 1.4.6 and Table 1.4.

The Five-Phase Framework

Table 1.4 The five phases. Nothing here is optional, and the order is not negotiable.

Phase	Name	What you produce
1	Understand	A restatement in your own words, the exact input/output/constraints, and a small example worked by hand
2	Abstract	The mathematical core of the problem, with the story stripped away
3	Design	The algorithm in plain English or pseudocode, plus the data structures it needs
4	Analyze	A correctness proof, a time complexity, a space complexity, and a check against the constraints
5	Implement	Clean code that is a direct translation of Phase 3

A PHASE YOU CANNOT SHOW AN ARTIFACT FOR IS A PHASE YOU SKIPPED. Table 1.4 names what each phase must *produce*: no hand-worked example means no Phase 1; no complexity figure means no Phase 4.

Every problem in the book — including every problem of the four evaluation chapters — is solved with these five phases. Book: Section 1.5 and Table 1.5.

The Phases in Practice

Understand. Restate the problem without looking; write the input, output and constraints; **work a small example by hand**. If it disagrees with the sample, you misread the problem — stop and find out why.

Abstract. Strip the story away. Prove a **lower bound** L, then **construct** something achieving L: the answer is exactly L.

Design. Choose a paradigm; write numbered steps in English or pseudocode; name the data structures. No C++ yet.

Analyze. A proof of correctness; time and space in Big-O; the largest input checked against 10^8 operations and the memory limit (10^8 `int`s are 400 MB). If it does not fit, return to Phase 2.

Implement. A short, direct translation of Phase 3. Complicated code means an unfinished design.

The six programs that follow are eight to twenty lines each — because the thinking was done first. Book: Sections 1.5.1–1.5.5.

Problem 1 — *Columns* (E-Olymp 10281)

Columns (E-Olymp 10281)

Time limit **1 s** Memory limit **128 MB**

Statement

Given n columns of cubes, the i-th of height a_i, find the smallest number of colors sufficient to color all the cubes so that every horizontal run of cubes in a row (without gaps) and every column has pairwise different colors.

Input

The first line contains one integer n $(1 \le n \le 1000)$, the number of columns. The second line contains n integers $a_1, a_2, \ldots, a_n$ $(1 \le a_i \le 1000)$, the heights of the columns.

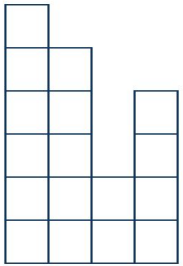


Figure 1.3 *Columns*, E-Olymp problem 10281, restated from the judge: the statement, the picture of four columns of cubes, and the input format.

Let the class read Figure 1.3 before anybody speaks. Paradigm: greedy / direct computation. Book: Section 1.6.1 and Figure 1.3.

Columns — Understand

n columns of cubes, the i-th a_i tall, $1 \le n, a_i \le 1000$. Paint every cube with as few colors as possible: **every column** and **every substring** must have all colors different.

THE ONE WORD THE PROBLEM INVENTS. A **substring** is a horizontal run of adjacent cubes at one height **with no gap**. A column too short to reach that height splits the row into two substrings that may repeat colors.

HAND-WORKED — heights $[6,5,2,4]$, top row first. Row 6: `[X]`. Row 5: `[X][X]`. Rows 4 and 3: `[X][X]` gap `[X]`. Rows 2 and 1: `[X][X][X][X]`. The tallest column stacks **6** cubes; the bottom row is an unbroken run of **4**. Six is the larger demand.

Half a minute with a pencil, and the picture contains the whole solution. Book: Section 1.6.1.

Columns — The Judge's Sample

Columns (E-Olymp 10281)

Time limit 1 s Memory limit 128 MB

Output

Print one number: the minimum number of colors required.

Sample

Input #1	Answer #1
4 6 5 2 4	6

Note

One possible painting of the sample. The third row from the bottom holds color 3 twice; this is allowed because the empty space in column 3 separates the two cubes.

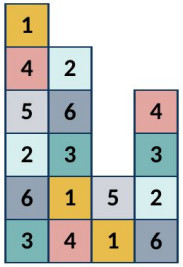


Figure 1.4 The sample test of *Columns* and one valid painting with six colors. Note the third row from the bottom: it contains the color 3 twice, which is legal because an empty space separates the two cubes.

The third row from the bottom of Figure 1.4 reads 2, 3, *gap,* 3*: color* 3 *twice in one row, and legal, because the gap splits the row. Book: Section 1.6.1 and Figure 1.4.*

Columns — Lower Bound and Construction

Vertical: the tallest column has $H = \max_i a_i$ cubes that must differ, so $C \geq H$. **Horizontal:** every $a_i \geq 1$, so the bottom row is one substring of length n, and $C \geq n$. Together, Equation (1.2):

$$C \geq \max\left(n, \max_i a_i\right). \tag{1.2}$$

Construction. With $C = \max(n, \max_i a_i)$, columns $j = 0, \ldots, n-1$ and rows $r = 0, 1, \ldots$ from the bottom, paint by Equation (1.3):

$$\text{color}(j, r) = ((j + r) \bmod C) + 1. \tag{1.3}$$

A column gives $a_j \leq C$ **consecutive** values of $j + r$; a gap-free run gives at most $n \leq C$ consecutive values. Consecutive integers, at most C of them, have distinct remainders mod C. Both rules hold.

KEY INSIGHT. answer $= \max(n, \max(a_1, a_2, \ldots, a_n))$ — a lower bound that is achieved is exact.

The chapter's brute-force verifier (Listing 1.7) checks every column and every run for the sample and prints `VERIFIED: no clash anywhere`. *Book: Section 1.6.2 and Equations (1.2)–(1.3).*

Columns — Design, Analyze, Implement

Design. $ans \leftarrow n$; for each height, $ans \leftarrow \max(ans, a_i)$; print. No array. **Analyze.** Invariant: after i heights, $ans = \max(n, a_1, \ldots, a_i)$. Time $O(n)$, space $O(1)$.

Listing 1.7 The complete solution to *Columns*, E-Olymp 10281.

```cpp
#include <cstdio>
#include <algorithm>
using namespace std;

int n, height, ans;

int main() {
    scanf("%d", &n);
    ans = n;                        // the bottom row alone already forces n colors
    for (int i = 0; i < n; i++) {
        scanf("%d", &height);
        ans = max(height, ans);     // a column of this height forces `height` colors
    }
    printf("%d\n", ans);
    return 0;
}
```

```
6
```

On the sample `4` / `6 5 2 4`, Listing 1.7 printed `6`; on `5` / `1 1 1 1 1` it printed `5`, because the bottom row of five demands five colors. Book: Sections 1.6.3–1.6.5 and Listing 1.8.

Problem 2 — *Journey* (E-Olymp 10280)

n cities on a **straight line**, city i at x_i; distance $|x_i - x_j|$ (absolute value). Cossack Vus visits every city and returns to his start; he chooses the start and the route. Report the minimum distance. $1 \le n \le 100$, $1 \le x_i \le 1000$.

> **HAND-WORKED — the two samples.** Cities at 1 and 4: walk 3 right, 3 back, total **6**. Cities at $1, 3, 1$: two share coordinate 1; walk 2 right, 2 back, total **4**.

It *sounds* like the traveling salesman problem, one of the hardest problems in computer science. On a line it is not: the only moves are left and right.

Paradigm: greedy / mathematical observation. Book: Section 1.7.1 and Figures 1.5–1.6.

Journey — The Bound and Its Proof

Let $R = x_{\max} - x_{\min}$, the **range**. A closed route is a loop, so read it from a moment when it stands on $x_{\min}$: it goes **out** to $x_{\max}$ (at least R) and **back** to $x_{\min}$ (at least R). So every route costs at least $2R$.

By contradiction: a route shorter than $2R$ has an out-piece or a back-piece shorter than R, yet each joins two points exactly R apart. Impossible.

Achieved: from $x_{\min}$ sweep right to $x_{\max}$ and back — $2R$, passing every city. From any start s: $(s - x_{\min}) + (x_{\max} - x_{\min}) + (x_{\max} - s)$; the s terms cancel, giving Equation (1.4):

$$-x_{\min} + x_{\max} - x_{\min} + x_{\max} = 2(x_{\max} - x_{\min}) = 2R. \tag{1.4}$$

KEY INSIGHT. answer $= 2 \times (\max_i x_i - \min_i x_i)$. Samples: $2(4-1) = 6$ and $2(3-1) = 4$.

The number of cities is irrelevant; only the two extremes matter. Book: Section 1.7.2 and Equation (1.4).

Journey — The Program

Design. Start $d_{\min}=2000$ (above every coordinate) and $d_{\max}=0$ (below every one); update both per coordinate; print $2(d_{\max}-d_{\min})$. Invariant: they are the extremes of the coordinates read so far. $O(n)$ time, $O(1)$ space.

Listing 1.8 The complete solution to *Journey*, E-Olymp 10280.

```cpp
#include <cstdio>
using namespace std;

int main() {
    int n, dist;
    scanf("%d", &n);
    int dmin = 2000, dmax = 0;     // 2000 is above every legal coordinate, 0 below every one
    for (int i = 0; i < n; i++) {
        scanf("%d", &dist);
        dmin = dmin < dist ? dmin : dist;
        dmax = dmax < dist ? dist : dmax;
    }
    printf("%d\n", 2 * (dmax - dmin));
    return 0;
}
```

`dmin < dist ? dmin : dist` is the **conditional operator**: the value before `:` if the test holds, the value after it otherwise — here, the smaller of the two.

Listing 1.8 printed 6 on sample 1 and 4 on sample 2. Book: Sections 1.7.3–1.7.5 and Listing 1.9.

Problem 3 — *Unique Numbers* (SPOJ DCEPC901)

DCEPC901 — Unique Numbers (SPOJ)

Time limit 0.204 s Source limit 50 000 B Memory limit 1536 MB

Statement

A teacher writes all the numbers from 0 to $n-1$ and then performs k operations on them, of two types: `0 x` replaces each number i by $(i+x)$ mod n; `1 x` replaces each number i by $(i \times x)$ mod n. After each operation, report how many distinct numbers remain.

Note: each operation is applied to the numbers produced by the previous operation, not to the original ones.

Input

The first line contains n, the second k, the number of operations. Each of the next k lines contains two integers p and x.

Output

Output k lines, each containing how many distinct numbers are left.

Figure 1.5 *Unique Numbers*, SPOJ problem DCEPC901, with its two operation types and the crucial note that operations compose.

Note the sentence in bold on the judge page of Figure 1.5: the operations compose. Paradigm: number theory, the greatest common divisor. Book: Section 1.8.1 and Figure 1.7.

Unique Numbers — Understand

Start with $0, 1, \ldots, n-1$. Apply k operations (p, x), each to the result of the previous one: $p=0$ replaces each i by $(i+x) \bmod n$; $p=1$ replaces each i by $(i \times x) \bmod n$.

After each operation, print how many **distinct** values remain. Constraints: $1 \le n \le 10^8$, $1 \le k \le 20$, $0 \le x \le 10^8$.

Sample: $n=30$, operations `1 8`, `0 3`, `1 12`; expected output 15, 15, 5.

> **READ $n \le 10^8$ AGAINST** Table 1.2**.** The list needs 400 MB and 20 passes cost 2×10^9 steps. The intended solution never builds the list.

Book: Section 1.8.1 and Figure 1.8.

Unique Numbers — Abstract

Addition never changes the count. If $(a+x)\equiv(b+x)\pmod n$ then $a\equiv b$, and with $a,b\in\{0,\dots,n-1\}$ that forces $a=b$. The map is injective, hence a bijection on a finite set: a shuffle.

Multiplication shrinks the universe. With $g=\gcd(n,x)$, every output $i\,x \bmod n$ is a multiple of g; by Bézout ($u\,x+v\,n=g$) every multiple of g is reached. So, by Equation (1.5):

$$\left|\{(i\times x)\bmod n : 0\le i\le n-1\}\right|=\frac{n}{\gcd(n,x)}, \tag{1.5}$$

Chain. The survivors $0,g,2g,\dots$ behave like a fresh $\{0,\dots,n'-1\}$ with $n'=n/g$, so replace n by n' and carry on.

KEY INSIGHT. Keep n as a running variable: on a multiplication set $n\leftarrow n/\gcd(n,x)$; on an addition do nothing; print n.

We never track the numbers, only how many there are. Book: Section 1.8.2 and Equation (1.5).

Unique Numbers — Multiplication Modulo 15

op	x	$(i \times x) \bmod 15$														
i		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	0	3	6	9	12	0	3	6	9	12	0	3	6	9	12
i		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	5	0	5	10	0	5	10	0	5	10	0	5	10	0	5	10
i		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	0	2	4	6	8	10	12	14	1	3	5	7	9	11	13
i		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	4	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11

Figure 1.6 Multiplication modulo 15. Each pair of rows shows the inputs $0..14$ above and the outputs $(i \times x) \bmod 15$ below, for $x = 3, 5, 2, 4$ in turn.

Check Figure 1.6 against Equation (1.5): $x=3$ gives five values ($15/3$); $x=5$ three ($15/5$); $x=2$ and $x=4$ all fifteen ($\gcd=1$). Four confirmations by hand before any code. Book: Section 1.8.2 and Figure 1.9.

Unique Numbers — Worked, Then Coded

HAND-WORKED — the sample, $n=30$. `1 8`: $\gcd(30,8)$: $30 \bmod 8=6$, $8 \bmod 6=2$, $6 \bmod 2=0$, so 2; $n=15$, **print 15**. `0 3`: a bijection, **print 15**. `1 12`: $\gcd(15,12)=3$; $n=5$, **print 5**.

Listing 1.9 The complete solution to *Unique Numbers*, SPOJ DCEPC901.

```cpp
#include <cstdio>
#include <algorithm>
using namespace std;

int main() {
    int n, k, p, x;
    scanf("%d %d", &n, &k);
    while (k--) {
        scanf("%d %d", &p, &x);
        if (p & 1) n /= __gcd(n, x);   // only a multiplication shrinks the universe
        printf("%d\n", n);
    }
    return 0;
}
```

A TRAP WORTH KNOWING. $x=0$ is allowed. By convention $\gcd(n,0)=n$, so $n \leftarrow 1$ — exactly one value survives, and the count can never rise again.

Listing 1.9 printed `15`, `15`, `5` on the sample, and `1`, `1` on `30` / `2` / `1 0` / `0 5`. `__gcd` is GNU's Euclid (`std::gcd` in C++17); `p & 1` tests oddness. Time $O(k \log n)$, space $O(1)$. Book: Sections 1.8.3–1.8.5 and Listing 1.10.

Problem 4 — *Minimum and Maximum Numbers* (SPOJ NEWLANG)

NEWLANG — Minimum and maximum numbers (SPOJ)

Time limit 0.1 s Source limit 50 000 B Memory limit 1536 MB

Statement

Given N and K, find the smallest and the largest positive integers that each have exactly N digits and exactly K distinct digits (0–9). Leading zeros are not allowed.

Input

The first line contains T, the number of test cases (about 200). Each of the next T lines contains `N K`, where $1 \le N \le 18$ and $1 \le K \le 10$. The input guarantees that an answer exists.

Output

For each test case, output the two numbers, separated by a space, on a new line.

Sample

Input	Answer
2 1 1 3 3	1 9 102 987

Figure 1.7 *Minimum and maximum numbers*, SPOJ problem NEWLANG, with its constraints and sample test.

Figure 1.7. Paradigm: greedy construction, with a one-sentence proof that the greedy choice is safe. Book: Section 1.9.1 and Figure 1.10.

Min and Max — Understand, and Why Greedy Is Safe

Given N and K $(1 \le N \le 18,\ 1 \le K \le 10)$, print the smallest and largest N-digit numbers using **exactly K distinct digits**, no leading zero. Sample: `1 1` gives `1 9`; `3 3` gives `102 987`.

Distinctness is about the *set* of digits: 121 uses $\{1,2\}$ — two. Check: 100 and 101 use only $\{0,1\}$, so 102 is the first; 988–999 have at most two distinct digits, so 987 is the last.

> **WHY GREEDY IS SAFE HERE.** The leading digit weighs 10^{N-1}, more than everything to its right ($10^{N-1}-1$ at most). So the best legal digit for position 1 is never a mistake, and the same holds for position 2 among $2..N$, and so on.

$K=1$: one non-zero digit repeated — $\underbrace{11\cdots1}_{N}$ and $\underbrace{99\cdots9}_{N}$.

We print digits as characters, so the size of the numbers never matters. Book: Sections 1.9.1–1.9.2.

Min and Max — The Two Constructions

Minimum, $K \geq 2$**:** lead with 1, then as many 0s as possible — $(N-1)-(K-2) = N-K+1$ — then $2, 3, \ldots, K-1$ in increasing order, Equation (1.6):

$$\text{minimum} \;=\; 1 \; \underbrace{00\cdots0}_{N-K+1} \; 2\,3\,\cdots\,(K-1). \tag{1.6}$$

Maximum, $K \geq 2$**:** $N-K+1$ nines, then $8, 7, \ldots, 10-K$ in decreasing order, Equation (1.7):

$$\text{maximum} \;=\; \underbrace{99\cdots9}_{N-K+1} \; 8\,7\,\cdots\,(10-K). \tag{1.7}$$

HAND-WORKED — $N=5$**,** $K=3$**.** Minimum: 1, three 0s, then 2: **10002**. Maximum: three 9s, then 8, 7: **99987**. Both have five digits and exactly three distinct ones.

Digit counts: $1+(N-K+1)+(K-2) = N$ *and* $(N-K+1)+(K-1) = N$*; distinct digits* $[0, \ldots, K-1]$ *and* $[10-K, \ldots, 9]$*. Book: Section 1.9.2 and Equations (1.6)–(1.7).*

Min and Max — The Table

Minimum

$K \backslash N$	1	2	3	4	5	6	7	8	9	10
1	1	11	111	1111	11111	111111	1111111	11111111	111111111	1111111111
2	–	10	100	1000	10000	100000	1000000	10000000	100000000	1000000000
3	–	–	102	1002	10002	100002	1000002	10000002	100000002	1000000002
4	–	–	–	1023	10023	100023	1000023	10000023	100000023	1000000023
5	–	–	–	–	10234	100234	1000234	10000234	100000234	1000000234

Maximum

$K \backslash N$	1	2	3	4	5	6	7	8	9	10
1	9	99	999	9999	99999	999999	9999999	99999999	999999999	9999999999
2	–	98	998	9998	99998	999998	9999998	99999998	999999998	9999999998
3	–	–	987	9987	99987	999987	9999987	99999987	999999987	9999999987
4	–	–	–	9876	99876	999876	9999876	99999876	999999876	9999999876
5	–	–	–	–	98765	998765	9998765	99998765	999998765	9999998765

Figure 1.8 The minimum (top, shaded amber) and maximum (bottom, shaded gray) for $N = 1..10$ and $K = 1..5$. Each row is one value of K; read across for increasing N.

Row $K = 5$ of Figure 1.8 at $N = 10$: $1\,000000\,234$ and $999999\,8765$ — six zeros or six nines, exactly as Equation (1.6) and Equation (1.7) predict. Book: Section 1.9.2 and Figure 1.11.

Min and Max — The Program

Listing 1.10 The complete solution to *Minimum and maximum numbers*, SPOJ NEWLANG.

```cpp
#include <cstdio>

int main() {
    int t, n, k;
    scanf("%d", &t);
    while (t--) {
        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 9
102 987
```

`i + 1` *for* $i = 1..K-2$ *prints* $2..K-1$ *ascending;* `9 - i` *for* $i = 1..K-1$ *prints* $8..10-K$ *descending; at* $K = 2$ *the first loop runs zero times. Listing 1.10 also printed* `10002 99987` *for* `5 3`*. Time* $O(T\,N)$*. Book: Sections 1.9.3–1.9.4 and Listing 1.11.*

Problem 5 — *Interesting Sequence* (E-Olymp 4894)

Interesting sequence (E-Olymp 4894)

Time limit 1 s Memory limit 128 MB

Statement

The sequence 011212201220200112… is built as follows: first comes 0; then the following action is repeated: append the already existing sequence to the right, replacing 0 with 1, 1 with 2 and 2 with 0, that is

$0 \to 01 \to 0112 \to 01121220 \to \ldots$

Find the m-th digit of the sequence.

Input

The first line contains the number of tests n ($1 \le n \le 1000$). Each of the next n lines contains m ($1 \le m \le 2^{63} - 1$), the position of the required digit.

Output

For each test, print on a separate line the digit at the m-th place of the sequence.

Figure 1.9 *Interesting sequence*, E-Olymp problem 4894. The sequence is built by repeatedly appending a shifted copy of itself.

The phrase to notice in Figure 1.9 is "append the sequence to itself": doubling, which means binary. Book: Section 1.10.1 and Figure 1.12.

Interesting Sequence — Understand

Start with `0`. Repeat forever: append a copy of the sequence with $0 \to 1$, $1 \to 2$, $2 \to 0$.

> **BY HAND.** `0` → `01` → `0112` → `01121220` → `0112122012202001`. With two characters of the next stage, `0112122012202001`**`12`** — exactly the prefix in the statement.

Task: given m, print the m-th digit (from 1). Up to 1000 queries, $1 \le m \le 2^{63} - 1 \approx 9.22 \times 10^{18}$.

Sample: $m = 17,239,385$ gives $1, 0, 2$. By Table 1.2, nothing that counts up to m survives: there is a formula.

Book: Section 1.10.1 and Figure 1.13.

Interesting Sequence — Abstract

Name the transformation: $0 \to 1 \to 2 \to 0$ is simply $v \mapsto (v+1) \bmod 3$.

Recursion on the index p (from 0): after k stages there are 2^k digits. If $p < 2^k$, the digit is in the first half and unchanged; if $p = 2^k + q$, it is the digit at q plus one, mod 3.

In **binary**, "$p \geq 2^k$?" asks whether the leading bit is 1, and $q = p - 2^k$ deletes it. Each 1-bit adds one; each 0-bit adds nothing. Hence Equation (1.8):

$$\text{value}(p) = (\text{number of } 1\text{-bits in the binary form of } p) \bmod 3. \tag{1.8}$$

KEY INSIGHT. The digits are numbered from 1, so answer $=$ popcount$(m-1) \bmod 3$.

The number of 1*-bits is the* ***population count****, or popcount. Book: Section 1.10.2 and Equation (1.8).*

Interesting Sequence — Proof by Induction

Let S_k be the sequence after k stages and $P(p)$ the popcount of p. Claim: $S_k[p] = P(p) \bmod 3$ for $0 \le p < 2^k$.

Base. $S_0 = [0]$ and $P(0) = 0$. True.

Step, first half. For $p < 2^k$, $S_{k+1}[p] = S_k[p] = P(p) \bmod 3$.

Step, second half. For $p = 2^k + q$ with $q < 2^k$, the bit of weight 2^k in q is 0, so $P(p) = P(q) + 1$, and $S_{k+1}[p] = (P(q) \bmod 3 + 1) \bmod 3 = P(p) \bmod 3$. ∎

The chapter also builds 128 real terms by brute force (Listing 1.12) and finds 0 mismatches with Equation (1.8). Book: Section 1.10.2.

Interesting Sequence — The Table Is the Sequence

N	Binary				#1-bits	(#1-bits) mod 3
0	0	0	0	0	0	**0**
1	0	0	0	1	1	**1**
2	0	0	1	0	1	**1**
3	0	0	1	1	2	**2**
4	0	1	0	0	1	**1**
5	0	1	0	1	2	**2**
6	0	1	1	0	2	**2**
7	0	1	1	1	3	**0**
8	1	0	0	0	1	**1**
9	1	0	0	1	2	**2**
10	1	0	1	0	2	**2**
11	1	0	1	1	3	**0**
	MSB			LSB		

Figure 1.10 Indices 0 to 11 in binary, with the number of 1-bits and that count modulo 3. The last column is the sequence itself.

The last column of Figure 1.10 reads 0,1,1,2,1,2,2,0,1,2,2,0 *— the first twelve digits we built by hand. Book: Section 1.10.2 and Figure 1.14.*

Interesting Sequence — Worked, Then Coded

HAND-WORKED — the samples. $m=17$: $p=16=10000_2$, one bit, **1**. $m=239$: $p=238=128+64+32+8+4+2=11101110_2$, six bits, **0**. $m=385$: $p=384=256+128=110000000_2$, two bits, **2**.

Listing 1.11 The complete solution to *Interesting sequence*, E-Olymp 4894.

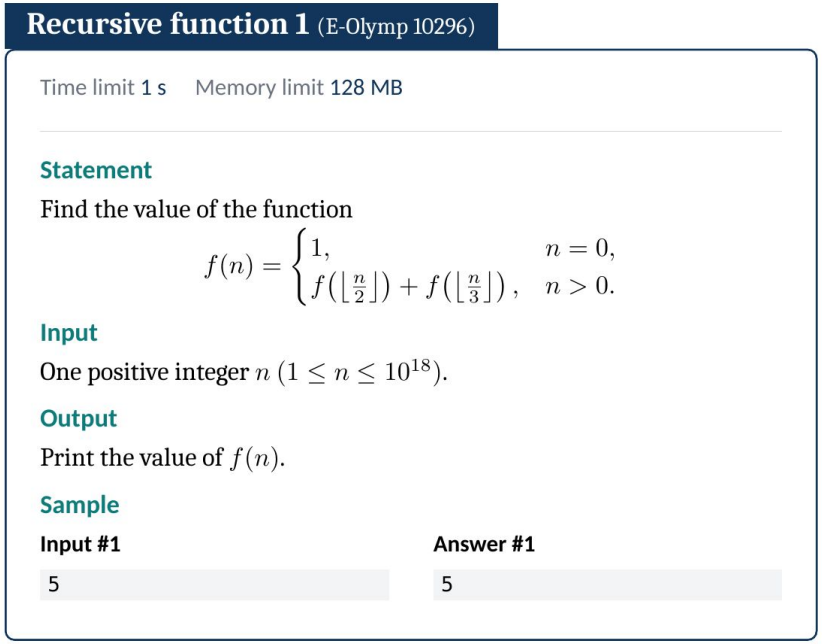
```cpp
#include <iostream>
using namespace std;
typedef unsigned long long LL;

int main() {
    LL n, x;
    cin >> n;
    while (n--) {
        cin >> x;
        LL tmp = __builtin_popcountll(x - 1);   // 1-indexed m becomes 0-indexed m-1
        cout << tmp % 3 << endl;
    }
    return 0;
}
```

```
1
0
2
```

`unsigned long long` reaches $2^{64}-1$; an `int` would overflow. `__builtin_popcountll` is one machine instruction (`std::popcount` in C++20). Listing 1.11 prints `0 1 1 2 1 2 2 0` for $m=1..8$ — the stage-3 sequence. Book: Sections 1.10.3–1.10.4 and Listing 1.13.

Problem 6 — *Recursive Function 1* (E-Olymp 10296)



Recursive function 1 (E-Olymp 10296)

Time limit 1 s Memory limit 128 MB

Statement

Find the value of the function

$$f(n) = \begin{cases} 1, & n = 0, \\ f\left(\lfloor \frac{n}{2} \rfloor\right) + f\left(\lfloor \frac{n}{3} \rfloor\right), & n > 0. \end{cases}$$

Input

One positive integer n ($1 \le n \le 10^{18}$).

Output

Print the value of $f(n)$.

Sample

Input #1	Answer #1
5	5

Figure 1.11 *Recursive function 1*, E-Olymp problem 10296. The definition, the single-integer input, and the sample $f(5)=5$.

Figure 1.11. Paradigm: dynamic programming by memoization — and a lesson in choosing a container. Book: Section 1.11.1 and Figure 1.15.

Recursive Function 1 — Understand

The function, Equation (1.9), for one input n with $1 \le n \le 10^{18}$ and a one-second limit:

$$f(n) = \begin{cases} 1, & n = 0, \\ f(\lfloor \frac{n}{2} \rfloor) + f(\lfloor \frac{n}{3} \rfloor), & n > 0. \end{cases} \tag{1.9}$$

$\lfloor x \rfloor$, the **floor**, is the largest whole number not greater than x: $\lfloor 7/2 \rfloor = 3$, $\lfloor 1/2 \rfloor = 0$. In C++, `n / 2` on non-negative values already is $\lfloor n/2 \rfloor$.

HAND-WORKED — $n=5$. $f(5)=f(2)+f(1)$; $f(2)=f(1)+f(0)$; $f(1)=f(0)+f(0)=2$; so $f(2)=3$ and $f(5)=3+2=5$. Notice $f(1)$ is needed **twice**.

Book: Section 1.11.1 and Equation (1.9).

Recursive Function 1 — Count the Work

Naive calls. In the call tree every inner node has two children and every leaf $f(0)$ returns 1, so $f(n)$ is the number of leaves and the calls number Equation (1.10):

$$\text{calls}(n) = f(n) + (f(n) - 1) = 2f(n) - 1. \tag{1.10}$$

For $n = 5$: $2 \cdot 5 - 1 = 9$ calls, matching the hand trace. For $n = 10^{18}$, $f \approx 2.24 \times 10^{14}$, so about 4.5×10^{14} calls — some **fifty-two days**.

Distinct sub-problems have the form Equation (1.11), with $a \le 60$ and $b \le 38$: at most $61 \times 39 = 2,379$.

$$\left\lfloor \frac{n}{2^a 3^b} \right\rfloor \text{ for whole numbers } a, b \ge 0. \tag{1.11}$$

A `set`-based walk from 10^{18} (Listing 1.14) finds exactly **1,012**. And an array indexed by n would need eight million terabytes; a `std::map` stores only the keys met.

The algorithm does not change; the data structure does. Book: Section 1.11.2 and Equations (1.10)–(1.11).

Recursive Function 1 — The Cache, Filled

HAND-WORKED — $f(12)$. $f(12)=f(6)+f(4)$; $f(6)=f(3)+f(2)$; $f(3)=f(1)+f(1)$; $f(1)=2$ (**cache**); $f(3)=4$ (**cache**); $f(2)=2+1=3$ (**cache**); $f(6)=7$ (**cache**); $f(4)=f(2)+f(1)=5$, both from the cache; $f(12)=7+5=12$.

Table 1.5 The memoization table built while computing $f(12)$, in the order the entries were filled.

u	$\lfloor u/2 \rfloor$	$\lfloor u/3 \rfloor$	$f(u)$	When it was cached
0	—	—	1	seeded before the run
1	0	0	1+1=2	first
2	1	0	2+1=3	third
3	1	1	2+2=4	second
4	2	1	3+2=5	fifth
6	3	2	4+3=7	fourth
12	6	4	7+5=12	last

Table 1.5 is the walkthrough as the table the program holds: seven distinct values, deepest first. $f(12)=12$ is a coincidence: $f(11)=9$. Book: Section 1.11.2 and Table 1.6.

Recursive Function 1 — The Program

Listing 1.12 The complete solution to *Recursive function 1*, E-Olymp 10296.

```cpp
#include <map>
#include <cstdio>
using namespace std;
typedef long long ll;

map<ll, ll> mp;

ll rec(ll u) {
    if (mp.count(u)) return mp[u];      // already solved: hand back the stored answer
    ll nval = rec(u / 2) + rec(u / 3);
    mp[u] = nval;
    return nval;
}

int main() {
    mp[0] = 1;                          // the base case, seeded into the cache
    ll u;
    scanf("%lld", &u);
    printf("%lld\n", rec(u));
    return 0;
}
```

No `if (u == 0) return 1;`: seeding `mp[0] = 1` lets the cache test catch the base case too.

Correct by strong induction, since $\lfloor u/2 \rfloor, \lfloor u/3 \rfloor < u$. **Time** $O(D \log D)$ with $D = 1{,}012$, i.e. $O(\log^2 n \cdot \log \log n)$. **Fits:** $f(10^{18}) = 224{,}412{,}958{,}156{,}019$, well inside `long long`.

Listing 1.12 printed `5` on the sample, `12` on `12`, and `224412958156019` instantly on 10^{18}. Book: Sections 1.11.3–1.11.4 and Listing 1.15.

All Six Problems at a Glance

Table 1.6 The six problems of this chapter, their paradigms, and the single sentence that solves each.

Problem	Paradigm	Core technique	Time	The answer in one sentence
Columns	Greedy	Running maximum	$O(n)$	$\max(\text{column count}, \text{tallest height})$
Journey	Greedy	Range of the array	$O(n)$	$2\times(\max-\min)$ on a line
Unique Numbers	Number theory	GCD reduction	$O(k\log n)$	A multiplication shrinks the universe by its GCD
Min/Max Numbers	Constructive greedy	Greedy digit placement	$O(N)$	Extreme digits at the most significant end
Interesting Sequence	Bit tricks	Popcount modulo 3	$O(1)$	Doubling means binary; count the 1-bits
Recursive Function 1	Memoization	Map-based dynamic programming	$O(\log^2 n)$	About a thousand sub-problems despite $n=10^{18}$

In Table 1.6 nothing is worse than $O(n)$ and no program exceeds twenty lines — yet each needed a genuine insight in Phase 2. Phrases in a statement point to a technique: “minimum/maximum” → one pass; “remainders” → gcd; “doubles each step” → binary; “$f(n)$ from $f(n/a)$” → memoization in a map.

The chapter’s symptom-and-diagnosis table (Table 1.8) lists eight such phrases. Book: Sections 1.12–1.12.1 and Tables 1.7–1.8.

Judges, Research, and the Road Map

An **online judge** compiles your program, runs it on secret tests and returns a verdict: *Accepted*, *Wrong Answer*, *Time Limit Exceeded*, *Memory Limit Exceeded*, *Runtime Error*, *Compilation Error*. We use **E-Olymp** and **SPOJ**; UVa and beecrowd are worth knowing.

On *Wrong Answer*, return to Phase 2; on *Time Limit Exceeded*, return to Phase 4. Never shuffle code and resubmit.

Judge problems are research too: ITS papers on SPOJ PERIOD5 (periodic extrapolation, ranked 2nd) and HIR (digit recognition under 0.1 s and 150 kB, top two entries).

Books: **CLRS**, 4th ed., proves; **CP4** drills. Sixteen chapters; Chapters 4, 8, 12 and 16 are **Evaluation and Discussion I–IV**, which set problems on the three chapters before and work every one in full.

Book: Sections 1.13–1.15, Figures 1.16–1.24 and Tables 1.9–1.12.

Your Workbench — Three Systems

Two things: a **compiler** (GCC's `g++` on Windows and Linux, Clang's `clang++` on macOS, both C++17) and an **editor** (VS Code). Compiler first, then editor, then connect them — Table 1.7.

Table 1.7 The workbench on three operating systems, from an empty computer to a running program. Other Linux distributions use `sudo dnf install gcc-c++ gdb` (Fedora) or `sudo pacman -S gcc gdb` (Arch) in step 1.

Step	Windows 10/11	MacOS	Linux (Ubuntu/Debian)
1. Install the compiler	Install MSYS2 from msys2.org, open the **MSYS2 UCRT64** terminal, run `pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain`	Open **Terminal**, run `xcode-select --install`	Open a terminal, run `sudo apt update && sudo apt install build-essential gdb`
2. Make it findable	Add `C:\msys64\ucrt64\bin` to the user `Path` variable, then open a new terminal	Nothing to do	Nothing to do
3. Check it	`g++ --version`	`clang++ --version`	`g++ --version`
4. Install VS Code	Installer from code.visualstudio.com; tick “Add to PATH”	Download, drag *Visual Studio Code* into *Applications*	`sudo snap install code --classic`, or the `.deb` from the website
5. Add C++ support	Extensions view (`Ctrl+Shift+X`): install **C/C++ Extension Pack** (Microsoft)	Same, with `Cmd+Shift+X`	Same, with `Ctrl+Shift+X`
6. Compile a file	`g++ -std=c++17 -O2 -Wall -o hello hello.cpp`	`clang++ -std=c++17 -O2 -Wall -o hello hello.cpp`	`g++ -std=c++17 -O2 -Wall -o hello hello.cpp`
7. Run it	`.\hello.exe`	`./hello`	`./hello`

Keep Table 1.7 as the one-page reference. Book: Section 1.16 and Table 1.13.

Installing the Compiler, Step by Step

Windows. Install MSYS2 from msys2.org into `C:\msys64`. Open **MSYS2 UCRT64** and run `pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain` (Enter, then `Y`). Add `C:\msys64\ucrt64\bin` to your user **Path**. In a *new* terminal, `g++ --version` must print a version.

macOS. In Terminal run `xcode-select --install` and click **Install**; check with `clang++ --version`. (`g++` also works on a Mac; it quietly runs Clang.)

Linux. `sudo apt update && sudo apt install build-essential gdb`, then `g++ --version`. Fedora: `dnf install gcc-c++ gdb`; Arch: `pacman -S gcc gdb`.

VS Code, every system. Install from code.visualstudio.com (on Windows tick **Add to PATH**); in Extensions (`Ctrl+Shift+X`, `Cmd+Shift+X` on a Mac) install the **C/C++ Extension Pack** by Microsoft; open a `DAA` folder with one subfolder per chapter.

If `g++` is "not recognized" on Windows, the Path entry was not saved or the terminal is old: open a new one. Book: Sections 1.16.1–1.16.4.

Your First Program

Listing 1.13 The first program: it proves that the compiler, the editor and the terminal all work together.

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string name;
    cout << "What is your name? ";
    if (!(cin >> name)) name = "friend";      // no input? greet a friend
    cout << "Hello, " << name << "! Your C++17 compiler works." << endl;
    cout << "__cplusplus = " << __cplusplus << endl;
    return 0;
}
```

```
g++ -std=c++17 -O2 -Wall -o hello hello.cpp
```

```
What is your name? Irfan
Hello, Irfan! Your C++17 compiler works.
__cplusplus = 201703
```

`-std=c++17` selects the language version, `-O2` optimizes, `-Wall` turns on warnings, `-o hello` names the program; silence means success. `201703` proves Listing 1.13 was compiled as C++17. Book: Section 1.16.5 and Listing 1.16.

Feeding Input, and Watching the Program Think

Judges feed programs through *standard input*. Save the sample in `input.txt` and **redirect**: `./columns < input.txt` (macOS, Linux, Command Prompt) or `Get-Content input.txt | .\columns.exe` (PowerShell).

Debugging. Click left of a line number to set a **breakpoint**; press `F5` and choose **C++ (GDB/LLDB)**. The **Variables** panel shows every value; `F10` steps a line, `F11` steps into a function, `F5` continues.

> **WHEN SOMETHING GOES WRONG.** "`iostream`: No such file" — the compiler came from the wrong MSYS2 terminal; repeat in **UCRT64**. The window flashes and closes — run it from the terminal. Squiggles everywhere but it compiles — run **C/C++: Select IntelliSense Configuration**.

Watching a loop invariant hold in the Variables panel, pass after pass, is one of the best ways to understand it. Book: Sections 1.16.6–1.16.7.

Profitina in Practice

Profitina (`app.profitina.com`) is an AI-visibility (GEO) platform. Near-duplicate detection by all pairs is $\Theta(P^2)$ — 2.5×10^9 at $P = 50{,}000$ pages; fingerprints make it $O(P)$. The app advises; the user decides.

Profitina Laundry, the teaching case: a van visiting drop-off points on Jalan Kertajaya is *Journey*; labels on a drying rack are *Columns*. Listing 1.14 solves both.

Listing 1.14 The Profitina Laundry planner: the van's shortest round trip and the number of rack labels required.

```
#include <cstdio>
#include <algorithm>
using namespace std;

int main() {
    int n;
    scanf("%d", &n);                        // number of drop-off points on Jalan Kertajaya

    int x, lo = 1000000000, hi = -1000000000;
    for (int i = 0; i < n; i++) {           // meter mark of each drop-off point
        scanf("%d", &x);
        lo = min(lo, x);
        hi = max(hi, x);
    }

    int h, labels = n;                      // the bottom shelf alone needs n labels
    for (int i = 0; i < n; i++) {           // how many baskets each point sends back
        scanf("%d", &h);
        labels = max(labels, h);            // a tall stack needs that many labels
    }

    printf("van route            : %d meters\n", 2 * (hi - lo));
    printf("distinct rack labels : %d\n", labels);
    return 0;
}
```

```
van route            : 1810 meters
distinct rack labels : 9
```

Input `5` / `120 340 75 980 410` / `3 7 2 9 4`*: range* $980-75=905$*, round trip* 1810*; labels* $\max(5,9)=9$*. Book: Section 1.17 and Listing 1.17.*

Common Mistakes

Coding before understanding, and **testing instead of proving**: close the editor, do Phases 1 and 2, and prove every loop, greedy choice and recursion.

Ignoring the constraints and **optimizing the wrong thing**: put the largest input into Table 1.2 first; `scanf` for `cin` will not rescue $O(n^2)$ at $n = 10^6$.

Forgetting legal edge cases: $x = 0$ in *Unique Numbers*, $n = 1$ or shared coordinates in *Journey*, $K = 1$ in *Min and Max*.

Integer overflow (an `int` stops near 2.1×10^9) and **the wrong container** (an array where a thousand states arise).

"Sorted" is not "correct", a **misread defined term** such as *substring*, and **the judge used as a compiler**.

Book: Section 1.18.

Summary — Five Things to Remember

1. An **algorithm** is the idea, a **program** is the idea in a language, a **data structure** makes the steps cheap. The algorithm is the part that lasts.

2. **Correct** means right for *every* allowed input, which only a proof establishes: direct, contradiction, loop invariant or induction.

3. About 10^8 operations per second: **the constraints decide the design** before any code is written.

4. Five paradigms — incremental, greedy, divide and conquer, dynamic programming, mathematical — and five phases, with the code last.

5. The central move of Phase 2: prove a **lower bound**, then **construct** something that attains it.

Columns $\max(n, \max a_i)$*; Journey* $2(x_{\max} - x_{\min})$*; Unique Numbers* $n \leftarrow n/\gcd(n, x)$*; Min and Max greedy digits; Interesting Sequence* $\text{popcount}(m-1) \bmod 3$*; Recursive Function 1 a memo* `map`*. Book: Section 1.19.*

Next Time

Chapter 2 — Getting Started: the RAM model, insertion sort traced and counted line by line, and linear and binary search.

The loop invariant returns in full — initialization, maintenance, termination — and "how long does it take?" becomes a formula, not a guess.

Before we meet: install the workbench and run Listing 1.13; submit *Columns* and *Journey* on E-Olymp after working all five phases on paper.

Exercises 1.1–1.16 practice every idea of today, with full solutions in Section 1.21.

Book: Sections 1.20–1.21, then Chapter 2.

Design and Analysis of Algorithms (EF234405)

Lecture 2 — Getting Started: the RAM Model, Insertion Sort, and Searching

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*, Chapter 2

Lecturer: Irfan Subakti — `yifana@gmail.com`

Department of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya

Today "how long does it take?" stops being a guess. Book: Chapter 2, and Cormen et al., Introduction to Algorithms, 4th ed., Chapter 2.

Agenda

1. Five words used precisely: problem, instance, algorithm, program, data structure — and a short history of the idea.

2. The **RAM model**: what counts as one step, and what does not.

3. Insertion Sort done completely: traced, coded, proved correct with a loop invariant, and counted line by line.

4. Best, worst and average case, checked against a real machine; the growth of functions.

5. Linear search and **binary search**, with their invariants, their $\Theta(n)$ and $\Theta(\lg n)$, and the two off-by-one traps.

Book: Chapter 2, Sections 2.1–2.19.

Where We Left Off

Chapter 1 was a chapter of *doing*: six judge problems, the five-phase framework, accepted C++.

What we did not do was **justify** it. "This is $O(n)$, so it passes" used a notation we had not defined.

Today: agree on words, agree on a **machine**, then take one algorithm and do *everything* to it — describe, trace, prove, count, derive. Then the same, faster, for searching.

> **KEY INSIGHT.** Chapter 1 gave the *habit* of solving problems; this chapter gives the *vocabulary and the arithmetic* to say why a solution is good. The habit without the arithmetic is guesswork.

Chapter 3 completes the payment with formal O, Θ and Ω. Book: Section 2.1.

What This Course Is About

Shortest paths from Keputih to Rungkut, airline gates, web search, airflow over a turbine, video compression: real problems, modeled computationally.

Three activities: *designing* algorithms (the creative act), *analyzing* them (the mathematical act), *choosing data structures* (so the algorithm can work cheaply).

Not the subject: programming languages (C++ is our notation), computer architecture (abstracted by the RAM model), software engineering, complexity theory in depth.

Our focus is the **algorithm**: the abstract idea of how to solve a problem, independent of language and machine.

Book: Section 2.2.

The Word, and the First Algorithm

The word comes from **al-Khwarizmi** (c. 780–850), whose book on decimal calculation was Latinized as *Algorismus*. His gift was a *procedure* so clear that one could follow it without knowing why it works.

The oldest non-trivial algorithm still in use: **Euclid's**, c. 300 BCE — replace (a, b) by $(b, a \bmod b)$ until $b = 0$.

> **BY HAND —** $\gcd(48, 18)$. $48 \bmod 18 = 12 \Rightarrow (18, 12)$; $18 \bmod 12 = 6 \Rightarrow (12, 6)$; $12 \bmod 6 = 0 \Rightarrow (6, 0)$; report 6. No computer, no language — and it runs unchanged on every machine ever built.

Book: Sections 2.3.1–2.3.2.

Al-Khwarizmi



Figure 2.1 Muhammad ibn Musa al-Khwarizmi (c. 780–850 CE), whose Latinized name *Algorismus* became the word *algorithm*.

Figure 2.1: we are borrowing a name more than a thousand years old. Book: Section 2.3.1 and Figure 2.1.

Machines, and What Computation Is

Babbage designed the Analytical Engine (mill, store, punched cards). **Ada Lovelace** wrote the first algorithm meant for a machine — the Bernoulli numbers — and foresaw machines manipulating any symbols.

Turing (1936) defined computability with a tape, a head and a table of rules, and proved some well-posed problems have *no* algorithm.

Church's lambda calculus defines exactly the same computable functions — the Church–Turing thesis.

Von Neumann (1945): the program lives in the same memory as its data. The RAM model is an idealization of this design.

Algorithms outlive hardware: Euclid has outlived the Roman Empire and the floppy disk. Book: Sections 2.3.3–2.3.4 and Figures 2.2–2.6.

Problem and Instance

DEFINITION. An **algorithmic problem** specifies a desired input–output relationship: *what* must be computed, not *how*. An **instance** is one particular legal input.

THE SORTING PROBLEM. *Input:* a sequence $\langle a_1, a_2, \ldots, a_n \rangle$. *Output:* a permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ with $a'_1 \le a'_2 \le \cdots \le a'_n$.

n is the **size of the input**; $\langle \cdots \rangle$ is a sequence (order matters, repeats allowed); a'_1 is the first *output* element.

$\langle 5,2,4,6,1,3 \rangle$ is an instance. **Correctness is about the problem; running time is usually about the instance**: $\langle 1,\ldots,6 \rangle$ and $\langle 6,\ldots,1 \rangle$ are the same size and cost Insertion Sort wildly different amounts.

Book: Sections 2.4.1–2.4.2.

Algorithm: Five Properties

DEFINITION. An **algorithm** is a well-defined computational procedure that takes some value(s) as **input** and produces some value(s) as **output**.

Knuth's criteria: **input** (zero or more values), **output** (at least one), **definiteness** ("compare $A[i]$ with key", not "stir until it looks right").

Finiteness on *every* legal input — `while (true) i = i + 1;` is definite and useless. **Effectiveness** — each step doable with pencil and paper.

EXAMPLE. "Generate every ordering; stop at the first sorted one." All five properties hold, so it *is* an algorithm — and for $n=20$ it may try $20! \approx 2.4 \times 10^{18}$ orderings. Being an algorithm is a low bar.

Book: Section 2.4.3.

Program, Data Structure, Correct, Efficient

A **program** implements an algorithm in a language for a machine. We analyze algorithms, so the analysis covers every implementation at once.

A **data structure** organizes data so some operations are cheap: an array reads position 7 instantly but inserts there expensively; a linked list does the reverse.

DEFINITION. An algorithm is **correct** if on **every** legal instance it halts with an output meeting the specification. "Right on every instance but one" is incorrect.

DEFINITION. Efficiency is the resource used *as a function of* n: chiefly time (primitive steps), secondly space (memory cells).

The whole book in a sentence: among infinitely many correct algorithms, find the efficient ones. Book: Sections 2.4.4–2.4.6 and Figure 2.7.

The Five Design Goals

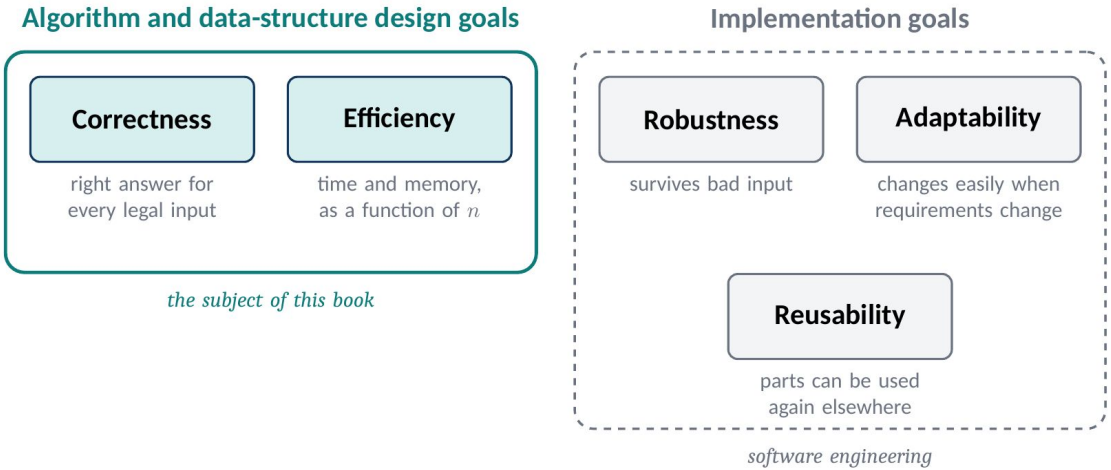


Figure 2.2 The five goals, in two groups. Correctness and efficiency are design goals and are the subject of this book; robustness, adaptability and reusability are implementation goals belonging to software engineering.

Figure 2.2: correctness and efficiency are design goals and this book's subject; robustness, adaptability and reusability are implementation goals. Correctness is proved (loop invariants, induction), not tested. Book: Section 2.5 and Figure 2.8.

The RAM Model

Rules. Numbered memory cells, **any** cell read or written in one step (*random access*); one instruction at a time; each **simple** instruction costs one step; values of reasonable size (a word holds at least $\lg n$ bits).

One step: arithmetic (`+ - * / %`, floor, comparison, bitwise ops); data movement, including **indexing** `A[i]`; control (a test, a jump, a call, a return).

```
x = y + z;        // one addition, plus the movement -- counted as constant
if (a > b) ...    // one comparison
t = A[i];         // one array access
i = i - 1;        // one subtraction and one store
```

$\lg n$ *means* $\log_2 n$*; Section 2.12 defines it with care. Book: Sections 2.6.1–2.6.2.*

What Is *Not* One Step — and Why This Model

A loop is not one step: its body runs n times. **A call is not one step:** `sort(v)` is one line and perhaps $n \lg n$ work. **Unbounded data is not one step:** copying a string of length m costs m; 2^k costs about $\lg k$ multiplications.

THINK. Does the work of this line depend on n? If not, it is one step. If so, open it and count.

The model is a deliberate lie (caches, pipelines, cores). We use it because it is **machine-independent**, **predictive** — it ranks algorithms correctly once n is large — and **simple enough to compute with**.

A model is a decision about what to ignore. Book: Sections 2.6.3–2.6.4.

How to Analyze Running Time

DEFINITION. The **running time** $T(n)$ is the number of primitive RAM steps executed on an input of size n — a count of operations, never seconds.

Method: (1) give each line a constant cost $c_1, c_2, \ldots$; (2) count how many times each line runs; (3) sum cost × times. All the skill is in step 2.

Define n first: elements for sorting and searching; the dimension for matrices (n^2 numbers); $|V|$ and $|E|$ for graphs; the number of **bits** for number theory.

0.8 seconds tells you about your laptop; "about $n^2/2$ operations" will still be true in fifty years.

Book: Sections 2.7–2.7.2.

A First Complete Example

Pseudocode 2.1 Finding the largest of n numbers with one pass.

```
FIND-MAX(A, n)
1  max = A[1]
2  for i = 2 to n
3      if A[i] > max
4          max = A[i]
5  return max
```

Counts for Pseudocode 2.1: line 1 once; line 2's test n times ($n-1$ successes and one final failure); line 3 $n-1$ times; line 4 some k times with $0 \le k \le n-1$; line 5 once. By Equation (2.1):

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4 k + c_5. \tag{2.1}$$

Both extremes of k give the form $an+b$: $T(n)$ is linear whatever the input. k changed the constants, not the shape.

That pattern — the data moves the constants, not the power of n — recurs all semester. Book: Section 2.7.3, Pseudocode 2.1 and Equation (2.1).

The Sorting Problem, Formally

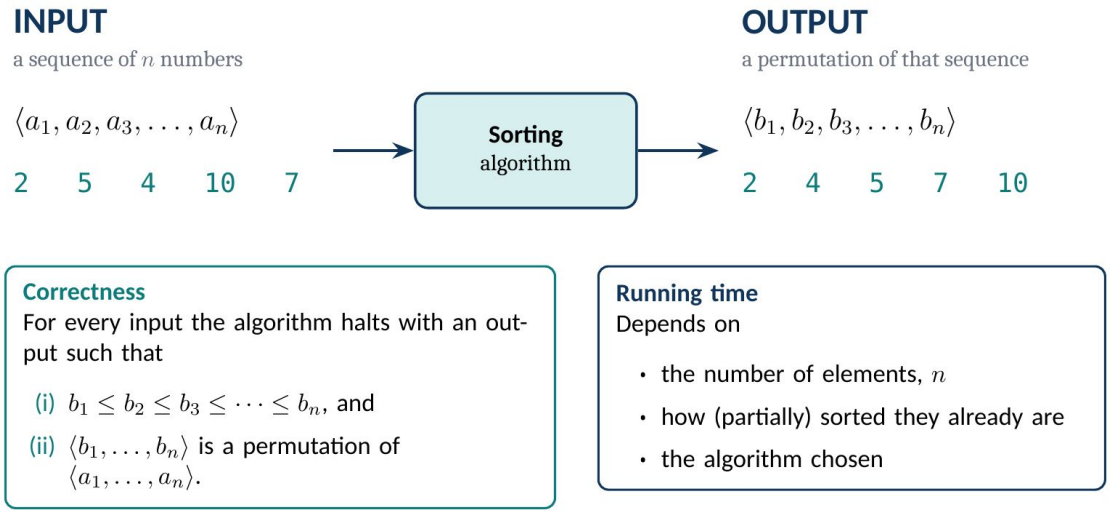


Figure 2.3 The sorting problem. The output $\langle b_1, \ldots, b_n \rangle$ must be both *ordered* ($b_1 \leq b_2 \leq \cdots \leq b_n$) and a *permutation* of the input — two conditions, and an algorithm that meets only one of them is wrong. The running time depends on how many elements there are, how nearly sorted they already are, and which algorithm we chose.

The bottom-left box of Figure 2.3 holds the trap: sortedness and permutation. $\langle 1,2,3,4,5 \rangle$ for every input is ordered and wrong. Book: Section 2.8 and Figure 2.9.

The Card Player's Method

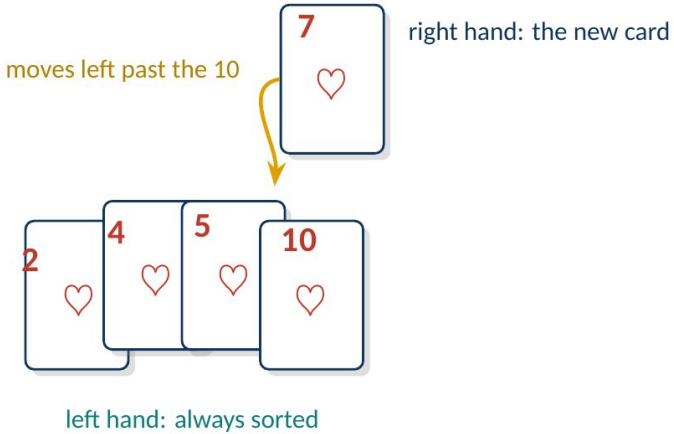


Figure 2.4 The card-player's method. The left hand holds a sorted run, 2–4–5–10; the right hand has just picked up the 7 and will move it leftwards past the 10 to reach its place.

In Figure 2.4 the sorted run grows by one card per step (the incremental method), and the place is found by scanning right to left, stopping at the first card not bigger — that early stop is why the best case is so good. Book: Section 2.8.1 and Figure 2.10.

Insertion Sort in Pseudocode

Pseudocode 2.2 Insertion Sort in the pseudocode used throughout this book, with 1-based indices.

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

Line 2 saves `key` before line 5 overwrites $A[j]$ — omit it and the array fills with duplicates.

Line 4: `i > 0` stops us at the left end; `A[i] > key` gives the early exit. The order matters: `&&` short-circuits, so $A[0]$ is never read.

Line 7 writes at $A[i+1]$, not $A[i]$: the loop exits one step *past* the last element moved, so $i+1$ is the hole.

Pseudocode 2.2 counts from 1, as CLRS does, because it keeps the mathematics clean. Book: Section 2.8.2 and Pseudocode 2.2.

Tracing by Hand

Table 2.1 Insertion Sort on $A = \langle 5, 2, 4, 6, 1, 3 \rangle$, one outer iteration per row. The bar between the sorted prefix and the untouched suffix moves one place right on every row.

Step	Key	Action	Result
$j = 2$	2	$2 < 5$, so shift 5 right. Insert 2 at position 1.	$[2, 5, 4, 6, 1, 3]$
$j = 3$	4	$4 < 5$, shift 5 right. $4 > 2$, stop. Insert 4 at position 2.	$[2, 4, 5, 6, 1, 3]$
$j = 4$	6	$6 > 5$, no shifts needed. Already in place.	$[2, 4, 5, 6, 1, 3]$
$j = 5$	1	$1 < 6$, shift 6. $1 < 5$, shift 5. $1 < 4$, shift 4. $1 < 2$, shift 2. Insert 1 at position 1.	$[1, 2, 4, 5, 6, 3]$
$j = 6$	3	$3 < 6$, shift 6. $3 < 5$, shift 5. $3 < 4$, shift 4. $3 > 2$, stop. Insert 3 at position 3.	$[1, 2, 3, 4, 5, 6]$

Check every row of Table 2.1 with the array in front of you: the sorted prefix grows by one each row.

Book: Section 2.8.3 and Table 2.1.

The Row j = 5, Step by Step

Start $[2,4,5,6,1,3]$; `key` $=1$, $i=4$.

Test: $4>0$ and $6>1$ — shift: $[2,4,5,6,6,3]$, $i=3$. Test: $5>1$ — shift: $[2,4,5,5,6,3]$, $i=2$.

Test: $4>1$ — shift: $[2,4,4,5,6,3]$, $i=1$. Test: $2>1$ — shift: $[2,2,4,5,6,3]$, $i=0$.

Test: $0>0$ is **false** — we stop on the first condition and never read $A[0]$. Line 7: $A[1] \leftarrow 1$, giving $[1,2,4,5,6,3]$.

The line-4 test ran **five** times: four successes, one failure. Remember "five" — it will be called t_5.

Sixteen lines to move one card. Book: Section 2.8.3.

Insertion Sort in C++

Listing 2.1 Insertion Sort in C++, with 1-based indexing so that the code matches the pseudocode exactly.

```cpp
// Insertion Sort, written to match the pseudocode line for line.
// The array is 1-based: A[0] is deliberately left unused so that the C++
// indices are exactly the indices used in the book's pseudocode.
#include <iostream>
#include <vector>
using namespace std;

void insertionSort(vector<int>& A, int n) {
    for (int j = 2; j <= n; ++j) {
        int key = A[j];          // the card we pick up
        int i = j - 1;           // start comparing with the card to its left
        while (i > 0 && A[i] > key) {
            A[i + 1] = A[i];     // slide the bigger card one place right
            i = i - 1;
        }
        A[i + 1] = key;          // drop the card into the hole we opened
    }
}

void print(const vector<int>& A, int n) {
    for (int i = 1; i <= n; ++i) cout << A[i] << (i < n ? ' ' : '\n');
}

int main() {
    vector<int> A = {0, 5, 2, 4, 6, 1, 3};   // A[0] unused
    int n = 6;
    cout << "before: "; print(A, n);
    insertionSort(A, n);
```

```
    cout << "after:  "; print(A, n);
    return 0;
}
```

```
before: 5 2 4 6 1 3
after:  1 2 3 4 5 6
```

Listing 2.1 leaves `A[0]` unused so the C++ sits line for line against the pseudocode; the answer matches Table 2.1. Book: Section 2.8.4 and Listing 2.1.

The Machine Traces Itself

Listing 2.2 Insertion Sort instrumented to print its own trace, one line per outer iteration.

```cpp
// Insertion Sort with a printed trace: one line per outer iteration,
// reproducing by machine exactly the hand trace done in the text.
#include <iostream>
#include <vector>
using namespace std;

void show(const vector<int>& A, int n) {
    cout << '[';
    for (int i = 1; i <= n; ++i) cout << A[i] << (i < n ? "," : "");
    cout << ']';
}

int main() {
    vector<int> A = {0, 5, 2, 4, 6, 1, 3};
    int n = 6;
    cout << "start        "; show(A, n); cout << '\n';
    for (int j = 2; j <= n; ++j) {
        int key = A[j];
        int i = j - 1;
        int shifts = 0;
        while (i > 0 && A[i] > key) { A[i + 1] = A[i]; --i; ++shifts; }
        A[i + 1] = key;
        cout << "j=" << j << " key=" << key
             << " shifts=" << shifts
             << " insert at " << (i + 1) << "  ";
        show(A, n);
        cout << '\n';
    }
```

```
    return 0;
}
```

```
start        [5,2,4,6,1,3]
j=2 key=2 shifts=1 insert at 1  [2,5,4,6,1,3]
j=3 key=4 shifts=1 insert at 2  [2,4,5,6,1,3]
j=4 key=6 shifts=0 insert at 4  [2,4,5,6,1,3]
j=5 key=1 shifts=4 insert at 1  [1,2,4,5,6,3]
j=6 key=3 shifts=3 insert at 3  [1,2,3,4,5,6]
```

Every row of Listing 2.2's output agrees with Table 2.1, including the shift counts $1,1,0,4,3$*. Trace by hand and instrument; where they disagree, one is a bug. Book: Section 2.8.4 and Listing 2.2.*

What a Loop Invariant Is

> **DEFINITION.** A **loop invariant** is a statement about the state of the computation that is true immediately before every iteration of a loop and again immediately after the loop finishes.

Initialization — true before the first iteration. **Maintenance** — if true before an iteration, still true before the next.

Termination — at exit, the invariant plus the reason the loop stopped gives the property we wanted.

The first two are exactly induction (base case, inductive step); termination is where we cash the invariant in.

> **WHY THE THIRD STEP IS NOT OPTIONAL.** "j is an integer" is invariant and useless. A good invariant is one whose truth *at exit* is the thing you set out to prove.

Book: Section 2.9.1.

The Invariant, and Initialization

> **LOOP INVARIANT (Insertion Sort).** At the start of each iteration of the `for` loop of line 1, the subarray $A[1\ldots j-1]$ consists of the elements originally in $A[1\ldots j-1]$, but in sorted order.

Two clauses, both needed: **sorted**, and **the same elements**. Without the second, an algorithm that writes zeros would satisfy it.

Check it on the trace: at $j=5$, $A[1..4]=[2,4,5,6]$ is sorted, and $\{2,4,5,6\}$ were the original occupants $5,2,4,6$.

Initialization. $j=2$: $A[1\ldots1]$ is one element — sorted vacuously (no adjacent pair can fail), and nothing has moved.

"Every unicorn in this room is purple" is true for the same reason: an empty collection has no counterexample. Book: Sections 2.9.2–2.9.3.

Maintenance: an Invariant Inside

> **INNER LOOP INVARIANT.** At each test of line 4: (a) $A[1\ldots i]$ is unchanged; (b) $A[i{+}2\ldots j]$ holds the old $A[i{+}1\ldots j-1]$, moved one place right; (c) $A[i{+}1]$ is a *hole*; (d) every value in $A[i{+}2\ldots j]$ exceeds `key`.

It holds on entry ($i=j-1$; the ranges in (b) and (d) are empty; $A[j]$ is a hole because `key` saved it) and **each shift preserves it** (line 5 fills the hole, line 6 moves it left).

Exit with $i=0$**:** everything in $A[2..j]$ exceeds `key`; `key` goes into $A[1]$. **Exit with** $A[i]\le key$**:** left block $\le key$, right block $> key$, `key` into the hole $A[i{+}1]$.

Either way $A[1\ldots j]$ is sorted and no value was created or destroyed: the outer invariant holds for $j{+}1$.

This settles the off-by-one: the key goes to $i{+}1$ because line 6 always moves i one place past the last element shifted. Book: Section 2.9.4.

Termination

The loop must stop: j starts at 2, grows by one, and is tested against the fixed bound n. It exits at $j = n+1$.

Substitute $j = n+1$: $A[1 \ldots n]$ holds the original elements of $A[1 \ldots n]$ in sorted order.

That is the entire array, and exactly both clauses of the sorting specification. Insertion Sort is correct. ▰

> **THINK.** The invariant alone proves nothing about the answer; the invariant *evaluated at the exit condition* does.

Write the invariant before the loop, and the loop usually comes out right the first time. Book: Section 2.9.5.

The Cost-and-Times Table

DEFINITION. t_j is the number of times the `while` **test** on line 4 runs for that j. The body (lines 5–6) runs $t_j - 1$ times.

Table 2.2 The cost of each line of Insertion Sort and the number of times it runs. The symbol t_j is the number of times the line-4 test is evaluated for that value of j.

Line	Cost	Times executed
1: `for j = 2 to A.length`	c_1	n
2: `key = A[j]`	c_2	$n-1$
3: `i = j - 1`	c_3	$n-1$
4: `while i > 0 and A[i] > key`	c_4	$\sum_{j=2}^{n} t_j$
5: `A[i+1] = A[i]`	c_5	$\sum_{j=2}^{n} (t_j - 1)$
6: `i = i - 1`	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7: `A[i+1] = key`	c_7	$n-1$

Line 1 runs n times: $n-1$ entries plus one failing test. In the trace, $t_5=5$ and lines 5–6 ran 4 times — `shifts=4`.

A count you cannot justify is a count you guessed; Table 2.2 justifies every row. $\sum_{j=2}^{n} t_j$ *means* $t_2+t_3+\cdots+t_n$. *Book: Section 2.10.1 and Table 2.2.*

The General Formula

Summing cost × times down Table 2.2 gives the running time for **any** input, Equation (2.2):

$$T(n)=c_1 n+c_2(n-1)+c_3(n-1)+c_4\sum_{j=2}^{n} t_j+c_5\sum_{j=2}^{n}(t_j-1)+c_6\sum_{j=2}^{n}(t_j-1)+c_7(n-1). \tag{2.2}$$

The c_k depend on compiler and processor — we never need their values. n is the size. The t_j depend on **which array we were given**.

> **KEY INSIGHT.** $T(n)$ is not determined by n alone: the formula has a hole shaped like $\sum t_j$ that only the input can fill. Hence three cases.

Book: Section 2.10.2 and Equation (2.2).

Best, Worst, Average

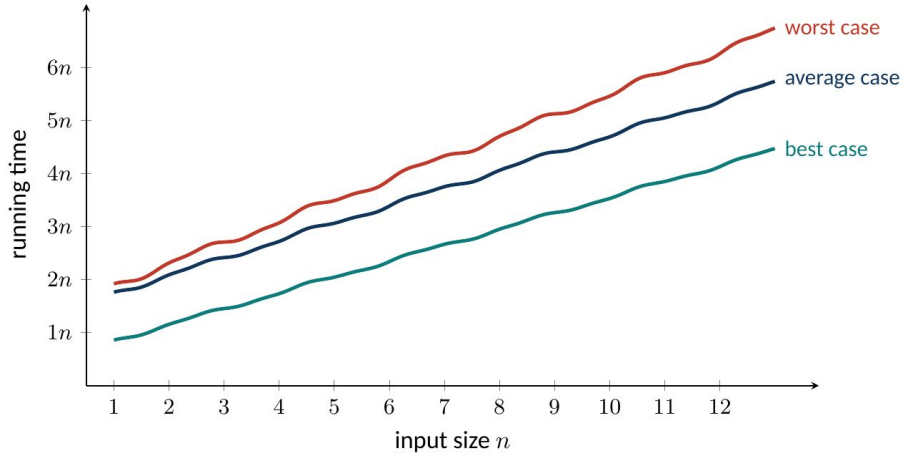


Figure 2.5 For a given input size there are many possible inputs and therefore many possible running times. The worst case is the upper envelope, the best case the lower, and the average case lies between them.

In Figure 2.5, for each size there is a band of running times: worst case on top, best case at the bottom, average in between. Book: Section 2.11 and Figure 2.11.

The Best Case: Already Sorted

Input $\langle 1,2,3,4,5,6\rangle$: $A[j-1] \le A[j]$, so the line-4 test fails at once. $t_j = 1$ for every j; lines 5–6 never run.

$\sum_{j=2}^{n} t_j = n-1$ and $\sum_{j=2}^{n} (t_j - 1) = 0$. Substituting and collecting:

$T(n) = (c_1+c_2+c_3+c_4+c_7)n - (c_2+c_3+c_4+c_7)$, which is Equation (2.3):

$$T(n) = an + b. \tag{2.3}$$

KEY INSIGHT. Linear — $\Theta(n)$ in Chapter 3's notation. It cannot be faster: it must look at every element to know the array is sorted.

Book: Section 2.11.1 and Equation (2.3).

The Worst Case: Two Sums

Input $\langle 6,5,4,3,2,1\rangle$: every element left of j is larger, so the test succeeds $j-1$ times and fails once: $t_j = j$ (check $t_2 = 2$).

Gauss's trick: write $S = 2+3+\cdots+n$ forward and backward; each of the $n-1$ columns adds to $n+2$, so $2S = (n-1)(n+2)$ and Equation (2.4):

$$\sum_{j=2}^{n} t_j = \frac{n(n+1)}{2} - 1. \tag{2.4}$$

With $t_j - 1 = j - 1$, the shifts are $1+2+\cdots+(n-1)$, Equation (2.5):

$$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j-1) = 1+2+3+\cdots+(n-1) = \frac{n(n-1)}{2}, \tag{2.5}$$

Check with $n = 6$: $2+3+4+5+6 = 20 = 21-1$ and $1+2+3+4+5 = 15 = 6\cdot 5/2$. Book: Section 2.11.2 and Equations (2.4)–(2.5).

The Worst Case: Quadratic

Substitute Equation (2.4) and Equation (2.5) into Equation (2.2), expand $\frac{n(n\pm1)}{2}=\frac{n^2}{2}\pm\frac{n}{2}$, and collect by powers of n:

$$T(n)=an^2+bn+c \tag{2.6}$$

Here $a=(c_4+c_5+c_6)/2>0$: in Equation (2.6) the worst case is **quadratic**, $\Theta(n^2)$ — derived, not asserted.

> **KEY INSIGHT.** A reverse-sorted million elements need about $10^{12}/2$ shifts: over eight minutes at 10^9 operations per second, for a list `std::sort` handles in under a second.

The machine changes a; it never changes the n^2. That is why this analysis can be done on paper, before the hardware exists. Book: Section 2.11.2 and Equation (2.6).

The Average Case

Assume all $n!$ orderings are equally likely. `key` lands in any of j places with equal chance, so t_j is equally likely to be $1, 2, \ldots, j$: average $\frac{j+1}{2}$ — about halfway along the prefix.

Then $\sum_{j=2}^{n} \frac{j+1}{2} = \frac{n^2+3n-4}{4} \approx \frac{n^2}{4}$ (check $n=6$: 12.5), and the shifts are also about $\frac{n^2}{4}$ — half the worst case.

The n^2 coefficient halves, giving Equation (2.7):

$$T(n) \approx \frac{a}{2} n^2 + b'n + c' \tag{2.7}$$

> **KEY INSIGHT.** Same order of growth as the worst case. If eight minutes is too slow, four minutes is too slow. Only a new growth rate — $n \lg n$, Merge Sort in Chapter 3 — solves it.

Book: Section 2.11.3 and Equation (2.7).

Counting on a Real Machine

Listing 2.3 Counting $\sum_{j=2}^{n} t_j$ for ascending, descending and random input, alongside the predicted worst-case value.

```cpp
// Counting the work Insertion Sort really does, in the three classic cases.
// We count exactly the quantity called t_j in the cost table: the number of
// times the while-loop TEST is evaluated for each j.
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <iomanip>
using namespace std;

long long sortAndCountTests(vector<int> A) {      // A is taken by value: a copy
    int n = (int)A.size() - 1;                    // A[0] unused
    long long tests = 0;
    for (int j = 2; j <= n; ++j) {
        int key = A[j], i = j - 1;
        ++tests;                                  // the test that may fail at once
        while (i > 0 && A[i] > key) {
            A[i + 1] = A[i]; --i;
            ++tests;                              // every re-test counts too
        }
        A[i + 1] = key;
    }
    return tests;
}

int main() {
    cout << left << setw(6) << "n" << setw(12) << "best"
         << setw(12) << "worst" << setw(12) << "random"
```

```
         << setw(14) << "n(n+1)/2-1" << '\n';
    mt19937 rng(12345);                              // fixed seed: reproducible
    for (int n : {8, 16, 32, 64, 128}) {
        vector<int> asc(n + 1), desc(n + 1), rnd(n + 1);
        for (int i = 1; i <= n; ++i) { asc[i] = i; desc[i] = n - i + 1; }
        rnd = asc;
        shuffle(rnd.begin() + 1, rnd.end(), rng);
        cout << left << setw(6) << n
             << setw(12) << sortAndCountTests(asc)
             << setw(12) << sortAndCountTests(desc)
             << setw(12) << sortAndCountTests(rnd)
             << setw(14) << ((long long)n * (n + 1) / 2 - 1) << '\n';
    }
    return 0;
}
```

```
n     best        worst       random      n(n+1)/2-1
8     7           35          25          35
16    15          135         59          135
32    31          527         340         527
64    63          2079        1045        2079
128   127         8255        4453        8255
```

Listing 2.3: best is exactly $n-1$; worst is identical to $n(n+1)/2-1$; random sits near half the worst ($4453/8255=0.539$). Book: Section 2.11.4 and Listing 2.3.

A Stopwatch and the Doubling Test

Listing 2.4 Timing Insertion Sort's worst case at five input sizes, each double the last. The `ratio` column is the observed cost multiplier for doubling n.

```cpp
// Does the quadratic really show up on a real machine? Double n and see.
// Insertion Sort is run on reverse-sorted input, which is its worst case,
// so the count of while-loop tests is exactly n(n+1)/2 - 1 every time.
#include <iostream>
#include <vector>
#include <chrono>
#include <iomanip>
using namespace std;
using namespace std::chrono;

double timeWorstCase(int n) {
    vector<int> A(n + 1);
    for (int i = 1; i <= n; ++i) A[i] = n - i + 1;   // n, n-1, ..., 1
    auto t0 = steady_clock::now();
    for (int j = 2; j <= n; ++j) {
        int key = A[j], i = j - 1;
        while (i > 0 && A[i] > key) { A[i + 1] = A[i]; --i; }
        A[i + 1] = key;
    }
    auto t1 = steady_clock::now();
    return duration<double, milli>(t1 - t0).count();
}

int main() {
    cout << setw(8) << "n" << setw(14) << "time (ms)"
         << setw(12) << "ratio" << setw(18) << "n(n+1)/2-1" << '\n';
    double prev = 0;
    for (int n : {5000, 10000, 20000, 40000, 80000}) {
```

```
        double ms = timeWorstCase(n);
        cout << setw(8) << n << setw(14) << fixed << setprecision(2) << ms;
        if (prev > 0) cout << setw(12) << setprecision(2) << (ms / prev);
        else          cout << setw(12) << "-";
        cout << setw(18) << ((long long)n * (n + 1) / 2 - 1) << '\n';
        prev = ms;
    }
    return 0;
}
```

```
       n     time (ms)       ratio        n(n+1)/2-1
    5000          4.10           -          12502499
   10000         16.09        3.93          50004999
   20000         64.06        3.98         200009999
   40000        256.48        4.00         800019999
   80000       1034.66        4.03        3200039999
```

*Your milliseconds from Listing 2.4 will differ; the **ratio** travels, circling four, because of Equation (2.8) below. Book: Section 2.11.4 and Listing 2.4.*

Why Four — and Why the Worst Case

For $T(n) = an^2 + bn + c$ the n^2 term dominates, so Equation (2.8):

$$\frac{T(2n)}{T(n)} \approx \frac{a(2n)^2}{an^2} = \frac{4an^2}{an^2} = 4. \tag{2.8}$$

Double n: a linear algorithm doubles, a quadratic one quadruples, a logarithmic one adds a constant. If you believe $O(n \lg n)$ and see 4, something is wrong.

Why we study the worst case: it is a **guarantee**; some systems (pacemakers, brakes) require one; it happens often (absent keys, sorted data, adversaries); the average is often the same order; and the average needs a distribution you rarely know.

> **INTUITION.** "Arrives within 5 days" is a promise you can plan around; "usually 3 days" is a hope. Worst-case analysis sells guarantees.

Book: Sections 2.11.4–2.11.5 and Equation (2.8).

Logarithms and the Growth Table

$\lg n = \log_2 n$ answers: *how many times can I halve n before reaching 1?* $\lg 8 = 3$, $\lg 1024 = 10$, $\lg 10^6 \approx 19.93$.

Table 2.3 How five common running times grow. The final entry is so large that it is often written simply as ∞; the computed value has over three hundred thousand digits.

n	$\lg n$	n	$n \lg n$	n^2	2^n
10	3.3	10	33	100	1,024
100	6.6	100	664	10,000	1.27×10^{30}
1,000	10.0	1,000	9,966	1,000,000	1.07×10^{301}
10^6	19.9	10^6	1.99×10^7	10^{12}	9.90×10^{301029}

At $n = 10^6$ and 10^9 operations per second: $\lg n$ — 20 operations; n — a millisecond; $n \lg n$ — twenty milliseconds; n^2 — sixteen minutes forty seconds; 2^n — no unit of time exists.

Every entry of Table 2.3 was computed by the chapter's Listing 2.5, which prints 2^{10^6} without ever storing it. Book: Sections 2.12.1–2.12.2 and Table 2.3.

Growth, Drawn on Log–Log Axes

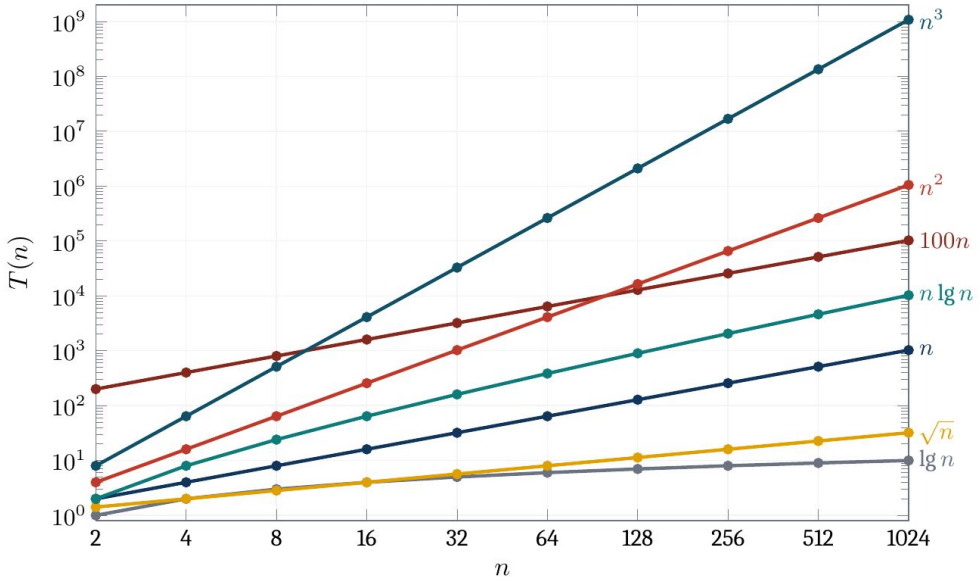


Figure 2.6 Growth of $\lg n$, $\sqrt{n}$, n, $n \lg n$, $100\,n$, n^2 and n^3 with both axes logarithmic. On such a plot a function n^k becomes a straight line whose slope is k, so steeper means worse.

In Figure 2.6 each n^k is a straight line of slope k. $100\,n$ starts fifty times above n^2, the lines cross near $n = 100$, and n^2 then climbs away forever: a large constant hides a bad growth rate only for small n. Book: Section 2.12.3 and Figure 2.12.

Add the Exponential

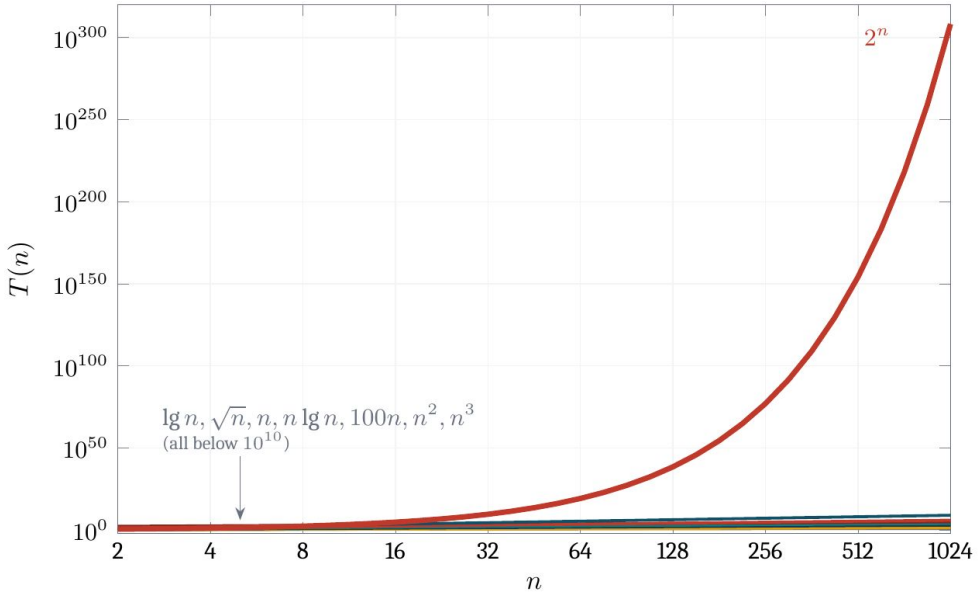


Figure 2.7 The same growth rates with 2^n added. The vertical axis now runs to about 10^{308}; every polynomial curve is crushed into the bottom of the chart, indistinguishable from the axis.

In Figure 2.7 every polynomial is crushed into the axis: against 2^n, all polynomials are fast. That is why the main dividing line runs between polynomial and exponential. Book: Section 2.12.3 and Figure 2.13.

The Hierarchy and the Budget

$$O(1)<O(\lg n)<O(\sqrt{n})<O(n)<O(n\lg n)<O(n^2)<O(n^3)<O(2^n)<O(n!) \tag{2.9}$$

Read Equation (2.9) with examples: `A[i]`; binary search; trial division; one pass; Merge Sort; Insertion Sort's worst case; naive matrix product; all subsets ($n\approx 30$ at most); all orderings ($n\approx 12$).

Table 2.4 The largest input size each complexity class can handle within a one- to two-second limit. Read it backward to choose an algorithm: the problem statement gives you n, and this table gives you the complexity you must achieve.

Max n	Allowed complexity	Operations	Example
$n\le 10$	$O(n!), O(2^n)$	$\sim 10^6$	Brute force / backtracking
$n\le 1,000$	$O(n^2)$	$\sim 10^6$	Insertion Sort, Bubble Sort
$n\le 100,000$	$O(n\lg n)$	$\sim 1.7\times 10^6$	Merge Sort, Quicksort
$n\le 10^6$	$O(n)$ or $O(n\lg n)$	$\sim 10^7$	Linear scan, counting sort
$n\le 10^{18}$	$O(\lg n)$ or $O(1)$	~ 60	Binary search, closed-form formula

Use Table 2.4 backward: the statement gives n, the table gives the complexity you must reach. $O(n^2)$ at $n=10^6$ is five hundred times over budget — no micro-optimization closes that. Book: Sections 2.12.4–2.12.5, Equation (2.9) and Table 2.4.

The Searching Problem



Figure 2.8 The searching problem. On the instance $A = \langle 2, 5, 4, 10, 7 \rangle$ with $q = 5$ the answer is the index 2; with $q = 9$ the answer is NIL, because 9 does not occur.

Figure 2.8: return an ***index*** j *with* $A[j] = q$*, or NIL. NIL must be a value no real index can take — 0 with 1-based indices,* -1 *with 0-based. Book: Section 2.13.1 and Figure 2.14.*

Linear Search and Its Invariant

Pseudocode 2.3 Linear search: examine the elements from left to right until q is found or the array runs out.

```
LINEAR-SEARCH(A, q)
1  j = 1
2  while j <= A.length and A[j] != q
3      j = j + 1
4  if j <= A.length
5      return j
6  else
7      return NIL
```

LOOP INVARIANT (linear search). At the start of each iteration, none of $A[1], \ldots, A[j-1]$ equals q.

Initialization: an empty range. **Maintenance:** the body runs only when $A[j] \neq q$. **Termination:** either $A[j] = q$ (return j) or $j = n+1$ and no element equals q (return NIL). ∎

In Pseudocode 2.3, `j <= A.length` must be tested first, so we never read past the end. Book: Sections 2.13.2–2.13.3 and Pseudocode 2.3.

Linear Search: Analysis and Code

Worst (absent or last): $\Theta(n)$. **Best** ($A[1]=q$): $\Theta(1)$. **Average**, q present and uniform, Equation (2.10):

$$\frac{1}{n}(1+2+\cdots+n)=\frac{1}{n}\cdot\frac{n(n+1)}{2}=\frac{n+1}{2}, \tag{2.10}$$

Listing 2.5 Linear search in C++, on the same data as Figure 2.8.

```cpp
// Linear search on an UNSORTED array, exactly as the pseudocode has it.
// Indices are 1-based; A[0] is unused. NIL is represented by 0, which is
// never a legal index, so it cannot be confused with a genuine answer.
#include <iostream>
#include <vector>
using namespace std;

const int NIL = 0;

int linearSearch(const vector<int>& A, int n, int q) {
    int j = 1;
    while (j <= n && A[j] != q) ++j;    // stop at the first match, or run off the end
    if (j <= n) return j;               // j is a real index: we found q there
    return NIL;                         // we walked off the end: q is absent
}

int main() {
    vector<int> A = {0, 2, 5, 4, 10, 7};   // the data of the search-problem figure
    int n = 5;
    for (int q : {5, 9}) {
        int j = linearSearch(A, n, q);
        cout << "q=" << q << " -> ";
        if (j == NIL) cout << "NIL\n"; else cout << j << '\n';
    }
    return 0;
}
```

```
q=5 -> 2
q=9 -> NIL
```

$\Theta(n)$ is optimal on unsorted data: an unexamined position could hold q. To go faster, change the data. Listing 2.5 reproduces Figure 2.8. Book: Sections 2.13.4–2.13.6, Equation (2.10) and Listing 2.6.

The Sorted Searching Problem

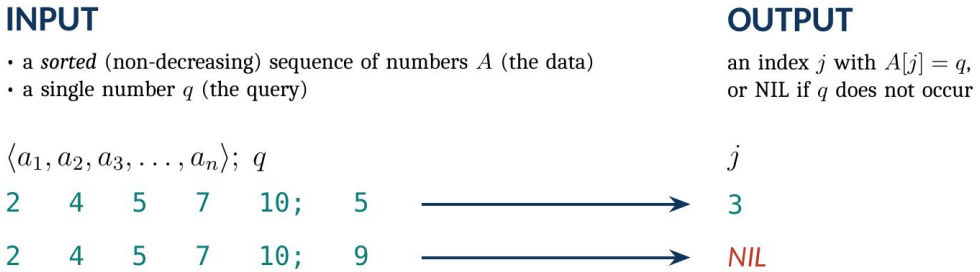


Figure 2.9 The sorted searching problem. The one added word — *sorted* — in the input specification is what makes an exponential speed-up possible.

One added word, sorted, in Figure 2.9: now one comparison with $A[5] > q$ rules out positions 5, 6, 7, … at once. Binary search on unsorted data silently returns wrong answers. Book: Section 2.14.1 and Figure 2.15.

Binary Search

Compare q with the **middle**: equal — found; $A[mid]>q$ — discard the right half; $A[mid]<q$ — discard the left half. Like opening a phone book near the middle.

Pseudocode 2.4 Binary search on a sorted array: compare with the middle element and discard the half that cannot contain q.

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

LOOP INVARIANT (binary search). At the start of each iteration, if q occurs in A, it occurs within the window $[left, right]$.

In Pseudocode 2.4, $(left, right)$ is the window where q may still hide; an empty window means absent. Book: Sections 2.14.2–2.14.3 and Pseudocode 2.4.

Binary Search Is Correct and Terminates

Initialization: the window is the whole array. **Maintenance:** if $A[mid] > q$, every $A[k]$ with $k \geq mid$ exceeds q, so q can only be in $[left, mid-1]$; symmetric for $<$.

Termination: at $left > right$ the window is empty, so q is absent and NIL is right; the other exit returns mid with $A[mid] = q$.

Progress: $left \leq mid \leq right$, so $mid-1$ and $mid+1$ strictly shrink the window. ▰

TRAP. Write `right = mid` and the window $[1,2]$ with $A[1] > q$ becomes $[1,1]$, then $[1,1]$ forever: the program hangs.

Book: Section 2.14.3.

A Trace, and Why Logarithmic

Table 2.5 Binary search for $q=7$ in $A=\langle 2,4,5,7,10\rangle$. Two comparisons suffice where linear search would need four.

Step	Left	Right	Mid	Action
1	1	5	3	$A[3]=5<7 \Rightarrow$ search the right half: $left=4$
2	4	5	4	$A[4]=7=7 \Rightarrow$ found! Return 4

After k comparisons the window holds at most $n/2^k$ elements; it is exhausted once $n/2^k \le 1$, Equation (2.11):

$$\frac{n}{2^k} \le 1 \Leftrightarrow n \le 2^k \Leftrightarrow \lg n \le k. \tag{2.11}$$

So $T(n)=\Theta(\lg n)$: **20** comparisons for 10^6 elements, **60** for 10^{18}. Every doubling of n adds exactly one comparison.

In Table 2.5, one comparison discards three of the five positions. Book: Sections 2.14.4–2.14.5, Table 2.5 and Equation (2.11).

Binary Search in C++

Listing 2.6 Binary search in C++, iterative form, instrumented to print the window at each step.

```cpp
// Binary search on a SORTED array, with a printed trace of left, right, mid.
// 1-based indices; NIL is 0. The midpoint is computed as
// left + (right - left) / 2, never (left + right) / 2 -- see the text.
#include <iostream>
#include <vector>
using namespace std;

const int NIL = 0;

int binarySearch(const vector<int>& A, int n, int q, bool trace) {
    int left = 1, right = n, step = 0;
    while (left <= right) {                 // an empty window means "absent"
        int mid = left + (right - left) / 2;
        if (trace)
            cout << "  step " << ++step << ": left=" << left
                 << " right=" << right << " mid=" << mid
                 << " A[mid]=" << A[mid] << '\n';
        if (A[mid] == q) return mid;
        else if (A[mid] > q) right = mid - 1;   // q can only be to the left
        else                 left  = mid + 1;   // q can only be to the right
    }
    return NIL;
}

int main() {
    vector<int> A = {0, 2, 4, 5, 7, 10};    // sorted; A[0] unused
    int n = 5;
    cout << "searching for 7, with trace:\n";
```

```
    int j = binarySearch(A, n, 7, true);
    cout << "  result = " << j << "\n\n";
    for (int q : {5, 9}) {
        int r = binarySearch(A, n, q, false);
        cout << "q=" << q << " -> ";
        if (r == NIL) cout << "NIL\n"; else cout << r << '\n';
    }
    return 0;
}
```

```
searching for 7, with trace:
  step 1: left=1 right=5 mid=3 A[mid]=5
  step 2: left=4 right=5 mid=4 A[mid]=7
  result = 4

q=5 -> 3
q=9 -> NIL
```

The traced steps of Listing 2.6 equal Table 2.5; the last two lines reproduce Figure 2.9. Note the midpoint: `left + (right - left) / 2`. *Book: Section 2.14.6 and Listing 2.7.*

The Midpoint Overflow Trap

Listing 2.7 Why the midpoint must be computed as `left + (right - left) / 2`.

```cpp
// The midpoint overflow trap, demonstrated rather than asserted.
// Both expressions are mathematically the same. In 32-bit arithmetic only
// one of them is safe, because left + right can exceed INT_MAX.
#include <iostream>
#include <climits>
using namespace std;

int main() {
    int left  = 1'500'000'000;
    int right = 2'000'000'000;
    cout << "INT_MAX           = " << INT_MAX << '\n';
    cout << "left              = " << left  << '\n';
    cout << "right             = " << right << '\n';
    cout << "true midpoint     = " << 1'750'000'000 << '\n';

    // Signed overflow is undefined behavior, so we do the addition in a wider
    // type and then cut it down to int -- that reproduces what the machine does
    // without the compiler being free to assume it never happens.
    long long sum = (long long)left + right;
    int wrapped = (int)(unsigned int)sum;          // the bit pattern int would hold
    cout << "(left+right)      = " << sum << " which does not fit in int\n";
    cout << "(left+right)/2    = " << wrapped / 2 << "   <-- wrong, negative\n";
    cout << "left+(right-left)/2 = " << left + (right - left) / 2 << "   <-- right\n";
    return 0;
}
```

```
INT_MAX           = 2147483647
left              = 1500000000
right             = 2000000000
true midpoint     = 1750000000
(left+right)      = 3500000000 which does not fit in int
(left+right)/2    = -397483648   <-- wrong, negative
left+(right-left)/2 = 1750000000   <-- right
```

This bug hid for two decades in Programming Pearls and for years in Java's library. Listing 2.7 shows why: $right - left$ *is always small,* $left + right$ *need not be. Book: Section 2.14.6 and Listing 2.8.*

Binary Search as Recursion

Listing 2.8 Binary search as a recursive divide-and-conquer procedure, printing the window at each level of recursion.

```cpp
// Binary search written recursively: the divide-and-conquer shape made visible.
// The recursion is on the window [left..right]; an empty window is the base case.
#include <iostream>
#include <vector>
using namespace std;

const int NIL = 0;

int bsRec(const vector<int>& A, int left, int right, int q, int depth) {
    cout << string(2 * depth, ' ') << "window [" << left << ".." << right << "]";
    if (left > right) { cout << " is empty -> NIL\n"; return NIL; }   // base case
    int mid = left + (right - left) / 2;
    cout << " mid=" << mid << " A[mid]=" << A[mid] << '\n';
    if (A[mid] == q) return mid;
    if (A[mid] > q) return bsRec(A, left, mid - 1, q, depth + 1);
    return               bsRec(A, mid + 1, right, q, depth + 1);
}

int main() {
    vector<int> A = {0, 1, 3, 5, 7, 9, 11, 13, 15};   // 8 sorted elements
    int n = 8;
    for (int q : {13, 6}) {
        cout << "looking for " << q << ":\n";
        int r = bsRec(A, 1, n, q, 1);
        cout << "result: ";
        if (r == NIL) cout << "NIL\n\n"; else cout << r << "\n\n";
    }
    return 0;
```

```
}
```

```
looking for 13:
  window [1..8] mid=4 A[mid]=7
    window [5..8] mid=6 A[mid]=11
      window [7..8] mid=7 A[mid]=13
result: 7

looking for 6:
  window [1..8] mid=4 A[mid]=7
    window [1..3] mid=2 A[mid]=3
      window [3..3] mid=3 A[mid]=5
        window [4..3] is empty -> NIL
result: NIL
```

In Listing 2.8 one call on half the window plus constant work gives the recurrence Equation (2.12), solved mechanically in Chapter 3. Book: Section 2.14.7 and Listing 2.9.

A Recurrence, and the Price of Sorting

$$T(n) = T(n/2) + \Theta(1), \tag{2.12}$$

Equation (2.12): the time for n elements is the time for $n/2$ plus one comparison. Its solution is $\Theta(\lg n)$.

Trade-off for k searches: linear costs $\Theta(k\,n)$; sort then search costs $\Theta(n \lg n + k \lg n)$.

At $n = 10^6$: one search — linear wins; one hundred — binary wins five times over; ten thousand — five hundred times over. Break-even is near $k \approx \lg n \approx 20$.

> **KEY INSIGHT.** Pay once to build structure, then query cheaply forever — the bargain behind hash tables, search trees, heaps and prefix sums.

Book: Sections 2.14.7–2.14.8 and Equation (2.12).

Design Paradigms Revisited

Incremental (Insertion Sort): extend a solution for $j-1$ items to j. Simple, good on small or nearly sorted data, *online*; library sorts use it below about sixteen elements. Weakness: often $\Theta(n^2)$.

Divide and conquer (binary search, Merge Sort): divide, conquer, combine. Often $\Theta(n \lg n)$ or $\Theta(\lg n)$; parallelizes; costs recursion overhead and sometimes $\Theta(n)$ scratch memory.

Ad hoc: a trick for one problem — *Columns*' $\max(n, \max_i a_i)$, *Interesting Sequence*'s popcount. Fastest when found; found only by staring at small cases.

Learn the paradigms so you always have a move; keep looking for the trick so that sometimes you have a better one. Book: Section 2.15.

Chapter 1, Re-Examined

Table 2.6 The six problems of Chapter 1, re-examined with the analysis tools of Chapter 2. Every complexity claim made informally in Chapter 1 is now something you can derive.

Chapter 1 problem	Paradigm	Chapter 2 concept applied	Complexity
Columns	Greedy / direct	Single-pass incremental scan	$O(n)$ — linear, best possible
Journey	Greedy	Minimum and maximum in one pass	$O(n)$ — the lower bound for reading the input
Unique Numbers	Number theory	Euclid's GCD, $O(\lg n)$ per call	$O(k \lg n)$ worst case
Min/Max Numbers	Greedy construction	Print digits incrementally	$O(N)$ per test case
Interesting Sequence	Bit manipulation	$O(1)$ built-in CPU instruction	$O(1)$ — constant time
Recursive Function 1	Memoization / DP	Map access $O(\lg D)$ per lookup, D distinct keys	$O(\lg^2 n \cdot \lg \lg n)$

Columns and *Journey* cannot beat $O(n)$: any algorithm must read n numbers. *Recursive Function 1*: $D = O(\lg^2 n)$ keys, each lookup $O(\lg \lg n)$ — against $2f(n) - 1 \approx 4.5 \times 10^{14}$ naive calls.

Every informal claim of Chapter 1 in Table 2.6 is now something you can derive. Book: Section 2.16 and Table 2.6.

Profitina in Practice

Input size first: a check over all pairs of pages is invisible at 40 pages (1,600 comparisons) and minutes of CPU at 40,000 (1.6 billion) — so every signal has a stated complexity budget.

Nearly sorted data: weekly scores arrive almost in order, so inserting each point runs near the $\Theta(n)$ best case. **Scheduler:** sort once, binary-search many times. **Worst case is the service-level promise** — and a scanned site is adversarial input.

Profitina Laundry: ten tickets arriving nearly in order cost **3 shifts** against a worst case of 45; finding a ticket by binary search costs **4 probes**, present or absent, where $\lceil \lg(n+1) \rceil = 4$.

The app advises; the user decides. The full counter-book program is Listing 2.10. Book: Section 2.17.

Common Mistakes

Blaming the language, calling a loop one step, quoting wall-clock time as complexity. Count operations; use the doubling test.

Insertion Sort: losing the key; inserting at $A[i]$ instead of $A[i+1]$; testing $A[i] > key$ before $i > 0$.

Binary search: `mid = (left + right) / 2`; `right = mid`; running it on unsorted data; a NIL value that can be a real index.

Economics: sorting to search once; tuning constants when the growth rate is wrong (Table 2.3).

Proving by testing. Ten passing tests prove ten inputs. State the invariant and check its three parts.

Book: Section 2.18.

Summary — Five Things to Remember

1. Problem, instance, algorithm, program, data structure are five different things; correctness is about all instances, running time about one.

2. In the RAM model, $T(n)$ = sum over lines of cost × times. A loop is never one step.

3. Insertion Sort is correct by its invariant; it is $\Theta(n)$ at best ($t_j = 1$), $\Theta(n^2)$ at worst ($t_j = j$) and on average.

4. Growth rate dominates: $O(1) < O(\lg n) < O(n) < O(n \lg n) < O(n^2) < O(2^n)$; we analyze the worst case because it is a guarantee.

5. Linear search is optimal on unsorted data at $\Theta(n)$; binary search is $\Theta(\lg n)$ on sorted data — speed bought by structure.

"An algorithm must be seen to be believed" (Knuth). Trace every algorithm by hand. Book: Section 2.19 and Figure 2.16.

Next Time

Chapter 3 — Asymptotic Notation, Recurrences, and the Master Theorem.

O, Ω, Θ defined with explicit constants and proved; summations; recurrences solved by substitution, recursion trees and the Master Theorem.

Merge Sort derived from nothing: $T(n) = 2T(n/2) + \Theta(n)$ gives $\Theta(n \lg n)$, finally beating today's quadratic.

Before we meet: Exercises 2.1, 2.5, 2.10 and 2.12 — trace, prove, count, and break a binary search. Solutions are in Section 2.21.

Book: Sections 2.20–2.21, then Chapter 3.

Design and Analysis of Algorithms (EF234405)

Lecture 3 — Asymptotic Notation, Recurrences, and the Master Theorem

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*, Chapter 3

Lecturer: Irfan Subakti — `yifana@gmail.com`

Department of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya

Today every "about n^2" becomes a statement with constants you can write down. Book: Chapter 3, and Cormen et al., Introduction to Algorithms, 4th ed., Chapters 3–4.

Agenda

1. Partial and total correctness; why constants are thrown away.

2. The five notations O, Ω, Θ, o, ω — proofs *and* disproofs — and their algebra.

3. The common functions and the summations of loops: arithmetic, geometric, harmonic, telescoping.

4. Recurrences, solved three ways: substitution, recursion tree, and the **Master Theorem** with its gaps.

5. Merge Sort derived from nothing: algorithm, trace, invariant, recurrence, program.

Book: Chapter 3, Sections 3.1–3.18.

Where We Left Off

The RAM model: count instructions, not seconds. Seconds belong to a laptop on a Tuesday; instruction counts belong to the algorithm.

The loop invariant: initialization, maintenance, termination — induction wearing different clothes. Insertion Sort was proved correct with it.

Three holes remain. *Language*: what does "about n^2" mean? *Arithmetic*: closed forms for the sums that loops produce. *Recursion*: Merge Sort calls itself, so it needs a recurrence.

This chapter fills all three. Book: Sections 3.1 and 3.1.2.

Partial and Total Correctness

> **DEFINITION.** An algorithm is **partially correct** if, *whenever it terminates*, its output is correct. It is **totally correct** if it always terminates on every legal input *and* its output is then correct.

Total correctness = partial correctness + termination. "*If* the chef finishes, the dish is delicious" versus "the chef finishes, *and* it is delicious".

No program can check correctness for us: deciding whether an algorithm halts is the **Halting Problem**, which Turing proved undecidable in 1936.

Termination is usually obvious (a counter climbs to a limit; a recursion shrinks its input); the hard work is partial correctness.

Book: Section 3.1.1.

The Two Guarantees, Drawn

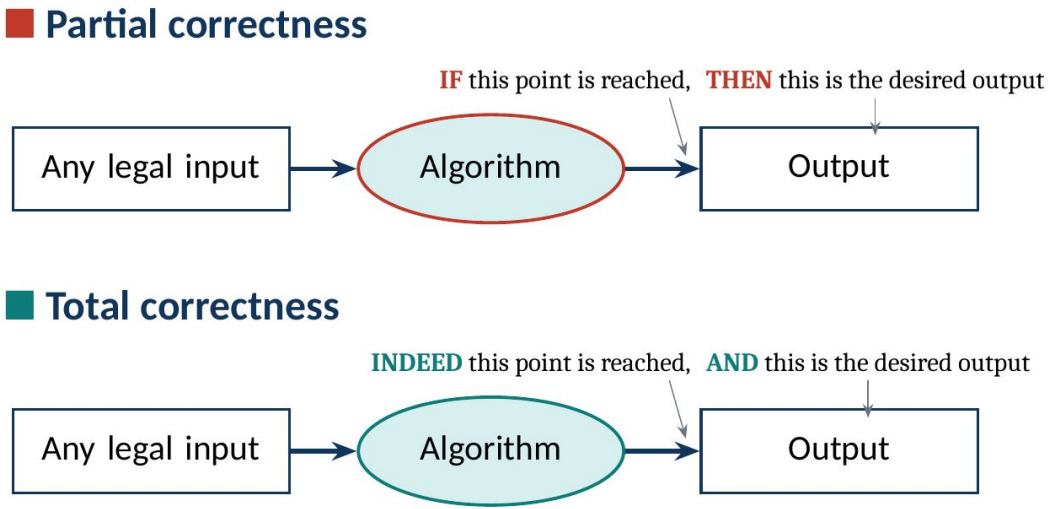


Figure 3.1 Partial correctness against total correctness. In the upper diagram the guarantee is conditional — **IF** control reaches the output box, **THEN** the output is the desired one. In the lower diagram the guarantee is unconditional — control **INDEED** reaches the output box, **AND** the output is the desired one.

*Figure 3.1: the upper diagram promises **IF** control reaches the output, **THEN** it is right; the lower promises control **INDEED** reaches it, **AND** it is right. Book: Section 3.1.1 and Figure 3.1.*

Why a Language for Growth

A sorts $1,000$ records in 0.8 s, B in 0.1 s. At $1,000,000$: A $(n \lg n)$ needs 0.8×2000 s ≈ 27 minutes; B (n^2) needs 0.1×10^6 s ≈ 28 hours.

> **KEY INSIGHT.** Drop **constant factors** $(3n^2 \to n^2)$ and **lower-order terms** $(5n^2+3n+7 \to n^2)$: what the machine decides is thrown away; what the algorithm decides is kept.

Table 3.1 The growth classes, what happens to each when the input size doubles, and where each one shows up in this book.

Growth	Name	Doubling n ...	Example from this book
$\Theta(1)$	constant	changes nothing	reading $A[i]$
$\Theta(\lg n)$	logarithmic	adds one step	binary search
$\Theta(n)$	linear	doubles the work	scanning an array once
$\Theta(n \lg n)$	linearithmic	slightly more than doubles it	Merge Sort, Heapsort
$\Theta(n^2)$	quadratic	multiplies work by 4	Insertion Sort, worst case
$\Theta(n^3)$	cubic	multiplies work by 8	naive matrix multiplication
$\Theta(2^n)$	exponential	*squares* the work	brute-force subset search
$\Theta(n!)$	factorial	worse still	brute-force permutations

Memorize the doubling column of Table 3.1. For small n constants still rule: Insertion Sort beats Merge Sort on ten elements, which is why library sorts switch to it near sixteen. Listing 3.1 prints the raw numbers. Book: Sections 3.2–3.2.1 and Table 3.1.

Big-O: the Upper Bound

n is the input size; $f(n)$ the function studied; $g(n)$ the simple comparison function; c a **fixed** positive multiplier; n_0 a threshold below which we promise nothing.

DEFINITION (Big-O). $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that Equation (3.1) holds:

$$0 \le f(n) \le c \cdot g(n) \text{ for all } n \ge n_0. \tag{3.1}$$

In words: f **eventually stays under some fixed multiple of** g**, forever.** You may pick the multiple and ignore small inputs; you may not change your choice as n grows.

Big-O states worst-case upper bounds. Book: Section 3.3 and Equation (3.1).

Big-O, Drawn

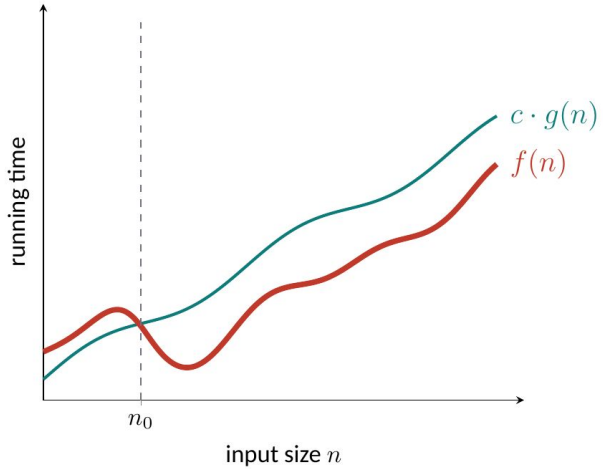


Figure 3.2 Big-O as an upper bound. Beyond the threshold n_0, the running time $f(n)$ never rises above $c \cdot g(n)$. What happens to the left of n_0 is irrelevant to the claim.

In Figure 3.2, left of n_0 the curve f may wander above $c \cdot g$; from n_0 on it is trapped underneath. Book: Section 3.3 and Figure 3.2.

Proving a Big-O Claim

CLAIM: $7n - 3 = O(n)$. Subtracting makes things smaller, so $7n - 3 \le 7n$ for every n. Witnesses: $c = 7$, $n_0 = 1$. ▰ Any working pair is a complete proof; $c = 100$, $n_0 = 50$ would do too.

CLAIM: $3n^2 + 5n + 7 = O(n^2)$. For $n \ge 1$, $n \le n^2$ and $1 \le n^2$, so $3n^2 + 5n + 7 \le 15n^2$: $c = 15$, $n_0 = 1$.

Pushing c down: $c = 4$ needs $5n + 7 \le n^2$, false at $n = 6$ ($37 > 36$), true from $n = 7$. So $c = 4$, $n_0 = 7$ also works — a smaller constant bought with a larger threshold.

A proof of an O-claim is always the same shape: exhibit the witnesses, verify the inequality. Book: Sections 3.3.1–3.3.2.

Disproving a Big-O Claim

Suppose, for contradiction, that $n^2 \le c \cdot n$ for all $n \ge n_0$.

Divide by $n \ge 1$: then $n \le c$ for all $n \ge n_0$ — the integer n never exceeds a fixed number.

Take $n = \max(n_0, \lfloor c \rfloor + 1)$: then $n > c$. Contradiction, so $n^2 \neq O(n)$. ∎

> **KEY INSIGHT.** To *prove* $f = O(g)$, exhibit one pair (c, n_0). To *disprove* it, show that *every* pair fails. Proving is existential; disproving is universal.

Book: Section 3.3.3.

Letting the Machine Hunt for Witnesses

Listing 3.1 Hunting numerically for the constants c and n_0 in the definition of Big-O.

```cpp
// witness.cpp - Searching for the constants c and n0 that witness an O-claim,
// and watching the search fail when the claim is false.
// Compile: g++ -std=c++17 -O2 -o witness witness.cpp
#include <iostream>
#include <functional>
#include <string>
#include <cmath>
using namespace std;

// Try every c in 1..cMax; for each, find the smallest n0 such that
// f(n) <= c*g(n) holds for every n in [n0, nMax]. Report the first success.
bool findWitness(const string& claim,
                 function<double(double)> f,
                 function<double(double)> g,
                 int cMax, int nMax) {
    for (int c = 1; c <= cMax; ++c) {
        int n0 = -1;
        // Scan downward from nMax: the last n that violates the bound fixes n0.
        for (int n = nMax; n >= 1; --n) {
            if (f(n) > c * g(n)) { n0 = n + 1; break; }
            if (n == 1) n0 = 1;
        }
        if (n0 != -1 && n0 <= nMax) {
            cout << claim << " : witnessed by c = " << c
                 << ", n0 = " << n0 << "\n";
            return true;
        }
    }
```

```
    cout << claim << " : NO witness with c <= " << cMax
         << " on n <= " << nMax << " -- the claim looks false\n";
    return false;
}

int main() {
    findWitness("7n - 3 = O(n)      ", [](double n){ return 7*n - 3; },
                                       [](double n){ return n; }, 20, 100000);
    findWitness("3n^2+5n+7 = O(n^2) ", [](double n){ return 3*n*n + 5*n + 7; },
                                       [](double n){ return n*n; }, 20, 100000);
    findWitness("2n+10 = O(n)       ", [](double n){ return 2*n + 10; },
                                       [](double n){ return n; }, 20, 100000);
    findWitness("n^2 = O(n)         ", [](double n){ return n*n; },
                                       [](double n){ return n; }, 20, 100000);
    findWitness("2^n = O(n^10)      ", [](double n){ return (n<1000)?pow(2.0,n):1e300; },
                                       [](double n){ return pow(n,10); }, 20, 200);
    return 0;
}
```

```
7n - 3 = O(n)       : witnessed by c = 7, n0 = 1
3n^2+5n+7 = O(n^2)  : witnessed by c = 4, n0 = 7
2n+10 = O(n)        : witnessed by c = 3, n0 = 10
n^2 = O(n)          : NO witness with c <= 20 on n <= 100000 -- the claim looks false
2^n = O(n^10)       : NO witness with c <= 20 on n <= 200 -- the claim looks false
```

Listing 3.1 confirms $c=4, n_0=7$. But a program only checks finitely many n: it can suggest a claim is false, never prove it. Book: Section 3.3.4 and Listing 3.2.

Big-Omega: the Lower Bound

DEFINITION (Big-Ω). $f(n)=\Omega(g(n))$ if there exist positive constants c and n_0 such that Equation (3.2) holds:

$$0 \le c \cdot g(n) \le f(n) \text{ for all } n \ge n_0. \tag{3.2}$$

$7n-3=\Omega(n)$: $c=1$ needs $n \le 7n-3$, i.e. $n \ge 1/2$, so $n_0=1$. $c=6$ needs $n \ge 3$. $c=7$ never works.

$1/2\,n^2-3n=\Omega(n^2)$: $c=1/4$ needs $3n \le 1/4\,n^2$, i.e. $n \ge 12$.

Ω states best cases and, more importantly, **problem lower bounds**: searching an unsorted array is $\Omega(n)$ for *every* algorithm.

Book: Section 3.4 and Equation (3.2).

Big-Omega, Drawn

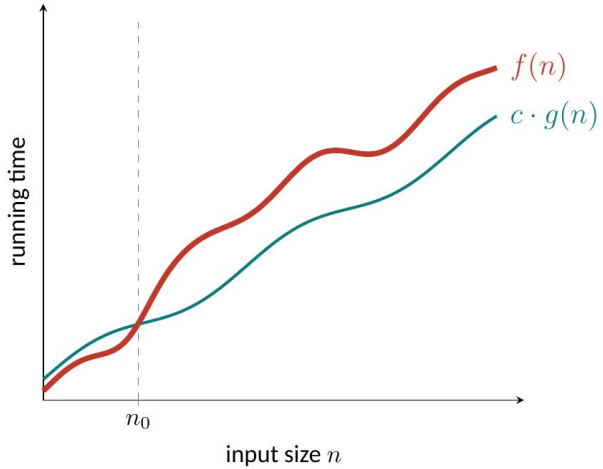


Figure 3.3 Big-Ω as a lower bound. Beyond n_0, the running time $f(n)$ never falls below $c \cdot g(n)$.

In Figure 3.3, f may dip below $c \cdot g$ early on, then climbs above it at n_0 and never comes back. Book: Section 3.4 and Figure 3.3.

Big-Theta: the Tight Bound

DEFINITION (Big-Θ). $f(n)=\Theta(g(n))$ if there exist positive constants c_1, c_2 and n_0 such that Equation (3.3) holds:

$$0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n) \text{ for all } n \ge n_0. \tag{3.3}$$

THEOREM. $f(n)=\Theta(g(n))$ if and only if $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$. (Take c_2 from the O-bound, c_1 from the Ω-bound, n_0 the larger threshold.)

f is eventually sandwiched between two fixed multiples of g: same growth rate, up to constants. Practice: prove the upper bound, prove the lower bound, combine.

Book: Section 3.5 and Equation (3.3).

Big-Theta, Drawn

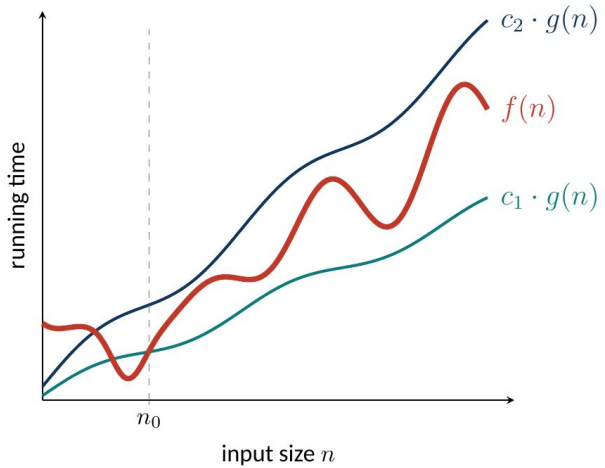


Figure 3.4 Big-Θ as a tight bound. Past n_0 the function $f(n)$ is trapped in the corridor between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$, so f and g grow at the same rate.

In Figure 3.4, past n_0 the function runs in the corridor between $c_1 g(n)$ and $c_2 g(n)$. Book: Section 3.5 and Figure 3.4.

A Big-Theta Proof in Full

CLAIM: $1/2\,n^2 - 3n = \Theta(n^2)$. *Upper:* $-3n \le 0$, so $1/2\,n^2 - 3n \le 1/2\,n^2$: $c_2 = 1/2$ from $n \ge 1$. *Lower:* $c_1 n^2 \le 1/2\,n^2 - 3n \Leftrightarrow n \ge \dfrac{3}{\frac{1}{2} - c_1}$; with $c_1 = 1/4$, $n \ge 12$. *Combine:* $c_1 = 1/4$, $c_2 = 1/2$, $n_0 = 12$. ▰

THINK. $7n = O(n^5)$ is true and useless. Report the tightest bound you can prove: "Insertion Sort is $\Theta(n^2)$ in the worst case" says the bound cannot be improved.

Book: Section 3.5.

Little-o and Little-omega

DEFINITION (little-o). $f(n)=o(g(n))$ if for **every** positive constant c there is an n_0 with Equation (3.4); equivalently $f(n)/g(n)\to 0$.

$$0\le f(n)<c\cdot g(n) \text{ for all } n\ge n_0. \tag{3.4}$$

Big-O says "**there exists** a c"; little-o says "**for every** c". Little-ω is the mirror image: $f/g\to\infty$, and $f=\omega(g)\Leftrightarrow g=o(f)$. $2n=o(n^2)$: given c, take $n_0=\lceil 2/c\rceil+1$. $2n^2\neq o(n^2)$: $c=1$ already fails. $\lg n=o(n)$; $n^{100}=o(2^n)$.

One bad c is enough to kill a little-o claim. Book: Section 3.6 and Equation (3.4).

The Number Analogy

Table 3.2 The five asymptotic notations and the numerical comparison each one imitates.

Notation	Meaning for growth	Number analogy
$f(n)=O(g(n))$	f grows no faster than g	$f \le g$
$f(n)=\Omega(g(n))$	f grows at least as fast as g	$f \ge g$
$f(n)=\Theta(g(n))$	f grows at the same rate as g	$f = g$
$f(n)=o(g(n))$	f grows strictly slower than g	$f < g$
$f(n)=\omega(g(n))$	f grows strictly faster than g	$f > g$

$O(g(n))$ is really a **set**; $f(n)=O(g(n))$ means $f(n)\in O(g(n))$. The equals sign is one-way: $2n=O(n^2)$, never "$O(n^2)=2n$". Inside a formula, $\Theta(n)$ means *some anonymous function* of that class: $T(n)=2T(n/2)+\Theta(n)$.

Table 3.2 is a good servant and a bad master: it breaks at trichotomy (next page but one). Book: Sections 3.6.1–3.6.2 and Table 3.2.

The Algebra of the Notations

Transitivity (all five): $f \le c_1 g$ and $g \le c_2 h$ give $f \le (c_1 c_2) h$ beyond $\max(n_1, n_2)$.

Reflexivity: $f = O(f) = \Omega(f) = \Theta(f)$; never $f = o(f)$, just as $x < x$ is false.

Symmetry only for Θ; **transpose symmetry**: $f = O(g) \Leftrightarrow g = \Omega(f)$ and $f = o(g) \Leftrightarrow g = \omega(f)$.

Sum rule $f_1 + f_2 = O(\max(g_1, g_2))$; **product rule** $f_1 f_2 = O(g_1 g_2)$; **constant rule** $k f_1 = O(g_1)$.

The sum rule is where "drop lower-order terms" comes from; the constant rule is where "drop constants" comes from. Book: Sections 3.7.1–3.7.4.

Why We Drop Terms — and Where the Analogy Breaks

Every polynomial is Θ of its leading power. For $p(n)=a_d n^d+\cdots+a_0$ with $a_d>0$: upper $p(n)\le A n^d$ with $A=\sum|a_i|$; lower $p(n)\ge a_d n^d - B n^{d-1}\ge \frac{a_d}{2} n^d$ once $n\ge 2B/a_d$.

$50n\log n \to \Theta(n\log n)$; $7n-3 \to \Theta(n)$; $8n^2\log n+5n^2+n \to \Theta(n^2\log n)$.

> **COMMON MISCONCEPTION.** "Any two functions can be compared." False: n and $n^{1+\sin n}$ (exponent swinging between 0 and 2) are neither O nor Ω of each other. **Trichotomy fails.**

Real running times are built from polynomials, logs and exponentials, which are all comparable — so the analogy works day to day. Book: Sections 3.7.5–3.7.6.

Floors, Logarithms, Stirling

$x-1<\lfloor x\rfloor\le x\le\lceil x\rceil<x+1$ and $\lceil n/2\rceil+\lfloor n/2\rfloor=n$ — why Merge Sort's halves always add back to n.

$\lg n=\log_2 n$, $\ln n=\log_e n$, $\lg^k n=(\lg n)^k$ — **not** $\lg\lg n$. Change of base makes every logarithm a constant multiple of every other, Equation (3.5):

$$\log_2 n=\Theta(\log_{10} n)=\Theta(\ln n)=\Theta(\log_b n)\text{ for any fixed }b>1. \tag{3.5}$$

Every exponential beats every polynomial: $n^d=o(a^n)$ for $a>1$. Stirling, Equation (3.6):

$$n!=\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right). \tag{3.6}$$

So $n!=o(n^n)$, $n!=\omega(2^n)$ and $\lg(n!)=\Theta(n\lg n)$ — the reason no comparison sort beats $n\lg n$ (Chapter 5). The base matters only in exponents such as $n^{\log_b a}$. Book: Sections 3.8.2–3.8.7, Table 3.3 and Equations (3.5)–(3.6).

The Logarithm Identities

Table 3.3 The logarithm identities used in this chapter. The last two are the ones that make the Master Theorem work.

Identity	Why it is true
$\log_b(xy) = \log_b x + \log_b y$	exponents add when powers multiply
$\log_b(x/y) = \log_b x - \log_b y$	exponents subtract when powers divide
$\log_b(x^a) = a \log_b x$	$(b^y)^a = b^{ya}$
$\log_b x = \dfrac{\log_c x}{\log_c b}$	change of base; $\log_c b$ is a constant
$b^{\log_b x} = x$	the definition, read backward
$x^{\log_b y} = y^{\log_b x}$	take $\log_b$ of both sides: both give $\log_b x \cdot \log_b y$
$a^{\log_b n} = n^{\log_b a}$	the previous line, with $x = a$, $y = n$
$\log_b a = \dfrac{1}{\log_a b}$	change of base with $c = a$

The heart of the Master Theorem: $a^{\log_b n} = n^{\log_b a}$. Take $\log_b$ of both sides: each gives $(\log_b n)(\log_b a)$, and $\log_b$ is one-to-one. ∎

Table 3.3: the last two rows are the ones the Master Theorem uses. Book: Section 3.8.6 and Table 3.3.

The Common Functions, in Order

Table 3.4 The common functions in increasing order of growth. Each row is $o(\cdot)$ of every row below it.

Function	Growth class	Value at n=1024
1	constant	1
$\lg^* n$	iterated log	4
$\lg \lg n$	log log	≈ 3.3
$\lg n$	logarithmic	10
$\sqrt{n} = n^{1/2}$	root	32
n	linear	1024
$n \lg n$	linearithmic	10240
n^2	quadratic	1048576
n^3	cubic	$\approx 1.07 \times 10^9$
2^n	exponential	astronomically large
$n!$	factorial	larger still
n^n	super-factorial	larger still again

$\lg^* n$ counts how often you take $\lg$ before reaching 1: from 65536 to 2^{65536} it rises by one. Fibonacci grows like ϕ^n with $\phi \approx 1.618$ — why naive recursive Fibonacci is unusable.

Each row of Table 3.4 is $o(\cdot)$ of every row below it; Listing 3.3 checks $\lg^*$, Stirling and Fibonacci numerically. Book: Sections 3.8.8–3.8.9 and Table 3.4.

The Arithmetic Series

DEFINITION (arithmetic series). Equation (3.7):

$$\sum_{i=1}^{n} i = 1+2+3+\cdots+n = \frac{n(n+1)}{2} = \Theta(n^2). \tag{3.7}$$

Gauss's trick: write $S = 1+2+\cdots+n$ forward and backward and add column by column. Each of the n columns totals $n+1$, so $2S = n(n+1)$.

By induction: $S(1) = 1 = \frac{1\cdot 2}{2}$; and $S(n) = S(n-1)+n = \frac{(n-1)n}{2}+n = \frac{n(n+1)}{2}$.

It is $\Theta(n^2)$ ($\frac{n^2}{2} \le S \le n^2$), and it *is* Insertion Sort's worst case: $\sum_{j=2}^{n}(j-1) = \frac{(n-1)n}{2}$.

Book: Sections 3.9.1–3.9.2, Equation (3.7) and Figures 3.5–3.8.

Gauss's Pairing, Drawn

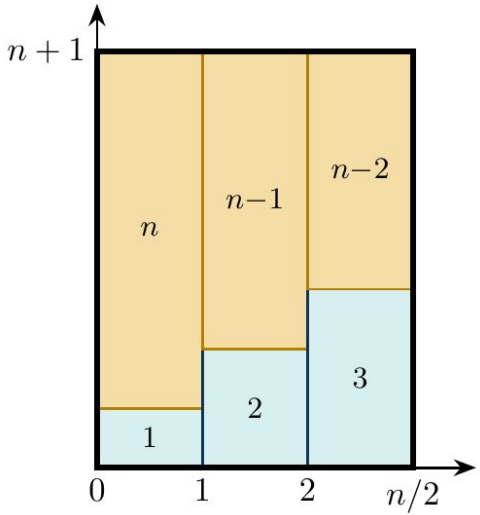


Figure 3.5 Gauss's pairing made geometric. Pairing the first bar with the last, the second with the second-last, and so on, repacks the staircase into a rectangle of width $n/2$ and height $n+1$, of area $n(n+1)/2$.

Figure 3.5 folds the staircase into a rectangle $n/2$ wide and $n+1$ tall: area $n(n+1)/2$. Book: Section 3.9.1 and Figure 3.6.

The Geometric Series

DEFINITION (geometric series). For real $a \neq 1$, Equation (3.8):

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{1 - a^{n+1}}{1 - a}. \tag{3.8}$$

Derivation: $S - aS = 1 - a^{n+1}$, because every middle term cancels. (For $a = 1$ the sum is $n+1$.)

$a < 1$**:** converges to $\frac{1}{1-a}$ — $\Theta(1)$, the **first** term dominates. $a > 1$**:** $\Theta(a^n)$, the **last** term dominates. $a = 1$**:** $\Theta(n)$, every term counts equally.

KEY INSIGHT. Recognizing which of the three regimes you are in is the whole skill of recursion-tree analysis — and it is exactly the three cases of the Master Theorem.

Book: Section 3.9.3 and Equation (3.8).

Harmonic, Telescoping, Integrals

Harmonic, Equation (3.9), with $\gamma \approx 0.5772$; grouping terms in doubling blocks, each at least $1/2$, gives $H_n \geq 1/2 \lg n$:

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \ln n + \gamma + O\left(\frac{1}{n}\right) = \Theta(\lg n). \tag{3.9}$$

Telescoping: $\sum_{i=1}^{n} (a_i - a_{i-1}) = a_n - a_0$, e.g. $\sum_{i=1}^{n} \frac{1}{i(i+1)} = 1 - \frac{1}{n+1}$.

Integral bound for increasing f, Equation (3.10) (reversed for decreasing f); it gives $\lg(n!) = \Theta(n \lg n)$ without Stirling:

$$\int_{m-1}^{n} f(x)\,dx \leq \sum_{i=m}^{n} f(i) \leq \int_{m}^{n+1} f(x)\,dx. \tag{3.10}$$

Book: Sections 3.9.4–3.9.6 and Equations (3.9)–(3.10).

The Summation Reference Card

Table 3.5 The summations used throughout this book, with closed forms and asymptotic classes.

Sum	Closed form	Asymptotics
$\sum_{i=1}^{n} 1$	n	$\Theta(n)$
$\sum_{i=1}^{n} i$	$\frac{n(n+1)}{2}$	$\Theta(n^2)$
$\sum_{i=1}^{n} i^2$	$\frac{n(n+1)(2n+1}{6}$	$\Theta(n^3)$
$\sum_{i=1}^{n} i^k$	—	$\Theta(n^{k+1})$ for $k \geq 0$
$\sum_{i=0}^{n} a^i, a \neq 1$	$\frac{1-a^{n+1}}{1-a}$	$\Theta(1)$ if $a<1$; $\Theta(n)$ if $a=1$; $\Theta(a^n)$ if $a>1$
$\sum_{i=0}^{\infty} a^i$ with $\lvert a\rvert<1$	$\frac{1}{1-a}$	$\Theta(1)$
$\sum_{i=1}^{n} \frac{1}{i}$	$\ln n+\gamma+O(1/n$	$\Theta(\lg n)$

Table 3.5 (Continued) *The summations used throughout this book, with closed forms and asymptotic classes.*

Sum	Closed form	Asymptotics
$\sum_{i=1}^{n} \lg i$	$\lg(n!)$	$\Theta(n \lg n)$
$\sum_{i=1}^{n} (a_i - a_{i-1})$	$a_n - a_0$	telescoping

Table 3.5 is the card to keep. Listing 3.4 checks four rows by brute force — Gauss's 5050 *appears, and after a million terms* H_n *is only* 14.39*. Book: Section 3.9 and Table 3.5.*

Recurrences

DEFINITION. A **recurrence** describes a function by its values on smaller inputs, plus **boundary conditions** for the smallest inputs.

Most of ours have the divide-and-conquer form Equation (3.11): $a \geq 1$ subproblems, each of size n/b with $b>1$, plus $f(n)$ for dividing and combining.

$$T(n) = aT(n/b) + f(n), \tag{3.11}$$

Merge Sort $a=b=2$; binary search $a=1, b=2$; Strassen $a=7, b=2$ — a and b are independent.

We may ignore **floors and ceilings**, omit **boundary conditions** ($T(n)=\Theta(1)$ for small n), and must track $\leq$ / $\geq$ / $=$ (O / Ω / Θ).

Merge Sort's recurrence as a two-case definition is Figure 3.10. Book: Sections 3.10.1–3.10.3 and Equation (3.11).

Three Methods

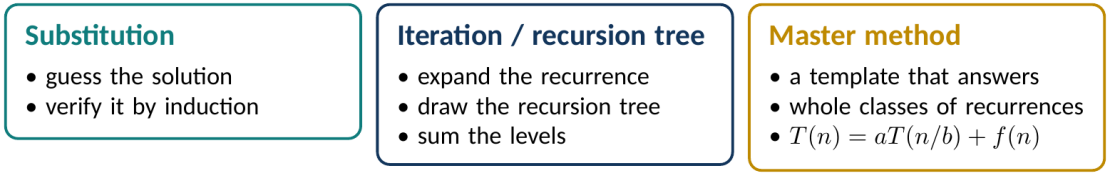


Figure 3.6 The three methods for solving recurrences: substitution (guess, then verify by induction), iteration or recursion tree (expand the recurrence and draw it), and the master method (a template that answers whole classes of recurrences at once).

*Figure 3.6: the **recursion tree** finds an answer, **substitution** proves it, the **master method** skips both when the form fits. Try the Master Theorem first; if it fails, draw a tree, then prove by substitution. Book: Section 3.10.4 and Figure 3.11.*

Substitution: a Cubic Upper Bound

Guess $T(n) \le c\,n^3$; **assume** it for all $k < n$, in particular $k = n/2$.

$$T(n) = 4\,T(n/2) + n \le 4\,c\,\frac{n^3}{8} + n = \frac{c\,n^3}{2} + n = c\,n^3 - \left(\frac{c\,n^3}{2} - n\right).$$

$\le c\,n^3$ provided $\frac{c\,n^3}{2} \ge n$, i.e. $c \ge 2/n^2$: $c \ge 2$ works for every $n \ge 1$.

Base: $T(1) = 1 \le c$. So $T(n) \le 2\,n^3$, and $T(n) = O(n^3)$. ∎

The move to learn is writing $c\,n^3 - (\text{slack})$: the goal on the left, everything to dispose of in one bracket. Book: Section 3.11.1.

The Failed Attempt, and the Fix

Guess $T(n) \le c\,n^2$: $4\,c\frac{n^2}{4} + n = c\,n^2 + n$, and $c\,n^2 + n \le c\,n^2$ needs $n \le 0$. Impossible for every c.

> **COMMON TRAP.** "$c\,n^2 + n$ is $O(n^2)$, done" is wrong: the induction must reproduce the hypothesis *exactly*, same c. Leftovers accumulate over the levels.

Strengthen: $T(n) \le c_1 n^2 - c_2 n$. Then $4\left(c_1\frac{n^2}{4} - c_2\frac{n}{2}\right) + n = c_1 n^2 - c_2 n - (c_2 n - n) \le c_1 n^2 - c_2 n$ when $c_2 \ge 1$.

> **KEY INSIGHT.** Left with (bound) $+$ (positive leftover)? **Subtract** a term of the leftover's order from the guess. A stronger hypothesis is also a stronger assumption.

The chapter's Listing 3.5 computes $T(n)$ exactly: $T(n) = 2n^2 - n$ at every power of two, so $c_1 = 2$, $c_2 = 1$ is best possible; Merge Sort's recurrence gives exactly $n \lg n + n$. Book: Sections 3.11.2–3.11.3, 3.11.5 and Figures 3.12–3.13.

Base Cases and Changing Variables

Move the base case. For $T(n)=2T(\lfloor n/2 \rfloor)+n$, $T(1)=1$, the guess $c\,n\lg n$ fails at $n=1$ ($\lg 1=0$). Prove the base at $n=2,3$ instead: $4\le 2c$ and $5\le 3c\lg 3$, both true for $c\ge 2$.

Change variables. $T(n)=2T(\sqrt{n})+\lg n$: put $m=\lg n$ and $S(m)=T(2^m)$ to get $S(m)=2S(m/2)+m=\Theta(m\lg m)$.

Back in n: $T(n)=\Theta(\lg n\cdot\lg\lg n)$.

Asymptotic bounds need hold only for $n\ge n_0$, which is what licenses moving the base. Book: Section 3.11.4.

Iteration: Unrolling a Recurrence

$$T(n) = n + 3T(n/4) = n + \frac{3n}{4} + 9T(n/16) = n + \frac{3n}{4} + \frac{9n}{16} + 27T(n/64) = \cdots$$

$$T(n) \le n \sum_{i \ge 0} (3/4)^i + \Theta(n^{\log_4 3}) = 4n + \Theta(n^{0.793}).$$

The leaves cost $n^{\log_4 3} = o(n)$, so $T(n) = O(n)$ — and $\Theta(n)$, since the root alone costs n.

KEY INSIGHT. Ratio below 1 ⇒ the series converges ⇒ the **root** dominates ⇒ $T(n) = \Theta(f(n))$.

Book: Section 3.12.1 and Figure 3.14.

A Tree Whose Levels Shrink

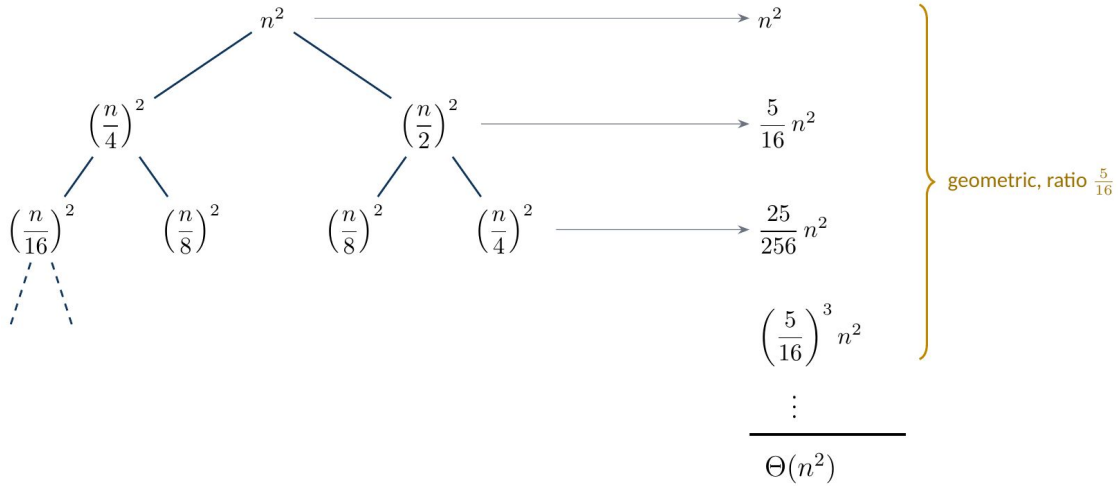


Figure 3.7 The completed recursion tree for $T(n)=T(n/4)+T(n/2)+n^2$. The level totals $n^2, \frac{5}{16}n^2, \frac{25}{256}n^2, \ldots$ form a geometric series of ratio $5/16<1$, summing to $\Theta(n^2)$.

Figure 3.7: for $T(n)=T(n/4)+T(n/2)+n^2$ each level costs $(1/4)^2+(1/2)^2=\frac{5}{16}$ of the one above. Book: Section 3.12.2 and Figure 3.16.

Summing That Tree

Level 0: n^2. Level 1: $\frac{n^2}{16}+\frac{n^2}{4}=\frac{5n^2}{16}$. Level 2: $\frac{n^2+4n^2+4n^2+16n^2}{256}=\left(\frac{5}{16}\right)^2 n^2$.

$T(n)\le n^2\sum_{i\ge 0}\left(\frac{5}{16}\right)^i=\frac{16}{11}n^2\approx 1.4545\,n^2$, and $T(n)\ge n^2$ from the root alone.

So $T(n)=\Theta(n^2)$: everything below the root adds only 45 %. Unequal subproblems — the Master Theorem cannot touch this; the tree can.

The chapter's Listing 3.6 sums seven levels numerically: $327680/1048576=5/16$ *exactly. Book: Section 3.12.2 and Figure 3.15.*

A Tree Whose Levels Are Equal

$$T(n) = T(n/3) + T(2n/3) + n$$

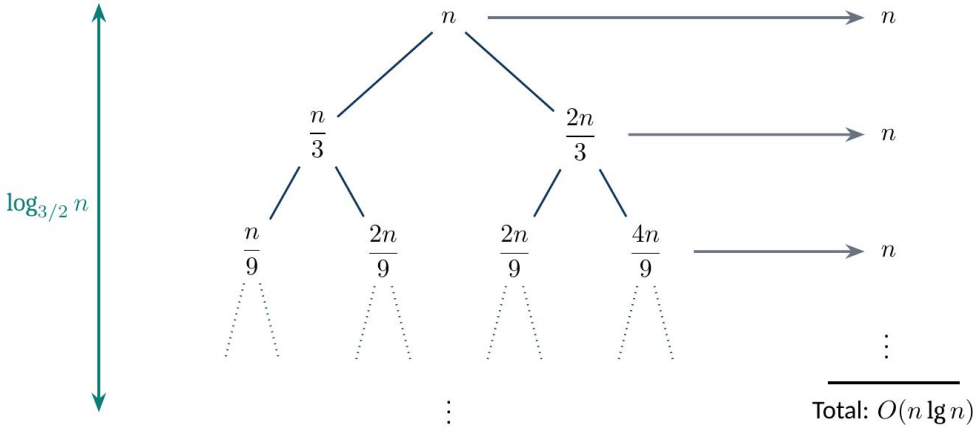


Figure 3.8 The recursion tree for $T(n) = T(n/3) + T(2n/3) + n$. Every level sums to exactly n, and the longest root-to-leaf path has $\log_{3/2} n$ levels, so the total is $O(n \lg n)$.

Figure 3.8: for $T(n) = T(n/3) + T(2n/3) + n$ every level sums to n ($\frac{n}{9} + \frac{2n}{9} + \frac{2n}{9} + \frac{4n}{9} = n$). The longest path has $\log_{3/2} n$ levels, the shortest $\log_3 n$, so $T(n) = \Theta(n \lg n)$ — about $1.71 n \lg n$ at most. Book: Section 3.12.3 and Figure 3.17.

The Leaf Count, and Three Shapes

For $aT(n/b)+f(n)$: level i has a^i nodes of size n/b^i; depth $\log_b n$; so the leaves number Equation (3.12) — the most important formula in the chapter:

$$a^{\log_b n} = n^{\log_b a}, \tag{3.12}$$

Adding level costs and leaves gives Equation (3.13):

$$\text{cost of level } i \quad = \quad a^i f\left(\frac{n}{b^i}\right), \qquad \text{and} \qquad T(n) \quad = \quad \underbrace{\sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)}_{\text{divide and combine, all levels}} + \underbrace{\Theta\left(n^{\log_b a}\right)}_{\text{all the leaves}}. \tag{3.13}$$

Table 3.6 The three shapes a recursion tree can take, and what each one implies. These correspond exactly to Cases 3, 2 and 1 of the Master Theorem.

Type	Level ratio	Dominated by	Result
Decreasing	ratio < 1	the root (top level)	$\Theta(f(n))$
Equal	ratio $= 1$	all levels equally	$\Theta(f(n)\log n)$
Increasing	ratio > 1	the leaves (bottom level)	$\Theta\left(n^{\log_b a}\right)$

Merge Sort: $n^{\log_2 2} = n$ *leaves; binary search:* $n^{\log_2 1} = 1$ *leaf. Table 3.6 is the Master Theorem in disguise. Book: Sections 3.12.4–3.12.5, Equations (3.12)–(3.13) and Table 3.6.*

The Master Theorem

THE MASTER THEOREM. Let $a \geq 1$ and $b > 1$ be constants, $f(n)$ asymptotically positive, and $T(n) = aT(n/b) + f(n)$ (with n/b read as $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$). **Case 1.** If $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. **Case 2.** If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$. **Case 3.** If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, **and** $af(n/b) \leq cf(n)$ for some $c < 1$ and all large n, then $T(n) = \Theta(f(n))$.

THE ONE IDEA. Compare the root cost $f(n)$ with the leaf cost $n^{\log_b a}$. Polynomially bigger wins; a tie pays for all $\lg n$ levels.

a is the branching factor, b the shrink factor, $f(n)$ the local work, $n^{\log_b a}$ the leaf count. Book: Sections 3.13.1–3.13.2.

Epsilon and Regularity

ε means "polynomially". Write f and $n^{\log_b a}$ as powers of n and subtract exponents. A factor of $n^{0.0001}$ counts; a factor of $\lg n$ does **not**, because $\lg n = o(n^{\varepsilon})$ for every $\varepsilon > 0$.

Regularity, Equation (3.14): the children's total work is at most a fixed fraction of the parent's, so level costs shrink geometrically:

$$a f\left(\frac{n}{b}\right) \le c f(n) \text{ for some constant } c < 1 \text{ and all large } n. \tag{3.14}$$

Automatic for polynomials: $f(n) = n^k$ with $k > \log_b a$ gives $c = a/b^k < 1$.

It can fail: $T(n) = T(n/2) + n(2 - \cos n)$. Where $\cos n \approx 1$ and $\cos(n/2) \approx -1$, the condition forces $c \ge 1.5$.

Book: Sections 3.13.3–3.13.4 and Equation (3.14).

The Summary Card, and Five Steps

Table 3.7 The three cases of the Master Theorem for $T(n) = aT(n/b) + f(n)$, with the intuition each one expresses.

Case	Condition on $f(n)$	Conclusion	Intuition
1	$f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$	$T(n) = \Theta(n^{\log_b a})$	leaves dominate
2	$f(n) = \Theta(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \lg n)$	every level costs the same
3	$f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, **and** $af(n/b) \le cf(n)$ with $c < 1$	$T(n) = \Theta(f(n))$	the root dominates

1. Extract a, b, $f(n)$; check $a \ge 1$, $b > 1$, f positive. **2.** Compute $n^{\log_b a}$. **3.** Compare exponents. **4.** Decide the case — or recognize a gap. **5.** In Case 3, check regularity.

Table 3.7 is the card to memorize. Book: Sections 3.13.5 and 3.13.7 and Table 3.7.

Where the Theorem Is Silent

Gap 1–2: $T(n)=2T(n/2)+n/\lg n$ — smaller than n, not polynomially. True answer $\Theta(n\lg\lg n)$.

Gap 2–3: $T(n)=2T(n/2)+n\lg n$. Not Case 2 ($\lg n\to\infty$), not Case 3 ($\lg n\geq c\,n^{\varepsilon}$ fails). Level i costs $n(\lg n-i)$, giving Equation (3.15):

$$T(n)=\sum_{i=0}^{\lg n-1} n(\lg n-i)=n\sum_{j=1}^{\lg n} j=n\cdot\frac{\lg n(\lg n+1)}{2}=\Theta(n\lg^2 n). \tag{3.15}$$

COMMON TRAP. "Case 3, so $\Theta(n\lg n)$" is wrong twice: the case does not apply, and the answer is $\Theta(n\lg^2 n)$.

Out of scope: f not positive ($4T(n/2)-n^2$); a not constant ($2^nT(n/2)$); unequal parts ($T(n/3)+T(2n/3)$); subtraction ($T(n-1)+n=\Theta(n^2)$ by telescoping).

Book: Section 3.13.6 and Equation (3.15).

Worked: Four Recurrences

$T(n/2)+1$: $n^{\log_2 1}=1=f$. Case 2: $\Theta(\lg n)$ — binary search.

$9T(n/3)+n$: $n^{\log_3 9}=n^2$; $1<2$, $\varepsilon=1$. Case 1: $\Theta(n^2)$ — the leaves swamp the root.

$2T(n/2)+\Theta(n)$: exact tie with n. Case 2: $\Theta(n\lg n)$ — Merge Sort.

$3T(n/4)+n\lg n$: $\log_4 3\approx 0.7925$; with $\varepsilon=0.2$, $n\lg n/n^{0.9925}\to\infty$. Regularity: $3\cdot\frac{n}{4}\lg\frac{n}{4}\le 3/4\,n\lg n$. Case 3: $\Theta(n\lg n)$.

Book: Sections 3.14.1–3.14.4 and Pseudocode 3.1.

Worked: Same a and b, Three Cases

$a=4$, $b=2$, $n^{\log_2 4}=n^2$: $+n$ Case 1, $\Theta(n^2)$; $+n^2$ Case 2, $\Theta(n^2 \lg n)$; $+n^3$ Case 3 with $c=1/2$, $\Theta(n^3)$.

$8T(n/2)+n^2$**:** Case 1, $\Theta(n^3)$ — divide-and-conquer matrix product, no better than three loops.

Strassen $7T(n/2)+n^2$**:** $\log_2 7 \approx 2.807$, Case 1: $\Theta(n^{\lg 7})$. One multiplication fewer cuts the exponent.

$T(n/2)+n$**:** Case 3, $c=1/2$: $\Theta(n)$ $(n+\frac{n}{2}+\cdots<2n)$. $2T(n/2)+n^2/\lg n$**:** regularity is a real calculation — ratio $\frac{\lg n}{2(\lg n-1)} \le \frac{2}{3}$ for $n \ge 16$ — giving $\Theta(n^2/\lg n)$.

Book: Sections 3.14.5–3.14.20.

All Twenty at a Glance

Table 3.8 All twenty worked problems of $3.14 at a glance, with the case each falls into.

Recurrence	$\log_b a$	$f(n)$	Case	$T(n)$
$T(n)=T(n/2)+1$	0	1	2	$\Theta(\lg n)$
$T(n)=9T(n/3)+n$	2	n	1	$\Theta(n^2)$
$T(n)=2T(n/2)+n$	1	n	2	$\Theta(n\lg n)$
$T(n)=3T(n/4)+n\lg n$	0.7925	$n\lg n$	3	$\Theta(n\lg n)$
$T(n)=4T(n/2)+n^3$	2	n^3	3	$\Theta(n^3)$
$T(n)=2T(n/2)+n\lg n$	1	$n\lg n$	gap	$\Theta(n\lg^2 n)$
$T(n)=4T(n/2)+n$	2	n	1	$\Theta(n^2)$
$T(n)=4T(n/2)+n^2$	2	n^2	2	$\Theta(n^2\lg n)$
$T(n)=8T(n/2)+n^2$	3	n^2	1	$\Theta(n^3)$
$T(n)=7T(n/2)+n^2$	2.807	n^2	1	$\Theta(n^{\lg 7})$
$T(n)=8T(n/2)+n^3$	3	n^3	2	$\Theta(n^3\lg n)$
$T(n)=2T(n/4)+\sqrt{n}$	0.5	$\sqrt{n}$	2	$\Theta(\sqrt{n}\lg n)$
$T(n)=3T(n/3)+n/2$	1	$n/2$	2	$\Theta(n\lg n)$

Table 3.8 (Continued) *All twenty worked problems of §3.14 at a glance, with the case each falls into.*

Recurrence	$\log_b a$	$f(n)$	Case	$T(n)$
$T(n)=16T(n/4)+n^2$	2	n^2	2	$\Theta(n^2 \lg n)$
$T(n)=T(n/2)+n$	0	n	3	$\Theta(n)$
$T(n)=2T(n/2)+n/\lg n$	1	$n/\lg n$	gap	$\Theta(n \lg \lg n)$
$T(n)=5T(n/2)+n^2$	2.322	n^2	1	$\Theta(n^{\lg 5})$
$T(n)=T(2n/3)+1$	0	1	2	$\Theta(\lg n)$
$T(n)=2T(n/2)+n^2/\lg n$	1	$n^2/\lg n$	3	$\Theta(n^2/\lg n)$
$T(n)=T(n/2)+n(2-cc$	0	oscillating	none	regularity fails

Table 3.8: the chapter's Master Theorem solver (Listing 3.7) reproduces every row by machine and recognizes both gaps as gaps. Book: Section 3.14.21 and Table 3.8.

Merge Sort: Blueprint and Algorithm

> **DEFINITION. Divide and conquer**: DIVIDE into smaller subproblems of the same kind; CONQUER them recursively (directly if small); COMBINE their solutions.

For sorting: **divide** at the midpoint ($\Theta(1)$ with indices); **conquer** each half (one element is sorted); **combine** by **merging** two sorted halves.

Pseudocode 3.1 Merge Sort on the slice $A[p..r]$, with MERGE described in words; Listing 3.2 gives the full code.

```
MERGE-SORT(A, p, r)
    if p < r then
        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)
    Take the smaller of the two front elements of
    A[p..q] and A[q+1..r], place it into the result. Repeat
    until both are empty. Copy the result back into A[p..r].
```

In Pseudocode 3.1, `if p < r` is the base case in disguise: one element or none means nothing to do. Book: Sections 3.15.1–3.15.2 and Pseudocode 3.2.

A Complete Trace

```
Level 0: [1, 5, 2, 4 | 6, 3, 2, 6]
Level 1: [1, 5 | 2, 4]   [6, 3 | 2, 6]
Level 2: [1|5] [2|4]     [6|3] [2|6]
Level 3: [1][5] [2][4]   [6][3] [2][6]      <- singletons: the base cases
Level 3 -> 2: [1]+[5]=[1,5]   [2]+[4]=[2,4]   [6]+[3]=[3,6]   [2]+[6]=[2,6]
Level 2 -> 1: [1,5]+[2,4]=[1,2,4,5]           [3,6]+[2,6]=[2,3,6,6]
Level 1 -> 0: [1,2,4,5]+[2,3,6,6] = [1, 2, 2, 3, 4, 5, 6, 6]
```

The last merge: $1 < 2$ take 1; tie at 2 — take the **left** one (stability); then $2, 3, 4, 5$; the left list is empty, copy $6, 6$.

Book: Section 3.15.3.

MERGE Is Correct

> **LOOP INVARIANT.** At the start of each iteration: (a) $A[p..k-1]$ holds the $k-p$ smallest elements of $L \cup R$, sorted; (b) $L[i]$ and $R[j]$ are the smallest elements of L and R not yet copied back.

Initialization: $k = p$, the prefix is empty; $i = j = 0$ point at the minima of sorted L and R.

Maintenance: if $L[i] \le R[j]$, $L[i]$ beats everything left in both lists, so appending it keeps (a); i advances, keeping (b). Symmetric otherwise.

Termination: one list is empty; the rest of the other is sorted and no smaller than what was placed; the clean-up loops append it. At most $r - p + 1$ iterations: **totally correct**, $\Theta(n)$. ∎

MERGE-SORT then follows by strong induction on the slice length. Book: Section 3.15.4.

The Recurrence, Three Ways

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2\,T(n/2) + \Theta(n) & \text{if } n > 1, \end{cases} \tag{3.16}$$

Tree: every level of Equation (3.16) costs $c\,n$, and there are $\lg n + 1$ levels: $c\,n \lg n + c\,n = \Theta(n \lg n)$.

Master: $a = b = 2$, $n^{\log_2 2} = n = f(n)$: Case 2, $\Theta(n \lg n)$.

Substitution: $2c\frac{n}{2}\lg\frac{n}{2} + n = c\,n\lg n - c\,n + n \le c\,n\lg n$ when $c \ge 1$.

Three methods, one answer — and the exact solution with $T(1) = 1$ is $n \lg n + n$. Book: Section 3.15.5 and Equation (3.16).

Merge Sort in C++

Listing 3.2 Merge Sort in C++, with a printed trace and a measured comparison count.

```cpp
// mergesort.cpp - Merge Sort in full: the merge step, the recursive sort,
// a printed trace of the divide and combine phases, and a count of the
// comparisons actually performed against the n lg n prediction.
// Compile: g++ -std=c++17 -O2 -o mergesort mergesort.cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <random>
#include <cmath>
using namespace std;

long long comparisons = 0;   // every key-to-key comparison the merge performs

static string show(const vector<int>& a, int p, int r) {
    string s = "[";
    for (int i = p; i <= r; ++i) { s += to_string(a[i]); if (i < r) s += ", "; }
    return s + "]";
}

// MERGE: A[p..q] and A[q+1..r] are each sorted; afterward A[p..r] is sorted.
// Loop invariant: at the top of each iteration, A[p..k-1] holds the k-p
// smallest elements of L and R, in sorted order, and L[i] and R[j] are the
// smallest elements of their arrays not yet copied back.
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);
    size_t i = 0, j = 0;
```

```
    int k = p;
    while (i < L.size() && j < R.size()) {
        ++comparisons;
        if (L[i] <= R[j]) A[k++] = L[i++];   // <= keeps the sort stable
        else              A[k++] = R[j++];
    }
    while (i < L.size()) A[k++] = L[i++];
    while (j < R.size()) A[k++] = R[j++];
}
```

Listing 3.2 (Continued) *Merge Sort in C++, with a printed trace and a measured comparison count.*

```
void mergeSort(vector<int>& A, int p, int r, int depth, bool trace) {
    if (p >= r) return;                          // one element: already sorted
    int q = p + (r - p) / 2;                     // floor((p+r)/2) without overflow
    if (trace) cout << string(2 * depth, ' ') << "divide " << show(A, p, r)
                    << " -> " << show(A, p, q) << " " << show(A, q + 1, r) << "\n";
    mergeSort(A, p, q, depth + 1, trace);
    mergeSort(A, q + 1, r, depth + 1, trace);
    merge(A, p, q, r);
    if (trace) cout << string(2 * depth, ' ') << "merge  -> " << show(A, p, r) << "\n";
}

int main() {
    vector<int> A = {1, 5, 2, 4, 6, 3, 2, 6};
    cout << "input  " << show(A, 0, (int)A.size() - 1) << "\n\n";
    mergeSort(A, 0, (int)A.size() - 1, 0, true);
    cout << "\nsorted " << show(A, 0, (int)A.size() - 1) << "\n";
    cout << "comparisons for n = 8: " << comparisons << "\n\n";

    cout << "n          comparisons      n lg n    ratio\n";
    cout << "-------------------------------------------\n";
    mt19937 rng(12345);
    for (int n : {16, 64, 256, 1024, 4096, 65536}) {
        vector<int> B(n);
        for (int& x : B) x = (int)(rng() % 1000000);
        comparisons = 0;
        mergeSort(B, 0, n - 1, 0, false);
        double nlgn = n * log2((double)n);
        cout << left << setw(11) << n << setw(17) << comparisons
```

```
                << setw(12) << fixed << setprecision(0) << nlgn
                << setprecision(3) << comparisons / nlgn << "\n";
    }
    return 0;
}
```

Listing 3.2 merges with `<=` to stay stable and computes `q = p + (r - p) / 2` to avoid overflow. Book: Section 3.15.6 and Listing 3.8.

What the Program Prints

```
input  [1, 5, 2, 4, 6, 3, 2, 6]

divide [1, 5, 2, 4, 6, 3, 2, 6] -> [1, 5, 2, 4] [6, 3, 2, 6]
  divide [1, 5, 2, 4] -> [1, 5] [2, 4]
    divide [1, 5] -> [1] [5]
    merge  -> [1, 5]
    divide [2, 4] -> [2] [4]
    merge  -> [2, 4]
  merge  -> [1, 2, 4, 5]
  divide [6, 3, 2, 6] -> [6, 3] [2, 6]
    divide [6, 3] -> [6] [3]
    merge  -> [3, 6]
    divide [2, 6] -> [2] [6]
    merge  -> [2, 6]
  merge  -> [2, 3, 6, 6]
merge  -> [1, 2, 2, 3, 4, 5, 6, 6]

sorted [1, 2, 2, 3, 4, 5, 6, 6]
comparisons for n = 8: 16

n          comparisons      n lg n    ratio
-------------------------------------------
16         47               64          0.734
64         307              384         0.799
256        1725             2048        0.842
1024       8962             10240       0.875
4096       43970            49152       0.895
65536      965956           1048576     0.921
```

The ratio to $n \lg n$ *stays between* 0.73 *and* 0.93 *as* n *grows four thousandfold —* $\Theta(n \lg n)$ *measured. Merge Sort is* $\Theta(n \lg n)$ *in every case and stable, but needs* $\Theta(n)$ *extra memory. Book: Sections 3.15.6–3.15.7.*

Profitina in Practice

A scan is a recurrence: split a site's crawl into a shards of size n/b and the Master case says whether more shards help (Case 1) or parsing is the bottleneck (Case 3).

Sorting must be $n \lg n$, and stable: trends, deduplication and rankings sort; tied scores must keep a reproducible order — Merge Sort's `<=`.

Profitina Laundry: sort twelve baskets by pickup time and count **inversions** in the same merge — each time the right half wins, add $|L| - i$. Result: **32** out-of-order pairs of a possible 66, in $\Theta(n \lg n)$.

The app advises; the brand owner decides; personal data falls under UU PDP 27/2022. The full inversion-counting program is Listing 3.9. Book: Section 3.16.

Common Mistakes

O **for** Θ, and reading $=$ backward. State the tightest bound; read $=$ as "is a member of".

A positive leftover in substitution, and **a forgotten base case**. Reproduce the hypothesis exactly; strengthen by subtracting; move the base up.

"Larger" for "polynomially larger", and **skipping regularity**. Compare exponents; a log factor means a gap.

The Master Theorem on the wrong form — unequal parts, $T(n-1)$, non-constant a.

$\lg^2 n$ **for** $\lg\lg n$ (256 against 4 at $n=65536$); **worrying about log bases inside** Θ; **assuming comparability**; **trusting asymptotics on tiny inputs**.

Book: Section 3.17.

Summary — Five Things to Remember

1. Total correctness = partial correctness + termination; invariants are induction on loops.

2. O, Ω, Θ, o, ω are claims about constants you can name: exhibit them to prove, defeat every choice to disprove.

3. Arithmetic $\Theta(n^2)$; geometric dominated by its first term, its last, or all equally; harmonic $\Theta(\lg n)$.

4. Tree to guess, substitution to prove, Master Theorem when the form fits: compare $f(n)$ with $n^{\log_b a}$, and know the gaps.

5. Merge Sort: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$ in every case; correct by invariant; stable; not in place.

Tables 3.9 and 3.10 of the book put the whole chapter, and every recurrence so far, on one page each. Book: Section 3.18.

Next Time

Chapter 4 — Evaluation and Discussion I, on Chapters 1–3.

Two SPOJ problems, LDCSUMAB and TOHU, taken from the statement to a proved formula and a program; then twelve worked problems on invariants, counting, growth and recurrences.

Before we meet: Exercises 3.1, 3.4, 3.8, 3.13 and 3.16 — prove, rank, apply the theorem, spot the non-gap, strengthen a guess.

Then try both SPOJ problems yourself before reading their solutions.

Book: Sections 3.19–3.20, then Chapter 4.

Design and Analysis of Algorithms (EF234405)

Lecture 4 — Evaluation and Discussion I: Foundations, Analysis, and First Problems

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*, Chapter 4

Lecturer: Irfan Subakti — `yifana@gmail.com`

Department of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya

No new theory today: fourteen problems on Chapters 1–3, each solved from nothing with every number written down. Book: Chapter 4.

Agenda

1. A **seven-step method** for reading any algorithms problem.

2. LDCSUMAB (SPOJ): unranking the sums $2^m + 2^n$ — a formula, a proof, three programs.

3. TOHU (SPOJ): a hidden sequence recovered by differencing and summed by telescoping.

4. Worked Problems **A–L**: invariants and traces; counting and growth; recurrences; closed forms and a hidden Fibonacci; brute force against sort-and-search, timed.

5. The fourteen problems compared, Profitina, common mistakes, summary.

Before each solution, cover it and try for ten minutes; the check yourself variants in the book test that you really understood. Book: Sections 4.1–4.13.

Where We Stand

Chapter 1: algorithms, correctness and efficiency, five paradigms, the five phases *Understand, Abstract, Design, Analyze, Implement*.

Chapter 2: the RAM model; loop invariants (initialization, maintenance, termination); Insertion Sort in three cases; linear and binary search.

Chapter 3: O, Ω, Θ, o, ω with explicit constants; standard functions; summations; substitution, recursion trees and the Master Theorem.

> **HOW TO USE THIS CHAPTER.** Read the statement, cover the solution, try for ten minutes. A solution you struggled with is understood in a way a solution you only read never is.

Book: Section 4.1.

The Seven Steps: Understand

The steps are the five phases made concrete: Steps 1–2 *Understand*, 3–5 *Abstract*, 6 *Design* plus correctness, 7 *Implement* after the cost check.

Step 1 — copy the formats verbatim: input format, output format (lines, precision, wording), every constraint, the sample input and output character for character.

Many *Wrong Answer* verdicts are format errors: TOHU wants exactly eleven digits after the point; print ten and a perfect algorithm fails.

Step 2 — name the given and the wanted, each with a symbol. LDCSUMAB: *given* $k \le 10^9$, up to $T \le 10$ times; *wanted* $m<n$ with 2^m+2^n the k-th smallest such number.

Until you have symbols you cannot do algebra, and algebra solves most of this chapter. Book: Sections 4.2–4.2.2.

Step 3 — Constraints Become a Time Budget

A judge does about 10^8–10^9 simple operations per second, and limits are one to three seconds: **your program may perform about 10^8 operations in total.**

Table 4.1 The complexity budget. Read the bound on n, read off the affordable complexity, and reject every idea that exceeds it before you write a line of code.

Bound on n	Affordable per test case	Typical shape
$n \le 10$	$O(n!), O(2^n \cdot n)$	brute force, permutations
$n \le 20$	$O(2^n)$	subset enumeration, bitmask DP
$n \le 500$	$O(n^3)$	Floyd–Warshall, matrix chain
$n \le 5000$	$O(n^2)$	pairwise DP, simple DP tables
$n \le 10^6$	$O(n)$ or $O(n \log n)$	one pass, sorting, sieve, stack
$n \le 10^{12}$	$O(\sqrt{n})$	divisor blocks, trial division
$n \le 10^{18}$	$O(\log n)$	binary search, fast exponentiation
n unbounded	$O(1)$	a closed form

When a statement names its complexity — TOHU says $O(1)$ — it is telling you the shape of the answer: a closed form.

Table 4.1 is the RAM model turned into a planning tool: reject every idea that exceeds the budget before writing a line. Book: Section 4.2.3 and Table 4.1.

Steps 4 to 7

Step 4 — compute the first five cases by hand. TOHU's terms $2/3, 1/24, 1/60, 1/120$; LDCSUMAB's $n = 1, 2, 2, 3, 3, 3, 4, 4, 4, 4$.

Step 5 — name the shape: closed form from a sum; inverting a counting formula; a loop with an invariant; counting steps with sums; a recurrence; brute force, then better.

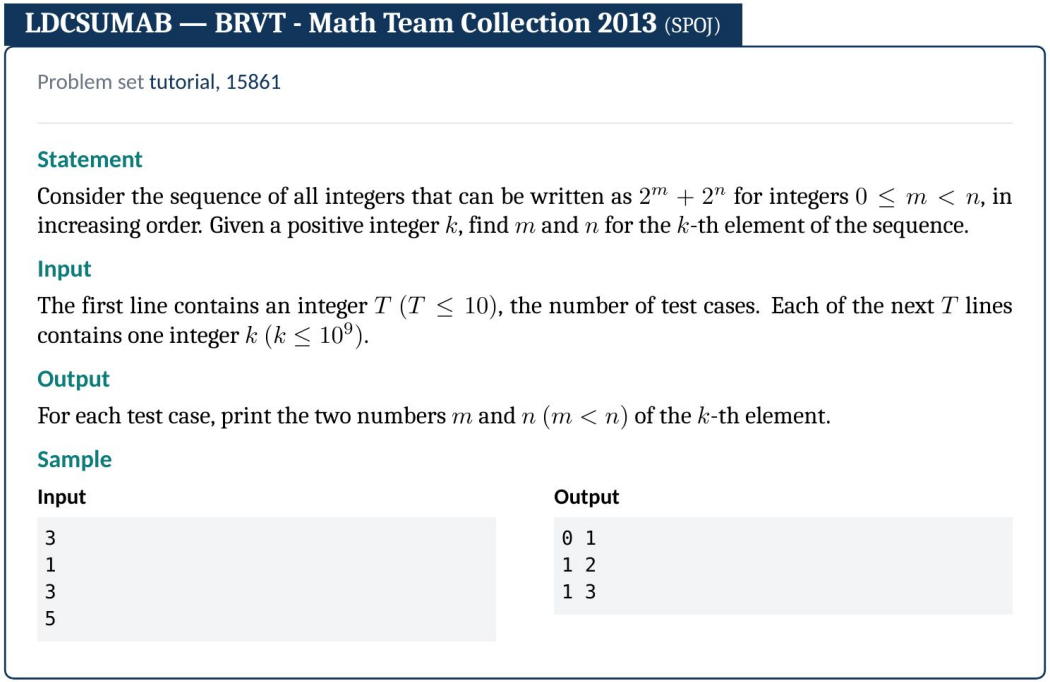
Step 6 — write the invariant, recurrence or formula, and prove it. A pattern read off a table is a conjecture until then.

Step 7 — only now, the program. Check overflow (`int` $\approx 2.1 \times 10^9$, `long long` $\approx 9.22 \times 10^{18}$), precision (`double` 15–16 digits, `float` 7) and input speed (`scanf`/`printf`).

> **THE TABLE IS THE ALGORITHM.** Most solutions here are visible in a hand table of five to ten cases. No pattern? Then look for invariants, orderings or exchange arguments.

Book: Sections 4.2.4–4.2.7.

Problem 1 — LDCSUMAB



LDCSUMAB — BRVT - Math Team Collection 2013 (SPOJ)

Problem set tutorial, 15861

Statement

Consider the sequence of all integers that can be written as $2^m + 2^n$ for integers $0 \leq m < n$, in increasing order. Given a positive integer k, find m and n for the k-th element of the sequence.

Input

The first line contains an integer T ($T \leq 10$), the number of test cases. Each of the next T lines contains one integer k ($k \leq 10^9$).

Output

For each test case, print the two numbers m and n ($m < n$) of the k-th element.

Sample

Input	Output
3 1 3 5	0 1 1 2 1 3

Figure 4.1 SPOJ problem set (tutorial) 15861, *LDCSUMAB — BRVT - Math Team Collection 2013*, restated from the judge: statement, input and output format, and the sample test.

Figure 4.1: a SPOJ tutorial problem that uses only three ideas — the arithmetic series, the ceiling function, and binary search. Book: Section 4.3 and Figure 4.1.

LDCSUMAB — the Problem and Its Budget

> **PROBLEM.** List all integers 2^m+2^n with $0 \le m < n$ in increasing order: $3, 5, 6, 9, 10, 12, 17, \ldots$ Given a position k ($1 \le k \le 10^9$, at most 10 queries), print `m n` for the k-th number. Sample `3 / 1 / 3 / 5` gives `0 1 / 1 2 / 1 3`.

Order matters: m first, then n — reversed is *Wrong Answer* on line one.

Price the ideas. Build the list: $\Theta(k)$, 10^{10} for ten queries — rejected. Walk the blocks: about $\sqrt{2k} \approx 44,721$ steps — fits. Binary search: about 16 steps. Closed form: $O(1)$.

The budget says which ideas are forbidden, not which is best; we derive the formula because it teaches the most. Book: Sections 4.3.1–4.3.2.

LDCSUMAB — the First Ten by Hand

Table 4.2 The first ten numbers of the form 2^m+2^n, computed by hand. The column n reads one 1, two 2s, three 3s, four 4s; inside each block m counts $0, 1, 2, \ldots$.

k	Value	m	n	Position inside block n
1	3	0	1	1st of 1
2	5	0	2	1st of 2
3	6	1	2	2nd of 2
4	9	0	3	1st of 3
5	10	1	3	2nd of 3
6	12	2	3	3rd of 3
7	17	0	4	1st of 4
8	18	1	4	2nd of 4
9	20	2	4	3rd of 4
10	24	3	4	4th of 4

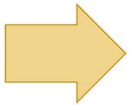
Candidates with $n \le 4$ are ten; the smallest with $n=5$ is 33, larger than all of them — so these are exactly the first ten.

Read Table 4.2 by columns: block n holds n entries; inside it $m = 0, 1, \ldots, n-1$; blocks end at $1, 3, 6, 10$ — the triangular numbers.

Book: Section 4.3.3 and Table 4.2.

The Same Observation, Drawn

k	m	n	$2^m + 2^n$
1	0	1	3
2	0	2	5
3	1	2	6
4	0	3	9
5	1	3	10
6	2	3	12
7	0	4	17
8	1	4	18
9	2	4	20
10	3	4	24



n	k
1	**1**
2	**2**
2	**3**
3	**4**
3	**5**
3	**6**
4	**7**
4	**8**
4	**9**
4	**10**

$$k = \frac{n(n+1)}{2}$$

$$n^2 + n - 2k = 0$$

$$n = \frac{-1 + \sqrt{1 + 8k}}{2}$$

Figure 4.2 The first ten terms of the LDCSUMAB sequence (left), collapsed to the map from k to n (middle), and the inversion $k = \frac{n(n+1)}{2}$, $n^2 + n - 2k = 0$, $n = \frac{-1 + \sqrt{1+8k}}{2}$ (right).

Figure 4.2: the list, the map from k to n, and the inversion we derive next. Book: Section 4.3.3 and Figure 4.2.

The Order Is Proved, the Blocks Are Counted

Claim 1 — blocks never interleave. The largest value of block n is $2^{n-1}+2^n=3\cdot 2^{n-1}$; the smallest of block $n+1$ is $1+4\cdot 2^{n-1}$, which is larger.

Claim 2 — inside a block, values grow with m. So the list is the pairs in lexicographic order, n first, then m.

Blocks 1 to n hold, by the arithmetic series of Section 3.9.1, Equation (4.1) — the triangular number T_n, with $T_0=0$ and $T_n=T_{n-1}+n$:

$$1+2+3+\cdots+n=\frac{n(n+1)}{2} \tag{4.1}$$

Position k lies in block n exactly when $T_{n-1}<k\le T_n$: **n is the smallest integer with $T_n\ge k$.**

A table of ten rows is evidence; the two claims make it hold forever. This family is ranking and unranking. Book: Section 4.3.4 and Equation (4.1).

Solving for n and m

$\frac{n(n+1)}{2} = k \Rightarrow n^2 + n - 2k = 0 \Rightarrow n = \frac{-1 \pm \sqrt{1+8k}}{2}$; keep the plus sign. The smallest integer at or above that root is Equation (4.2):

$$n = \lceil \frac{-1+\sqrt{1+8k}}{2} \rceil \tag{4.2}$$

Ceiling, not floor: at $k=5$ the root is about 2.70 and position 5 is in block 3; at a triangular k the root is already an integer. Inside block n, position k carries $m = k - T_{n-1} - 1$; with $r = T_n$ that is Equation (4.3):

$$m = k - (r - n) - 1 = n + k - (r+1) \tag{4.3}$$

Book: Section 4.3.4 and Equations (4.2)–(4.3).

Working the Cases by Hand

$k=1$: $\sqrt{9}=3$, $n=1$, $r=1$, $m=1+1-2=0$ → `0 1`. $k=3$: $\sqrt{25}=5$, $n=2$, $r=3$, $m=1$ → `1 2`.

$k=5$: $\sqrt{41}\approx 6.4031$, root ≈ 2.7016, $n=3$, $r=6$, $m=3+5-7=1$ → `1 3`, the value 10.

$k=6$ **(boundary):** $\sqrt{49}=7$, $n=3$, $m=2$ → `2 3`. $k=7$ **(new block):** root ≈ 3.2749, $n=4$, $r=10$, $m=0$ → `0 4`.

$k=10^9$: $\sqrt{8,000,000,001}\approx 89,442.719$, so $n=44,721$; check $T_{44720}=999,961,560<10^9\le T_{44721}=1,000,006,281$; $m=38,439$ → `38439 44721`.

Book: Section 4.3.4.

The Formula Is Correct for Every k

THEOREM. For every integer $k \geq 1$, let n be the smallest integer with $T_n \geq k$ and $m = k - T_{n-1} - 1$. Then $0 \leq m < n$, and $2^m + 2^n$ is the k-th smallest integer of the form $2^a + 2^b$ with $0 \leq a < b$.

By Claims 1–2 the pairs with exponent at most $n-1$ fill the first T_{n-1} positions, those with exponent at most n the first T_n. By the choice of n, $T_{n-1} < k \leq T_n$: the exponent at position k is exactly n, and inside the block it carries $m = k - T_{n-1} - 1$. $T_{n-1} < k$ gives $m \geq 0$; $k \leq T_n = T_{n-1} + n$ gives $m \leq n - 1$. ∎

Book: Section 4.3.5.

Making the Square Root Safe

Pseudocode 4.1 Answering one LDCSUMAB query: a floating-point first guess for the block, two correcting loops, and the position inside the block.

```
LDCSUMAB-QUERY(k)
1  n = floor((-1 + sqrt(1 + 8k)) / 2)     // first guess, from the quadratic formula
2  while n(n+1)/2 < k                      // guess too small: move up
3      n = n + 1
4  while n >= 1 and (n-1)n/2 >= k          // guess too large: move down
5      n = n - 1
6  r = n(n+1)/2                            // last position of block n
7  m = n + k - (r + 1)
8  return (m, n)
```

After lines 2–3: $T_n \geq k$ (the loop stops only when its test fails, and T_n grows without bound).

Lines 4–5 keep the invariant $T_n \geq k$**:** the body runs only when $T_{n-1} \geq k$. At exit $T_{n-1} < k \leq T_n$ ($n = 0$ is impossible, since $T_0 = 0 < k$).

Lines 6–7 compute Equation (4.3), which the Theorem proves correct.

Pseudocode 4.1 uses the floating-point root only as a first guess; two small loops pin the exact integer. Book: Section 4.3.5 and Pseudocode 4.1.

LDCSUMAB — Cost and Overflow

At a boundary $k = T_n$, $1 + 8k = (2n+1)^2$ — an exact odd square. A `double` holds integers up to 2^{53} exactly and `sqrt` is correctly rounded, so for $k \le 10^9$ the first guess is already exact and **neither loop runs**.

Why keep them? If the limit became $k \le 10^{18}$, $1 + 8k$ would pass 2^{53} and the guess could be off by one; the loops remove that dependence at no cost.

Cost: $O(1)$ per query, $\Theta(T)$ in total, $O(1)$ space. **Overflow:** $n(n+1) \approx 2.0 \times 10^9$ is a whisker below the `int` limit — use `long long`.

The chapter also shows a compact solution with `ceil` beside its table (Figure 4.3). Book: Section 4.3.6.

LDCSUMAB — the Program

Listing 4.1 LDCSUMAB — unranking the sequence 2^m+2^n, in $O(1)$ per query after the square root.

```
#include <cstdio>
#include <cmath>
using namespace std;
typedef long long LL;

int main() {
    int T;
    if (scanf("%d", &T) != 1) return 0;
    while (T--) {
        LL k;
        if (scanf("%lld", &k) != 1) break;
        // smallest n with n(n+1)/2 >= k, found by the quadratic formula then corrected
        LL n = (LL)floor((-1.0 + sqrt(1.0 + 8.0 * (double)k)) / 2.0);
        while (n * (n + 1) / 2 < k) ++n;          // correct upward  (guards sqrt error)
        while (n >= 1 && (n - 1) * n / 2 >= k) --n;  // correct downward
        LL r = n * (n + 1) / 2;                   // how many pairs use an exponent <= n
        LL m = n + k - (r + 1);
        printf("%lld %lld\n", m, n);
    }
    return 0;
}
```

```
0 1
1 2
1 3
2 3
0 4
998 999
0 1000
38439 44721
```

Listing 4.1 on the sample, then on $6, 7, 499500, 499501, 10^9$*:* $499, 500 = T_{999}$ *ends block* 999*; the next position opens block* 1000*. Compile with* `g++ -std=c++17 -O2 -Wall`*, run with* `./ldcsumab < sample.txt`*. Book: Section 4.3.7 and Listing 4.1.*

A Second Road: Binary Search on the Block

"Is $T_n \geq k$?" is *no* for small n and *yes* from some point on: a monotone question, answered by binary search. $T_{50000} = 1,250,025,000 \geq 10^9$, so search $[1, 50000]$.

Invariant: the answer lies in $[lo, hi]$. If $T_{mid} \geq k$ set $hi = mid$, else $lo = mid + 1$; stop at $lo = hi$.

> **BY HAND — $k = 5$ in $[1, 8]$.** $mid = 4$: $T_4 = 10 \geq 5$, $hi = 4$. $mid = 2$: $T_2 = 3 < 5$, $lo = 3$. $mid = 3$: $T_3 = 6 \geq 5$, $hi = 3$. Now $lo = hi = 3$, and $m = 5 - T_2 - 1 = 1 \rightarrow$ 1 3.

No floating point at all, and a correctness argument of one invariant. Book: Section 4.3.8.

The Binary-Search Program

Listing 4.2 LDCSUMAB by binary search on the block number, $\Theta(\lg 50000) \approx 16$ steps per query, with no floating point at all.

```cpp
#include <cstdio>
using namespace std;
typedef long long LL;

// Smallest n >= 1 with n(n+1)/2 >= k, by binary search on the answer.
// Invariant: the answer lies in [lo, hi].
LL blockOf(LL k) {
    LL lo = 1, hi = 50000;                  // 50000*50001/2 > 10^9, so hi is big enough
    while (lo < hi) {
        LL mid = lo + (hi - lo) / 2;        // never overflows, unlike (lo + hi) / 2
        if (mid * (mid + 1) / 2 >= k) hi = mid;   // mid is large enough: answer <= mid
        else lo = mid + 1;                         // mid is too small:    answer >  mid
    }
    return lo;
}

int main() {
    int T;
    if (scanf("%d", &T) != 1) return 0;
    while (T--) {
        LL k;
        if (scanf("%lld", &k) != 1) break;
        LL n = blockOf(k);
        LL m = k - (n - 1) * n / 2 - 1;     // position inside block n, counted from 0
        printf("%lld %lld\n", m, n);
    }
    return 0;
}
```

Listing 4.2 prints the same lines as Listing 4.1 on every input tried; it costs about 16 *iterations per query. Note the overflow-safe midpoint of Section 2.14.6. Book: Section 4.3.8 and Listing 4.2.*

Checking Against Brute Force

Listing 4.3 A brute-force cross-check for LDCSUMAB: the list built and sorted explicitly, compared with the formula at all 820 positions.

```cpp
#include <cstdio>
#include <cmath>
#include <vector>
#include <tuple>
#include <algorithm>
using namespace std;
typedef long long LL;

LL formulaN(LL k) {                          // the method of the closed form
    LL n = (LL)floor((-1.0 + sqrt(1.0 + 8.0 * (double)k)) / 2.0);
    while (n * (n + 1) / 2 < k) ++n;
    while (n >= 1 && (n - 1) * n / 2 >= k) --n;
    return n;
}

int main() {
    // Brute force: build every value 2^m + 2^n with 0 <= m < n <= 40 and sort by value.
    vector<tuple<LL, int, int>> seq;
    for (int n = 1; n <= 40; ++n)
        for (int m = 0; m < n; ++m)
            seq.emplace_back((1LL << m) + (1LL << n), m, n);
    sort(seq.begin(), seq.end());
    printf(" k  value  m  n\n");
    for (int k = 1; k <= 10; ++k)
        printf("%2d  %5lld  %d  %d\n", k, get<0>(seq[k - 1]), get<1>(seq[k - 1]), get<2>(seq[k - 1]));
    int bad = 0;
    for (LL k = 1; k <= 820; ++k) {          // 820 = 40*41/2 terms, all with n <= 40
        LL n = formulaN(k), m = n + k - (n * (n + 1) / 2 + 1);
```

```
        if (m != get<1>(seq[k - 1]) || n != get<2>(seq[k - 1])) ++bad;
    }
    printf("terms compared: 820, mismatches: %d\n", bad);
    return 0;
}
```

```
 k  value  m  n
 1      3  0  1
 2      5  0  2
 3      6  1  2
 4      9  0  3
 5     10  1  3
 6     12  2  3
 7     17  0  4
 8     18  1  4
 9     20  2  4
10     24  3  4
terms compared: 820, mismatches: 0
```

Listing 4.3 builds all $820 = T_{40}$ *values with* $n \le 40$*, sorts them, and compares with the formula: no mismatches, and the first ten rows are Table 4.2. Book: Section 4.3.9 and Listing 4.3.*

Problem 2 — TOHU: the Problem and Its Budget

A sequence $a_1, a_2, \ldots$ is known only through its weighted sums, Equation (4.4):

$$a_1 + 2\,a_2 + 3\,a_3 + \cdots + k\,a_k = \frac{k+1}{k+2} \text{ for every } k \geq 1 \tag{4.4}$$

> **PROBLEM.** For each of up to $20,000$ queries n, print $S_n = a_1 + \cdots + a_n$ with eleven digits after the decimal point. Sample `2 / 2 / 5` gives `0.70833333333 / 0.73809523810`. The statement notes: *time complexity* $O(1)$.

Two unknowns: the hidden terms a_k, then their plain sum. **Budget:** summing n terms per query is $\Theta(T\,n)$, up to 2×10^{13} — so there must be a closed form. Eleven digits rules out `float`.

TOHU exercises the other great summation tool of Chapter 3: the telescoping sum. Book: Sections 4.4.1–4.4.2 and Equation (4.4).

TOHU — the First Five by Hand

With $W_k = \frac{k+1}{k+2}$ and $W_0 = 0$, one new term is added each step: $k\,a_k = W_k - W_{k-1}$.

Table 4.3 The first five terms of the TOHU sequence, recovered by hand from the weighted sums. From $k = 2$ on, a_k is the reciprocal of three consecutive integers; a_1 breaks the pattern.

k	$W_k = (k+1)/(k+2)$	$k\,a_k = W_k - W_{k-1}$	a_k	a_k Factored	S_k	S_k As a decimal
1	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	—	$\frac{2}{3}$	0.66666666667
2	$\frac{3}{4}$	$\frac{1}{12}$	$\frac{1}{24}$	$\frac{1}{2\cdot 3\cdot 4}$	$\frac{17}{24}$	0.70833333333
3	$\frac{4}{5}$	$\frac{1}{20}$	$\frac{1}{60}$	$\frac{1}{3\cdot 4\cdot 5}$	$\frac{29}{40}$	0.72500000000
4	$\frac{5}{6}$	$\frac{1}{30}$	$\frac{1}{120}$	$\frac{1}{4\cdot 5\cdot 6}$	$\frac{11}{15}$	0.73333333333
5	$\frac{6}{7}$	$\frac{1}{42}$	$\frac{1}{210}$	$\frac{1}{5\cdot 6\cdot 7}$	$\frac{31}{42}$	0.73809523810

The S_2 and S_5 rows are the sample outputs. From $k=2$, $a_k=\frac{1}{k(k+1)(k+2)}$ — but $a_1=\frac{2}{3}$, not $\frac{1}{6}$. **The table also shows where the pattern breaks.**

Table 4.3: every subtraction is done over a common denominator, e.g. $2\,a_2=\frac{9}{12}-\frac{8}{12}$. *Book: Section 4.4.3 and Table 4.3.*

Recovering the Terms by Differencing

Write Equation (4.4) at k and at $k-1$ (for $k \geq 2$), one above the other, and subtract: everything cancels except $k\,a_k = \frac{k+1}{k+2} - \frac{k}{k+1}$.

Over $(k+1)(k+2)$: $\frac{(k^2+2k+1)-(k^2+2k)}{(k+1)(k+2)} = \frac{1}{(k+1)(k+2)}$. Divide by k to get Equation (4.5):

$$a_k = \frac{1}{k(k+1)(k+2)} \text{ for } k \geq 2, a_1 = \frac{2}{3} \tag{4.5}$$

Why $k=1$ differs: the formula for the previous weighted sum would claim $W_0 = 1/2$, but the empty sum is 0. So the pattern's $1/6$ is short by $1/2$: $1/6 + 1/2 = 2/3$.

Book: Section 4.4.4 and Equation (4.5).

Summing by Telescoping

$\frac{1}{k(k+1)} - \frac{1}{(k+1)(k+2)} = \frac{2}{k(k+1)(k+2)}$, so each term splits as Equation (4.6):

$$\frac{1}{k(k+1)(k+2)} = \frac{1}{2}\left[\frac{1}{k(k+1)} - \frac{1}{(k+1)(k+2)}\right] \tag{4.6}$$

Summed from $k=2$ to n, each negative term cancels the next positive one: $\sum_{k=2}^{n} a_k = \frac{1}{12} - \frac{1}{2(n+1)(n+2)}$. Adding $a_1 = \frac{2}{3} = \frac{8}{12}$ gives Equation (4.7):

$$S_n = \frac{3}{4} - \frac{1}{2(n+1)(n+2)} \tag{4.7}$$

Check: $n=2$: $\frac{3}{4} - \frac{1}{24} = \frac{17}{24}$ ✓. $n=5$: $\frac{3}{4} - \frac{1}{84} = \frac{31}{42} \approx 0.73809523810$ ✓.

Book: Section 4.4.5 and Equations (4.6)–(4.7).

A Second Road: Partial Fractions

Look for A, B, C with $\frac{1}{k(k+1)(k+2)} = \frac{A}{k} + \frac{B}{k+1} + \frac{C}{k+2}$.

Multiply out: $1 = (A+B+C)k^2 + (3A+2B+C)k + 2A$. Match coefficients: $2A = 1$, $2A+B = 0$, $A+B+C = 0$.

So $A = 1/2$, $B = -1$, $C = 1/2$: $\frac{1}{k(k+1)(k+2)} = \frac{1}{2k} - \frac{1}{k+1} + \frac{1}{2(k+2)}$.

Grouped as $1/2\left[1/k - 1/k+1\right] - 1/2\left[1/k+1 - 1/k+2\right]$ it turns back into Equation (4.6): both roads reach Equation (4.7).

The first road is shorter; the second works for any rational function. Book: Section 4.4.5.

TOHU — Proof by Induction

THEOREM. For every $n \geq 1$, $S_n = \frac{3}{4} - \frac{1}{2(n+1)(n+2)}$.

Base, $n=1$**:** $S_1 = \frac{2}{3}$, and $\frac{3}{4} - \frac{1}{12} = \frac{8}{12} = \frac{2}{3}$ ✓ — the exceptional a_1 is absorbed into the constant $\frac{3}{4}$.

Step, $n \geq 2$**:** $S_n = S_{n-1} + a_n = \frac{3}{4} - \frac{1}{2n(n+1)} + \frac{1}{n(n+1)(n+2)}$.

Over $2n(n+1)(n+2)$: $-\frac{(n+2)-2}{2n(n+1)(n+2)} = -\frac{1}{2(n+1)(n+2)}$. ∎

As n grows, S_n climbs towards $\frac{3}{4}$ without reaching it: every answer lies in $\left[\frac{2}{3}, \frac{3}{4}\right)$. Book: Section 4.4.6.

TOHU — Cost, Precision, Overflow

Time: a handful of operations per query — $\Theta(1)$, $\Theta(T)$ overall, about 10^5 operations in all.

Precision: eleven decimals of a number below 0.75 need about eleven significant digits; `double` has fifteen to sixteen, `float` seven and fails on the sample.

Overflow: `2*(n+1)*(n+2)` in `int` passes $2,147,483,647$ at $n = 32,767$. Convert each factor to `double` first; at $n = 10^9$ the correction $\approx 5 \times 10^{-19}$ vanishes and the program prints `0.75000000000`.

Book: Section 4.4.7.

TOHU — the Program

Pseudocode 4.2 The TOHU algorithm: one evaluation of the closed form Equation (4.7) per query.

```
TOHU()
1  read T
2  for q = 1 to T
3      read n
4      print 3/4 - 1 / (2(n+1)(n+2)) with 11 decimal places
```

Listing 4.4 TOHU — Help Tohu, in $O(1)$ per test case, with no special case for $n=1$.

```cpp
#include <cstdio>
using namespace std;

int main() {
    int T;
    if (scanf("%d", &T) != 1) return 0;
    while (T--) {
        long long n;
        if (scanf("%lld", &n) != 1) break;
        // S(n) = 3/4 - 1 / (2 (n+1)(n+2))   -- valid for every n >= 1
        double denom = 2.0 * (double)(n + 1) * (double)(n + 2);
        printf("%.11f\n", 0.75 - 1.0 / denom);
    }
    return 0;
}
```

```
0.66666666667
0.70833333333
0.72500000000
0.73333333333
0.74621212121
0.75000000000
```

Pseudocode 4.2 and Listing 4.4, on $n = 1, 2, 3, 4, 10, 10^9$*: the first four lines are the decimal column of Table 4.3; the fifth is* $\frac{197}{264}$*. The sample gives* `0.70833333333` *and* `0.73809523810`*. Book: Section 4.4.8, Pseudocode 4.2 and Listing 4.4.*

Another Correct Program

```c
#include <stdio.h>
int main() {
    int t, n;
    double formula;
    scanf("%d", &t);
    while (t--) {
        scanf("%d", &n);
        if (n==1) {
            printf("%.11lf\n",2.0/3.0);
        } else {
            formula = ((double) 1.0/12.0) - 1/ ((double) 2 * (n+1) * (n+2));
            formula = formula + ((double) 2.0/3.0);
            printf("%.11lf\n", formula);
        }
    }
    return 0;
}
```

Figure 4.3 An alternative TOHU program with a separate branch `if (n==1) printf("%.11lf\n", 2.0/3.0);` and a general branch that evaluates $\frac{1}{12} - \frac{1}{2(n+1)(n+2)} + \frac{2}{3}$.

Figure 4.3 special-cases $n=1$ — harmless but unnecessary, because the induction's base case shows Equation (4.7) already gives $\frac{2}{3}$. Its cast `(double) 2 * (n+1) * (n+2)` *avoids overflow correctly but fragilely. Book: Section 4.4.8 and Figure 4.4.*

Exact Fractions, with No Formula Assumed

Listing 4.5 TOHU from first principles: each a_k recovered exactly from the weighted-sum condition, with no formula assumed, and compared with the closed form.

```cpp
#include <cstdio>
#include <numeric>
using namespace std;
typedef long long LL;

struct Frac {                                   // an exact fraction p/q, always in lowest terms
    LL p, q;
    Frac(LL a = 0, LL b = 1) { LL g = gcd(a, b); p = a / g; q = b / g; }
};
Frac operator+(Frac a, Frac b) { return Frac(a.p * b.q + b.p * a.q, a.q * b.q); }
Frac operator-(Frac a, Frac b) { return Frac(a.p * b.q - b.p * a.q, a.q * b.q); }

int main() {
    // Recover a_k from the defining condition, exactly, with no formula assumed:
    // k*a_k = (k+1)/(k+2) - (sum of j*a_j for j < k).
    Frac weighted(0, 1), S(0, 1);
    printf(" k   a_k        S_k (exact)     S_k (decimal)   closed form\n");
    for (LL k = 1; k <= 10; ++k) {
        Frac kak = Frac(k + 1, k + 2) - weighted;        // this is k * a_k
        Frac ak(kak.p, kak.q * k);                       // divide by k
        weighted = weighted + kak;
        S = S + ak;
        double closed = 0.75 - 1.0 / (2.0 * (k + 1) * (k + 2));
        printf("%2lld  %lld/%-6lld  %6lld/%-8lld  %.11f  %.11f\n",
               k, ak.p, ak.q, S.p, S.q, (double)S.p / S.q, closed);
    }
    return 0;
}
```

```
 k   a_k        S_k (exact)     S_k (decimal)   closed form
 1  2/3            2/3         0.66666666667  0.66666666667
 2  1/24          17/24        0.70833333333  0.70833333333
 3  1/60          29/40        0.72500000000  0.72500000000
 4  1/120         11/15        0.73333333333  0.73333333333
 5  1/210         31/42        0.73809523810  0.73809523810
 6  1/336         83/112       0.74107142857  0.74107142857
 7  1/504        107/144       0.74305555556  0.74305555556
 8  1/720         67/90        0.74444444444  0.74444444444
 9  1/990         41/55        0.74545454545  0.74545454545
10  1/1320       197/264       0.74621212121  0.74621212121
```

Listing 4.5 recovers each a_k from Equation (4.4) alone, in exact arithmetic with `std::gcd`: useless as a solution ($\Theta(n)$ per query), excellent as a check. Book: Section 4.4.9 and Listing 4.5.

Mistakes on the Two Judge Problems

LDCSUMAB: assuming the blocks do not interleave without proof; floor for ceiling or $>$ for $\geq$ — test $k=1,3,6,10$ and $2,4,7,11$; printing `n m`; $n(n+1)$ in `int`; building the list.

TOHU: looping $k=1..n$ per query; differencing backwards and getting $-k\,a_k$.

TOHU: trusting the pattern at $k=1$, which gives $\frac{1}{4}-\frac{1}{2(n+1)(n+2)}$ — wrong by exactly $\frac{1}{2}$.

TOHU: `float` or `%.10f`; `(n+1)*(n+2)` in `int`; writing `3/4`, which is `0` in C++.

Book: Sections 4.3.10 and 4.4.10.

Worked Problem A — Horner's Rule

Pseudocode 4.3 Horner's rule: evaluating $a_0+a_1x+\cdots+a_nx^n$ from the highest coefficient down.

```
HORNER(A, n, x)
1  p = 0
2  for i = n downto 0
3      p = A[i] + x * p
4  return p
```

Idea: $a_0+a_1x+a_2x^2+a_3x^3=a_0+x(a_1+x(a_2+x\,a_3))$ — one multiplication per level.

LOOP INVARIANT (Horner's rule). At the start of each iteration of lines 2–3, with index i, Equation (4.8) holds:

$$p=\sum_{k=0}^{n-(i+1)} A[k+i+1]x^k \tag{4.8}$$

Initialization: $i=n$, the sum runs to -1 — empty, so 0, which is line 1's p. ✓

In words: p is the polynomial with coefficients $A[i+1],\ldots,A[n]$, lowest first. Pseudocode 4.3 claims $n+1$ multiplications. Book: Section 4.5.1, Pseudocode 4.3 and Equation (4.8).

Horner — Maintenance, Termination, Trace

Maintenance: $p' = A[i] + x\sum_{k} A[k+i+1]x^{k} = \sum_{j=0}^{n-i} A[j+i]x^{j}$ (put $j = k+1$); with $i' = i-1$ this is Equation (4.8) again. ✓

Termination: at $i = -1$, $p = \sum_{k=0}^{n} a_k x^k = P(x)$. ∎ Cost $\Theta(n)$ against $\frac{(n+1)(n+2)}{2} = \Theta(n^2)$ for the naive method.

Table 4.4 Horner's rule on $2x^3 - 6x^2 + 2x - 1$ at $x = 3$. After each iteration, p equals the partial polynomial predicted by the invariant; after the last, it equals $P(3) = 5$.

Iteration i	$A[i]$	p Before	Line 3: $A[i] + 3p$	Invariant for the next index, evaluated directly
3	2	0	$2 + 3 \cdot 0 = 2$	$A[3] = 2$
2	-6	2	$-6 + 3 \cdot 2 = 0$	$A[2] + 3A[3] = -6 + 6 = 0$
1	2	0	$2 + 3 \cdot 0 = 2$	$A[1] + 3A[2] + 9A[3] = 2 - 18 + 18 = 2$
0	-1	2	$-1 + 3 \cdot 2 = 5$	$A[0] + 3A[1] + 9A[2] + 27A[3] = -1 + 6 - 54 + 54$

Table 4.4 evaluates the invariant independently in its last column: it holds at every step. Book: Section 4.5.1 and Table 4.4.

Horner — the Program

Listing 4.6 Horner's rule against naive evaluation, with a trace of the loop and a count of multiplications.

```cpp
#include <cstdio>
#include <vector>
using namespace std;
typedef long long LL;

// HORNER(A, n, x): p = 0; for i = n downto 0: p = A[i] + x*p
LL horner(const vector<LL>& A, LL x, LL& mults) {
    LL p = 0;
    int n = (int)A.size() - 1;
    for (int i = n; i >= 0; --i) {
        // invariant: p = sum_{k=0}^{n-(i+1)} A[k+i+1] * x^k
        p = A[i] + x * p;
        ++mults;
        printf("  i = %d   p = A[%d] + %lld * (previous p) = %lld\n", i, i, x, p);
    }
    return p;
}

// Naive evaluation: compute every power x^k from scratch.
LL naive(const vector<LL>& A, LL x, LL& mults) {
    LL sum = 0;
    for (int k = 0; k < (int)A.size(); ++k) {
        LL power = 1;
        for (int t = 0; t < k; ++t) { power *= x; ++mults; }   // k multiplications
        sum += A[k] * power; ++mults;                          // one more for A[k]*x^k
    }
    return sum;
}
```

```
int main() {
    vector<LL> A = {-1, 2, -6, 2};          // P(x) = 2x^3 - 6x^2 + 2x - 1, A[k] = coefficient of x^k
    LL x = 3, mh = 0, mn = 0;
    printf("Horner on P(x) = 2x^3 - 6x^2 + 2x - 1 at x = 3\n");
    LL h = horner(A, x, mh);
    LL v = naive(A, x, mn);
    printf("Horner: P(3) = %lld using %lld multiplications\n", h, mh);
    printf("Naive : P(3) = %lld using %lld multiplications\n", v, mn);
```

Listing 4.6 (Continued) *Horner's rule against naive evaluation, with a trace of the loop and a count of multiplications.*

```
    // Operation counts for larger degrees (all coefficients 1, x = 1, so no overflow).
    printf("\n degree n   Horner mults   naive mults\n");
    for (int n : {3, 10, 100, 1000}) {
        vector<LL> B(n + 1, 1);
        LL a = 0, b = 0;
        LL p = 0; for (int i = n; i >= 0; --i) { p = B[i] + 1 * p; ++a; }
        naive(B, 1, b);
        printf(" %8d   %12lld   %11lld\n", n, a, b);
    }
    return 0;
}
```

```
Horner on P(x) = 2x^3 - 6x^2 + 2x - 1 at x = 3
  i = 3   p = A[3] + 3 * (previous p) = 2
  i = 2   p = A[2] + 3 * (previous p) = 0
  i = 1   p = A[1] + 3 * (previous p) = 2
  i = 0   p = A[0] + 3 * (previous p) = 5
Horner: P(3) = 5 using 4 multiplications
Naive : P(3) = 5 using 10 multiplications

 degree n   Horner mults   naive mults
        3              4            10
       10             11            66
      100            101          5151
     1000           1001        501501
```

Listing 4.6: $1,001$ *multiplications against* $501,501$ *at degree* 1000. *Check yourself:* `r = 1; for i = 1 to n: r = r * x` *has invariant* $r = x^{i-1}$. *Book: Section 4.5.1 and Listing 4.6.*

Worked Problem B — Tracing Insertion Sort

Table 4.5 Insertion Sort on $\langle 8,3,6,3,9,1 \rangle$. The total number of shifts, 9, equals the number of inversions; t_j counts executions of the `while` test on line 4 and is always one more than the shifts.

j	Key	Array after the iteration	Comparisons	Shifts	t_j
2	3	$\langle 3,8,6,3,9,1 \rangle$	1	1	2
3	6	$\langle 3,6,8,3,9,1 \rangle$	2	1	2
4	3	$\langle 3,3,6,8,9,1 \rangle$	3	2	3
5	9	$\langle 3,3,6,8,9,1 \rangle$	1	0	1
6	1	$\langle 1,3,3,6,8,9 \rangle$	5	5	6
		total	**12**	**9**	**14**

On $\langle 8,3,6,3,9,1 \rangle$: at $j=4$ the test $3>3$ is **false**, so the two 3s keep their order — stability from the strict `>`.

Inversions of the original: $4+1+2+1+1=9$ — exactly the shifts, always, because each shift removes one inversion and creates none.

t_j = shifts $+1$ (one failing test ends each loop): $\sum t_j = 14$.

Table 4.5: 12 comparisons, 9 shifts. The chapter's Listing 4.7 prints the same trace. Check yourself: $\langle 4,1,3,9,7 \rangle$ needs 3 shifts, and has 3 inversions. Book: Section 4.5.2 and Table 4.5.

Worked Problem C — Tracing Binary Search

Table 4.6 Binary search in a 15-element array for a present key (44) and an absent key (20). The window shrinks $15 \to 7 \to 3 \to 1 \to 0$; four probes suffice in every case.

q	Probe	*left*	*right*	*mid*	$A[mid]$	Decision	Window size after
44	1	1	15	8	31	too small: $left = 9$	7
44	2	9	15	12	51	too big: $right = 11$	3
44	3	9	11	10	40	too small: $left = 11$	1
44	4	11	11	11	44	found, return 11	—
20	1	1	15	8	31	too big: $right = 7$	7
20	2	1	7	4	15	too small: $left = 5$	3
20	3	5	7	6	22	too big: $right = 5$	1
20	4	5	5	5	19	too small: $left = 6$	0

Array $\langle 3, 8, 11, 15, 19, 22, 27, 31, 36, 40, 44, 51, 57, 62, 70 \rangle$: $q = 44$ is found at 11; $q = 20$ ends with $left = 6 > right = 5$ — it would sit between 19 and 22, where no position exists.

With $15 = 2^4 - 1$ elements the window shrinks $15 \to 7 \to 3 \to 1 \to 0$: at most $\lfloor \lg 15 \rfloor + 1 = 4$ probes.

Table 4.6; the chapter's Listing 4.8 prints one line per probe. Check yourself: $q = 70$ and $q = 3$ also take 4 probes. Book: Section 4.5.3 and Table 4.6.

Worked Problem D — Counting a Triple Loop

Pseudocode 4.4 A triple loop whose innermost line is to be counted exactly.

```
LOOPS(n)
1  count = 0
2  for i = 1 to n
3      for j = i to n
4          for k = i to j
5              count = count + 1
6  return count
```

k loop: $j-i+1$ values. j loop: $\sum_{t=1}^{n-i+1} t = \frac{(n-i+1)(n-i+2)}{2}$. i loop, with $m=n-i+1$: $\frac{1}{2}(\sum m^2+\sum m)$.

$\sum m^2 = \frac{n(n+1)(2n+1)}{6}$ by telescoping $(m+1)^3-m^3=3m^2+3m+1$. Together, Equation (4.9):

$$\text{executions of line } 5 = \sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 = \frac{n(n+1)(n+2)}{6} \tag{4.9}$$

Pseudocode 4.4: work from the inside out; every loop becomes a sum. Book: Section 4.6.1, Pseudocode 4.6 and Equation (4.9).

The Count, Checked

Table 4.7 Counting the innermost executions of LOOPS for $n=4$ by hand. The subtotals are the triangular numbers $10, 6, 3, 1$, and their sum is $\frac{4\cdot 5\cdot 6}{6}=20$.

i	Contributions $j-i+1$ for $j=i,\dots,4$	Subtotal
1	$1+2+3+4$	10
2	$1+2+3$	6
3	$1+2$	3
4	1	1
	total	**20**

$\frac{4\cdot 5\cdot 6}{6}=20$ ✓. The rows of Table 4.7 are triangular numbers; their sums $1, 4, 10, 20, 35, \dots$ are *tetrahedral* numbers — oranges in a triangular pyramid.

$\frac{n^3+3n^2+2n}{6}=\Theta(n^3)$ with constant $\frac{1}{6}$: a sixth of a plain triple loop's n^3.

> **CHECK YOURSELF.** `for i = 1 to n: for j = i+1 to n` runs $\frac{n(n-1)}{2}$ times; `i = n; while i > 1: i = floor(i/2)` runs $\lfloor \lg n \rfloor$ times ($10 \to 5 \to 2 \to 1$).

The chapter's Listing 4.9 confirms the formula up to $n = 500$ *(*$20,958,500$*). Book: Section 4.6.1 and Table 4.7.*

Worked Problem E — Explicit Constants

The definition, Equation (4.10); O is its right half alone:

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \tag{4.10}$$

(a) $5n^2 - 3n + 12 = \Theta(n^2)$**:** $c_2 = 5$ needs $n \ge 4$; $c_1 = 4$ needs $(n - 3/2)^2 + \frac{39}{4} \ge 0$, always. $n_0 = 4$ does real work: $f(3) = 48 > 45$.

(b) $2^{n+1} = O(2^n)$**:** $c = 2$. **(c)** $2^{2n} = O(2^n)$ **is false:** dividing gives $2^n \le c$ forever — broken at $n = \max(n_0, \lfloor \lg c \rfloor + 1)$. A constant in the exponent changes the base.

(d) $\lg(n!) = \Theta(n \lg n)$**:** $n! \le n^n$ gives $c_2 = 1$; the top half of the factors gives $n! \ge (n/2)^{n/2}$, so $c_1 = 1/4$ from $n_0 = 4$.

The chapter's Listing 4.10 checks these constants for every n up to 10^6 with no violation. Book: Section 4.6.2 and Equation (4.10).

Worked Problem F — Thirteen Functions Ranked

Simplify first: $2^{\lg n} = n$ and $4^{\lg n} = n^2$ (by $a^{\lg n} = n^{\lg a}$); $\lg(n!) = \Theta(n \lg n)$ by Problem E.

Table 4.8 The thirteen functions of Worked Problem F in increasing order of growth. Functions on the same row are Θ of each other; the last column justifies each step up.

Rank	Function(s)	Simplified class	Reason it beats the row above
1	100	$\Theta(1)$	—
2	$\lg n$	$\Theta(\lg n)$	grows without bound
3	$\lg^2 n$	$\Theta(\lg^2 n)$	ratio $\lg n \to \infty$
4	$\sqrt{n}$	$\Theta(n^{1/2})$	polylog $= o(n^{\varepsilon})$
5	$2^{\lg n}$	$= n$	ratio $\sqrt{n} \to \infty$
6	$n \lg n, \lg(n!)$	$\Theta(n \lg n)$	ratio $\lg n \to \infty$
7	$n^2, 4^{\lg n}$	$= n^2$	ratio $n/\lg n \to \infty$
8	n^3	$\Theta(n^3)$	ratio $n \to \infty$
9	$(3/2)^n$	exponential	exponential beats polynomial
10	2^n	exponential	ratio $(4/3)^n \to \infty$
11	$n!$	$\Theta\left(\sqrt{n}(n/e)^n\right)$	ratio $\geq \frac{1}{2}(3/2)^{n-2} \to \infty$

At small n the order lies: at $n=16$ the constant 100 beats n, and $(3/2)^{16}<16^3$; only by $n=2^{32}$ is every row above the one before.

Each step of Table 4.8 is justified by a ratio tending to 0 or ∞; the chapter's Listing 4.11 prints $\lg f(n)$ at four sizes. Check yourself: $n^{1/\lg n}=2$, and $n^{\lg 7}$ sits between n^2 and n^3. Book: Section 4.6.3 and Table 4.8.

Worked Problem G — Substitution, Strengthened

(a) $T(n)=T(\lfloor n/2 \rfloor)+n$, $T(1)=1$: guess $T(n)\le 2n$; $T(n)\le 2\lfloor n/2 \rfloor+n\le 2n$ ✓; and $T(n)\ge n$. So $\Theta(n)$ — the geometric series $n+n/2+\cdots<2n$.

(b) The recurrence Equation (4.11):

$$U(n)=2U(\lfloor n/2 \rfloor)+1, U(1)=1 \tag{4.11}$$

Guess $U(n)\le cn$: we get $cn+1$ — **fails for every** c, and "$cn+1$ is still $O(n)$" is exactly the error of Section 3.11.3.

Guess $U(n)\le cn-d$: $2(c\lfloor n/2 \rfloor-d)+1\le cn-2d+1\le cn-d$ when $d\ge 1$; the base needs $c\ge 2$. So $U(n)\le 2n-1$.

The weaker claim $U(n)\le 2n$ is true but unprovable this way; the stronger one proves. In fact $U(n)=2^{\lfloor \lg n \rfloor+1}-1$. Book: Section 4.7.1 and Equation (4.11).

Substitution, Checked by Machine

Listing 4.7 Both recurrences of Worked Problem G evaluated exactly, with the proved bounds checked for every $n \le 10^6$.

```cpp
#include <cstdio>
#include <vector>
using namespace std;
typedef long long LL;

int main() {
    const int N = 1000000;
    vector<LL> T(N + 1), U(N + 1);
    T[1] = 1; U[1] = 1;
    for (int n = 2; n <= N; ++n) {
        T[n] = T[n / 2] + n;          // T(n) = T(floor(n/2)) + n
        U[n] = 2 * U[n / 2] + 1;      // U(n) = 2U(floor(n/2)) + 1
    }
    printf("  n   T(n)  2n   U(n)  2n-1\n");
    for (int n = 1; n <= 10; ++n)
        printf("%3d  %5lld %3d  %5lld %5d\n", n, T[n], 2 * n, U[n], 2 * n - 1);
    int badT = 0, badU = 0, badUn = 0, firstBadUn = 0;
    for (int n = 1; n <= N; ++n) {
        if (!(n <= T[n] && T[n] <= 2LL * n)) ++badT;
        if (!(U[n] <= 2LL * n - 1)) ++badU;
        if (U[n] > n) { ++badUn; if (!firstBadUn) firstBadUn = n; }
    }
    printf("n <= T(n) <= 2n     fails for %d values of n up to 10^6\n", badT);
    printf("U(n) <= 2n - 1      fails for %d values of n up to 10^6\n", badU);
    printf("U(n) <= n (c = 1)   fails for %d values, the first at n = %d\n", badUn, firstBadUn);
    return 0;
}
```

```
  n   T(n)  2n   U(n)  2n-1
  1      1   2      1     1
  2      3   4      3     3
  3      4   6      3     5
  4      7   8      7     7
  5      8  10      7     9
  6     10  12      7    11
  7     11  14      7    13
  8     15  16     15    15
  9     16  18     15    17
 10     18  20     15    19
n <= T(n) <= 2n     fails for 0 values of n up to 10^6
U(n) <= 2n - 1      fails for 0 values of n up to 10^6
U(n) <= n (c = 1)   fails for 999981 values, the first at n = 2
```

Listing 4.7: U meets $2n-1$ exactly at powers of two, so $c=2$ cannot be improved. Check yourself: $T(n)=T(\lfloor n/2 \rfloor)+1$ gives $T(n)\le \lg n+1$. Book: Section 4.7.1 and Listing 4.12.

Worked Problem H — a Recursion Tree

$$T(n)=3T(n/4)+n^2, T(1)=1 \tag{4.12}$$

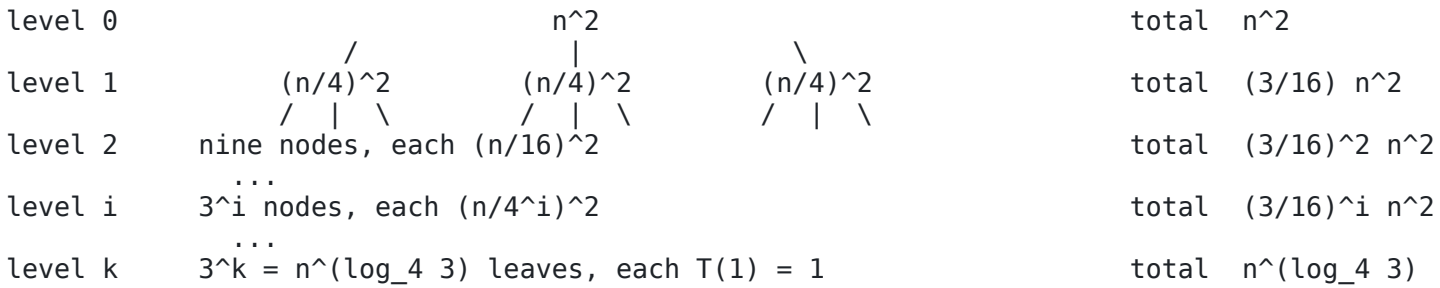


Table 4.9 The recursion tree of $T(n)=3T(n/4)+n^2$, level by level. Each level costs $\frac{3}{16}$ of the level above, so the root dominates.

Level i	Nodes	Size of each	Cost of each	Cost of the level
0	1	n	n^2	n^2
1	3	$n/4$	$n^2/16$	$\frac{3}{16}n^2$
2	9	$n/16$	$n^2/256$	$\left(\frac{3}{16}\right)^2 n^2$
i	3^i	$n/4^i$	$n^2/16^i$	$\left(\frac{3}{16}\right)^i n^2$
$k=\log_4 n$	$3^k=n^{\log_4 3}$	1	1	$n^{\log_4 3}$

For Equation (4.12) with $n=4^k$: level i has 3^i nodes costing $n^2/16^i$ each, so it costs $(3/16)^i n^2$; Table 4.9 shows the root dominating. Book: Section 4.7.2, Equation (4.12) and Table 4.9.

Summing the Tree, Then Two Confirmations

Internal levels: $n^2 \cdot \frac{1-3^k/n^2}{13/16} = \frac{16}{13}(n^2 - 3^k)$; add the 3^k leaves, giving Equation (4.13):

$$T(n) = \frac{16n^2 - 3^{k+1}}{13}, n = 4^k \tag{4.13}$$

Hand check: $T(4) = 3 + 16 = 19 = \frac{256-9}{13}$; $T(16) = 3 \cdot 19 + 256 = 313 = \frac{4096-27}{13}$. So $n^2 \le T(n) < \frac{16}{13}n^2$: $\Theta(n^2)$.

Substitution: $3 \cdot \frac{16}{13} \cdot \frac{n^2}{16} + n^2 = \frac{16}{13}n^2$ ✓. **Master:** $n^{\log_4 3} \approx n^{0.792}$, $f = n^2$, regularity $c = \frac{3}{16}$ — Case 3.

The tree finds, substitution proves; the regularity constant is exactly the level ratio. The chapter's Listing 4.13 shows $T(n)/n^2 \to 1.230769 = \frac{16}{13}$.

Book: Section 4.7.2 and Equation (4.13).

Worked Problem I — Four Recurrences, One Gap

Table 4.10 Four recurrences through the Master Theorem. The last one lies in the gap between Cases 2 and 3 and is solved by unrolling.

Recurrence	$\log_b a$	Compare $f(n)$	Case	Solution	Exact on powers of b
$8T(n/4)+n$	1.5	$n=O(n^{1.5-0.5})$	1	$\Theta(n^{1.5})$	$2n^{1.5}-n$
$27T(n/3)+n^3$	3	$n^3=\Theta(n^3)$	2	$\Theta(n^3\lg n)$	$n^3(\log_3 n+1)$
$5T(n/4)+n^2$	≈ 1.161	$n^2=\Omega(n^{1.161+0.839}),\ c=\frac{5}{16}$	3	$\Theta(n^2)$	$\frac{16n^2-5n^{\log_4 5}}{11}$
$T(n/2)+\lg n$	0	$\lg n$: not polynomially above 1	gap	$\Theta(\lg^2 n)$ by unrolling	$\frac{\lg n(\lg n+1)}{2}+1$

(a) $\log_4 8=1.5$, $\varepsilon=0.5$: Case 1. **(b)** $\log_3 27=3$: Case 2. **(c)** $\log_4 5\approx 1.161$, regularity $c=\frac{5}{16}$: Case 3.

(d) $T(n)=T(n/2)+\lg n$: $n^0=1$, and $\lg n$ is larger but not *polynomially* — **the gap**. Unroll: $T(2^k)=k+(k-1)+\cdots+1+1=\frac{k(k+1)}{2}+1=\Theta(\lg^2 n)$; at $n=16$, 11 ✓.

Table 4.10. Also outside the theorem: $2T(n-1)+1=\Theta(2^n)$, $1/2\,T(n/2)+n$ *(*$a<1$*),* $2^n T(n/2)+n$*. Book: Section 4.7.3 and Table 4.10.*

The Four, Checked by Machine

Listing 4.8 The four recurrences of Worked Problem I evaluated exactly and divided by their predicted bounds.

```cpp
#include <cstdio>
#include <cmath>
using namespace std;
typedef long double LD;

// Each recurrence is evaluated exactly on powers of b, with T(1) = 1,
// and divided by the bound the analysis predicts. A ratio that settles
// to a constant confirms the Theta bound.
LD Ta(LD n) { return n <= 1 ? 1 : 8 * Ta(n / 4) + n; }          // Theta(n^1.5)
LD Tb(LD n) { return n <= 1 ? 1 : 27 * Tb(n / 3) + n * n * n; } // Theta(n^3 lg n)
LD Tc(LD n) { return n <= 1 ? 1 : 5 * Tc(n / 4) + n * n; }      // Theta(n^2)
LD Td(LD n) { return n <= 1 ? 1 : Td(n / 2) + log2l(n); }       // Theta(lg^2 n)

int main() {
    printf("(a) T(n)=8T(n/4)+n      T(n)/n^1.5\n");
    for (int k = 2; k <= 14; k += 4) { LD n = powl(4, k); printf("    n=4^%-2d  %.6Lf\n", k, Ta(n) /
powl(n, 1.5L)); }
    printf("(b) T(n)=27T(n/3)+n^3   T(n)/(n^3 log3 n)\n");
    for (int k = 2; k <= 14; k += 4) { LD n = powl(3, k); printf("    n=3^%-2d  %.6Lf\n", k, Tb(n) / (n *
n * n * k)); }
    printf("(c) T(n)=5T(n/4)+n^2    T(n)/n^2\n");
    for (int k = 2; k <= 14; k += 4) { LD n = powl(4, k); printf("    n=4^%-2d  %.6Lf\n", k, Tc(n) / (n *
n)); }
    printf("(d) T(n)=T(n/2)+lg n    T(n)/lg^2 n\n");
    for (int k = 4; k <= 64; k *= 2) { LD n = powl(2, k); printf("    n=2^%-2d  %.6Lf\n", k, Td(n) /
((LD)k * k)); }
    return 0;
}
```

```
(a) T(n)=8T(n/4)+n      T(n)/n^1.5
    n=4^2   1.750000
    n=4^6   1.984375
    n=4^10  1.999023
    n=4^14  1.999939
(b) T(n)=27T(n/3)+n^3   T(n)/(n^3 log3 n)
    n=3^2   1.500000
    n=3^6   1.166667
    n=3^10  1.100000
    n=3^14  1.071429
(c) T(n)=5T(n/4)+n^2    T(n)/n^2
    n=4^2   1.410156
    n=4^6   1.454122
    n=4^10  1.454541
    n=4^14  1.454545
(d) T(n)=T(n/2)+lg n    T(n)/lg^2 n
    n=2^4   0.687500
    n=2^8   0.578125
    n=2^16  0.535156
    n=2^32  0.516602
    n=2^64  0.508057
```

In Listing 4.8 the ratios settle at 2, 1, $\frac{16}{11}$ *and* $\frac{1}{2}$. *A ratio drifting like* $1+1/k$, *as in (b) and (d), signals a logarithmic factor. Book: Section 4.7.3 and Listing 4.14.*

Worked Problem J — Laundry Tokens

LAUNDRT. $T_k = 1+2+\cdots+k$ tokens circulate after day k; day k's bonus is $1/T_k$. Given $n \le 10^9$ (up to $20,000$ times), print $V_n = \sum_{k=1}^{n} 1/T_k$ to 11 decimals, in $O(1)$.

By hand: $V_1, \ldots, V_5 = 1, 4/3, 3/2, 8/5, 5/3 = 2/2, 4/3, 6/4, 8/5, \frac{10}{6}$ — the pattern $\frac{2n}{n+1}$.

Proof: $\frac{1}{T_k} = \frac{2}{k(k+1)} = 2\left(\frac{1}{k} - \frac{1}{k+1}\right)$, which telescopes to Equation (4.14):

$$V_n = 2\sum_{k=1}^{n}\left(\frac{1}{k} - \frac{1}{k+1}\right) = 2\left(1 - \frac{1}{n+1}\right) = \frac{2n}{n+1} \tag{4.14}$$

$V_n \to 2$: *the total bonus is bounded however long the promotion runs. Book: Section 4.8.1 and Equation (4.14).*

Laundry Tokens — the Program

Pseudocode 4.5 The Laundry Tokens algorithm: one evaluation of Equation (4.14) per query.

```
LAUNDRT()
1  read T
2  for q = 1 to T
3      read n
4      print 2n / (n + 1) with 11 decimal places
```

Listing 4.9 LAUNDRT — Laundry Tokens, solved in $O(1)$ by telescoping.

```cpp
#include <cstdio>
using namespace std;

int main() {
    int T;
    if (scanf("%d", &T) != 1) return 0;
    while (T--) {
        long long n;
        if (scanf("%lld", &n) != 1) break;
        // sum_{k=1..n} 1/T_k  with T_k = k(k+1)/2   equals   2n/(n+1)
        printf("%.11f\n", 2.0 * (double)n / ((double)n + 1.0));
    }
    return 0;
}
```

```
1.00000000000
1.33333333333
1.50000000000
1.81818181818
1.99999999800
```

Pseudocode 4.5 and *Listing 4.9* on $n = 1, 2, 3, 10, 10^9$. *The trap:* `2 * n / (n + 1)` *in integers is* `1` *for every* $n \geq 1$ *— convert before dividing.*

Check yourself: $\sum_{k=1}^{n} \frac{1}{k(k+2)} = \frac{3}{4} - \frac{2n+3}{2(n+1)(n+2)}$. *Book: Section 4.8.1, Pseudocode 4.7 and Listing 4.15.*

Worked Problem K — Tiling a 2 × N Strip

> **STRIPTILE.** Cover a $2 \times N$ corridor with 1×2 tiles. How many ways, modulo 10^9+7? Up to 10^5 queries, $0 \le N \le 10^{18}$, target $O(\log N)$.

By hand: $D_0=1$ (lay nothing), $D_1=1$, $D_2=2$ (VV, HH), $D_3=3$, $D_4=5$.

The leftmost column holds one vertical tile (leaving $N-1$) or the left halves of two stacked horizontal tiles (leaving $N-2$) — disjoint and exhaustive, giving Equation (4.15):

$$D_N = D_{N-1} + D_{N-2}, D_0 = D_1 = 1 \tag{4.15}$$

That is Fibonacci shifted: $D_N = F_{N+1}$. A table of 10^{18} entries is not a table, it is a wish. Book: Section 4.8.2 and Equation (4.15).

Fast Doubling

$\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^n = \begin{pmatrix}F_{n+1} & F_n\\ F_n & F_{n-1}\end{pmatrix}$ by induction; squaring the matrix for k gives the one for $2k$, and hence Equation (4.16):

$$F_{2k} = F_k(2F_{k+1} - F_k), F_{2k+1} = F_k^2 + F_{k+1}^2 \tag{4.16}$$

Pseudocode 4.6 Fast doubling: the Fibonacci pair (F_n, F_{n+1}) modulo p in $\Theta(\log n)$ multiplications.

```
FIB(n)                          // returns the pair (F(n), F(n+1)) modulo p
1  if n == 0
2      return (0, 1)
3  (a, b) = FIB(floor(n / 2))    // a = F(k), b = F(k+1), with k = floor(n/2)
4  c = a * (2b - a) mod p        // F(2k)
5  d = (a*a + b*b) mod p         // F(2k+1)
6  if n is odd
7      return (d, c + d mod p)   // (F(2k+1), F(2k+2))
8  else return (c, d)            // (F(2k), F(2k+1))
```

Binet's formula would need $\sqrt{5}$ modulo p; fast doubling needs only integers. Pseudocode 4.6 halves n each call: $T(n) = T(n/2) + \Theta(1) = \Theta(\log n)$. Book: Section 4.8.2, Equation (4.16) and Pseudocode 4.8.

Fast Doubling by Hand

Table 4.11 Fast doubling computing $(F_6, F_7) = (8, 13)$ in four calls. Each row checks one instance of the doubling identities.

Call	$k = \lfloor n/2 \rfloor$	$(a,b) = (F_k, F_{k+1})$	$c = a(2b - a)$	$d = a^2 + b^2$	Returns
FIB(0)	—	—	—	—	$(0, 1)$
FIB(1)	0	$(0, 1)$	$0 \cdot (2 - 0) =$	$0 + 1 = 1$	odd: $(1, 0 + 1) = (1, 1)$
FIB(3)	1	$(1, 1)$	$1 \cdot (2 - 1) =$	$1 + 1 = 2$	odd: $(2, 1 + 2) = (2, 3)$
FIB(6)	3	$(2, 3)$	$2 \cdot (6 - 2) =$	$4 + 9 = 13$	even: $(8, 13)$

So $D_5 = F_6 = 8$, matching $D_4 + D_3 = 5 + 3$.

Depth $\lfloor \log_2 N \rfloor + 2 \le 61$ for $N \le 10^{18}$: about 6×10^6 operations for 10^5 queries — well inside the budget.

Each row of Table 4.11 checks one instance of the doubling identities. Book: Section 4.8.2 and Table 4.11.

STRIPTILE — the Program

Listing 4.10 STRIPTILE — tilings of a $2 \times N$ strip by fast doubling, $\Theta(\log N)$ per query.

```cpp
#include <cstdio>
#include <utility>
using namespace std;
typedef long long LL;
const LL MOD = 1000000007LL;

// fast doubling: returns (F(n) mod p, F(n+1) mod p) with F(0)=0, F(1)=1
pair<LL, LL> fib(LL n) {
    if (n == 0) return {0, 1};
    auto p = fib(n >> 1);
    LL a = p.first, b = p.second;              // a = F(k), b = F(k+1), k = n/2
    LL c = a * ((2 * b % MOD - a % MOD + MOD) % MOD) % MOD;   // F(2k)   = F(k)(2F(k+1)-F(k))
    LL d = (a * a % MOD + b * b % MOD) % MOD;                  // F(2k+1) = F(k)^2 + F(k+1)^2
    if (n & 1) return {d, (c + d) % MOD};
    return {c, d};
}

int main() {
    int T;
    if (scanf("%d", &T) != 1) return 0;
    while (T--) {
        LL n;
        if (scanf("%lld", &n) != 1) break;
        printf("%lld\n", fib(n + 1).first);    // T(n) = F(n+1)
    }
    return 0;
}
```

```
1
1
2
3
5
8
1346269
680057396
```

Listing 4.10 on $N = 0, 1, 2, 3, 4, 5, 30, 10^{18}$. Reduce every product mod p at once (two residues fit a `long long`, three do not), and add `MOD` before `%` because `2b - a` can go negative. Traps: F_N instead of F_{N+1}; forgetting $D_0 = 1$. Book: Section 4.8.2 and Listing 4.16.

Worked Problem L — Does Any Pair Sum to x?

Brute force: try all $\binom{n}{2}$ pairs — obviously correct, $\Theta(n^2)$ in the worst case.

Pseudocode 4.7 The sort-then-search algorithm for the pair-sum problem, in $\Theta(n \lg n)$.

```
PAIR-SUM(a[0..n-1], x)
1   sort a into increasing order
2   for i = 0 to n - 1
3       want = x - a[i]
4       lo = i + 1, hi = n            // search a[i+1 .. n-1]
5       while lo < hi                 // invariant: a[lo-1] < want <= a[hi], where they exist
6           mid = lo + floor((hi - lo) / 2)
7           if a[mid] < want then lo = mid + 1
8           else hi = mid
9       if lo < n and a[lo] == want
10          return TRUE
11  return FALSE
```

Correct: a TRUE answer names two different positions summing to x; if a pair $p<q$ exists after sorting, the search from $i=p$ finds $a[q]$ as the first value ≥ its target.

By hand: $\langle 1,3,3,6,8,9\rangle$, $x=12 \rightarrow 3+9$, TRUE; $x=16 \rightarrow$ FALSE (8 may not pair with itself).

Pseudocode 4.7: sort once ($\Theta(n \lg n)$), then n binary searches of $O(\lg n)$. Book: Section 4.9.1 and Pseudocode 4.9.

The Race, Timed

Listing 4.11 Brute force against sort-and-search for the pair-sum problem, timed on inputs of doubling size.

```cpp
#include <cstdio>
#include <vector>
#include <random>
#include <algorithm>
#include <chrono>
using namespace std;
typedef long long LL;

// Brute force: try every pair i < j.  Theta(n^2) comparisons in the worst case.
bool pairBrute(const vector<LL>& a, LL x) {
    size_t n = a.size();
    for (size_t i = 0; i < n; ++i)
        for (size_t j = i + 1; j < n; ++j)
            if (a[i] + a[j] == x) return true;
    return false;
}

// Better: sort once, then binary-search for the partner x - a[i] of every element.
// Theta(n lg n) for the sort plus n searches of Theta(lg n) each.
bool pairSorted(vector<LL> a, LL x) {              // takes a copy: the caller's array is untouched
    sort(a.begin(), a.end());
    size_t n = a.size();
    for (size_t i = 0; i < n; ++i) {
        LL want = x - a[i];
        size_t lo = i + 1, hi = n;                 // search the partner in a[i+1 .. n-1]
        while (lo < hi) {
            size_t mid = lo + (hi - lo) / 2;
            if (a[mid] < want) lo = mid + 1; else hi = mid;
```

```
        }
        if (lo < n && a[lo] == want) return true;
    }
    return false;
}

double millis(chrono::steady_clock::time_point s) {
    return chrono::duration<double, milli>(chrono::steady_clock::now() - s).count();
}
```

Listing 4.11 (Continued) *Brute force against sort-and-search for the pair-sum problem, timed on inputs of doubling size.*

```
int main() {
    mt19937_64 rng(2026);                              // fixed seed: the same data on every run
    printf("     n   brute (ms)   sorted (ms)   same answer?\n");
    for (int n = 2000; n <= 64000; n *= 2) {
        vector<LL> a(n);
        for (int i = 0; i < n; ++i) a[i] = 2LL * i + 1;    // n distinct odd numbers
        shuffle(a.begin(), a.end(), rng);
        LL x = 1;                                          // odd + odd is even, so no pair sums to 1:
                                                           // both methods must do their full work
        auto s1 = chrono::steady_clock::now(); bool r1 = pairBrute(a, x);  double t1 = millis(s1);
        auto s2 = chrono::steady_clock::now(); bool r2 = pairSorted(a, x); double t2 = millis(s2);
        printf("%6d   %10.2f   %11.3f   %s\n", n, t1, t2, r1 == r2 ? "yes" : "NO");
    }
    vector<LL> small = {8, 3, 6, 3, 9, 1};
    printf("small example {8,3,6,3,9,1}: x=12 -> %d %d, x=16 -> %d %d\n",
           pairBrute(small, 12), pairSorted(small, 12), pairBrute(small, 16), pairSorted(small, 16));
    return 0;
}
```

```
     n   brute (ms)   sorted (ms)   same answer?
  2000         0.63         0.125   yes
  4000         2.46         0.230   yes
  8000         9.87         0.480   yes
 16000        41.79         0.994   yes
 32000       173.16         2.452   yes
 64000       670.66         4.631   yes
small example {8,3,6,3,9,1}: x=12 -> 1 1, x=16 -> 0 0
```

Listing 4.11 uses n distinct odd numbers and $x = 1$, so no pair exists and both methods do all their work; the seed is fixed. Your milliseconds will differ; the ratios will not. Book: Section 4.9.1 and Listing 4.17.

Reading the Ratios

Table 4.12 Measured times from Listing 4.11. Doubling n multiplies the brute-force time by about 4, as $\Theta(n^2)$ predicts, and the sort-and-search time by about 2, as $\Theta(n \lg n)$ predicts; the advantage itself doubles with every row.

n	Brute force (ms)	Ratio to previous row	Sort and search (ms)	Ratio to previous row	Brute / better
2,000	0.63	—	0.125	—	5.0
4,000	2.46	3.90	0.230	1.84	10.7
8,000	9.87	4.01	0.480	2.09	20.6
16,000	41.79	4.23	0.994	2.07	42.0
32,000	173.16	4.14	2.452	2.47	70.6
64,000	670.66	3.87	4.631	1.89	144.8

Doubling n: brute force $\times 3.87$–4.23 (predicted 4); sort-and-search $\times$ about 2 (predicted a little over 2). The advantage grows from $5\times$ to $145\times$.

Extrapolate to 10^6: brute force about $164,000$ ms, nearly three minutes; sort-and-search about 90 ms.

Why only $145\times$, when operation counts differ about $1,000\times$? The brute force's inner loop is the cheapest thing a processor does; sorting and searching jump and branch. Constants matter for seconds, not for growth.

The ratios of Table 4.12 are what the analysis predicts. Check yourself: after sorting, two pointers from both ends finish in $\Theta(n)$ steps. Book: Section 4.9.1 and Table 4.12.

The Fourteen Problems Compared

Table 4.13 The fourteen problems of this chapter at a glance: the family each belongs to, the tool from Chapters 1 to 3 that solves it, and what the solution costs.

Problem	Section	Family (Section 4.2.5)	Main tool, and where it was taught	Cost of the solution
LDCSUMAB	4.3	inverting a counting formula	arithmetic series, ceiling (§3.9.1, §3.8.2); binary search (§2.14)	$O(1)$ per query
TOHU	4.4	closed form from a sum	differencing, telescoping, induction (§3.9.5)	$O(1)$ per query
A — Horner's rule	4.5.1	loop with an invariant	loop-invariant proof (§2.9)	$\Theta(n)$
B — Insertion Sort trace	4.5.2	loop with an invariant	exact tracing; t_j (§2.8, §2.10)	$\Theta(n+\text{inversions})$
C — binary search trace	4.5.3	loop with an invariant	window invariant; $\lfloor \lg n \rfloor + 1$ probes (§2.14)	$\Theta(\lg n)$
D — triple loop	4.6.1	counting steps with sums	nested sums, $\sum m^2$ (§3.9)	$\frac{n(n+1)(n+2)}{6}$ steps
E — explicit constants	4.6.2	asymptotic proof	definitions of O, Θ (§3.3, §3.5)	—
F — ranking functions	4.6.3	asymptotic proof	limits of ratios (§3.8)	—
G — substitution	4.7.1	recurrence	strengthened hypothesis (§3.11)	$\Theta(n)$
H — recursion tree	4.7.2	recurrence	geometric level sums (§3.12)	$\Theta(n^2)$

Table 4.13 (Continued) *The fourteen problems of this chapter at a glance: the family each belongs to, the tool from Chapters 1 to 3 that solves it, and what the solution costs.*

Problem	Section	Family (Section 4.2.5)	Main tool, and where it was taught	Cost of the solution
I — Master Theorem	4.7.3	recurrence	three cases and the gap (§3.13)	various
J — Laundry Tokens	4.8.1	closed form from a sum	telescoping (§3.9.5)	$O(1)$ per query
K — $2 \times N$ strip	4.8.2	recurrence	Fibonacci, fast doubling (§3.8.9)	$\Theta(\log N)$ per query
L — pair sum	4.9	brute force, then better	sorting and binary search (§2.14, §3.15)	$\Theta(n \lg n)$

THE PATTERN BEHIND THE PATTERN. Move 1: compute small cases by hand and tabulate them. Move 2: reorganize the computation so the work per answer drops — telescope, invert a count, carry an invariant, halve, or sort so a search replaces a scan. Then prove it still gives the right answer.

In Table 4.13, every problem had an obvious method that was wrong or too slow, every one yielded to a table, and every answer was checked twice. Book: Section 4.10 and Table 4.13.

Profitina in Practice

Budgets as a design rule: pages × signals × sites due must fit the time available. **Projections from closed forms:** re-scans over a horizon in $\Theta(1)$, not day by day.

Near-duplicate pages are Problem L on documents: canonical keys, then sort or hash — about $\Theta(n \lg n)$ instead of $\Theta(n^2)$. **Checking twice** is a release gate: over a thousand automated tests pass before any release.

Profitina Laundry: the most profitable run of consecutive days, by prefix sums $P[j]$ and an invariant on the earliest minimum — one $\Theta(n)$ pass. Fourteen days in Keputih: **days 2 to 7, Rp 430.000** $(340,000 - (-90,000))$.

The platform advises; the customer decides. The full program, with a tie-break demonstration, is Listing 4.18. Book: Section 4.11.

Common Mistakes

Code first; skipping small cases; trusting a pattern unproved (TOHU's a_1). Do Steps 1–6 first and prove what the table suggests.

Ignoring the budget — consult Table 4.1. **Off-by-one at boundaries** — ceiling vs floor, `mid - 1`, F_N vs F_{N+1}.

An invariant without termination; a drifting constant in substitution; constants that depend on n.

The Master Theorem in its gap ($T(n/2)+\lg n$) or off its form ($2T(n-1)+1$).

Arithmetic: integer division (`3/4` is `0`), silent overflow, `sqrt` trusted for integers, `float` for eleven decimals, timings read without ratios.

Book: Section 4.12.

Summary — Five Things to Remember

1. Seven steps: formats; given and wanted; budget; small cases; shape; prove; only then code.

2. LDCSUMAB: order proved, blocks counted by T_n, inverted by the quadratic formula and a ceiling, made safe by an invariant.

3. TOHU: differencing recovers a_k (with the exceptional a_1), telescoping gives $S_n = \frac{3}{4} - \frac{1}{2(n+1)(n+2)}$.

4. Problems A–L: invariants, exact counts, explicit constants, three recurrence methods and a gap, telescoping, fast doubling, a timed race.

5. Tabulate the small cases, then reorganize the computation — and prove the reorganized computation is still right.

Book: Section 4.13.

Next Time

Chapter 5 — Heapsort and Quicksort: two $\Theta(n \lg n)$ sorts on typical input, for completely different reasons.

Their analysis uses exactly what we practiced today: loop invariants, summations and recurrences.

Before we meet: Exercises 4.1–4.12 — among them ranking for LDCSUMAB (4.9) and a variant of TOHU where a_1 is not exceptional (4.10).

Full solutions are in Section 4.15; try each for ten minutes first.

Book: Sections 4.14–4.15, then Chapter 5.

Lecture 5 — Sorting in Practice: Heapsort and Quicksort

Course: Design and Analysis of Algorithms (EF234405)

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*, Chapter 5

Lecturer: Irfan Subakti — yifana@gmail.com

Bring the array $[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$ *with you: today we build a heap out of it, sort it, and then sort a second array a completely different way. Book: Chapter 5, Sections 5.1–5.15.*

What This Meeting Covers

1. **The gap** — every fast sort we know needs a second array; every in-place sort we know is quadratic.
2. **The binary heap** — a tree with no pointers, hidden in a flat array; `MAX-HEAPIFY`, `BUILD-MAX-HEAP` in $O(n)$, and `HEAPSORT`, each traced on the board.
3. **Priority queues** — `MAXIMUM`, `EXTRACT-MAX`, `INCREASE-KEY`, `INSERT`.
4. **Quicksort** — Lomuto's and Hoare's partitions, the four-region invariant, a full trace.
5. **Analysis** — best, worst, 9 : 1, average ($\approx 1.39\, n \lg n$), randomization, the stack, and the $\Omega(n \lg n)$ floor.
6. **Choosing a sort** — stability, cache behavior, and Profitina Laundry.

The order of this meeting follows the chapter section by section. Book: Chapter 5, Sections 5.1–5.15.

Where We Left Off: a Hole in the Table

Chapter 2 gave us Insertion Sort and the loop invariant; Chapter 3 gave us O, Ω, Θ, recurrences, the Master Theorem, and Merge Sort at $\Theta(n \lg n)$. Table 5.1 is the whole inventory — read only the columns **worst case** and **in-place**.

Table 5.1 The four sorting algorithms from Chapters 2 and 3. Notice that the fast one is the only one that is not in-place.

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
Merge Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	No	$O(n)$	Fast, but needs a second array

THE DREAM. A sort that runs in $O(n \lg n)$ time **and** in place, with $O(1)$ extra memory. Nothing in Table 5.1 does both. **Heapsort** fills the hole with a $\Theta(n \lg n)$ worst case and $O(1)$ extra space; **Quicksort** fills it with $\Theta(n \lg n)$ on average, $O(\lg n)$ extra space, and the best speed in practice.

***In-place** means rearranging inside the given array with a fixed number of extra variables. `MERGE` cannot do that. Book: Sections 5.1.1 and 5.1.2.*

Why $n \lg n$, and the Selection Sort Insight

Take $n = 10^7$: $n^2 = 10^{14}$ but $n \lg n \approx 2.3 \times 10^8$, a ratio of about $430,000$. At 10^9 operations per second that is a quarter of a second against **twenty-seven hours**.

Selection Sort repeats two steps: **Step A** scans $A[1 \ldots i]$ for the largest, $\Theta(i)$; **Step B** swaps it into place, $\Theta(1)$. So the total is $\Theta(n) \times (n-1) = \Theta(n^2)$, and **all** of it is Step A.

If finding and removing the maximum cost $O(\lg n)$ instead, the total would be $O(\lg n) \times (n-1) = O(n \lg n)$ — with the shape of the algorithm unchanged.

> **KEY IDEA.** Sometimes the way to speed up an algorithm is to change the **data structure** it keeps its data in. Same algorithm, different container: quadratic becomes $n \lg n$.

Why is $n \lg n$ the target rather than n? Section 5.11 proves no comparison sort can do better. Book: Sections 5.1.3 and 5.2.

A Tree Hidden Inside an Array

A binary tree is **nearly complete** when every level is full except possibly the last, and the last fills **from the left**. Its shape is forced by n alone, so we never store it.

Number the nodes $1, 2, \ldots, n$ in level order and put node i in $A[i]$ ($A[0]$ unused). Level d starts at index 2^d; node $i = 2^d + k$ has children $2^{d+1} + 2k = 2i$ and $2i+1$. That derivation gives Equation (5.1), and inverting it gives Equation (5.2):

$$Left(i) = 2i, Right(i) = 2i + 1. \tag{5.1}$$

$$Parent(j) = \lfloor j/2 \rfloor. \tag{5.2}$$

Table 5.2 Moving around the hidden tree with arithmetic alone. Every one of these is a single machine instruction.

Relationship	Formula	Example, $i=3$	Hardware trick
Parent of node i	$\lfloor i/2 \rfloor$	$Parent(3) =$	Right-shift by one bit
Left child of node i	$2i$	$Left(3) = 6$	Left-shift by one bit
Right child of node i	$2i+1$	$Right(3) = 7$	Left-shift, then set the low bit

Table 5.2 is the whole navigation system: no pointer is ever followed, which is why heaps are fast in practice. Book: Sections 5.3.1 and 5.3.2.

The Formulas, Drawn

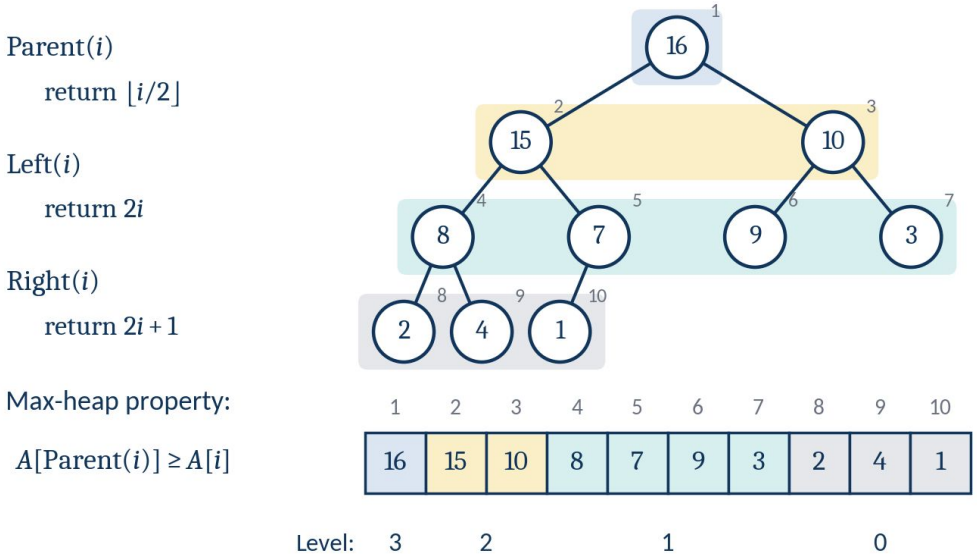


Figure 5.1 The three navigation formulas, the tree they describe, and the array that holds it. The colored bands group the nodes by depth and mark the matching runs of the array beneath; the row labeled “Level” counts from the bottom up, so the root’s band is 3 and the bottom row is 0.

Figure 5.1 draws the array $[16, 15, 10, 8, 7, 9, 3, 2, 4, 1]$ *— also a valid max-heap, which shows that the same numbers can form many heaps. Book: Section 5.3.2.*

The Max-Heap Property

DEFINITION. A **binary max-heap** is an array $A[1\ldots n]$, viewed as a nearly complete binary tree, in which every node is at least as large as each of its children:

$$A[Parent(i)] \geq A[i] \text{ for every } i = 2, 3, \ldots, n. \tag{5.3}$$

A **min-heap** reverses it: $A[Parent(i)] \leq A[i]$ for every $i > 1$.

Check $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$ against Equation (5.3): node 1: $16 \geq 14, 10$ ✔️; node 2: $14 \geq 8, 7$ ✔️; node 3: $10 \geq 9, 3$ ✔️; node 4: $8 \geq 2, 4$ ✔️; node 5: $7 \geq 1$ ✔️ (no right child, $11 > 10$). Nodes 6–10 have no children.

COMMON TRAP. A heap is **not** sorted: 9 at index 6 comes after 8 at index 4. The only global guarantee is that $A[1]$ is the maximum — by chaining $A[Parent(i)] \geq A[i]$ up to the root. ∎

The property is purely local: nothing about siblings or uncles. That weakness is exactly what makes a heap cheap to repair. Book: Sections 5.3.3–5.3.5.

The Same Ten Numbers, Seen Twice

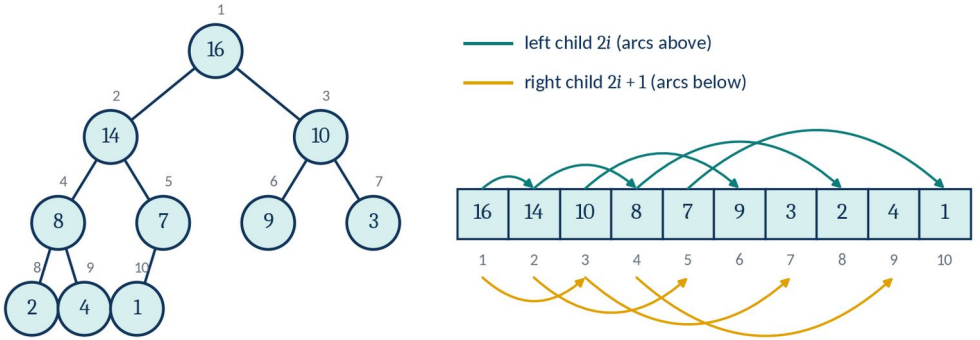


Figure 5.2 The same ten numbers seen twice. On the left, the nearly complete binary tree, with each node's array index printed above it. On the right, the array $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$, with an arc drawn from each node to each of its children — to the left child above the array, to the right child below it. The arcs are what the index arithmetic computes; no such arc is ever stored.

In Figure 5.2 the arcs are what the index arithmetic computes; no arc is ever stored. Book: Section 5.3.4.

Height, Leaves, and Node Counts

THEOREM. A heap with n elements has height $h = \lfloor \lg n \rfloor$. *Proof.* Levels $0 \dots h-1$ are full ($2^h - 1$ nodes) and level h holds between 1 and 2^h nodes, which gives Equation (5.4); taking lg gives $h \le \lg n < h+1$. ∎

$$2^h \le n < 2^{h+1}. \tag{5.4}$$

Leaves. Node i has a left child iff $2i \le n$, so the internal nodes are $1 \dots \lfloor n/2 \rfloor$ and the leaves are $\lfloor n/2 \rfloor + 1 \dots n$ — $\lceil n/2 \rceil$ of them, just over half.

Lemma. At most $\lceil n/2^{h+1} \rceil$ nodes have height h: strip the leaves and repeat. Half the nodes are leaves, a quarter have height 1, and only the root has height $\lfloor \lg n \rfloor$.

For $n = 10$, $h = 3$; for $n = 10^6$, $h = 19$. Hold on to the lemma: it is the whole of the $O(n)$ proof later. Book: Sections 5.3.6–5.3.9.

`MAX-HEAPIFY`: the One Repair

Precondition: the subtrees at $Left(i)$ and $Right(i)$ are max-heaps, but $A[i]$ may be too small. *Postcondition:* the subtree at i is a max-heap. The offending value **sinks**, as Pseudocode 5.1 shows.

Pseudocode 5.1 `MAX-HEAPIFY`, also called `HEAPIFY`: let the value at node i sink until it dominates both of its children.

```
HEAPIFY(A, i):
 1.  l ← LEFT(i)                          // l = 2i
 2.  r ← RIGHT(i)                         // r = 2i + 1
 3.  largest ← i
 4.  if l ≤ heap-size and A[l] > A[i]
 5.      then largest ← l
 6.  if r ≤ heap-size and A[r] > A[largest]
 7.      then largest ← r
 8.  if largest ≠ i
 9.      swap A[i] ↔ A[largest]
10.      HEAPIFY(A, largest)             // recurse down
```

THREE DETAILS. (1) Test `l ≤ heap-size` **before** reading $A[l]$. (2) Use `heap-size`, not the array length — Heapsort depends on it. (3) Line 6 compares with `A[largest]`, not `A[i]`.

In plain words: look at three values, find the largest; if it is not the node, swap and repeat one level down. Book: Sections 5.4.1 and 5.4.2.

`MAX-HEAPIFY(A, 2)` — Step (a)

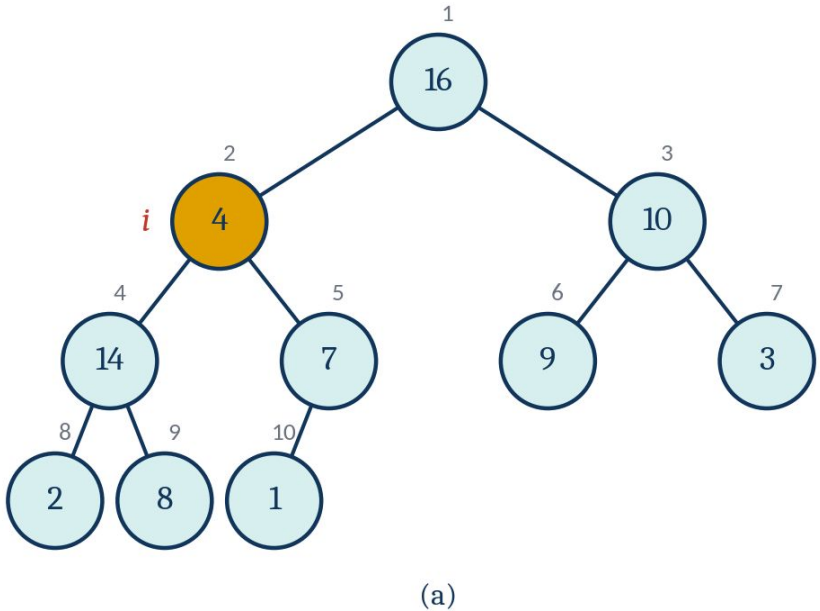


Figure 5.3 `MAX-HEAPIFY(A, 2)`, step (a). The call begins. Node $i = 2$, shaded, holds the value 4. Its children are $A[4] = 14$ and $A[5] = 7$. Both subtrees below it are already valid max-heaps; only node 2 is out of place.

Figure 5.3: $A = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]$. $l = 4$, $r = 5$; $14 > 4$ *so* $\text{largest} = 4$; $7 > 14$*? no. Swap* $A[2] \leftrightarrow A[4]$ *and recurse at* 4. *Book: Section 5.4.3.*

`MAX-HEAPIFY(A, 2)` — Step (b)

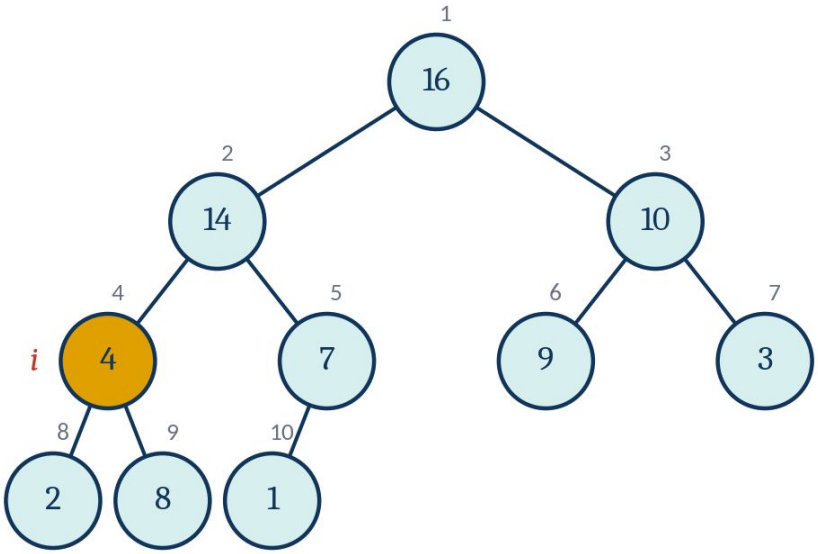
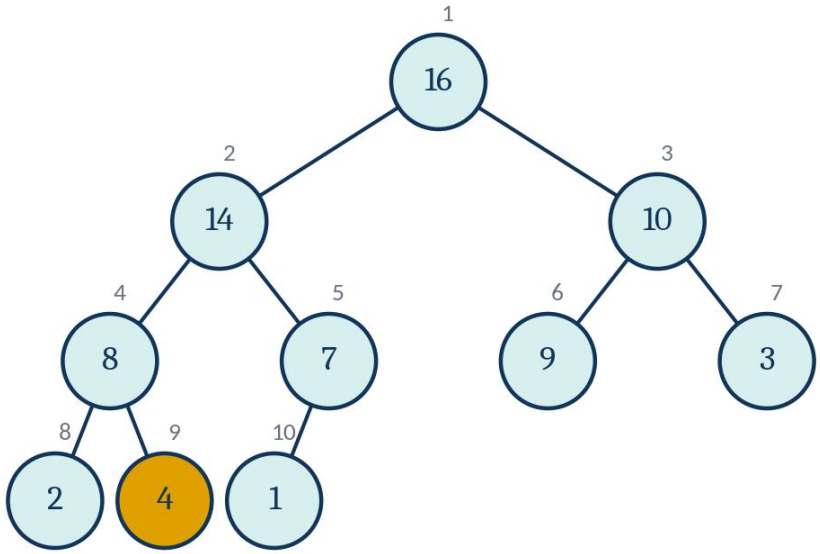


Figure 5.4 Step (b). After the first exchange. The value 14 has taken the place of 4 at index 2, and the offending 4 is now at index $i=4$, shaded. Its children are $A[8]=2$ and $A[9]=8$ — and $8>4$, so the sinking is not finished.

Figure 5.4: at $i=4$, $l=8$, $r=9$; $2>4$*? no;* $8>4$*? yes, so* $\text{largest}=9$. *Swap* $A[4] \leftrightarrow A[9]$ *and recurse at* 9. *Book: Section 5.4.3.*

`MAX-HEAPIFY(A, 2)` — Step (c)



(c)

Figure 5.5 Step (c). The repair is complete. The value 4 has come to rest at index 9, a leaf, and the whole tree is once again a max-heap. The value traveled the path $2 \to 4 \to 9$, which is one root-to-leaf path and no more.

Figure 5.5: at $i=9$, $18>10$, so there are no children; no swap; return. Result $[16,14,10,8,7,9,3,2,4,1]$ — three calls on one downward path. Book: Section 5.4.3.

Correct, and $O(\lg n)$

Correctness, by induction on the height of i. A leaf returns at once. Otherwise $A[\text{largest}]$ is the greatest of three; after the swap the untouched subtree is still a heap, and the other one has exactly `MAX-HEAPIFY`'s precondition at a lower height. ∎

Cost, in one line. $\Theta(1)$ per level, at most $\lfloor \lg n \rfloor$ levels: $O(\lg n)$.

Cost, via Chapter 3. With the bottom level exactly half full, the left subtree holds $\frac{2 \cdot 2^{h-1} - 1}{3 \cdot 2^{h-1} - 1} < \frac{2}{3}$ of the nodes, so we get Equation (5.5):

$$T(n) \le T(2n/3) + \Theta(1). \tag{5.5}$$

$a = 1$, $b = 3/2$, $n^{\log_b a} = n^0 = 1$, $f(n) = \Theta(1)$: Case 2 with $k = 0$, so $T(n) = \Theta(\lg n)$ and, the recurrence being an inequality, $O(\lg n)$.

A million elements: at most nineteen steps. A billion: at most twenty-nine. Book: Sections 5.4.4 and 5.4.5.

`BUILD-MAX-HEAP`: Three Lines

Pseudocode 5.2 `BUILD-MAX-HEAP`: repair every internal node, from the last one back up to the root.

```
BUILD-MAX-HEAP(A):
 1.  heap-size ← n
 2.  for i ← ⌊n/2⌋ downto 1
 3.      MAX-HEAPIFY(A, i)
```

Why start at $\lfloor n/2 \rfloor$? Nodes $\lfloor n/2 \rfloor + 1 \ldots n$ are leaves, already one-element heaps. **Why count down?** Children $2i$, $2i+1$ exceed i, so each child is finished before its parent — Pseudocode 5.2 always meets `MAX-HEAPIFY`'s precondition.

> **LOOP INVARIANT.** At the start of each iteration, each node $i+1, \ldots, n$ is the root of a max-heap. *Initialization:* those are leaves. *Maintenance:* both children of i are heap roots, so the call makes i one too. *Termination:* $i = 0$, so node 1 — the whole array — is a max-heap. ∎

*Counting **upward** breaks the precondition and gives a wrong answer, not merely a slow one. Book: Sections 5.5.1 and 5.5.2.*

The Input: No Order At All

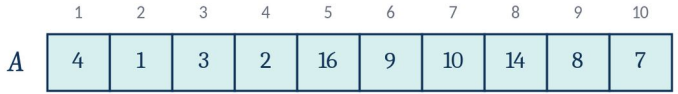


Figure 5.6 The input to `BUILD-MAX-HEAP`: the array $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$, in no particular order.

Figure 5.6: $n = 10$, so the loop runs $i = 5, 4, 3, 2, 1$; nodes 6–10 (holding $9, 10, 14, 8, 7$) are leaves and are skipped. Book: Section 5.5.3.

Build, Step (a): $i=5$

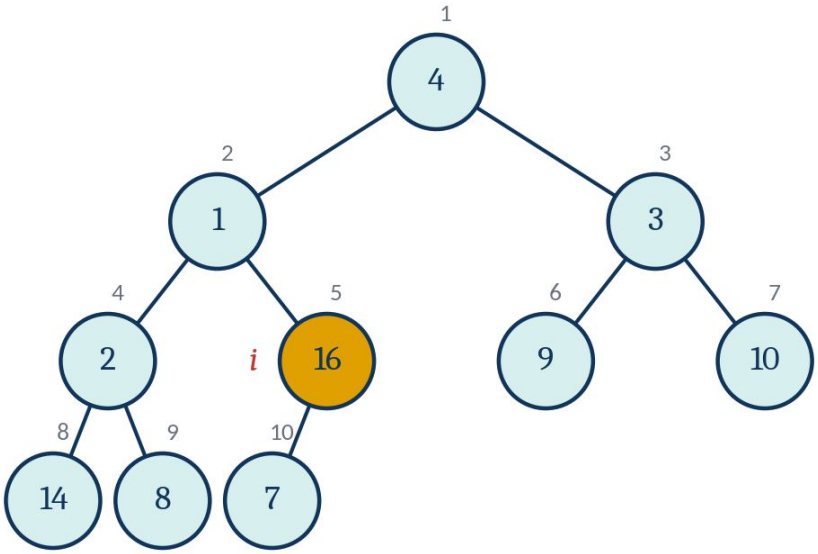
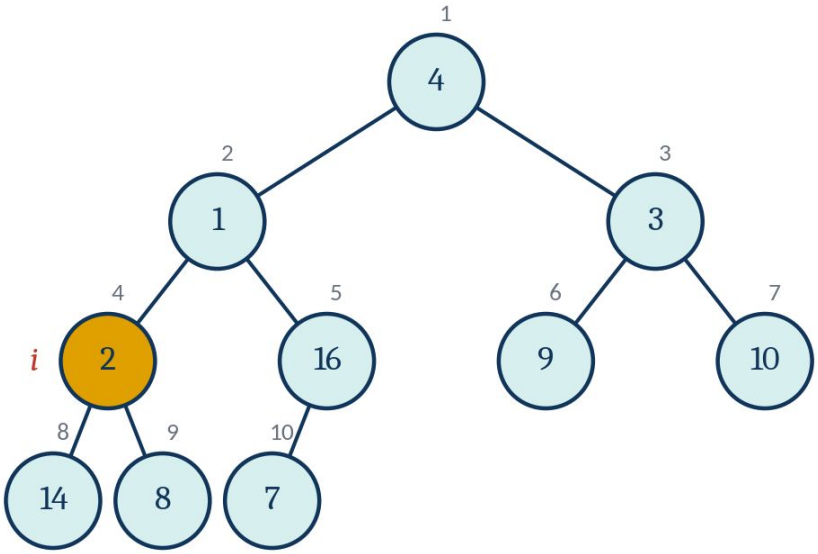


Figure 5.7 `BUILD-MAX-HEAP`, step (a): $i=5$. The array drawn as a nearly complete tree. Node 5, shaded, holds 16. Its only child is $A[10]=7$. The loop starts here because node 5 is the last node that has any children at all.

Figure 5.7: $A[5]=16$, only child $A[10]=7$; $16>7$, so no swap. The array is unchanged. Book: Section 5.5.3.

Build, Step (b): $i = 4$

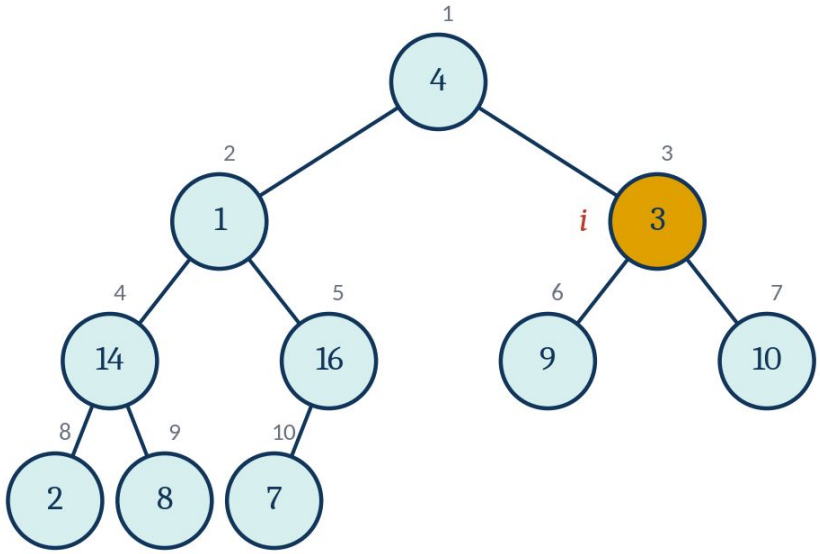


(b)

Figure 5.8 Step (b): $i = 4$. Node 4, shaded, holds 2, and its children $A[8] = 14$ and $A[9] = 8$ are both larger. The value 2 must sink.

Figure 5.8: largest of $[2, 14, 8]$ is 14 at index 8; swap, giving $[4, 1, 3, 14, 16, 9, 10, 2, 8, 7]$; node 8 is a leaf. Book: Section 5.5.3.

Build, Step (c): $i=3$



(c)

Figure 5.9 Step (c): $i=3$. Node 3, shaded, holds 3; its children are $A[6]=9$ and $A[7]=10$. The right child is the larger, so the exchange goes to the right.

Figure 5.9: largest of $[3,9,10]$ *is* 10 *at index* 7 *— the swap goes right:* $[4,1,10,14,16,9,3,2,8,7]$. *Book: Section 5.5.3.*

Build, Step (d): $i = 2$

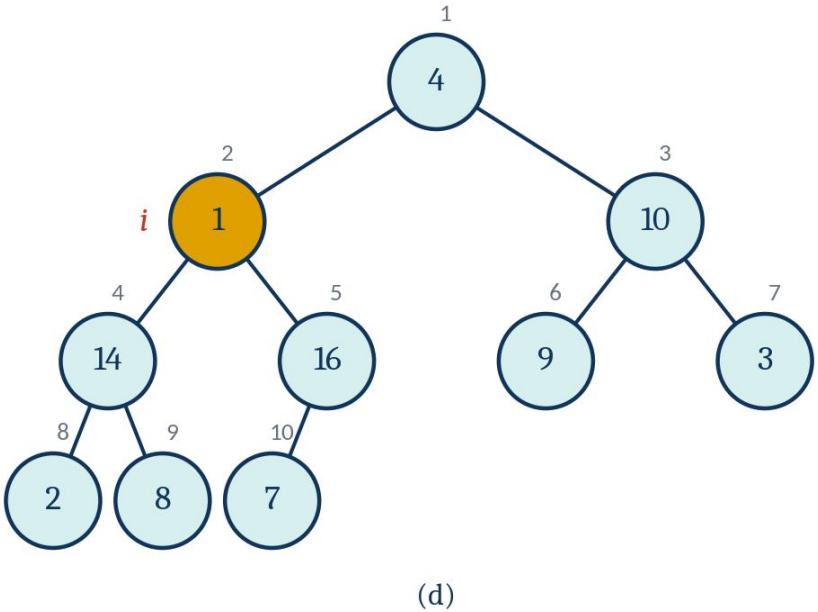


Figure 5.10 Step (d): $i = 2$. Node 2, shaded, holds 1, the smallest value in the array. It has a long way to sink: two levels, as it turns out.

Figure 5.10: 1 *swaps with* 16 *(index* 5*), then with* 7 *(index* 10*):* $[4, 16, 10, 14, 7, 9, 3, 2, 8, 1]$. *Two exchanges, two levels. Book: Section 5.5.3.*

Build, Step (e): $i=1$

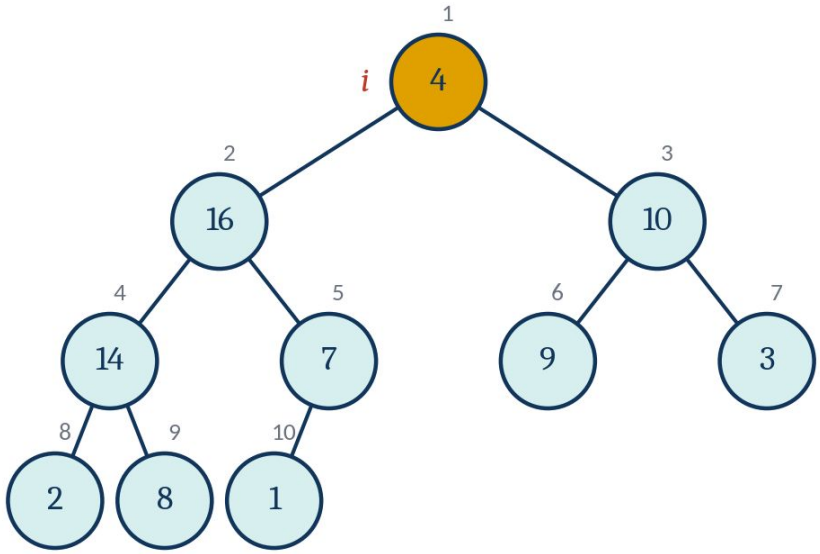
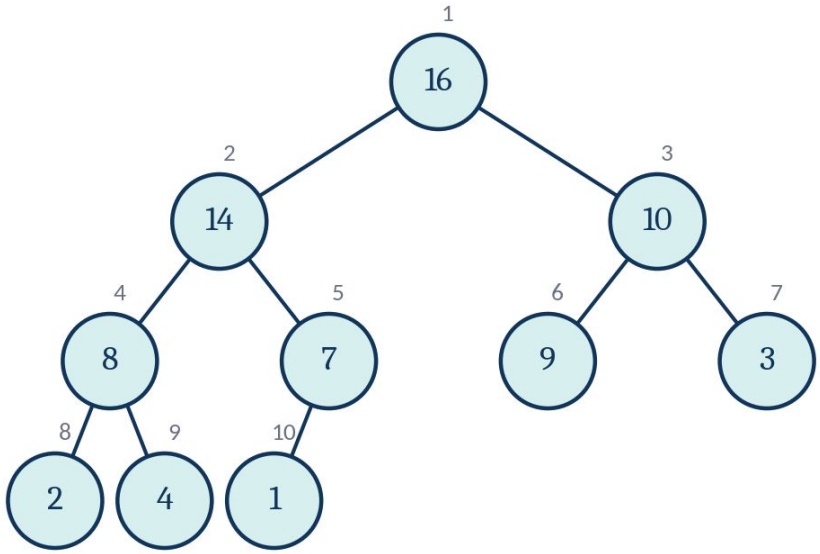


Figure 5.11 Step (e): $i=1$. The final iteration, at the root. Node 1, shaded, holds 4. Everything below it is already a valid heap, so this single `MAX-HEAPIFY` call finishes the job.

Figure 5.11: 4 *sinks* 1 → 2 → 4 → 9, *swapping with* 16, 14 *and* 8. *Three exchanges in this one call. Book: Section 5.5.3.*

Build, Step (f): the Finished Heap



(f)

Figure 5.12 Step (f): the finished max-heap $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$. Every parent dominates both of its children, and the maximum, 16, has arrived at the root.

Figure 5.12: seven exchanges in total for ten elements, and the last call did three of them. That asymmetry is the seed of the analysis. Book: Section 5.5.3.

Loose Bound $O(n \lg n)$, Tight Bound $O(n)$

Loose: $\lfloor n/2 \rfloor$ calls × $O(\lg n)$ each $= O(n \lg n)$. True, but not tight: a call on a node of height h costs $O(h)$, not $O(\lg n)$.

Steps 1–4: a call at height h costs $O(h)$; at most $\lceil n/2^{h+1} \rceil$ nodes have height h; heights run $0 \ldots \lfloor \lg n \rfloor$. Summing by height gives Equation (5.6):

$$T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} \underbrace{\left\lceil \frac{n}{2^{h+1}} \right\rceil}_{\text{how many nodes of height } h} \times \underbrace{O(h)}_{\text{cost of each}} . \tag{5.6}$$

Steps 5–8: $\lceil x \rceil \le 2x$ for $x \ge 1/2$, so $\lceil n/2^{h+1} \rceil \le n/2^h$; extend the sum to ∞; it equals 2; hence $T(n) = O\left(n \sum_h h/2^h\right) = O(2n) = O(n)$, and with $\Omega(n)$, $\Theta(n)$. ∎

The expensive calls are rare and the cheap calls are common: one node at height $\lg n$, half the nodes at height 0. Book: Sections 5.5.4 and 5.5.5.

Why $\sum_{h\ge 0} h/2^h = 2$

Start from $\sum_{i\ge 0} x^i = \frac{1}{1-x}$ for $|x|<1$. Differentiate term by term: $\sum_{i\ge 1} i\,x^{i-1} = \frac{1}{(1-x)^2}$. Multiply by x to get Equation (5.7):

$$\sum_{i=1}^{\infty} i\,x^i = \frac{x}{(1-x)^2}. \tag{5.7}$$

Put $x = 1/2$: $\frac{1/2}{(1/2)^2} = 2$. The $h=0$ term is 0, so Equation (5.8) follows:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2. \ \blacksquare \tag{5.8}$$

Measured: from $n = 10^3$ to $n = 10^7$, `BUILD-MAX-HEAP` makes about 0.74 exchanges per element at every size, while $\lg n$ climbs from 9 to 23.

Eight terms by hand already give 1.9296875. *Book: Sections 5.5.6 and 5.5.7.*

Heapsort: Selection Sort, Rebuilt

Pseudocode 5.3 HEAPSORT: build a max-heap, then repeatedly move the root to the end of the shrinking heap and repair the root.

```
HEAPSORT(A):
 1.  BUILD-MAX-HEAP(A)                    // O(n)
 2.  for i ← n downto 2                   // n − 1 times
 3.      swap A[1] ↔ A[i]                 // O(1)
 4.      heap-size ← heap-size − 1        // O(1)
 5.      MAX-HEAPIFY(A, 1)                // O(lg n)
```

After line 1 of Pseudocode 5.3, $A[1]$ is the maximum; line 3 puts it at $A[i]$ for good; line 4 shrinks the heap; line 5 repairs the root. The boundary between the heap $A[1\ldots\text{heap-size}]$ and the sorted tail moves one step left each iteration.

LOOP INVARIANT. At the start of each iteration, $A[1\ldots i]$ is a max-heap holding the i smallest elements, and $A[i{+}1\ldots n]$ holds the $n-i$ largest, sorted. At termination ($i=1$) the whole array is sorted. ∎

Heapsort is due to Williams (1964); the linear-time build is Floyd's, from the same year. Book: Sections 5.6.1 and 5.6.3.

Heapsort, Steps (a)–(c)

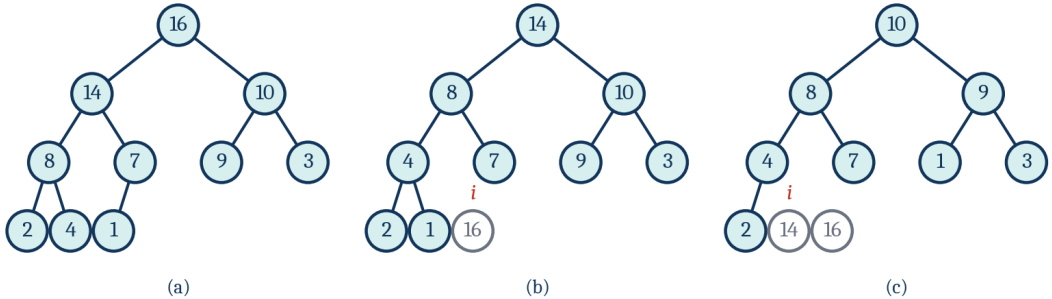


Figure 5.13 Heapsort, steps (a)–(c). Shaded nodes are still in the heap; unshaded ones have been placed in their final sorted positions and are no longer part of it. **(a)** the heap after `BUILD-MAX-HEAP`. **(b)** after exchanging 16 into position 10 and repairing the root. **(c)** after exchanging 14 into position 9 and repairing.

Figure 5.13: $i=10$ *gives* $[14,8,10,4,7,9,3,2,1 \mid 16]$; $i=9$ *gives* $[10,8,9,4,7,1,3,2 \mid 14,16]$. *The bar marks the sorted tail. Book: Section 5.6.2.*

Heapsort, Steps (d)–(f)

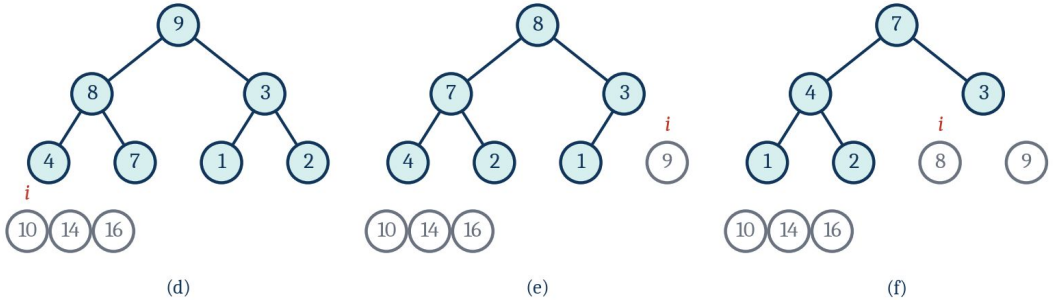


Figure 5.14 Heapsort, steps (d)–(f). The heap shrinks by one node each time and the sorted tail, drawn unshaded and laid out along the bottom, grows to meet it. **(d)** 10 placed. **(e)** 9 placed. **(f)** 8 placed.

Figure 5.14: $[9,8,3,4,7,1,2|10,14,16]$, *then* $[8,7,3,4,2,1|9,\ldots]$, *then* $[7,4,3,1,2|8,\ldots]$. *Book: Section 5.6.2.*

Heapsort, Steps (g)–(i)

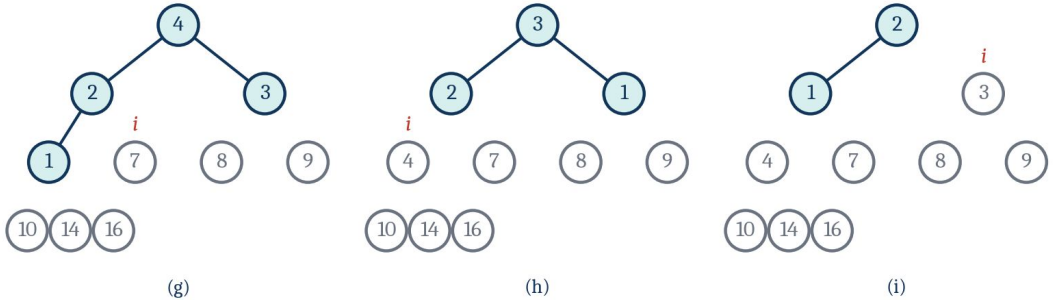


Figure 5.15 Heapsort, steps (g)–(i). With the heap down to four, three and two nodes the work per iteration has become trivial: the height is 2, then 1, then 1.

Figure 5.15: $[4,2,3,1|7,\ldots]$, *then* $[3,2,1|4,\ldots]$, *then* $[2,1|3,\ldots]$. *Book: Section 5.6.2.*

Heapsort, Steps (j)–(k)

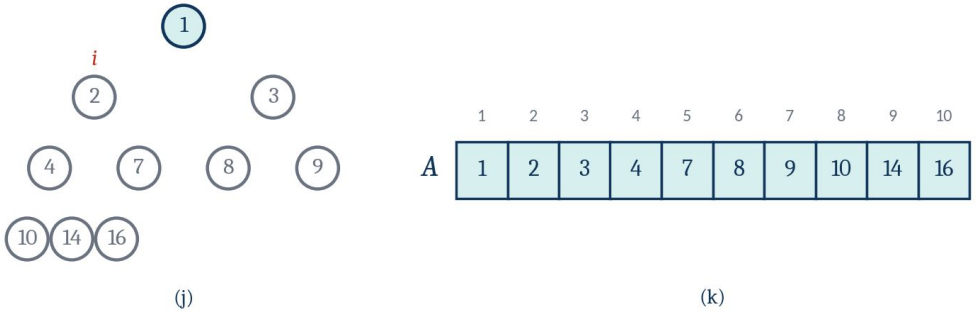


Figure 5.16 Heapsort, steps (j) and (k). (j) after the last exchange, the heap holds a single element. **(k)** the sorted array $A = [1, 2, 3, 4, 7, 8, 9, 10, 14, 16]$, in non-decreasing order.

Figure 5.16: the loop stops at $i = 2$; a one-element heap is already sorted. Result $[1, 2, 3, 4, 7, 8, 9, 10, 14, 16]$ ✔. Book: Section 5.6.2.

What Heapsort Costs, and Why It Is Not Stable

Table 5.3 The cost of Heapsort, line by line. The dominant term is the $n-1$ calls to `MAX-HEAPIFY`.

Step	Cost per execution	Number of executions	Total
`BUILD-MAX-HEAP` (line 1)	$O(n)$	1	$O(n)$
swap + shrink (lines 3–4)	$\Theta(1)$	$n-1$	$\Theta(n)$
`MAX-HEAPIFY` (line 5)	$O(\lg n)$	$n-1$	$O(n \lg n)$
Total			$O(n \lg n)$

Table 5.3: $O(n)+\Theta(n)+O(n\lg n)=O(n\lg n)$. The worst case is $\Theta(n\lg n)$ by Section 5.11; with distinct keys every case is $\Theta(n\lg n)$; if all keys are equal it is $\Theta(n)$.

NOT STABLE. Sort $\langle 1,a\rangle, \langle 1,b\rangle$ by the number: the build leaves them alone, then $i=2$ swaps them into $\langle 1,b\rangle, \langle 1,a\rangle$. Stability is lost on an array of length two.

Heapsort, summarized: $\Theta(n\lg n)$, $O(1)$ extra space, not stable — the dream of Section 5.1.2. Book: Sections 5.6.4 and 5.6.5.

Heapsort in C++

Listing 5.1 prints the array after every extraction, with a bar between the shrinking heap and the growing sorted tail.

Listing 5.1 A complete, runnable Heapsort that narrates every extraction.

```cpp
// HEAPSORT, printing the array after every extraction, with the sorted tail marked by |.
#include <iostream>
#include <vector>
#include <string>
using namespace std;

vector<int> A;
int heapSize;                                   // how much of A is still a heap
int n;                                          // how much of A exists at all

void printState(const string& label) {
    cout << label << "[";
    for (int i = 1; i <= n; ++i) {
        if (i == heapSize + 1 && heapSize < n) cout << "| ";
        cout << A[i] << (i < n ? " " : "");
    }
    cout << "]\n";
}

void maxHeapify(int i) {
    int l = 2 * i, r = 2 * i + 1, largest = i;
    if (l <= heapSize && A[l] > A[largest]) largest = l;
    if (r <= heapSize && A[r] > A[largest]) largest = r;
    if (largest != i) { swap(A[i], A[largest]); maxHeapify(largest); }
}

void buildMaxHeap() { for (int i = heapSize / 2; i >= 1; --i) maxHeapify(i); }
```

```
void heapsort() {
    heapSize = n;
    buildMaxHeap();
    printState("after BUILD-MAX-HEAP:  ");
    for (int i = n; i >= 2; --i) {
        cout << "  swap A[1]=" << A[1] << " with A[" << i << "]=" << A[i];
        swap(A[1], A[i]);
        --heapSize;                              // the element just parked is now frozen
        maxHeapify(1);
        cout << ", shrink to " << heapSize << ", heapify -> ";
        printState("");
    }
}
```

Listing 5.1 (Continued) *A complete, runnable Heapsort that narrates every extraction.*

```
int main() {
    A = {0, 4, 1, 3, 2, 16, 9, 10, 14, 8, 7};
    n = 10; heapSize = n;
    printState("input:                 ");
    heapsort();
    heapSize = n;                                  // the whole array is sorted; drop the divider
    printState("\nsorted:                ");

    bool ok = true;                                // independent check of the result
    for (int i = 2; i <= n; ++i) if (A[i - 1] > A[i]) ok = false;
    cout << "non-decreasing: " << (ok ? "yes" : "no") << "\n";
    return 0;
}
```

```
input:                [4 1 3 2 16 9 10 14 8 7]
after BUILD-MAX-HEAP:  [16 14 10 8 7 9 3 2 4 1]
  swap A[1]=16 with A[10]=1, shrink to 9, heapify -> [14 8 10 4 7 9 3 2 1 | 16]
  swap A[1]=14 with A[9]=1, shrink to 8, heapify -> [10 8 9 4 7 1 3 2 | 14 16]
  swap A[1]=10 with A[8]=2, shrink to 7, heapify -> [9 8 3 4 7 1 2 | 10 14 16]
  swap A[1]=9 with A[7]=2, shrink to 6, heapify -> [8 7 3 4 2 1 | 9 10 14 16]
  swap A[1]=8 with A[6]=1, shrink to 5, heapify -> [7 4 3 1 2 | 8 9 10 14 16]
  swap A[1]=7 with A[5]=2, shrink to 4, heapify -> [4 2 3 1 | 7 8 9 10 14 16]
  swap A[1]=4 with A[4]=1, shrink to 3, heapify -> [3 2 1 | 4 7 8 9 10 14 16]
  swap A[1]=3 with A[3]=1, shrink to 2, heapify -> [2 1 | 3 4 7 8 9 10 14 16]
  swap A[1]=2 with A[2]=1, shrink to 1, heapify -> [1 | 2 3 4 7 8 9 10 14 16]

sorted:               [1 2 3 4 7 8 9 10 14 16]
non-decreasing: yes
```

Every intermediate array matches panels (b)–(j) of the four figures. That is what correct looks like. Book: Section 5.6.6.

Priority Queues: the Heap's Real Use

DEFINITION. A **max-priority queue** keeps a set S of keyed elements and supports $Insert(S,x)$, $Maximum(S)$, $Extract\text{-}Max(S)$ and $Increase\text{-}Key(S,x,k)$ with k at least the current key. A min-priority queue mirrors it.

Pseudocode 5.4 `HEAP-MAXIMUM`: the largest key of a max-heap is always at the root.

```
HEAP-MAXIMUM(A):
 1.  if heap-size < 1 then error "heap underflow"
 2.  return A[1]
```

Table 5.4 The cost of the four priority-queue operations under three implementations. The heap is the only one with no $O(n)$ entry.

Operation	Heap-based queue	Unsorted array	Sorted array
`INSERT`	$O(\lg n)$	$O(1)$	$O(n)$
`MAXIMUM`	$O(1)$	$O(n)$	$O(1)$
`EXTRACT-MAX`	$O(\lg n)$	$O(n)$	$O(1)$
`INCREASE-KEY`	$O(\lg n)$	$O(1)$	$O(n)$

Pseudocode 5.4 is one array read. Table 5.4: a million inserts plus a million extracts cost about 10^{12} operations on an unsorted array, but 4×10^7 on a heap.

Uses: OS scheduling, event simulation, Dijkstra and Prim, Huffman coding, crawl scheduling. In C++, `std::priority_queue<int>` *is a max-heap; Java's* `PriorityQueue` *and Python's* `heapq` *are min-heaps (Python 3.14 adds separate* `*_max` *functions for max-heaps). Book: Sections 5.7.1–5.7.3, 5.7.7 and 5.7.9.*

EXTRACT-MAX

Pseudocode 5.5 HEAP-EXTRACT-MAX in pseudocode. Save the root; move the last element into the root; shrink the heap by one; call HEAPIFY on the root to restore order; return the saved value.

```
HEAP-EXTRACT-MAX(A):                        // remove and return the largest element of A
 1.  max ← A[1]
 2.  A[1] ← A[heap-size]
 3.  heap-size ← heap-size − 1
 4.  HEAPIFY(A, 1)                           // remake the heap
 5.  return max
```

Pseudocode 5.5 on $[16,14,10,8,7,9,3,2,4,1]$*: save* 16*;* $A[1] \leftarrow 1$*; shrink to* 9*;* 1 *sinks past* 14*,* 8*,* 4*; return* 16*, leaving* $[14,8,10,4,7,9,3,2,1]$*.* $O(\lg n)$*. Book: Section 5.7.4.*

`INCREASE-KEY`: the Value Floats Up

Pseudocode 5.6 `HEAP-INCREASE-KEY`: raise the key at position i, then float it up past every smaller parent.

```
HEAP-INCREASE-KEY(A, i, key):
 1.  if key < A[i] then error "new key is smaller than current key"
 2.  A[i] ← key
 3.  while i > 1 and A[PARENT(i)] < A[i]
 4.      swap A[i] ↔ A[PARENT(i)]
 5.      i ← PARENT(i)
```

Pseudocode 5.7 `MAX-HEAP-INSERT`: append a slot holding $-\infty$, then raise it to the new key.

```
MAX-HEAP-INSERT(A, key):
 1.  heap-size ← heap-size + 1
 2.  A[heap-size] ← −∞
 3.  HEAP-INCREASE-KEY(A, heap-size, key)
```

WORKED EXAMPLE. Heap $[10,8,9,4,7,3,2,1]$; raise position 8 from 1 to 20. Parent $A[4]=4<20$: swap, $i=4$. Parent $A[2]=8<20$: swap, $i=2$. Parent $A[1]=10<20$: swap, $i=1$. Stop. Result $[20,10,9,8,7,3,2,4]$.

Pseudocode 5.6 is the mirror of `MAX-HEAPIFY`: one leaf-to-root walk, $O(\lg n)$. Pseudocode 5.7 appends $-\infty$ and raises it — insertion needs no new code. Book: Sections 5.7.5 and 5.7.6.

`INSERT` Is `INCREASE-KEY` in Disguise

Pseudocode 5.8 `HEAP-INSERT` in pseudocode. Grow the heap by one slot at the end, then bubble the new key upward, moving each beaten parent down into the child's place, until a parent wins or the root is reached.

```
HEAP-INSERT(A, key):
 1.  heap-size ← heap-size + 1
 2.  i ← heap-size
 3.  while i > 1 and A[PARENT(i)] < key
 4.      A[i] ← A[PARENT(i)]              // shift the beaten parent down
 5.      i ← PARENT(i)
 6.  A[i] ← key                           // write the key once, at the end
```

Pseudocode 5.8 shifts each beaten parent down and writes the key once, at the end: the same result with half the memory writes. Book: Section 5.7.6.

Quicksort: Divide and Conquer, Work Moved

DEFINITION. Quicksort chooses a **pivot**, rearranges $A[p \ldots r]$ so that every element no greater than the pivot comes before every element no smaller than it, and sorts the two pieces recursively. There is no combine step.

Table 5.5 Merge Sort and Quicksort are mirror images. Each spends $\Theta(n)$ per level; they differ in *where* they spend it, and in whether the split is guaranteed.

	Merge Sort	Quicksort
Divide	trivial — compute the midpoint, $\Theta(1)$	expensive — partition, $\Theta(n)$
Conquer	two recursive calls	two recursive calls
Combine	expensive — merge, $\Theta(n)$	trivial — nothing, $\Theta(1)$
Split sizes	always exactly half and half	whatever the data gives you
Extra memory	$\Theta(n)$ for the merge buffer	none

Table 5.5: Merge Sort *buys* its even split with a merge buffer; Quicksort refuses to pay and accepts whatever split the data gives.

Tony Hoare invented Quicksort in 1959 and published it in 1962. Book: Sections 5.8.1 and 5.8.2.

Lomuto's PARTITION: Four Regions

Pseudocode 5.9 Lomuto's PARTITION: one left-to-right scan that grows a region of elements no larger than the pivot.

```
PARTITION(A, p, r):
 1.  x ← A[r]                        // the pivot is the last element
 2.  i ← p − 1                       // i marks the end of the "≤ x" region
 3.  for j ← p to r − 1
 4.      if A[j] ≤ x
 5.          i ← i + 1
 6.          swap A[i] ↔ A[j]
 7.  swap A[i+1] ↔ A[r]              // drop the pivot into the gap
 8.  return i + 1
```

LOOP INVARIANT. At the start of each iteration: $A[k] \le x$ for $p \le k \le i$; $A[k] > x$ for $i+1 \le k \le j-1$; $A[r] = x$. Indices $j \ldots r-1$ are unexamined. At termination line 7 drops the pivot into the gap, establishing Equation (5.9).

$$A[k] \le A[q] \text{ for } p \le k < q,\ A[k] > A[q] \text{ for } q < k \le r \tag{5.9}$$

Pseudocode 5.9 does one pass of $r - p$ iterations: $\Theta(n)$. Book: Section 5.8.3.

A Lomuto Partition, by Hand

$A=[2,8,7,1,3,5,6,4]$, pivot $x=A[8]=4$, $i=0$. Table 5.6 carries the loop.

Table 5.6 Lomuto's `PARTITION` on $[2,8,7,1,3,5,6,4]$ with pivot 4. The underlined prefix is region 1 — the elements known to be ≤ 4.

j	$A[j]$	$A[j]\le 4$?	Action	Array after
1	2	yes	$i\to 1$; swap $A[1]\leftrightarrow A[1]$	$[\underline{2},8,7,1,3,5,6,$
2	8	no	nothing	$[\underline{2},8,7,1,3,5,6,$
3	7	no	nothing	$[\underline{2},8,7,1,3,5,6,$
4	1	yes	$i\to 2$; swap $A[2]\leftrightarrow A[4]$	$[\underline{2,1},7,8,3,5,6,$
5	3	yes	$i\to 3$; swap $A[3]\leftrightarrow A[5]$	$[\underline{2,1,3},8,7,5,6,$
6	5	no	nothing	$[\underline{2,1,3},8,7,5,6,$
7	6	no	nothing	$[\underline{2,1,3},8,7,5,6,$

Then $i=3$, so line 7 swaps $A[4]\leftrightarrow A[8]$: $A=[2,1,3,4,7,5,6,8]$, $q=4$. Left $[2,1,3]\le 4$ ✔️, right $[7,5,6,8]>4$ ✔️.

The pivot 4 *is now in its final sorted position and never moves again. Book: Section 5.8.4.*

Hoare's `PARTITION`, Traced

```
PARTITION(A, p, r)
1   x ← A[r]
2   i ← p − 1
3   j ← r + 1
4   while TRUE
5       repeat j ← j − 1
6           until A[j] ≤ x
7       repeat i ← i + 1
8           until A[i] ≥ x
9       if i < j
10          exchange A[i] ↔ A[j]
11      else return j
```

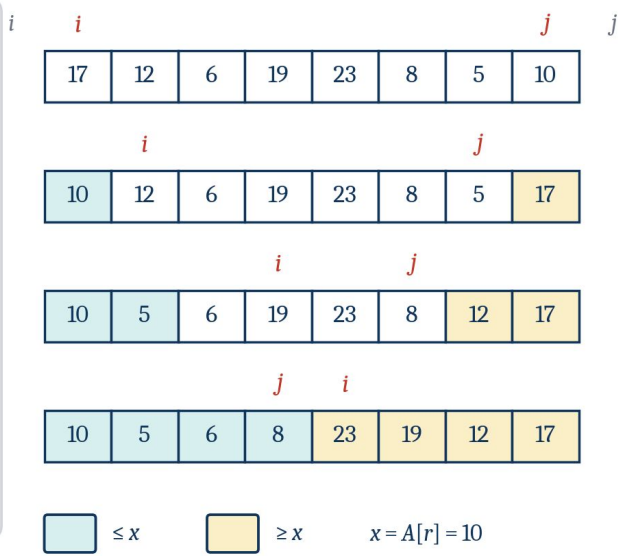


Figure 5.17 Hoare's `PARTITION`, with a full trace on $[17, 12, 6, 19, 23, 8, 5, 10]$. Teal marks the elements known to be $\le x$, amber those known to be $\ge x$. The pointers i and j begin just outside the array and close in; when they cross, the partition is finished and j is returned.

Figure 5.17, pivot $x = A[8] = 10$: swaps give $[10, 12, \ldots, 17]$, $[10, 5, 6, 19, 23, 8, 12, 17]$, $[10, 5, 6, 8, 23, 19, 12, 17]$; then $i = 5 > j = 4$: return 4. Book: Sections 5.8.5 and 5.8.6.

Hoare's Scheme, and Its Pivot Trap

Pseudocode 5.10 Hoare's PARTITION: two pointers close in from the ends and exchange misplaced pairs until they cross.

```
PARTITION(A, p, r):                    // Hoare, 1961
 1.  x ← A[r]                          // the pivot VALUE (the pivot slot does not move)
 2.  i ← p − 1
 3.  j ← r + 1
 4.  while TRUE
 5.      repeat j ← j − 1 until A[j] ≤ x
 6.      repeat i ← i + 1 until A[i] ≥ x
 7.      if i < j then swap A[i] ↔ A[j]
 8.      else return j
```

Lomuto returns the **pivot's final position**: recurse on $(p, q-1)$ and $(q+1, r)$. Hoare's Pseudocode 5.10 returns a **boundary**: recurse on (p, q) and $(q+1, r)$.

THE PIVOT TRAP. With $x = A[r]$ and the recursion (p, q), a maximum at $A[r]$ makes both pointers stop at r, so $q = r$ and the left call gets the same subarray for ever. Take $x = A[p]$, as CLRS does: with $x = A[1] = 17$ the partition returns $j = 5$, left $[10, 12, 6, 5, 8]$, right $[23, 19, 17]$.

Hoare's scheme makes about three times fewer exchanges and copes with equal keys; learn Lomuto first. Book: Sections 5.8.5 and 5.8.6.

Quicksort Itself — Four Lines

Pseudocode 5.11 `QUICKSORT` with Lomuto's partition: partition, then sort the two sides of the pivot recursively.

```
QUICKSORT(A, p, r):
 1.  if p < r
 2.      q ← PARTITION(A, p, r)
 3.      QUICKSORT(A, p, q − 1)          // Lomuto: the pivot at q is already final
 4.      QUICKSORT(A, q + 1, r)
```

The initial call of Pseudocode 5.11 is `QUICKSORT(A, 1, n)`. If $p \geq r$ the subarray has at most one element and is sorted; each call receives a strictly shorter subarray, so the recursion ends.

THE OFF-BY-ONE. With Lomuto's partition, writing `QUICKSORT(A, p, q)` recurses for ever when the pivot is the maximum. **Lomuto excludes the pivot from both calls; Hoare excludes it from neither.**

Write the partition and its two recursive calls at the same time, never separately. Book: Section 5.8.7.

Quicksort, Traced Completely

Listing 5.2 sorts $[2,8,7,1,3,5,6,4]$ and prints the array after each partition; brackets mark the current subarray and indentation shows the depth.

Listing 5.2 Quicksort with Lomuto partitioning, narrating the recursion. Indentation shows the depth of the call.

```cpp
// QUICKSORT with Lomuto partitioning, printing the whole array after every partition.
#include <iostream>
#include <vector>
#include <string>
using namespace std;

vector<int> A;
int calls = 0;

void showWindow(int p, int r, const string& note) {
    cout << "  ";
    for (int k = 0; k < (int)A.size(); ++k) {
        if (k == p) cout << "[";
        cout << A[k];
        if (k == r) cout << "]"; else cout << " ";
    }
    cout << "  " << note << "\n";
}

int partition(int p, int r) {
    int x = A[r], i = p - 1;
    for (int j = p; j <= r - 1; ++j)
        if (A[j] <= x) swap(A[++i], A[j]);
    swap(A[i + 1], A[r]);
    return i + 1;
}

void quicksort(int p, int r, int depth) {
```

```
    if (p >= r) {                                   // zero or one element: nothing to do
        if (p == r) cout << string(2 * depth + 2, ' ') << "QUICKSORT(" << p << "," << r
                         << "): one element, return\n";
        return;
    }
    ++calls;
    cout << string(2 * depth + 2, ' ') << "QUICKSORT(" << p << "," << r
         << "): pivot = A[" << r << "] = " << A[r] << "\n";
    int q = partition(p, r);
    cout << string(2 * depth + 2, ' ');
    showWindow(p, r, "after PARTITION, pivot " + to_string(A[q]) + " is final at index " + to_string(q));
    quicksort(p, q - 1, depth + 1);
    quicksort(q + 1, r, depth + 1);
}
```

Listing 5.2 (Continued) *Quicksort with Lomuto partitioning, narrating the recursion. Indentation shows the depth of the call.*

```
int main() {
    A = {2, 8, 7, 1, 3, 5, 6, 4};
    cout << "input: ";
    for (int x : A) cout << x << " ";
    cout << "\n\n";
    quicksort(0, (int)A.size() - 1, 0);
    cout << "\nsorted: ";
    for (int x : A) cout << x << " ";
    cout << "\npartition calls: " << calls << "\n";
    return 0;
}
```

```
input: 2 8 7 1 3 5 6 4

  QUICKSORT(0,7): pivot = A[7] = 4
    [2 1 3 4 7 5 6 8]  after PARTITION, pivot 4 is final at index 3
    QUICKSORT(0,2): pivot = A[2] = 3
      [2 1 3]4 7 5 6 8   after PARTITION, pivot 3 is final at index 2
      QUICKSORT(0,1): pivot = A[1] = 1
        [1 2]3 4 7 5 6 8   after PARTITION, pivot 1 is final at index 0
        QUICKSORT(1,1): one element, return
    QUICKSORT(4,7): pivot = A[7] = 8
      1 2 3 4 [7 5 6 8]  after PARTITION, pivot 8 is final at index 7
      QUICKSORT(4,6): pivot = A[6] = 6
        1 2 3 4 [5 6 7]8   after PARTITION, pivot 6 is final at index 5
        QUICKSORT(4,4): one element, return
        QUICKSORT(6,6): one element, return

sorted: 1 2 3 4 5 6 7 8
partition calls: 5
```

Five partitions place five pivots permanently; the other three elements fall out as one-element base cases. A pivot never moves again. Book: Section 5.8.8.

Everything Depends on the Split

`PARTITION` costs $\Theta(n)$ whatever happens. With k elements on the left, the general recurrence is Equation (5.10):

$$T(n) = T(k) + T(n-k-1) + \Theta(n), \tag{5.10}$$

Best case, the pivot always the median, gives Equation (5.11) — Merge Sort's recurrence; Case 2 of the Master Theorem gives $\Theta(n \lg n)$.

$$T(n) = 2T(n/2) + \Theta(n). \tag{5.11}$$

Worst case, $k = 0$ every time, gives Equation (5.12), which unrolls to $c \cdot n(n+1)/2 = \Theta(n^2)$.

$$T(n) = T(0) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n), \tag{5.12}$$

Substitution with the guess $T(n) \le c\,n^2$ confirms the worst case is never worse than $\Theta(n^2)$. Book: Sections 5.9.1–5.9.3.

The Best Case: Every Level Costs n

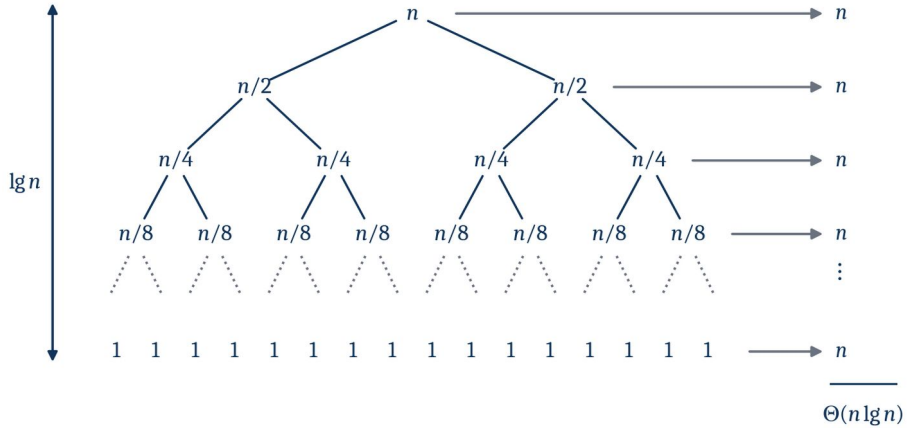


Figure 5.18 The best-case recursion tree for Quicksort. Each level's subproblem sizes halve — n, then $n/2$ twice, then $n/4$ four times — but the *total* work on each level stays n, because partitioning 2^d subarrays of size $n/2^d$ costs $2^d \times n/2^d = n$. There are $\lg n + 1$ levels, so the total is $\Theta(n \lg n)$.

Figure 5.18: 2^d subarrays of size $n/2^d$ cost n per level, over $\lg n + 1$ levels. The same picture explains every other case. Book: Section 5.9.2.

The Worst Case, Unrolled

$$T(n) = T(1) + T(n-1) + \Theta(n)$$

$$= \quad T(n-1) + \Theta(n)$$

$$= \quad \sum_{k=1}^{n} \Theta(k)$$

$$= \quad \Theta\left(\sum_{k=1}^{n} k \right)$$

$$= \quad \Theta(n^2)$$

Figure 5.19 The worst-case recurrence unrolled: $T(n) = T(1) + T(n-1) + \Theta(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$.

Figure 5.19 uses the arithmetic series $\sum_{k=1}^{n} k = n(n+1)/2$ *from Chapter 3. Book: Section 5.9.3.*

The Worst Case: a Tree That Is Not a Tree

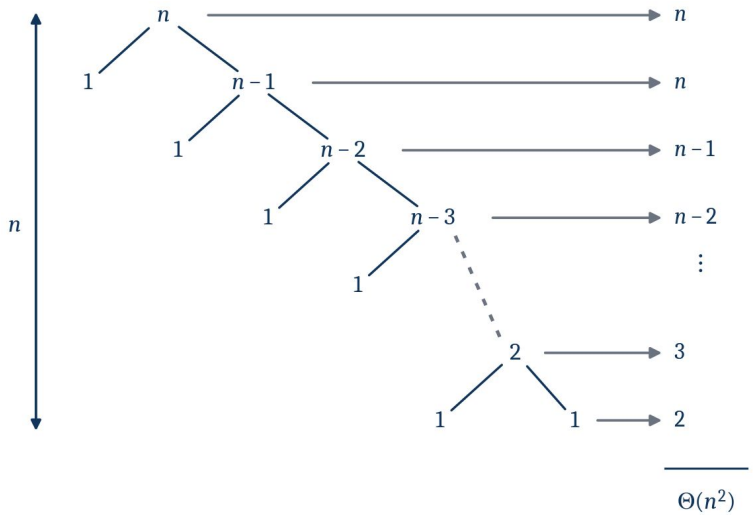


Figure 5.20 The worst-case recursion tree. Instead of branching, the tree degenerates into a single chain of length n — each node spawns one child of size one less and one child of size 0. The level costs are $n, n, n-1, n-2, \ldots, 2$, and summing them gives $\Theta(n^2)$.

*Figure 5.20: with Lomuto's last-element pivot this happens on **sorted** input — Insertion Sort's best case is naive Quicksort's worst. Book: Section 5.9.3.*

Even a $1:9$ Split Is Fine

A split that is always $10\ \%\ /\ 90\ \%$ gives Equation (5.13):

$$T(n)=T(n/10)+T(9n/10)+\Theta(n). \tag{5.13}$$

Every complete level costs exactly n. The shortest path ends after $\log_{10} n \approx 0.30 \lg n$ levels, the longest after $\log_{10/9} n \approx 6.58 \lg n$. Both are $\Theta(n \lg n)$.

> **THE CRUCIAL FACT. Any** split in a constant ratio gives $\Theta(n \lg n)$. Only the degenerate $1:(n-1)$ split, with a fixed *number* rather than a fixed *fraction* on the small side, gives $\Theta(n^2)$.

For $n=10^6$: a $99:1$ split every time costs about 1.4×10^9 operations; the degenerate split costs 5×10^{11}. Book: Section 5.9.4.

The $1:9$ Recursion Tree

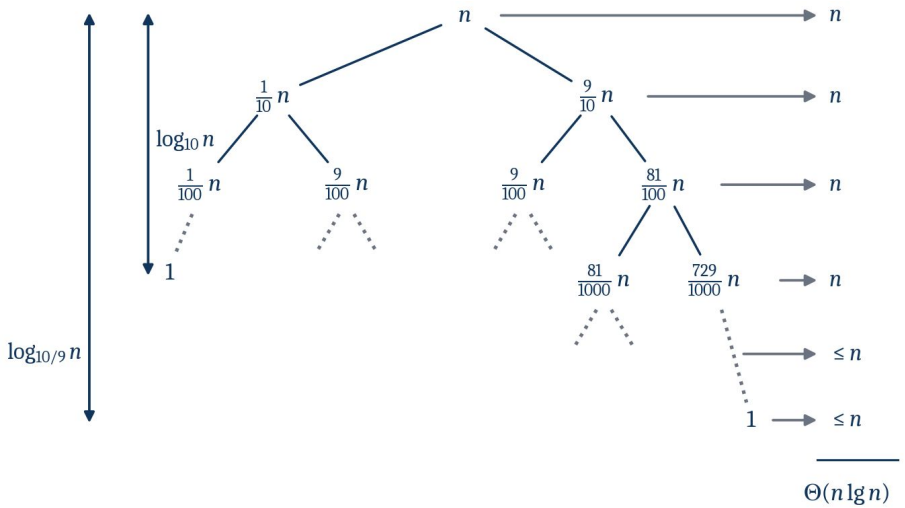


Figure 5.21 The recursion tree for a constant $1:9$ split. The left spine, whose sizes shrink by a factor of ten each level, bottoms out after $\log_{10} n$ levels. The right spine, whose sizes shrink only by a factor of $10/9$, survives to depth $\log_{10/9} n$. Between those two depths some levels are incomplete and cost less than n; above the shallower depth every level costs exactly n. The total is therefore between $n \log_{10} n$ and $n \log_{10/9} n$, and both are $\Theta(n \lg n)$.

Figure 5.21: lopsidedness costs a constant factor of about twenty-two, and nothing else. Book: Section 5.9.4.

Alternating Luck, Derived

$$L(n) = 2U(n/2) + \Theta(n) \qquad \text{lucky}$$

$$U(n) = L(n-1) + \Theta(n) \qquad \text{unlucky}$$

we consequently get

$$L(n) = 2(L(n/2-1) + \Theta(n/2)) + \Theta(n)$$

$$= \quad 2L(n/2-1) + \Theta(n)$$

$$= \quad \Theta(n \lg n)$$

Figure 5.22 The alternating-luck analysis. With $L(n)=2U(n/2)+\Theta(n)$ for a lucky level and $U(n)=L(n-1)+\Theta(n)$ for an unlucky one, substituting the second into the first gives $L(n)=2(L(n/2-1)+\Theta(n/2))+\Theta(n)=2L(n/2-1)+\Theta(n)=\Theta(n\lg n)$.

Figure 5.22: let luck alternate — $L(n)$ for a lucky (even) level, $U(n)$ for an unlucky ($1:n-1$) one. Substituting one recurrence into the other gives Merge Sort's recurrence again, so bad splits are absorbed by the good ones that follow. Book: Section 5.9.5.

Unlucky, Then Lucky

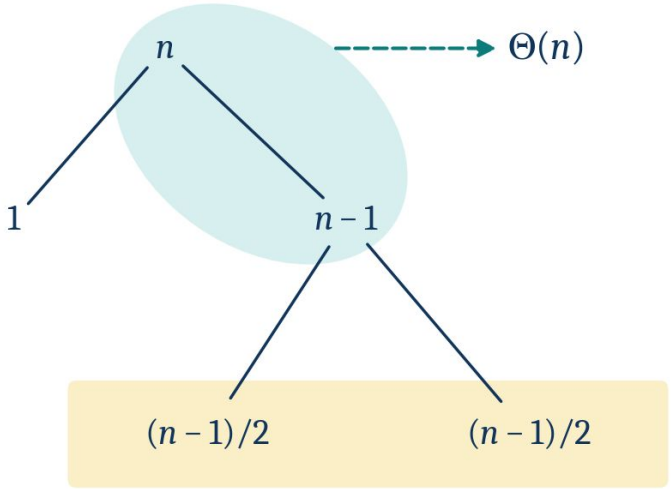


Figure 5.23 An unlucky split followed by a lucky one. The unlucky partition of n elements yields a piece of size 1 and a piece of size $n-1$; the lucky partition of that piece yields two pieces of size $(n-1)/2$. The shaded region shows the combined cost, $\Theta(n)+\Theta(n-1)=\Theta(n)$.

Figure 5.23: two levels of $\Theta(n)$ work end in two pieces of size about $(n-1)/2$. Book: Section 5.9.5.

One Lucky Split, for Comparison

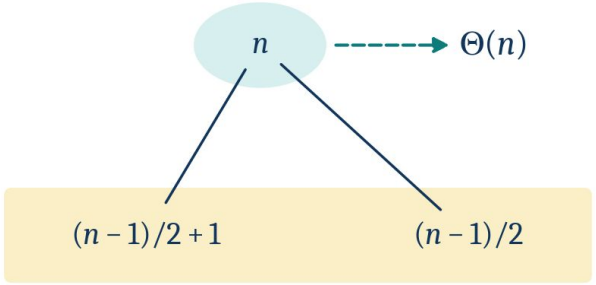


Figure 5.24 A single lucky split, for comparison: one partition of n elements, costing $\Theta(n)$, yielding two pieces of size about $(n-1)/2$.

Figure 5.24 ends in the same place as Figure 5.23 at the same cost to within a constant factor: an unlucky split does not double the running time, it adds one extra level. Book: Section 5.9.5.

The Expected Number of Comparisons

Rename the elements $z_1 < z_2 < \cdots < z_n$ and let $Z_{ij} = \{z_i, \ldots, z_j\}$. **Two elements are compared at most once** — only when one of them is the pivot.

z_i and z_j are compared exactly when the first pivot drawn from Z_{ij} is z_i or z_j. With a uniformly random pivot this gives Equation (5.14):

$$\Pr\{z_i \text{ is compared with } z_j\} = \frac{2}{j-i+1}. \tag{5.14}$$

Summing by linearity of expectation, $E[X] < 2(n-1)H_n = O(n \lg n)$; the exact value is Equation (5.15):

$$E[X] = 2(n+1)H_n - 4n \approx 2n \ln n - 2.85n \approx 1.386 n \lg n. \tag{5.15}$$

Measured at $n = 4000$: sorted input with a fixed pivot makes exactly $n(n-1)/2 = 7,998,000$ comparisons; with a random pivot, $57,639$ — against a prediction of $54,989$. Book: Sections 5.9.5 and 5.9.6.

Randomized Quicksort: Two Lines

Pseudocode 5.12 `RANDOMIZED-PARTITION` and `RANDOMIZED-QUICKSORT`: swap a uniformly random element into the pivot slot, then partition as before.

```
RANDOMIZED-PARTITION(A, p, r):
 1.  i ← RANDOM(p, r)                    // a uniformly random index in [p, r]
 2.  swap A[r] ↔ A[i]
 3.  return PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r):
 1.  if p < r
 2.      q ← RANDOMIZED-PARTITION(A, p, r)
 3.      RANDOMIZED-QUICKSORT(A, p, q − 1)
 4.      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Pseudocode 5.12 makes the running time independent of the input order: no specific input triggers the worst case. The worst case can still occur — by bad luck — with probability about Equation (5.16), roughly 10^{-128} for $n=100$.

$$\frac{2}{n}\cdot\frac{2}{n-1}\cdots\frac{2}{2}=\frac{2^{n-1}}{n!}. \tag{5.16}$$

Other defenses: median-of-three, dual-pivot (Java), **Introsort** (C++ `std::sort`: Quicksort, falling back to Heapsort past depth $2\lg n$, finished by Insertion Sort), and three-way partitioning for many equal keys.

We have not made the algorithm faster; we have made it predictable. Book: Sections 5.10.1–5.10.3.

Recurse on the Smaller Side

Naive Quicksort on sorted input recurses n deep and can exhaust the call stack before the $\Theta(n^2)$ time is noticed.

Pseudocode 5.13 `QUICKSORT-TAIL`: recurse only on the smaller piece and loop on the larger, so that the stack depth stays $O(\lg n)$.

```
QUICKSORT-TAIL(A, p, r):
 1.  while p < r
 2.      q ← PARTITION(A, p, r)
 3.      if (q − p) < (r − q)                    // the LEFT piece is smaller
 4.          QUICKSORT-TAIL(A, p, q − 1)         //   recurse on it ...
 5.          p ← q + 1                           //   ... and loop on the right
 6.      else                                    // the RIGHT piece is smaller
 7.          QUICKSORT-TAIL(A, q + 1, r)         //   recurse on it ...
 8.          r ← q − 1                           //   ... and loop on the left
```

THEOREM. Pseudocode 5.13 uses $O(\lg n)$ stack space. *Proof.* Each real recursive call is on the smaller piece, at most half its caller's size, so the depth d satisfies $n/2^d \geq 1$, that is, $d \leq \lg n$. ∎

Measured at $n = 64,000$: naive depth $64,000$ on sorted input, tail-recursive depth 2; on shuffled input 39 against $11 \leq \lg n = 15$. Book: Section 5.10.4.

The $\Omega(n \lg n)$ Floor

A **comparison sort** learns about its input only through questions "is $a_i \le a_j$?". For fixed n its run is a **decision tree**: internal nodes are comparisons, leaves are output permutations.

It needs at least $n!$ leaves (one per ordering), and a tree of height h has at most 2^h leaves — Equation (5.17):

$$n! \le \ell \le 2^h. \tag{5.17}$$

So $h \ge \lg(n!)$, and Stirling's formula gives Equation (5.18):

$$\lg(n!) \ge \lg\left(\left(\frac{n}{e}\right)^n\right) = n\lg n - n\lg e = n\lg n - 1.443\,n = \Omega(n\lg n). \tag{5.18}$$

Counting Sort and Radix Sort beat this by not being comparison sorts. For $n=3$: $\lg 6 = 2.585$, so three comparisons are needed. Book: Section 5.11.

Choosing a Sort: the Honest Comparison

Table 5.7 Every sorting algorithm in the course so far, with Θ bounds on the number of comparisons (Heapsort's best case assumes distinct keys). The space column counts extra memory beyond the input array.

Algorithm	Worst	Average	Best	Extra space	Stable?	In-place?	CLRS	Chapter
Insertion Sort	n^2	n^2	n	1	Yes	Yes	Ch. 2	2
Selection Sort	n^2	n^2	n^2	1	No	Yes	Ch. 2	2
Bubble Sort	n^2	n^2	n	1	Yes	Yes	Ch. 2	2
Merge Sort	$n \lg n$	$n \lg n$	$n \lg n$	n	Yes	No	Ch. 2	3
Heapsort	$n \lg n$	$n \lg n$	$n \lg n$	1	No	Yes	Ch. 6	5
Quicksort	n^2	$n \lg n$	$n \lg n$	$\lg n$	No	Yes	Ch. 7	5

Stability (from Table 5.7): Insertion and Merge Sort keep `1b 1e`, `2d 2g`, `3a 3c 3f`; Heapsort reverses every group; Quicksort gives `3c 3f 3a`. Tie-break on the original index to make any sort stable.

Cache: Quicksort scans memory in order; Heapsort jumps $i \to 2i \to 4i$. Measured at $n = 10^6$: Quicksort 66.79 ms, Merge Sort 97.02 ms, Heapsort 122.72 ms — yet at $n = 10^5$, which fits in cache, Heapsort was fastest.

Use the library's sort unless you have a reason not to; Heapsort for a guarantee, Merge Sort for stability, Insertion Sort for tiny or nearly sorted arrays, a heap for "the largest remaining". Book: Section 5.12.

Profitina in Practice: the Laundry Queue

Profitina's crawl queue is a max-priority queue keyed by urgency (`EXTRACT-MAX`, `INCREASE-KEY`, `INSERT`); its re-scan scheduler is a min-priority queue; its dashboard sorts.

Profitina Laundry asks "what do we wash next?" with the score Equation (5.19):

$$\text{urgency}(o) = (48 - \text{dueHour}) \times 10 + 40 \cdot [o \text{ is express}] - \text{kg}. \tag{5.19}$$

`PL-0344` is due at 13:00, the earliest, yet it is washed **third**: express `PL-0342` scores $340 + 40 - 3 = 377 > 346$. One hour is worth 10 points and express is worth 40.

The heap does not decide anything; the scoring function does. Orders carry an anonymous ticket code, never a name (UU PDP 27/2022). The app advises; the user decides. Book: Section 5.13.

Common Mistakes

Believing a heap is sorted, or mixing up `heap-size` and the array length.

Reading $A[l]$ before the bounds test, or comparing the right child with $A[i]$ instead of $A[\text{largest}]$.

Building by n inserts ($O(n \lg n)$) instead of `BUILD-MAX-HEAP` ($O(n)$); counting the build loop upward.

Mismatched recursion bounds — Lomuto $(p, q-1)$, $(q+1, r)$; Hoare (p, q), $(q+1, r)$ with $x = A[p]$.

A fixed pivot on real data; forgetting the stack; believing randomization removes the worst case; assuming Quicksort or Heapsort is stable; counting in `int` instead of `long long`.

Fifteen mistakes are listed in the book, each with its fix. Book: Section 5.14.

Summary: Five Things to Remember

1. A heap is a nearly complete tree in an array: $Parent(i) = \lfloor i/2 \rfloor$, children $2i$, $2i+1$, height $\lfloor \lg n \rfloor$; the max-heap property is local, and $A[1]$ is the maximum.

2. `MAX-HEAPIFY` is $O(\lg n)$; `BUILD-MAX-HEAP` is $\Theta(n)$ because $\sum_{h \geq 0} h/2^h = 2$; Heapsort is $\Theta(n \lg n)$, in place, not stable.

3. A heap-based priority queue does `MAXIMUM` in $O(1)$ and the other three operations in $O(\lg n)$.

4. Quicksort's cost depends on the splits: $\Theta(n \lg n)$ for any constant ratio, $\Theta(n^2)$ only for $1:(n-1)$; the expected cost with a random pivot is $\approx 1.39\, n \lg n$ comparisons.

5. No comparison sort beats $\Omega(n \lg n)$; among the optimal ones, cache behavior and guarantees decide.

Changing the data structure can change the complexity class: Selection Sort and Heapsort are the same algorithm. Book: Section 5.15.

Next Time

Chapter 6 — Binary Search Trees and AVL Trees. A heap finds only the maximum; a search tree finds *any* key. Search, insertion and deletion in $O(h)$ — and AVL rotations that keep $h = O(\lg n)$.

Before then: work Exercises 5.5, 5.6, 5.11 and 5.14 by hand.

Chapters 11 and 13 will cash in today's priority queue: Prim and Dijkstra are Section 5.7 wrapped around a graph. Book: Sections 5.15.1 and 5.16.

Lecture 6 — Binary Search Trees and AVL Trees

Course: Design and Analysis of Algorithms (EF234405)

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*, Chapter 6

Lecturer: Irfan Subakti — yifana@gmail.com

Today: a structure that keeps keys in order while they are added and removed at will — and a tree that repairs its own shape after every change.
Book: Chapter 6, Sections 6.1–6.24.

What This Meeting Covers

1. **The BST property** — and why it is not the heap property.
2. **Traversals and queries** — inorder is sorted (with proof); search, minimum, maximum, successor, predecessor, all $O(h)$.
3. **Insertion and the three deletion cases**, traced on the board.
4. **When h reaches n** — and the AVL cure: the invariant, the Fibonacci proof of $h \leq 1.44 \lg n$.
5. **Rotations and the four cases** — a five-step insertion example and a deletion example.
6. **Choosing** — AVL tree, red-black tree or hash table; Profitina Laundry's pickup board.

Every operation today costs $O(h)$; the second half is about one question: how do we stop h from growing? Book: Section 6.1.

Where We Left Off

Chapter 3 gave us asymptotic notation and recurrences. Today we use them on a *data structure*, whose operations are tangled together: one operation's speed depends on the shape the others leave behind.

A **sorted array** searches in $\Theta(\lg n)$ but inserts in $\Theta(n)$. A **linked list** splices in $\Theta(1)$ but finds the place in $\Theta(n)$.

The **binary search tree** refuses to choose: each comparison discards a whole side, as in binary search, and insertion changes only two or three pointers, as in a list.

A collection that is searched constantly and changed constantly needs both at once. Book: Section 6.1.

The Vocabulary of Trees

Root, parent, child, leaf, internal node, sibling; ancestor and descendant; the **subtree rooted at** x is x with all its descendants. **Depth** counts edges from the root; **height** counts edges down to the deepest leaf.

FIX THE CONVENTION. A single node has height 0 and the empty tree has height -1. Some figures count *nodes* instead (a leaf has height 1); the two differ by one everywhere, and balance factors — differences — agree.

A chain gives $h \le n-1$; a full tree holds at most $2^{h+1}-1$ nodes, so $h \ge \lceil \lg(n+1) \rceil - 1$. Together they give Equation (6.1):

$$\lceil \lg(n+1) \rceil - 1 \le h \le n-1 . \tag{6.1}$$

Equation (6.1) is the whole chapter in one line: the left end is about 20 *steps for a million keys, the right end a million steps. Book: Section 6.2.*

The Binary-Search-Tree Property

> **DEFINITION.** A binary tree is a **binary search tree** if, for every node x, every key in the left subtree of x is **less than** the key of x, and every key in the right subtree is **greater**. No two nodes hold the same key.

The words that do the work are **every** and **subtree**: not just the children, but every key below on that side.

The classic trap. Change 40 to 55 in the tree on the next page: node 30 still passes (55>30), but node 50 fails, because 55 sits in its left subtree. A tree can pass every parent–child test and still not be a BST.

Why it is useful: one comparison at x throws away a whole side, unseen. That is binary search performed on pointers. Book: Section 6.3.

A Binary Search Tree

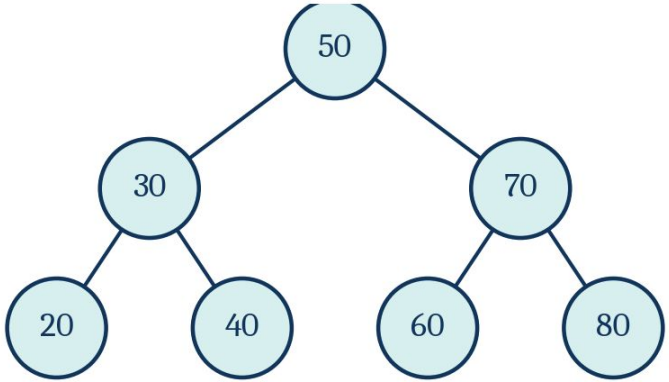


Figure 6.1 A binary search tree. The root is 50; everything in the left subtree $(30, 20, 40)$ is below 50, and everything in the right subtree $(70, 60, 80)$ is above it. The same test passes at every other node.

Figure 6.1, checked: at 50, $\{30, 20, 40\} < 50 < \{70, 60, 80\}$; *at* 30, $20 < 30 < 40$; *at* 70, $60 < 70 < 80$; *leaves always pass. Book: Section 6.3.*

BST Property ≠ Heap Property

A max-heap says only parent ≥ child — **vertical and local**. The BST property Equation (6.2) is **horizontal and global**:

$$\text{every key in left}(x) < \text{key}(x) < \text{every key in right}(x). \tag{6.2}$$

Table 6.1 The binary-search-tree property against the max-heap property. They are different rules, they support different operations, and they are stored differently.

Question	Binary search tree	Max-heap
The rule	left subtree $<$ node $<$ right subtree	parent ≥ each child
Scope of the rule	all descendants	immediate children only
Does left/right matter?	yes, they mean opposite things	no, the two children are interchangeable
Where is the smallest key?	leftmost node, $O(h)$ away	somewhere among the leaves, $O(n)$ to find
Where is the largest key?	rightmost node, $O(h)$ away	at the root, $O(1)$
Cost to search for an arbitrary key k	$O(h)$	$\Theta(n)$ — no better than scanning
What does an inorder walk print?	the keys in sorted order	nothing meaningful
Shape of the tree	whatever insertion order produced	always a complete tree, height exactly $\lfloor \lg n \rfloor$
Typical storage	nodes and pointers	a plain array, no pointers at all

Table 6.1: heaps are for repeatedly extracting the extreme; search trees are for looking things up. Using the wrong one costs a factor of n. Book: Section 6.4.

The Node in C++

Listing 6.1 declares a node — a key and two pointers — and wires up the seven nodes of Figure 6.1 by hand.

Listing 6.1 The node of a binary search tree, and the tree of Figure 6.1 assembled by hand from seven nodes. Compile with `g++ -std=c++17 -O2 -o node node.cpp`.

```
// The node of a binary search tree, and the tree of the first figure built by hand.
#include <iostream>

struct Node {
    int   key;
    Node* left;
    Node* right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

void inorder(const Node* x) {                  // left subtree, node, right subtree
    if (!x) return;                            // the empty tree: nothing to print
    inorder(x->left);
    std::cout << x->key << " ";
    inorder(x->right);
}

int main() {
    Node* root = new Node(50);                 // a whole tree is a pointer to its root
    root->left  = new Node(30);
    root->right = new Node(70);
    root->left->left   = new Node(20);
    root->left->right  = new Node(40);
    root->right->left  = new Node(60);
    root->right->right = new Node(80);

    std::cout << "root " << root->key << ", left child " << root->left->key
              << ", right child " << root->right->key << "\n";
```

```
    std::cout << "left child of 20 is " << (root->left->left->left ? "a node" : "nullptr")
              << " (the empty tree)\n";
    std::cout << "inorder: ";
    inorder(root);
    std::cout << "\n";

    Node* nodes[] = {root->left->left, root->left->right, root->right->left,
                     root->right->right, root->left, root->right, root};
    for (Node* p : nodes) delete p;            // children before parents
    return 0;
}
```

```
root 50, left child 30, right child 70
left child of 20 is nullptr (the empty tree)
inorder: 20 30 40 50 60 70 80
```

The empty tree is `nullptr`: height -1, no keys, and the base case of almost every recursion in the chapter. Book: Section 6.5.

Three Walks

DEFINITION. To traverse the subtree at x: **inorder** — left, x, right; **preorder** — x, left, right; **postorder** — left, right, x. The empty tree does nothing.

On Figure 6.1: inorder $20, 30, 40, 50, 60, 70, 80$; preorder $50, 30, 20, 40, 70, 60, 80$; postorder $20, 40, 30, 60, 80, 70, 50$.

Table 6.2 The three depth-first traversals and the jobs each one does.

Walk	Order	What it is good for
inorder	left, node, right	listing the keys in sorted order; range queries; checking that a tree really is a BST
preorder	node, left, right	copying or serializing a tree — re-inserting the preorder sequence rebuilds the identical shape
postorder	left, right, node	destroying a tree (free the children before the parent); computing a value at a node that depends on both subtrees, such as height

Table 6.2: each walk visits every node once, $\Theta(n)$ whatever the shape — the one operation here whose cost does not depend on h. Book: Section 6.6.

Why the Inorder Walk Is Sorted

Claim. An inorder traversal of a BST outputs its keys in strictly increasing order. **Proof** by induction on n. For $n=0$ the output is empty.

For $n \geq 1$ with root x: the subtrees L and R are BSTs with fewer nodes, so their walks are increasing by hypothesis. The output is walk(L), key(x), walk(R).

At the joints: every key of L is less than key(x), and every key of R is greater. So the whole output is increasing. ▰

A FREE SORT. Insert n keys, walk inorder: *tree sort*, $O(nh)$ — $\Theta(n \lg n)$ when the tree stays short, $\Theta(n^2)$ when it does not.

The height is now the difference between an excellent sorting algorithm and a terrible one. Book: Section 6.6.1.

Searching, Minimum, Maximum

TREE-SEARCH(x, k)**:** empty — fail; equal — found; smaller — go left; otherwise go right. Each comparison eliminates one entire side.

On the tree of the next page: SEARCH(10) visits $20 \to 8 \to 12 \to 10$ and succeeds. SEARCH(13) visits $20 \to 8 \to 12 \to 14$ and falls off as the left child of 14 — exactly where 13 would belong.

MINIMUM: keep going left $(20 \to 8 \to 4)$. **MAXIMUM:** keep going right $(20 \to 22)$. The minimum need not be a leaf.

Each step moves one level down, so all three cost $O(h)$. Book: Sections 6.7 and 6.8.

A Tree for Successors

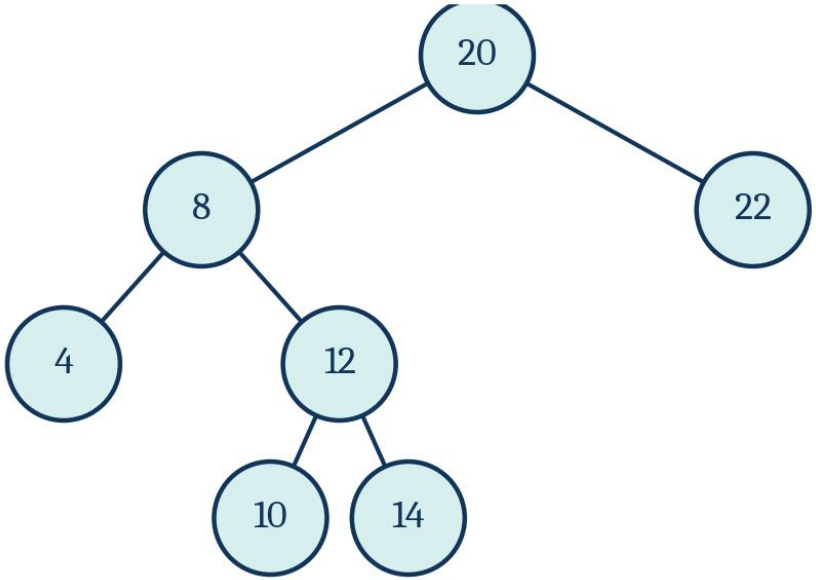


Figure 6.2 A tree for practicing inorder successors. Walking it inorder gives $4, 8, 10, 12, 14, 20, 22$, and reading that list is the easiest way to check any successor or predecessor claim.

Figure 6.2: the inorder walk $4, 8, 10, 12, 14, 20, 22$ *is the easiest check of any successor or predecessor claim. Book: Sections 6.7–6.9.*

Successor: Two Cases

Case 1 — x has a right subtree: the successor is the minimum of that subtree. For $x=8$: $12 \to 10$, so the successor is 10 ✓.

Case 2 — no right subtree: climb; while you arrive from a right child, keep climbing; the first parent reached from a left child is the successor. For $x=14$: $14 \to 12 \to 8$ (from the right), then $8 \to 20$ (from the left): 20 ✓. For $x=22$: off the top — no successor.

Case 2 needs to move upward: add a `parent` pointer, or carry the best candidate down from the root during an ordinary search.

Predecessor is the mirror image. Either way one path, down or up: $O(h)$. Book: Section 6.9.

Insertion: a Failed Search, Then a Leaf

TREE-INSERT(T, k)**:** descend as SEARCH would; when the child you want is empty, hang a new node holding k there. A new node is **always a leaf**.

`Insert(50)` on the next page: 50<100, left; 50>20, right; 50>30, right; empty — 50 becomes the right child of 30.

Check: $[50]$>30 ✓, $[30, 50]$>20 ✓, $[20, 10, 30, 50]$<100 ✓. The property survives because the path was chosen by the same comparisons.

Cost $O(h)$*. Order matters:* $1, 2, 3, 4, 5$ *gives a chain of height* 4*;* $3, 1, 4, 2, 5$ *gives height* 2*. Book: Section 6.10.*

`Insert(50)`

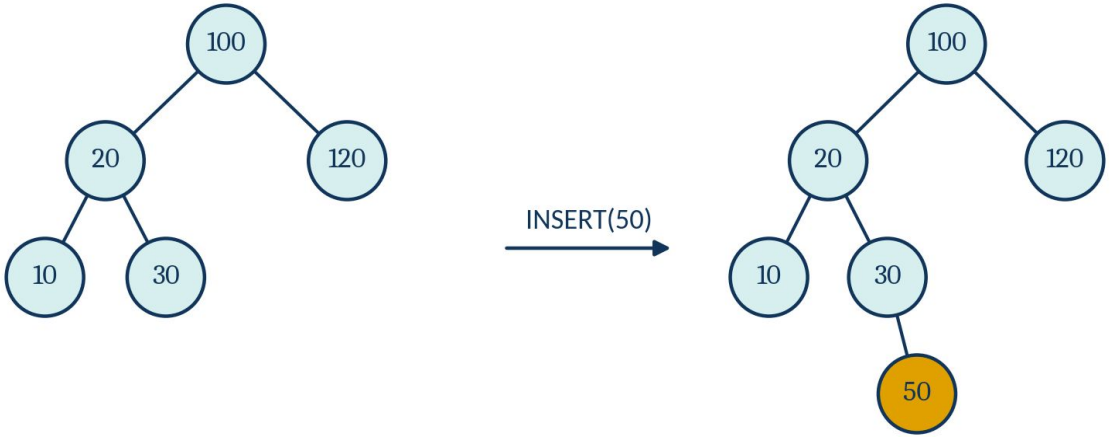


Figure 6.3 An insertion example. The key 50 is threaded down the tree — left at 100, right at 20, right at 30 — and lands as the new right child of 30, shown in amber. Nothing else in the tree moves.

Figure 6.3: nothing else in the tree moves; we only attach. Book: Section 6.10.

Deletion: Three Cases

Table 6.3 The three cases of deletion in a binary search tree, and the work each one costs beyond the initial search.

Case	Children of the doomed node	What to do	Cost
1	none	detach it from its parent and free it	$O(1)$ after the search
2	exactly one	promote that child's whole subtree into its place	$O(1)$ after the search
3	two	copy the inorder successor's key into it, then delete the successor's node (which is case 1 or 2)	one extra $O(h)$ descent

Table 6.3: the case is decided by how many children the doomed node has — none, one or two.

The next three pages delete 20, then 30, then 50 from the tree with root 50, children 30 and 70, and leaves $20, 40, 60, 80$.

Finding the node costs $O(h)$; case 3 adds one more $O(h)$ descent. Deletion is $O(h)$ overall. Book: Section 6.11.

Case 1 — a Leaf

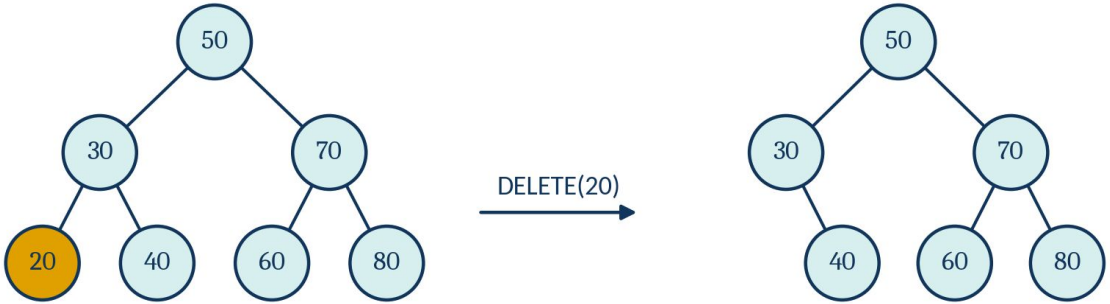


Figure 6.4 Case 1 of deletion: the node to be deleted, 20, is a leaf. It is simply removed; the rest of the tree is untouched.

Figure 6.4: search $50 \to 30 \to 20$*; 20 has no children; set* 30*'s left child to empty. Inorder:* $30, 40, 50, 60, 70, 80$*. Book: Section 6.11.1.*

Case 2 — One Child

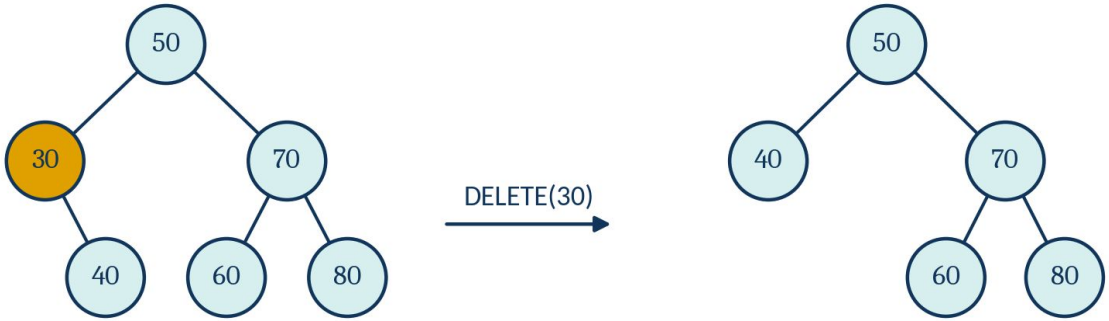


Figure 6.5 Case 2 of deletion: node 30 has only a right child, 40. Deleting 30 promotes 40 into its place.

Figure 6.5: 30 *has only the right child* 40*; promote* 40 *into* 30*'s place. Every key in the promoted subtree was already on the correct side of every ancestor. Inorder:* 40 , 50 , 60 , 70 , 80*. Book: Section 6.11.2.*

Case 3 — Two Children

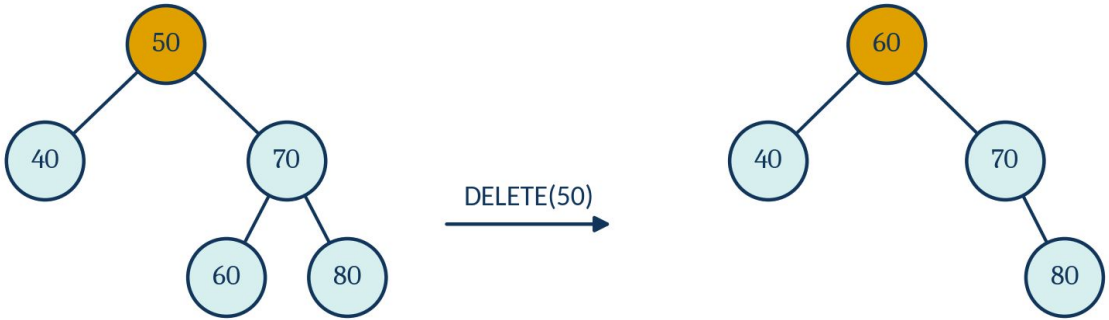


Figure 6.6 Case 3 of deletion: node 50 has two children. Its inorder successor is 60, the smallest key in its right subtree. The value 60 is copied up into the root and the original node 60 is then deleted, which is a case-1 deletion.

Figure 6.6: the successor of 50 *is* $70 \rightarrow 60$*; copy* 60 *into the root, then delete the old leaf* 60 *(case 1). Inorder:* $40, 60, 70, 80$*. Book: Section 6.11.3.*

Why Case 3 Is Legal, and Why It Stops

Legal. Left-subtree keys $< \text{key}(x) < \text{key}(s)$; and $\text{key}(s)$ is the smallest of the right-subtree keys, so all remaining right keys exceed it. The property holds at x.

Stops. The successor is the leftmost node of the right subtree, so it has no left child: deleting it is case 1 or case 2, never case 3 again.

The predecessor works too (max of the left subtree, no right child); here it would promote 40 instead of 60.

In code, case 3 copies a key, not a node: satellite data must move with it. Book: Sections 6.11.3 and 6.12.

When the Tree Goes Bad: h Can Reach n

Insert $10, 20, 30, 40, 50$ in that order: every key goes right, and the result is a chain of height 4 — a linked list wearing a tree's clothes.

Sorted input is the most common input there is: dates, invoice numbers, timestamps, primary keys.

Measured at $n = 16,000$: sorted input gives height $15,999$ and 7999.5 probes per insert; shuffled input gives height 32 and 16.03 probes. A factor of about **five hundred**, differing only in arrival order.

Random insertion order gives expected height $\Theta(\lg n)$, but real data is rarely random. We want a guarantee, not an expectation. Book: Section 6.13.

AVL Trees: Balance as an Invariant

DEFINITION. The **balance factor** of a node is Equation (6.3), with the empty subtree of height -1. An **AVL tree** is a BST satisfying Equation (6.4).

$$bf(x) = h(\text{right}(x)) - h(\text{left}(x)), \tag{6.3}$$

$$bf(x) \in \{-1, 0, +1\} \text{ for every node } x. \tag{6.4}$$

Negative means **left-heavy**, positive **right-heavy**. The invariant is about heights, not counts: AVL trees balance *depth*, not population.

Adelson-Velsky and Landis, 1962: after every change, find any lopsided node and straighten it with an $O(1)$ local rearrangement. Book: Section 6.14.

An AVL Tree

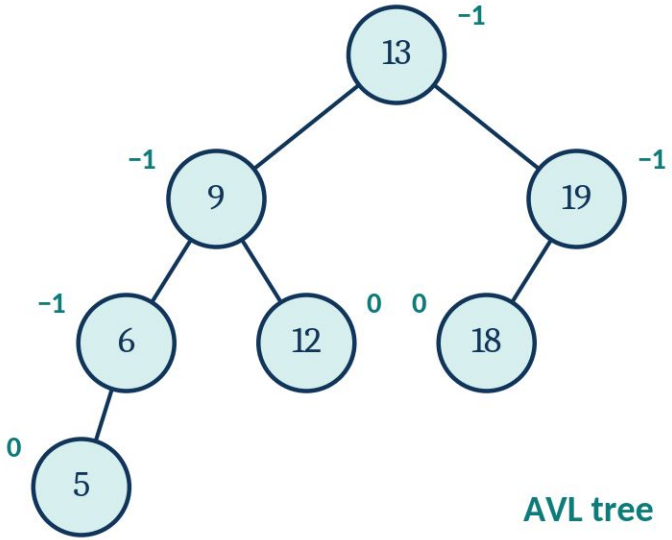


Figure 6.7 An AVL tree. Every node's two subtrees differ in height by at most one.

Figure 6.7: leaves $5, 12, 18$ *have* 0*; node* 6*:* $-1-0=-1$*; node* 9*:* $0-1=-1$*; node* 19*:* -1*; root* 13*:* $1-2=-1$*. All legal. Book: Section 6.14.*

And One That Is Not

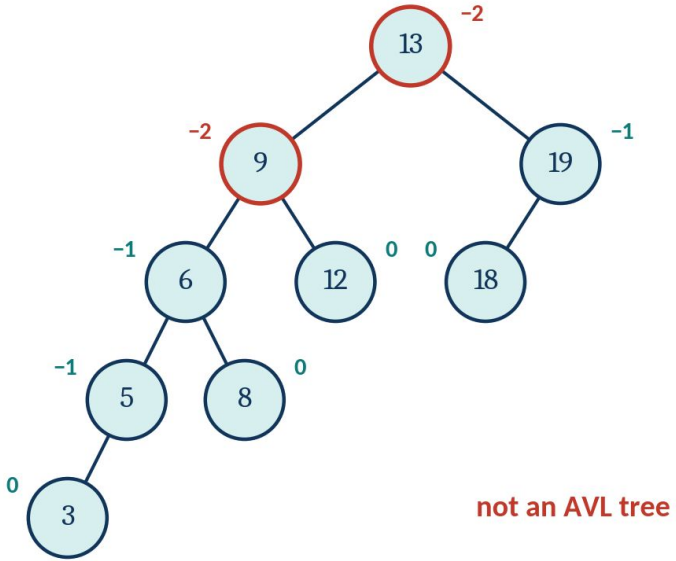


Figure 6.8 A binary search tree that is **not** an AVL tree. Adding the nodes 8 and 3 pushes two balance factors out of range: those of 9 and 13, both -2.

Figure 6.8: with 8 *under* 6 *and* 3 *under* 5*, node* 6 *has height* 2*, so node* 9 *has* $bf = 0 - 2 = -2$*. Still a valid BST — inorder* $3, 5, 6, 8, 9, 12, 13, 18, 19$ *— but not AVL. Book: Section 6.14.*

Why an AVL Tree Is Short

Ask the question inside out: let $N(h)$ be the **fewest nodes** an AVL tree of height h can have. $N(0)=1$, $N(1)=2$.

For $h \geq 2$: one subtree has height $h-1$; the invariant lets the other be as short as $h-2$; make both minimal and add the root. That is Equation (6.5):

$$N(h)=N(h-1)+N(h-2)+1, N(0)=1, N(1)=2. \tag{6.5}$$

BY HAND. $N(2)=2+1+1=4$; $N(3)=4+2+1=7$; $N(4)=12$; $N(5)=20$; $N(6)=20+12+1=33$. So 32 nodes can never form an AVL tree of height 6.

The trees that achieve these minima are called Fibonacci trees — and the name is a clue. Book: Section 6.15.

The Fibonacci Connection

Put $M(h) = N(h) + 1$: the $+1$ cancels and $M(h) = M(h-1) + M(h-2)$ with $M(0) = 2 = F(3)$ and $M(1) = 3 = F(4)$. Hence Equation (6.6):

$$N(h) + 1 = F(h+3). \tag{6.6}$$

Table 6.4 The fewest nodes an AVL tree of height h can contain, and the Fibonacci number it equals once you add one.

h	$N(h)$	$N(h)+1$	$F(h+3)$
0	1	2	2
1	2	3	3
2	4	5	5
3	7	8	8
4	12	13	13
5	20	21	21
6	33	34	34
7	54	55	55
8	88	89	89
9	143	144	144
10	232	233	233
11	376	377	377
12	609	610	610

Table 6.4: $N(4)+1=13=F(7)$ ✓, $N(6)+1=34=F(9)$ ✓. *Book: Section 6.15.1.*

From Fibonacci to $1.44 \lg n$

Binet's formula Equation (6.7); since $|\hat{\varphi}^m|/\sqrt{5} < 1/2$, it gives Equation (6.8):

$$F(m) = \frac{\varphi^m - \hat{\varphi}^m}{\sqrt{5}}, \varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803, \hat{\varphi} = \frac{1-\sqrt{5}}{2} \approx -0.61803. \tag{6.7}$$

$$F(m) + 1 \geq \frac{\varphi^m}{\sqrt{5}}. \tag{6.8}$$

Then $n + 2 \geq F(h+3) + 1 \geq \varphi^{h+3}/\sqrt{5}$; take lg and divide by $\lg \varphi \approx 0.69424$ to get Equation (6.9):

$$h \leq 1.4405 \lg(n+2) - 1.3277 \tag{6.9}$$

Never more than about 44% taller than a perfectly balanced tree: a billion keys, at most 41 *levels. The often-quoted* -0.3277 *counts nodes, not edges. Book: Section 6.15.2.*

Rotations: the One Tool

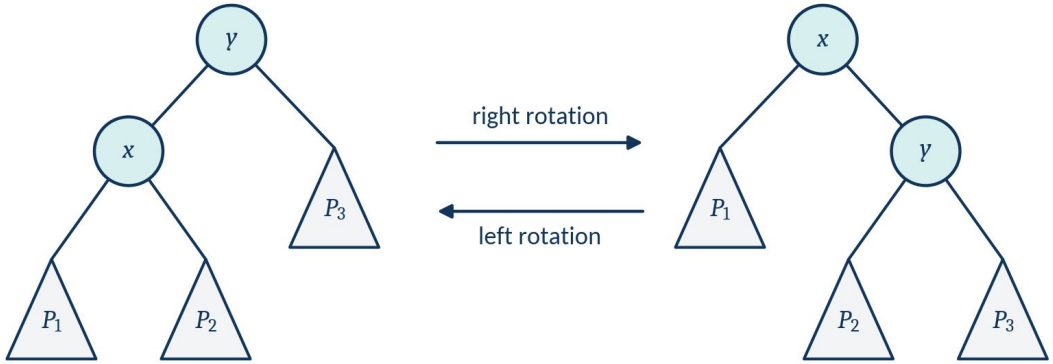


Figure 6.9 The two rotations. The same five items — the nodes x and y and the subtrees P_1, P_2, P_3 — appear in both pictures. A right rotation turns the left picture into the right one; a left rotation turns the right picture back into the left one. The two are exact inverses.

Figure 6.9: a right rotation about y lifts x; a left rotation about x lifts y. They are exact inverses and cost $\Theta(1)$. Book: Section 6.16.

A Rotation Keeps the Inorder Walk

In both pictures the BST property pins down Equation (6.10):

$$\text{keys}(P_1) < x < \text{keys}(P_2) < y < \text{keys}(P_3). \tag{6.10}$$

THEOREM. A rotation changes a tree's *shape* but not its *inorder sequence*; so a rotation on a BST always gives a BST.

Height: a right rotation lifts P_1 one level and lowers P_3 one level; P_2 only changes parent. A rotation moves height from one side to the other.

Only P_2 changes parent; the subtrees are re-pointed, never traversed. Book: Sections 6.16 and 6.16.3.

Right Rotation in C++

Listing 6.2 performs the pointer updates in order and rotates the chain 30 → 20 → 10; each node prints as `key[height]`.

Listing 6.2 A right rotation with its exact pointer updates, applied to the left-leaning chain $30 \to 20 \to 10$. Compile with `g++ -std=c++17 -O2 -o rotright rotright.cpp`.

```cpp
// A right rotation with its exact pointer updates, applied to a left-leaning chain.
#include <iostream>
#include <string>
#include <algorithm>

struct Node {
    int   key;
    int   height;                 // a leaf has height 0, the empty tree -1
    Node* left;
    Node* right;
    Node(int k) : key(k), height(0), left(nullptr), right(nullptr) {}
};

int  h(const Node* x) { return x ? x->height : -1; }
void refresh(Node* x) { x->height = 1 + std::max(h(x->left), h(x->right)); }

Node* rotateRight(Node* y) {
    Node* x  = y->left;         // 1. remember the node that will move up
    Node* P2 = x->right;        // 2. remember the subtree that will change parent
    x->right = y;               // 3. y becomes the right child of x
    y->left  = P2;              // 4. P2 becomes the left child of y
    refresh(y);                 // 5. y is now lower: recompute its height FIRST
    refresh(x);                 // 6. then x, which depends on y's new height
    return x;                   // 7. x is the new root of this subtree
}

std::string shape(const Node* x) {          // key[height]{left,right}; "." is empty
    if (!x) return ".";
```

```
    std::string s = std::to_string(x->key) + "[" + std::to_string(x->height) + "]";
    if (x->left || x->right) s += "{" + shape(x->left) + "," + shape(x->right) + "}";
    return s;
}

int main() {
    Node* a = new Node(30); Node* b = new Node(20); Node* c = new Node(10);
    a->left = b; b->left = c;                    // the chain 30 -> 20 -> 10, leaning left
    refresh(c); refresh(b); refresh(a);
    std::cout << "before: " << shape(a) << "\n";
    Node* root = rotateRight(a);
    std::cout << "after : " << shape(root) << "\n";
    delete a; delete b; delete c;
    return 0;
}
```

```
before: 30[2]{20[1]{10[0],.},.}
after : 20[1]{10[0],30[0]}
```

Refresh y ***before*** x*:* x*'s height is computed from* y*'s. Reversing them is the most common AVL bug, and it never crashes. Book: Sections 6.16.1 and 6.16.2.*

AVL Insertion: Four Steps, Four Cases

> **AVL INSERTION.** (1) Insert s as in a plain BST. (2) Walk up from s to the first node whose balance factor left $\{-1, 0, +1\}$; call it z. (3) Let y be z's child on that path and x the grandchild. (4) Rebalance at z.

LL: y left of z, x left of y. **LR:** y left, x right. **RR:** y right, x right. **RL:** y right, x left.

In the four figures, $P_1, \ldots, P_4$ are subtrees whose heights, after the insertion, are k or $k-1$; the insertion lifted x from height k to $k+1$.

"Left-Left" means the tree has leaned left twice in a row. Book: Section 6.17.

Case LL

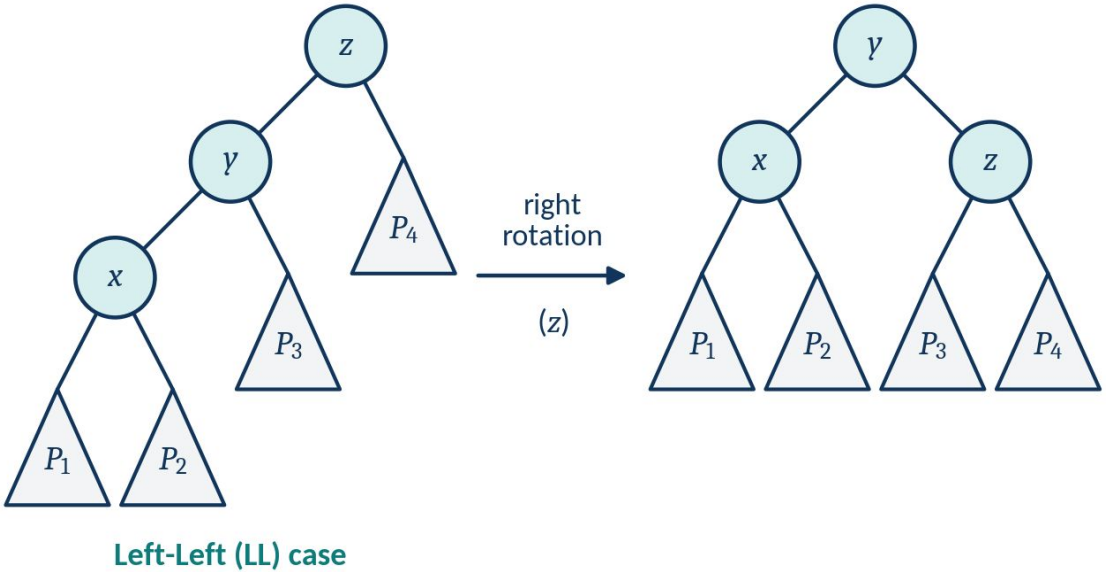


Figure 6.10 The Left-Left case, which also arises during deletion. A single right rotation about z restores the balance; y becomes the new root of the subtree, with x and z as its children.

Figure 6.10: $bf(z) = k - (k+2) = -2$ *and* $bf(y) = k - (k+1) = -1$. *One right rotation about* z. *Book: Section 6.17.1.*

The LL Repair

Pseudocode 6.1 The Left-Left repair: one right rotation about z, written as its four pointer assignments.

```
y       = z->left
P3      = y->right
y->right = z
z->left  = P3
refresh(z); refresh(y);        // z first, because it is now lower
```

Pseudocode 6.1 is the right rotation of Listing 6.2 with the roles renamed. Afterward $bf(z)=0$, $bf(y)=0$, and x keeps -1.

> **THE CRUCIAL OBSERVATION.** The subtree at z was $k+2$ tall before the insertion, $k+3$ after it, and is $k+2$ again after the rotation. Its ancestors notice nothing.

This observation is why insertion never needs more than one rebalancing. Book: Section 6.17.1.

Case LR

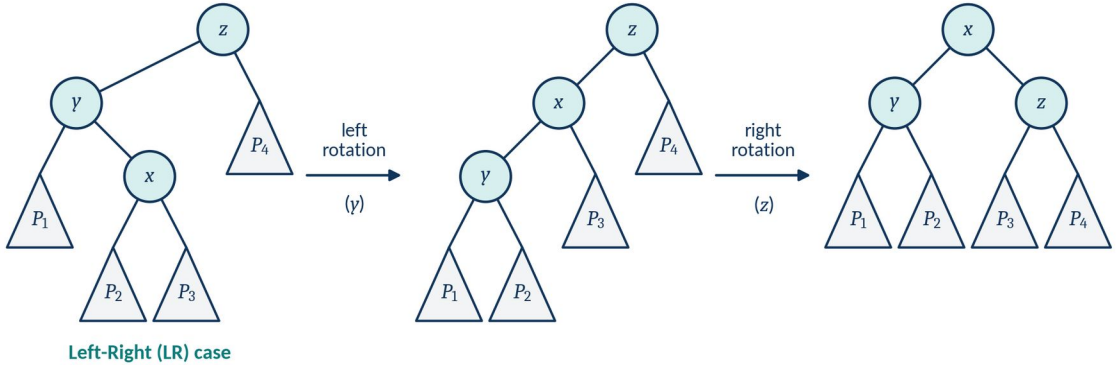


Figure 6.11 The Left-Right case, which also arises during deletion. Two rotations are needed: first a left rotation about y, which converts the problem into an LL case, then a right rotation about z.

Figure 6.11: $bf(z) = -2$ *but* $bf(y) = +1$ *— left, then right. First a left rotation about* y*, then a right rotation about* z*. Book: Section 6.17.2.*

The LR Repair

Why one rotation fails: a right rotation about z alone would leave $bf(y) = (k+2) - k = +2$ — still broken, leaning the other way. A single rotation cannot fix a zig-zag.

Pseudocode 6.2 The Left-Right repair: a left rotation about y turns the zig-zag into a Left-Left case, and a right rotation about z finishes.

```
z->left = rotateLeft(z->left);   // step 1: turns LR into LL
return rotateRight(z);           // step 2: the ordinary LL fix
```

Pseudocode 6.2: step 1 turns the zig-zag $z \to y \to x$ into the straight chain $z \to x \to y$ (an LL case); step 2 is the ordinary LL fix. Result: x on top, y and z below, height $k+2$.

All three balance factors end in $\{-1, 0, +1\}$. Book: Section 6.17.2.

Case RR

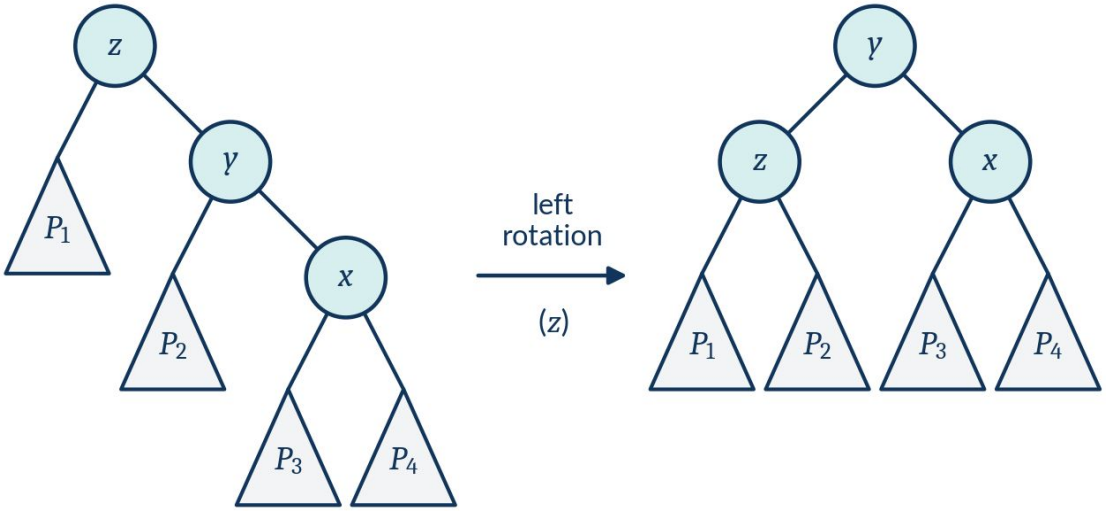


Figure 6.12 The Right-Right case, which also arises during deletion. The exact mirror image of Figure 6.10: a single left rotation about z.

Figure 6.12: $bf(z) = +2$, $bf(y) = +1$ — *two rightward leans. One left rotation about* z. *Book: Section 6.17.3.*

Case RL

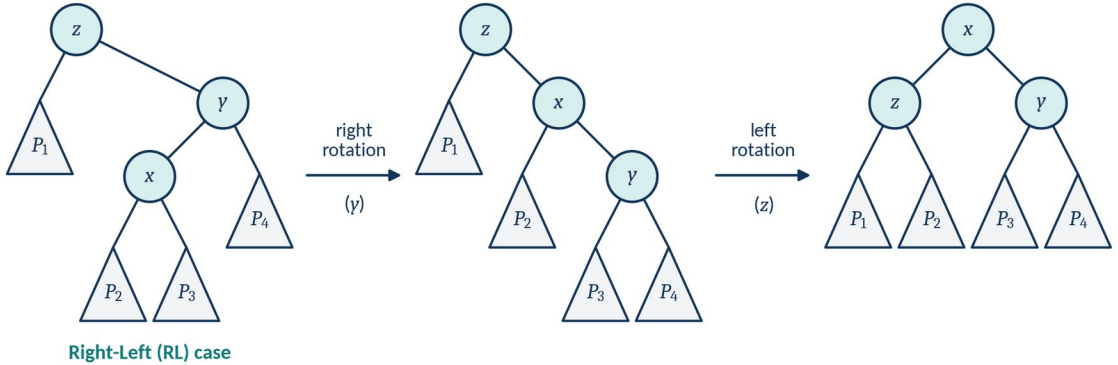


Figure 6.13 The Right-Left case, which also arises during deletion. The mirror of Figure 6.11: first a right rotation about y, which turns the zig-zag into an RR case, then a left rotation about z.

Figure 6.13: $bf(z) = +2$, $bf(y) = -1$ — *right, then left. A right rotation about* y*, then a left rotation about* z*. Book: Section 6.17.4.*

The RR and RL Repairs

Pseudocode 6.3 The Right-Right repair: one left rotation about z, the mirror image of Pseudocode 6.1.

```
y       = z->right
P2      = y->left
y->left  = z
z->right = P2
refresh(z); refresh(y);
```

Pseudocode 6.4 The Right-Left repair: a right rotation about y, then a left rotation about z.

```
z->right = rotateRight(z->right);   // step 1: turns RL into RR
return rotateLeft(z);               // step 2: the ordinary RR fix
```

Pseudocode 6.3 mirrors the LL repair; Pseudocode 6.4 mirrors the LR repair. In every case the subtree returns to height $k+2$.

Four cases, at most two rotations, $O(1)$ pointer writes each. Book: Sections 6.17.3 and 6.17.4.

Telling the Four Cases Apart

Table 6.5 The four rebalancing cases, identified by two balance factors, with the rotations each one needs.

bf(z)	bf(y), where y is z's taller child	Case	Rotations	Figure
< -1 (left-heavy)	negative	Left-Left	right rotation about z	Figure 6.10
< -1 (left-heavy)	positive	Left-Right	left about y, then right about z	Figure 6.11
$> +1$ (right-heavy)	positive	Right-Right	left rotation about z	Figure 6.12
$> +1$ (right-heavy)	negative	Right-Left	right about y, then left about z	Figure 6.13

Table 6.5 needs only two balance factors: the sign at z picks the side, the sign at y picks single or double.

> **WHAT IF $bf(y)=0$?** Impossible after an insertion, common after a deletion. Treat it as the **single**-rotation case: write `if (bf(z->left) <= 0)`, not `< 0`. The double rotation would leave y at ± 2.

Said out loud: LL is $(-)$ then $(-)$, LR $(-)$ then $(+)$, RR $(+)$ then $(+)$, RL $(+)$ then $(-)$. Book: Section 6.17.5.

Insertion Example, Step 1: Insert 14

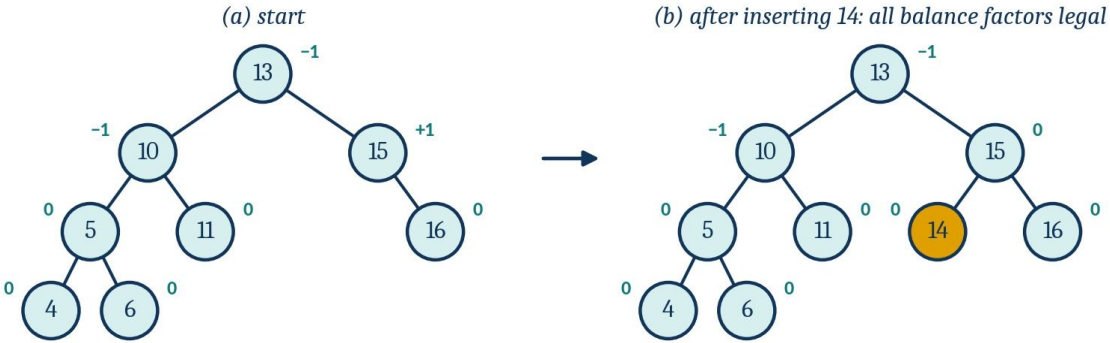


Figure 6.14 Inserting 14 into an AVL tree. Every balance factor stays inside $\{-1, 0, +1\}$, so no rotation is performed.

Figure 6.14: 14 *goes right at* 13*, left at* 15*.* $bf(15)$ *improves from* $+1$ *to* 0*;* $bf(13) = 1 - 2 = -1$*. No rotation — the common case. Book: Section 6.18.1.*

Step 2: Insert 3 — Left-Left

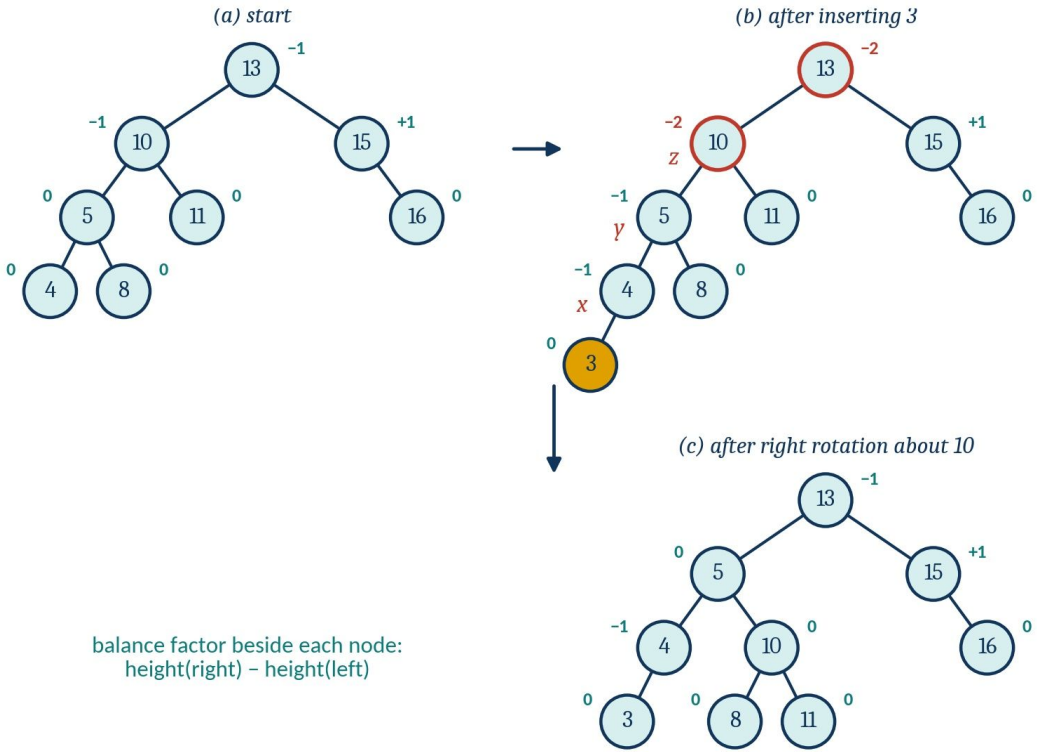


Figure 6.15 Inserting 3 produces a balance factor of -2 at node 10. The case is Left-Left, and one right rotation with 10 as the pivot repairs it.

Figure 6.15: 3 *hangs under* 4*;* $bf(4)=-1$*,* $bf(5)=-1$*,* $bf(10)=0-2=-2$*. So* $z=10$*,* $y=5$*,* $x=4$*: LL. Right rotation about* 10*;* 8 *changes parent. Tree height stays* 3*. Book: Section 6.18.2.*

Step 3: Insert 45 — Right-Right

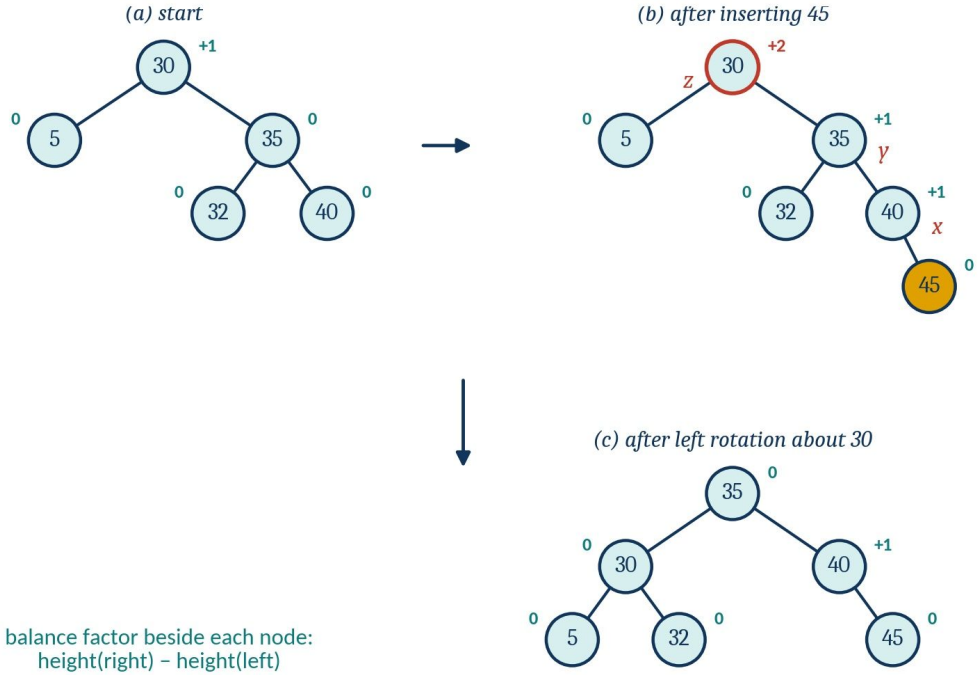


Figure 6.16 Inserting 45 makes node 30 right-heavy by two. The case is Right-Right, and one left rotation with 30 as the pivot repairs it.

Figure 6.16: $bf(40) = +1$, $bf(35) = +1$, $bf(30) = 2 - 0 = +2$. *So* $z = 30$, $y = 35$, $x = 40$*: RR. Left rotation about* 30*;* 32 *changes parent. New root* 35. *Book: Section 6.18.3.*

Step 4: Insert 7 — Left-Right

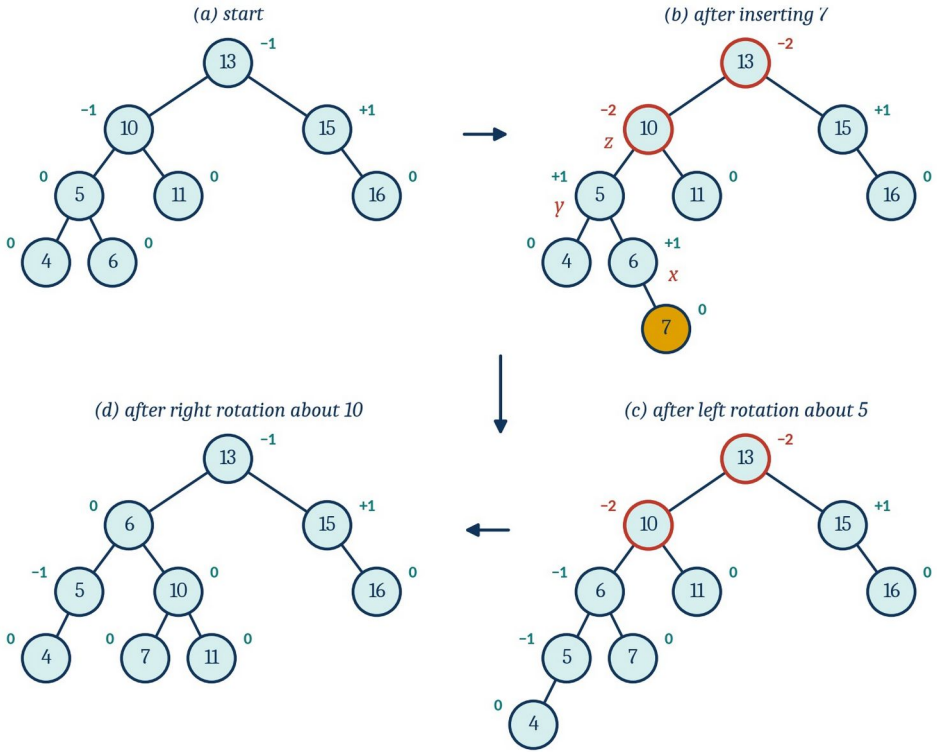


Figure 6.17 Inserting 7 creates a zig-zag — left at 10, then right at 5. Two rotations are needed: left with 5 as pivot, then right with 10 as pivot.

Figure 6.17: 7 hangs right of 6; $bf(5) = +1$, $bf(10) = -2$. So $z = 10$, $y = 5$, $x = 6$: LR. Left about 5, then right about 10: 6 ends on top with children 5 and 10. Book: Section 6.18.4.

Step 5: Insert 15 — Right-Left

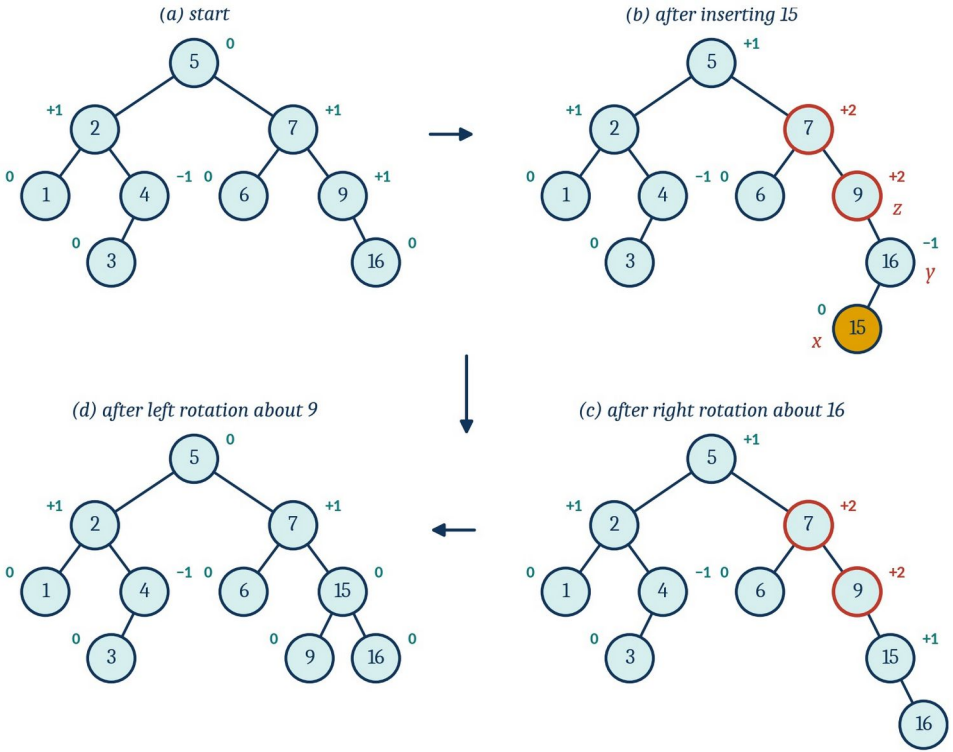


Figure 6.18 Inserting 15 makes node 9 right-heavy by two while node 16 leans left. The case is Right-Left: a right rotation with 16 as pivot, then a left rotation with 9 as pivot.

Figure 6.18: 15 *hangs left of* 16; $bf(16) = -1$, $bf(9) = 1 - (-1) = +2$. *So* $z = 9$, $y = 16$, $x = 15$: *RL. Right about* 16, *then left about* 9: 15 *ends on top with children* 9 *and* 16. *Book: Section 6.18.5.*

A Reference Implementation in Java, Part 1

```java
package AVLTree;

/**
 *
 * AVLTree class extended from Tree class.
 * Added a variable "height" for performing height-balanced efficiently
 */

public class AVLTree extends Tree {

    protected final boolean empty;
    protected final int value;
    protected final AVLTree left, right;
    protected final int height;

    /**
     * Creates a new tree whose root holds value and whose
     * left and right subtrees are left and right
     */
    public AVLTree(int value, AVLTree left, AVLTree right) {
        this.empty = false;
        this.value = value;
        this.left = left;
        this.right = right;
        this.height = 1 + Math.max(left.getHeight(), right.getHeight());
    }
```

Figure 6.19 The class `AVLTree.java`, first part. The class extends a plain `Tree` class and adds one field — `height` — "for performing height-balanced efficiently". The constructor sets `height = 1 + Math.max(left.getHeight(), right.getHeight())`.

Figure 6.19: an AVL tree is a tree with one extra field. `height = 1 + max(left, right)` is exactly the `refresh` of the C++ code. Book: Sections 6.19 and 6.19.1.

A Reference Implementation in Java, Part 2

```
26
27      /**
28       * Creates an empty tree
29       */
30      public AVLTree() {
31          this.empty = true;
32          this.value = 0;
33          this.left = null;
34          this.right = null;
35          this.height = 0;
36      }
37
38      /**
39       * Creates a tree with a single node
40       */
41      public AVLTree(int value) {
42          this.empty = false;
43          this.value = value;
44          this.left = new AVLTree();
45          this.right = new AVLTree();
46          this.height = 1;
47      }
48
49      /**
50       * Gets the height
51       */
52      private int getHeight() {
53          return height;
54      }
55
56  }
```

Figure 6.20 The class `AVLTree.java`, second part. The empty tree is an object with `height = 0`; a single-node tree has `height = 1`.

Figure 6.20 counts nodes: empty tree 0*, one node* 1*. Our C++ uses* -1 *and* 0*; the balance factors agree. Book: Section 6.19.1.*

AVL Deletion: Two Differences

Delete as in a plain BST; walk up to the first unbalanced node z; rebalance with the same four cases.

Difference 1: y is the child of z **with the greater height**, and x the taller child of y — the shrunken side is the short side, so the tall side must be found directly. On a tie, take x on the same side as y.

Difference 2: keep going. After a deletion the rotated subtree can end up shorter than before, its parent may now be unbalanced, and the repair propagates — up to $O(\lg n)$ rotations.

Measured on Fibonacci trees: one insertion always needs 1 *rebalancing; one deletion needs* $2, 3, 5, 6, 8$ *at heights* $4, 7, 10, 13, 16$*. Both stay* $O(\lg n)$*. Book: Sections 6.20–6.20.2.*

Deletion Example — Before

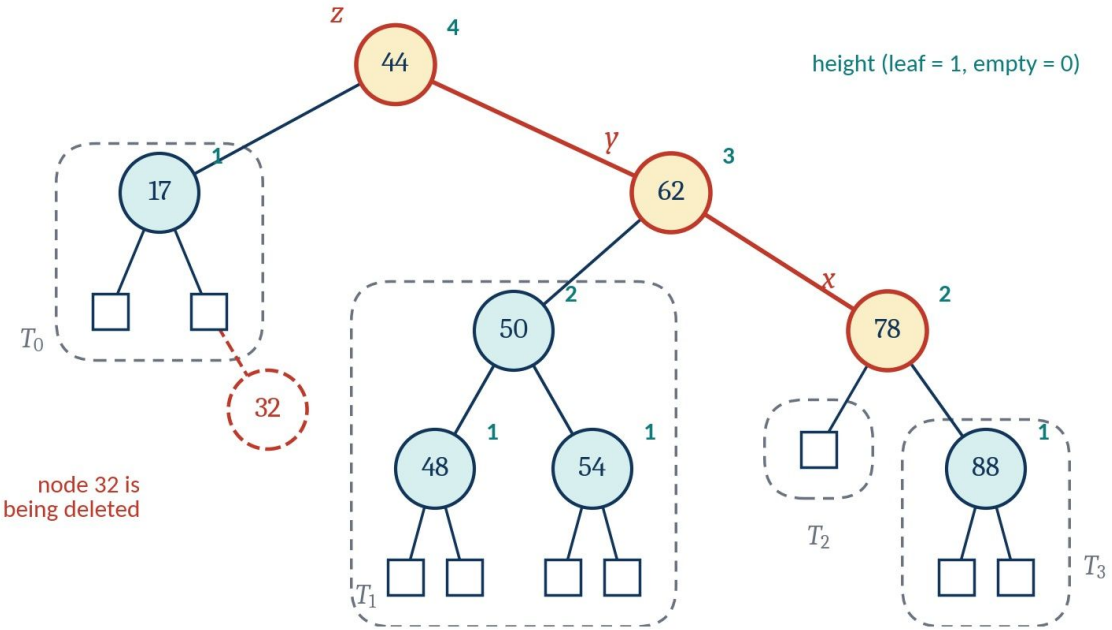


Figure 6.21 Deleting from an AVL tree: the node 32 is about to be deleted. Heights are written beside the nodes, and the first unbalanced ancestor z, its taller child y and y's taller child x are marked in red. The empty squares are null subtrees, and T_0 to T_3 name the four subtrees the rotation will move.

Figure 6.21 counts nodes (a leaf is 1*). Delete the leaf* 32*; at* 44 *the sides are* 1 *and* 3*, so* $z = 44$*,* $y = 62$*;* 50 *and* 78 *tie at* 2*, so* $x = 78$*: Right-Right. Book: Section 6.20.3.*

Deletion Example — After

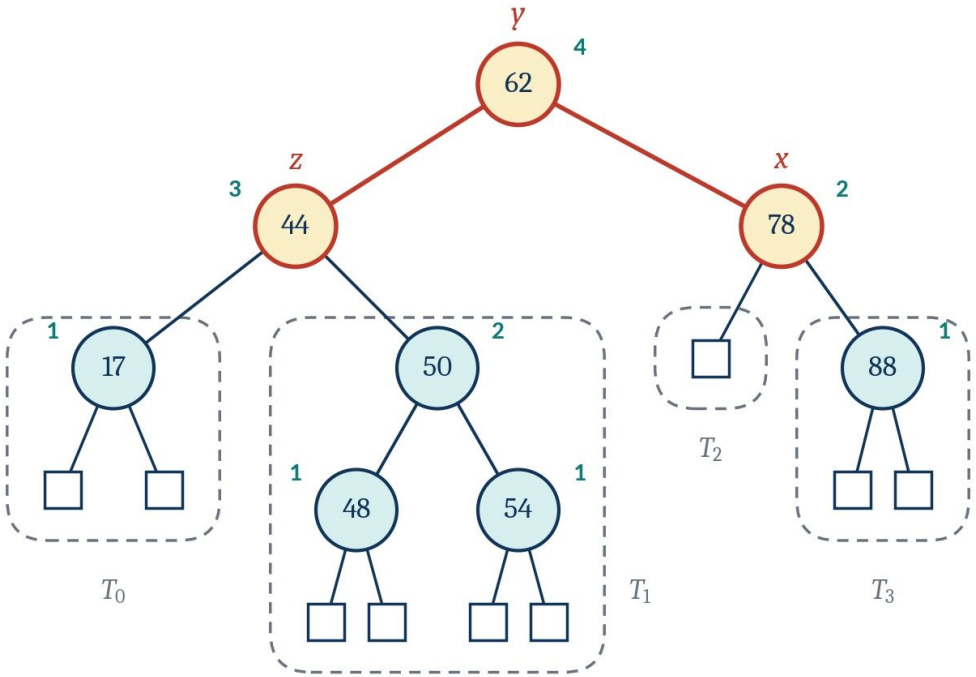


Figure 6.22 The tree after the left rotation. Node 62 is the new root; 44 has dropped to become its left child and has taken the subtree T_1 (rooted at 50) as its right child; 78 is unchanged as the right child of 62.

Figure 6.22: one left rotation about 44; 50 *(T_1) moves to* 44*'s right. Every node now differs by at most one, and one rotation sufficed here. Book: Section 6.20.3.*

AVL Tree, Red-Black Tree, or Hash Table?

A **red-black tree** stores one color bit; equal black counts on every path and no red–red edge give Equation (6.11), against about $1.44 \lg n$ for AVL.

$$h \le 2 \lg(n+1) \tag{6.11}$$

Red-black insertion needs ≤ 2 rotations and deletion ≤ 3; AVL deletion may need $\Theta(\lg n)$. **Read-heavy favors AVL; write-heavy favors red-black** (`std::map`).

A **hash table** gives expected $O(1)$ but no order: no successor, no minimum, no range query. Measured on $200,000$ keys: lookups 2.65 ms (hash) against 36.59 ms (AVL); a 1000-key range sum 0.51 ms (hash) against 0.01 ms (AVL).

Fed ascending keys, the AVL tree reached height 17*; a shuffled plain BST reached* 41*. Book: Section 6.21.*

What Each Operation Costs

Table 6.6 Worst-case costs for a plain binary search tree and for an AVL tree with n nodes.

Operation	Plain BST, worst case	AVL tree, worst case	Why
SEARCH	$\Theta(n)$	$\Theta(\lg n)$	one root-to-leaf path
MINIMUM, MAXIMUM	$\Theta(n)$	$\Theta(\lg n)$	one path, always the same direction
SUCCESSOR, PREDECESSOR	$\Theta(n)$	$\Theta(\lg n)$	one path down, or one path up
INSERT	$\Theta(n)$	$\Theta(\lg n)$, ≤ 2 rotations	descend, attach, rebalance once
DELETE	$\Theta(n)$	$\Theta(\lg n)$, $O(\lg n)$ rotations	descend, splice, rebalance all the way up
Inorder traversal	$\Theta(n)$	$\Theta(n)$	every node visited once
Range query returning k keys	$\Theta(n)$	$\Theta(\lg n + k)$	descend once, then walk the range
Build from n keys	$\Theta(n^2)$	$\Theta(n \lg n)$	n insertions
Height	up to $n-1$	$\leq 1.4405 \lg(n+2) - 1.3277$	the minimum-nodes recurrence

Table 6.6 is the summary to memorize: every query walks one path, so a guaranteed $O(\lg n)$ height makes every operation $O(\lg n)$.

Insertion rebalances once; deletion may rebalance at every level. Book: Section 6.24.

Profitina in Practice: the Pickup Board

In Profitina, the re-scan scheduler asks “which scan is due next?” (MINIMUM) and “what is due between 02:00 and 04:00?” (a range query); the weekly CSV export is an inorder walk; URL deduplication needs only equality, so it uses a hash table.

Profitina Laundry keeps its pickup board in an AVL tree keyed by (promised minute, ticket): insert on drop-off, successor for “what is due next?”, range for “what is due between 10:00 and 14:30?”, delete on collection.

Nine orders sit in a tree of height 3 — the shortest possible for nine nodes — even though tickets arrive in increasing time order.

The platform advises; the customer decides. A complete purge under UU PDP 27/2022 needs indexes that can delete cheaply. Book: Sections 6.22 and 6.22.1.

Checking a BST Correctly

The commonest bug checks only the children. Listing 6.3 carries an open interval down the tree instead, and shows a tree that fools the local test.

Listing 6.3 The wrong and the right way to verify the binary-search-tree property. Compile with `g++ -std=c++17 -O2 -o isbst isbst.cpp`.

```cpp
// Is this binary tree a binary search tree?
// The tempting local test -- "left child smaller, right child larger" -- is
// wrong. The correct test carries a permitted range down the tree.
#include <iostream>
#include <climits>

struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k, Node* l = nullptr, Node* r = nullptr) : key(k), left(l), right(r) {}
};

// WRONG: only compares a node with its two children.
bool looksLocallyFine(Node* x) {
    if (!x) return true;
    if (x->left  && x->left->key  >= x->key) return false;
    if (x->right && x->right->key <= x->key) return false;
    return looksLocallyFine(x->left) && looksLocallyFine(x->right);
}

// RIGHT: every key in the left subtree must be below x, every key in the
// right subtree above it, so pass the open interval (lo, hi) downward.
bool isBST(Node* x, long long lo, long long hi) {
    if (!x) return true;
    if (x->key <= lo || x->key >= hi) return false;
    return isBST(x->left, lo, x->key) && isBST(x->right, x->key, hi);
}
```

```
int main() {
    /*        20
     *       /  \
     *      10   30
     *          /  \
     *         5    40      <-- 5 is in the RIGHT subtree of 20: not a BST
     */
    Node* bad = new Node(20, new Node(10), new Node(30, new Node(5), new Node(40)));

    /*        20
     *       /  \
     *      10   30
     *     /       \
     *    5         40
     */
    Node* good = new Node(20, new Node(10, new Node(5)), new Node(30, nullptr, new Node(40)));
```

Listing 6.3 (Continued) *The wrong and the right way to verify the binary-search-tree property. Compile with g++ -std=c++17 -O2 -o isbst isbst.cpp.*

```
    std::cout << "bad  tree: local test says "
              << (looksLocallyFine(bad) ? "BST" : "not a BST")
              << ", correct test says "
              << (isBST(bad, LLONG_MIN, LLONG_MAX) ? "BST" : "not a BST") << "\n";
    std::cout << "good tree: local test says "
              << (looksLocallyFine(good) ? "BST" : "not a BST")
              << ", correct test says "
              << (isBST(good, LLONG_MIN, LLONG_MAX) ? "BST" : "not a BST") << "\n";
    return 0;
}
```

```
bad  tree: local test says BST, correct test says not a BST
good tree: local test says BST, correct test says BST
```

The interval is `long long` so that the sentinels lie strictly outside every `int` key. Book: Section 6.23, mistake 1.

Common Mistakes

Checking the BST property only against the children, or confusing it with the heap property.

Refreshing heights in the wrong order after a rotation; **mixing** the -1 and 0 height conventions for the empty tree.

Stopping after one rebalancing on deletion; using `< 0` instead of `<= 0` for the single-rotation test.

Assuming a plain BST is balanced because it usually is; copying a key but not its satellite data in case 3.

Using a tree where a hash table would do — trees earn their cost only when order matters.

Ten mistakes in all, each with its fix. Book: Section 6.23.

Summary: Five Things to Remember

1. Left subtree $<$ node $<$ right subtree, for *every* key below — not merely the children.

2. Inorder is sorted (proved by induction); every query and update walks one path: $O(h)$.

3. Deletion: detach a leaf; promote the only child; copy the successor, then delete it.

4. AVL: $bf \in \{-1, 0, +1\}$; $N(h)+1 = F(h+3)$ gives $h \le 1.4405 \lg(n+2) - 1.3277$.

5. Rotations preserve the inorder walk; four cases; insertion rebalances once, deletion possibly at every level.

A plain BST depends on your luck; an AVL tree does not. Book: Section 6.24.

Next Time

Chapter 7 — from divide and conquer to dynamic programming. Fibonacci again — this time as the smallest problem that breaks divide and conquer.

Memoization, bottom-up tables, matrix-chain multiplication, and the longest common subsequence.

Before then: redo the five insertion steps and the deletion example by hand, computing every balance factor.

Chapter 8 then works problems from Chapters 5 to 7, including these very trees. Book: Sections 6.24.1 and 6.25.

Lecture 7 — From Divide and Conquer to Dynamic Programming

Course: Design and Analysis of Algorithms (EF234405)

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*, Chapter 7

Lecturer: Irfan Subakti — yifana@gmail.com

Today we learn to tell, with evidence, whether a recursion is wasting its life recomputing answers it already has — and how to make it polynomial. Book: Chapter 7, Sections 7.1–7.13.

What This Meeting Covers

1. **Divide and conquer, restated** — and the one assumption it makes: the pieces are all different.
2. **Fibonacci** — $2F_{n+1} - 1$ calls for $n + 1$ questions, and four repairs: memoization, bottom-up, a sliding window, a matrix power.
3. **Optimization problems** — optimal substructure and the cut-and-paste proof.
4. **Matrix-chain multiplication** — the recurrence, the table filled by hand, the bracketing read back, $\Theta(n^3)$.
5. **The four-step recipe**, and **longest common subsequence** — table, arrows, reconstruction, $\Theta(\min(m, n))$ space.
6. **Overlapping versus independent subproblems** — measured; Profitina Laundry's slip matcher.

Two fully worked examples today: the matrix chain $d = \langle 10,20,3,5,30 \rangle$ *and the strings "ACGGTTA" and "CGTAT". Book: Chapter 7.*

Where We Left Off

Chapter 3 gave us the general divide-and-conquer recurrence Equation (7.1), and three ways to solve it: substitution, the recursion tree, the Master Theorem.

$$T(n) = a\,T(n/b) + f(n), \qquad a \geq 1,\ b > 1. \tag{7.1}$$

DEFINITIONS. A **subproblem** is a smaller instance of the *same* problem produced along the way. The **recursion tree** has the original call at the root and each call's direct calls as its children; each node is labeled with its subproblem.

The one observation: sometimes the same label appears on more than one node. When it does not, divide and conquer is right; when it does — catastrophically often — dynamic programming is right.

Chapter 3 silently assumed the pieces are different. When that fails the Master Theorem still answers correctly — with something like $\Theta(1.618^n)$. Book: Sections 7.1 and 7.1.1.

The Divide-and-Conquer Method

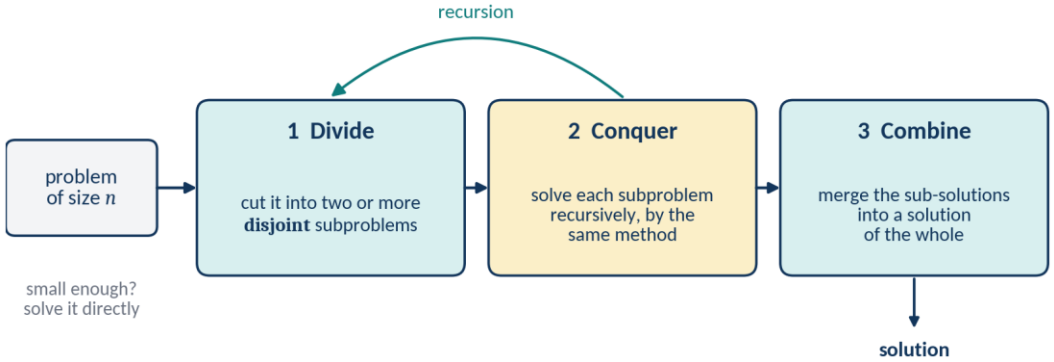


Figure 7.1 The divide-and-conquer method for algorithm design: divide the problem into **disjoint** subproblems, conquer them by recursion, then combine their solutions into a solution of the original.

***Figure 7.1**: **divide** into disjoint subproblems, **conquer** them recursively (with a base case), **combine** the answers. Book: Section 7.2.1.*

Merge Sort: Independent, All Different

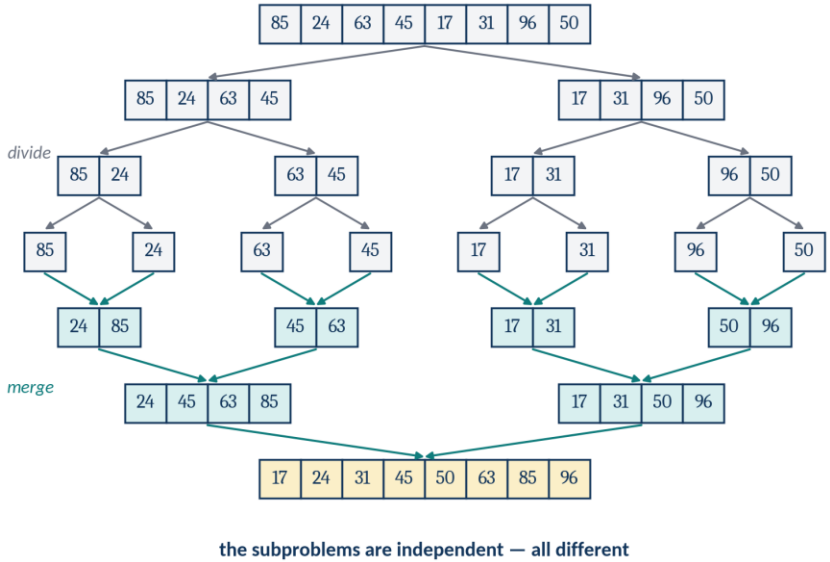


Figure 7.2 Merge Sort as divide and conquer. The array is split in half, each half is sorted recursively, and the two sorted halves are merged. The sentence that matters is written beneath the drawing: **the subproblems are independent, all different.**

Figure 7.2*: no label repeats anywhere in the tree, so* $T(n) = 2T(n/2) + \Theta(n)$ *tells the whole truth. Divide and conquer fits when the problem splits, the pieces are disjoint, and combining is cheap — Merge Sort, binary search, Strassen. Book: Sections 7.2.2 and 7.2.3.*

When Divide and Conquer Wastes Work

THE TRAP. Recursion is not slow. *Recomputation* is slow. A plain recursion has no memory: asked the same question a million times, it does a million full computations.

The smallest example is Fibonacci, defined by Equation (7.2):

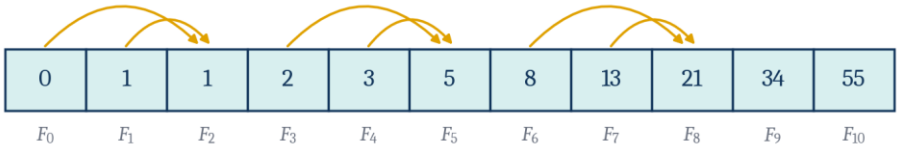
$$F_0 = 0, \qquad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2. \tag{7.2}$$

By hand: $F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, F_6 = 8, F_7 = 13, F_8 = 21, F_9 = 34, F_{10} = 55$.

The first two lines are the base cases; the third is the recursive case. Book: Sections 7.2.4 and 7.3.1.

The Fibonacci Numbers

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad (n \geq 2)$$



the straightforward recursive procedure is slow — why, and how slow?

Figure 7.3 The Fibonacci numbers, with the warning that follows immediately: *the straightforward recursive procedure is slow*. Why, and how slow, is the business of the next four sections.

Figure 7.3 *ends with the warning: the straightforward recursive procedure is slow. Why, and how slow, comes next. Book: Section 7.3.1.*

The Straightforward Recursion, Counted

Pseudocode 7.1 The straightforward recursion for F_n, transcribed directly from the definition.

```
long long fib(int n) {
    if (n <= 1) return n;            // base cases: F(0)=0 and F(1)=1
    return fib(n - 1) + fib(n - 2);  // recursive case
}
```

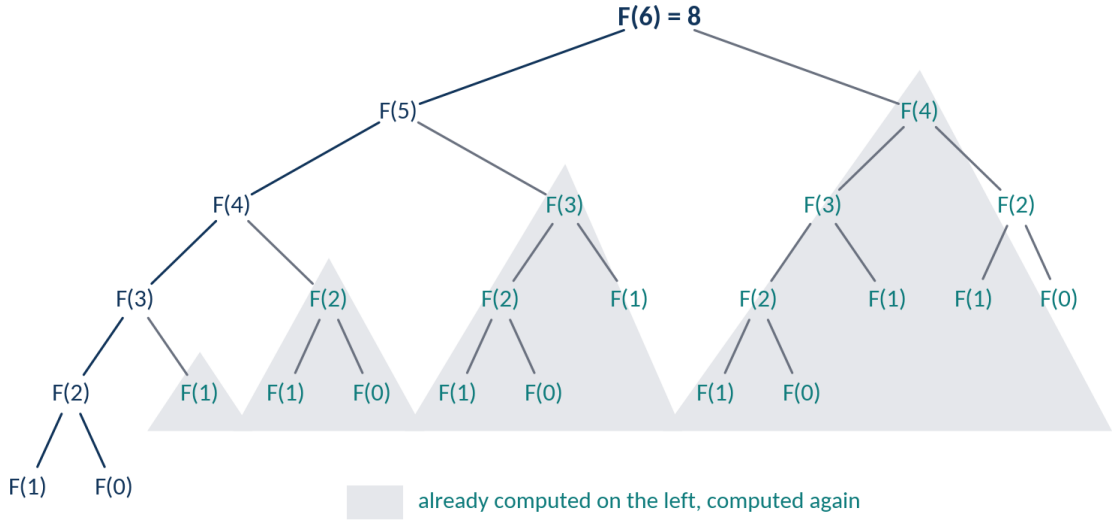
Let $C(n)$ count the calls **Pseudocode 7.1** makes: $C(0) = C(1) = 1$ and $C(n) = C(n-1) + C(n-2) + 1$.

CLAIM. $C(n) = 2F_{n+1} - 1$. *Proof* by strong induction: $C(n) = (2F_n - 1) + (2F_{n-1} - 1) + 1 = 2F_{n+1} - 1$. ■

Leaves: F_{n+1}; additions: $F_{n+1} - 1$. For $n = 6$: 25 calls, 13 leaves, 12 additions.

It is correct. It is also slow in the way that makes a program useless. Book: Sections 7.3.2 and 7.3.4.

The Recursion Tree of `fib(6)`



We keep calculating the same value over and over.

Figure 7.4 The recursion tree of the straightforward `fib(6)`. The gray triangles mark subtrees that have already been computed somewhere to their left and are being computed all over again. The whole right half of the tree — the entire $F(4)$ subtree — is a duplicate of a subtree already finished on the left.

***Figure 7.4**: $F(4)$ twice, $F(3)$ three times, $F(2)$ five times, $F(1)$ eight times. We keep calculating the same value over and over. Book: Section 7.3.3.*

Why the Cost Is $\Theta(\phi^n)$

$\phi = (1+\sqrt{5})/2 \approx 1.618$ and $\psi = (1-\sqrt{5})/2 \approx -0.618$ are the roots of $x^2 = x + 1$; fitting the base cases gives Binet's formula Equation (7.3):

$$F_n = \frac{\phi^n - \psi^n}{\sqrt{5}}. \tag{7.3}$$

Since $|\psi^n/\sqrt{5}| < 1/2$, F_n is the nearest integer to $\phi^n/\sqrt{5}$, and we get Equation (7.4):

$$C(n) = 2F_{n+1} - 1 = \Theta(\phi^n) = \Theta(1.618^n). \tag{7.4}$$

Only $n+1$ distinct questions exist, yet $F(k)$ is asked Equation (7.5):

$$\text{calls to } F(k) \;=\; F_{n-k+1} \quad (k \geq 1), \qquad \text{calls to } F(0) \;=\; F_{n-1}. \tag{7.5}$$

A computer a thousand times faster buys about fourteen more values of n. Book: Sections 7.3.5 and 7.3.6.

The Golden Ratio, in Outline

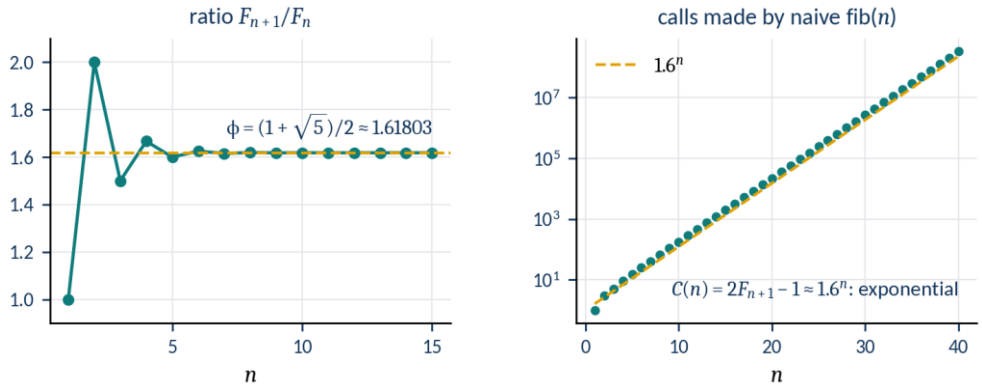


Figure 7.5 The argument in outline. Left: consecutive Fibonacci numbers approach the golden ratio $\phi \approx 1.61803$, so $F_n \approx 1.6^n$. Right: the call count $2F_{n+1} - 1$ of the naive procedure, on a logarithmic scale, runs parallel to 1.6^n — the tree has about 1.6^n leaves, and the running time is exponential.

***Figure 7.5**: consecutive Fibonacci numbers approach ϕ, so the tree has about 1.6^n leaves. Book: Section 7.3.5.*

Measuring It

Listing 7.1 counts every call, records how often each $F(k)$ is asked for, and times the recursion.

Listing 7.1 The naive Fibonacci recursion, instrumented. Save as `fib_naive.cpp`, compile with `g++ -std=c++17 -O2 -o fib_naive fib_naive.cpp`, and run `./fib_naive`.

```cpp
// fib_naive.cpp - the straightforward recursion, instrumented.
// Counts every call, counts how often each F(k) is recomputed, and times it.
#include <iostream>
#include <iomanip>
#include <vector>
#include <chrono>
using namespace std;
using namespace std::chrono;

long long calls = 0;                 // total number of invocations of fib()
vector<long long> perLevel;          // perLevel[k] = how many times F(k) was called

long long fib(int n) {
    ++calls;
    if (n < (int)perLevel.size()) ++perLevel[n];
    if (n <= 1) return n;            // base cases F(0)=0, F(1)=1
    return fib(n - 1) + fib(n - 2);
}

int main() {
    // Part 1: the call profile of fib(6), the tree drawn in Section 7.3.3.
    perLevel.assign(7, 0);
    calls = 0;
    long long v = fib(6);
    cout << "fib(6) = " << v << ", total calls = " << calls << "\n";
    cout << "calls to F(k), k = 0..6:";
    for (int k = 0; k <= 6; ++k) cout << " F(" << k << ")=" << perLevel[k];
    cout << "\n\n";
```

```
    // Part 2: growth of the call count and of the wall-clock time.
    cout << left << setw(5) << "n" << setw(14) << "F(n)"
         << setw(14) << "calls" << setw(12) << "ms"
         << setw(10) << "calls/prev" << "\n";
    long long prevCalls = 0;
    for (int n = 20; n <= 42; n += 2) {
        perLevel.assign(0, 0);
        calls = 0;
        auto t0 = steady_clock::now();
        long long f = fib(n);
        auto t1 = steady_clock::now();
        double ms = duration<double, milli>(t1 - t0).count();
        cout << left << setw(5) << n << setw(14) << f << setw(14) << calls
             << setw(12) << fixed << setprecision(2) << ms;
        if (prevCalls) cout << setw(10) << setprecision(3) << (double)calls / prevCalls;
        cout << "\n";
        prevCalls = calls;
```

Listing 7.1 (Continued) *The naive Fibonacci recursion, instrumented. Save as fib_naive.cpp, compile with g++ -std=c++17 -O2 -o fib_naive fib_naive.cpp, and run ./fib_naive.*

```
    }
    return 0;
}
```

```
fib(6) = 8, total calls = 25
calls to F(k), k = 0..6: F(0)=5 F(1)=8 F(2)=5 F(3)=3 F(4)=2 F(5)=1 F(6)=1

n    F(n)          calls         ms          calls/prev
20   6765          21891         0.05
22   17711         57313         0.12        2.618
24   46368         150049        0.34        2.618
26   121393        392835        0.86        2.618
28   317811        1028457       2.21        2.618
30   832040        2692537       5.83        2.618
32   2178309       7049155       16.27       2.618
34   5702887       18454929      40.88       2.618
36   14930352      48315633      108.97      2.618
38   39088169      126491971     280.85      2.618
40   102334155     331160281     735.49      2.618
42   267914296     866988873     1943.29     2.618
```

The profile of `fib(6)` is 25 *calls split* 5,8,5,3,2,1,1*; the ratio two steps apart is* $2.618 = \phi^2$*;* F_{42} *takes* 866 *million calls and nearly two seconds. Book: Section 7.3.7.*

The Repair, in Brief

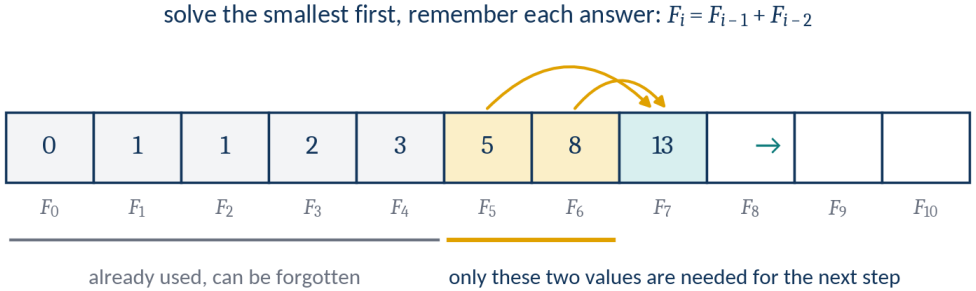


Figure 7.6 The repair, in brief: compute F_n in linear time by *remembering solutions to the solved subproblems* — dynamic programming — building the solution bottom-up, trading space for time. The highlighted pair is the observation that will become §7.4.3: only two values need to be remembered at any time.

***Figure 7.6**: remember solutions to solved subproblems — dynamic programming, named by Richard Bellman in the 1950s — and trade space for time. Book: Section 7.4.*

Repair 1: Memoization

DEFINITION. Memoization keeps a table of answers already computed, consults it at the top of every call, and returns the stored answer instead of recursing. The recursion's structure stays; only its repetition goes.

Pseudocode 7.2 Memoized Fibonacci: the same recursion, consulting and filling a table of answers.

```
vector<long long> memo;                    // memo[k] = F(k), or -1 if not yet known
long long fibMemo(int n) {
    if (memo[n] != -1) return memo[n];      // <-- already solved: do not recurse
    long long r = (n <= 1) ? n : fibMemo(n - 1) + fibMemo(n - 2);
    memo[n] = r;                            // <-- record before returning
    return r;
}
```

THE ACCOUNTING TRICK. Count the table entries and multiply by the cost of filling one, *ignoring* the recursive calls inside it (they are charged to their own entries): $n + 1$ entries $\times\, O(1) = \Theta(n)$.

Pseudocode 7.2 *shades every gray triangle of the recursion tree out of existence. "Memoization", from memo — not "memorization". Book: Section 7.4.1.*

Repairs 2 and 3: Bottom-Up, Then Two Variables

Pseudocode 7.3 Bottom-up Fibonacci: the table is filled from the smallest index upward.

```
long long fibTable(int n) {
    vector<long long> F(n + 1);
    F[0] = 0; if (n >= 1) F[1] = 1;
    for (int i = 2; i <= n; ++i) F[i] = F[i-1] + F[i-2];
    return F[n];
}
```

Pseudocode 7.3: compute $F_0, F_1, F_2, \ldots$ in order, so everything an entry needs is already there. $\Theta(n)$ time and space, no recursion. **The ordering is the whole art.**

Pseudocode 7.4 Fibonacci in constant space: a sliding window of two values replaces the table.

```
long long fibIter(int n) {
    if (n <= 1) return n;
    long long prev = 0, cur = 1;            // prev = F(i-2), cur = F(i-1)
    for (int i = 2; i <= n; ++i) {
        long long next = prev + cur;         // F(i)
        prev = cur;                          // slide the window one step right
        cur  = next;
    }
    return cur;
}
```

Pseudocode 7.4: F_i reaches back only two places, so a sliding window of two values replaces the table — $\Theta(n)$ time, $\Theta(1)$ space.

The same window trick on a two-dimensional table gives LCS in $\Theta(min(m, n))$ space later today. Book: Sections 7.4.2 and 7.4.3.

The Window, and Repair 4: the Matrix Power

Table 7.1 The sliding window of `fibIter` computing F_6. After the last iteration `cur` holds $8 = F_6$, and no array was ever allocated.

i	`prev` before	`cur` before	`next` = prev+cur	`prev` after	`cur` after
2	0	1	1	1	1
3	1	1	2	1	2
4	1	2	3	2	3
5	2	3	5	3	5
6	3	5	8	5	8

Table 7.1 traces the window for F_6. One step of the recurrence is a linear map, which gives Equation (7.6):

$$Q^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}, \tag{7.6}$$

Repeated squaring computes Q^n in $O(\lg n)$ arithmetic operations — counted in the RAM model; very large F_n need big integers, so the program works modulo 1 000 000 007.

F_n is the top-right entry of Q^n. Book: Sections 7.4.3 and 7.4.4.

All Four, Measured

Table 7.2 The five ways to compute F_n from this chapter, with the cost of each. The first row is the one to avoid; the last row is the one that wins when n is large.

Method	Time	Space	Notes
Naive recursion	$\Theta(\phi^n)$	$\Theta(n)$ stack	Unusable beyond $n \approx 45$
Memoization (top-down)	$\Theta(n)$	$\Theta(n)$ + $\Theta(n)$ stack	Easiest rewrite; stack-limited
Bottom-up table	$\Theta(n)$	$\Theta(n)$	No recursion, no stack risk
Sliding window	$\Theta(n)$	$\Theta(1)$	Cannot recover intermediate F_i
Matrix power	$\Theta(\lg n)$	$\Theta(1)$	Needs the linear-algebra insight

Table 7.2, measured: at $n = 40$ memoization takes 0.83 µs against 735,490 µs for the naive recursion — about 886,000 times faster, from two extra lines.

Memoization is the slowest of the linear three (call overhead) and overflowed the stack at $n = 1{,}000{,}000$; the window beats the table (no memory traffic); the matrix power took 0.13 µs at $n = 10^6$.

Asymptotics say the three linear methods are equal; the machine says otherwise. Both statements are true. Book: Section 7.4.5.

Optimization Problems

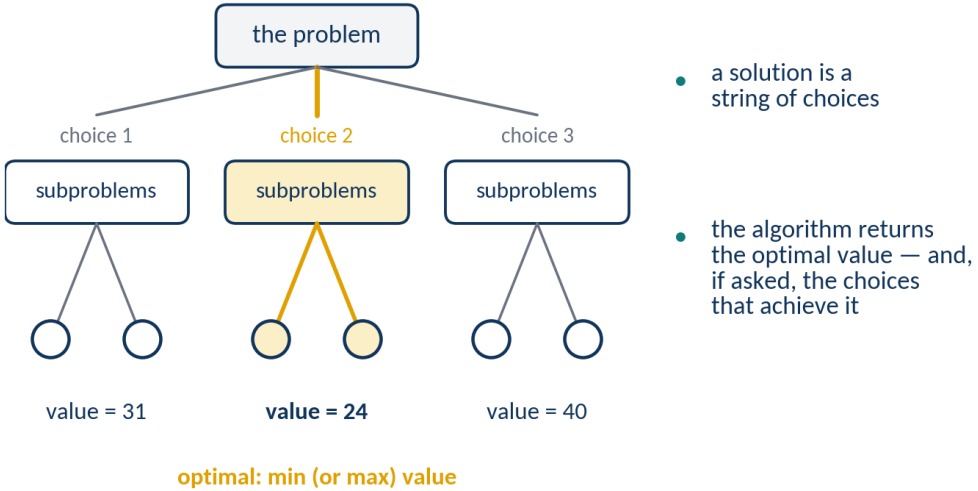


Figure 7.7 Optimization problems in outline: many solutions exist, one is optimal, a solution is a *string of choices*, and the algorithm computes the optimal value and — if we ask for it — the optimal solution itself.

***Figure 7.7**: many feasible solutions, each with a value; find an optimal one. A solution is a string of choices, and the algorithm returns the optimal value and, if asked, the solution. Book: Sections 7.5 and 7.5.1.*

Optimal Substructure, and Cut-and-Paste

DEFINITION. A problem has **optimal substructure** if an optimal solution contains optimal solutions to the subproblems it is composed of. (Not the converse: optimal pieces of the *wrong* split glue into rubbish — so we try every split.)

CUT AND PASTE. If some piece S_i were not optimal, cut it out and paste in a better S_i': the result is still **legal** and strictly better than S — a contradiction.

No substructure: longest simple path. On the square $q - r - t - s - q$, the longest $q \to t$ path is $q \to r \to t$, but the longest $q \to r$ path is $q \to s \to t \to r$; pasting it in repeats vertices, so the paste is illegal.

Prove optimal substructure — and check the paste is legal — before writing any table. Longest simple path is NP-hard. Book: Sections 7.5.2–7.5.4.

The Cost of One Matrix Product

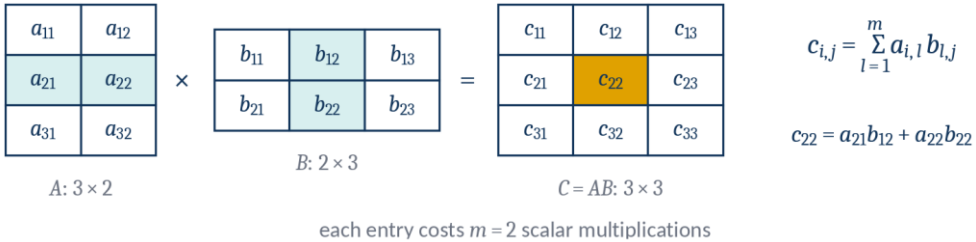


Figure 7.8 Multiplying an $n \times m$ matrix A by an $m \times k$ matrix B, drawn here with $n = 3, m = 2, k = 3$. The shaded row of A and shaded column of B combine to produce the single shaded entry c_{22} of the result, by the formula $c_{i,j} = \sum_{l=1}^{m} a_{i,l} \cdot b_{l,j}$.

Figure 7.8*: each of the n × k result entries costs m scalar multiplications. Book: Section 7.6.1.*

The Cost Rule, and Why Grouping Matters

Entry (i, j) of AB is Equation (7.7). So an $n \times m$ times $m \times k$ product costs $n \cdot m \cdot k$ scalar multiplications.

$$c_{i,j} = \sum_{l=1}^{m} a_{i,l} \cdot b_{l,j}. \tag{7.7}$$

A: 10×100, B: 100×5, C: 5×50. $(AB)C$ costs $5\,000 + 2\,500 = 7\,500$; $A(BC)$ costs $25\,000 + 50\,000 = 75\,000$. Same answer, ten times the work.

> **THE PROBLEM.** Given the dimensions of $A_1, \ldots, A_n$, find a full parenthesization minimizing the total number of scalar multiplications. The output is a plan — a set of brackets.

Matrix multiplication is associative in its answer, not in its cost. Book: Sections 7.6.1 and 7.6.2.

A Four-Matrix Example

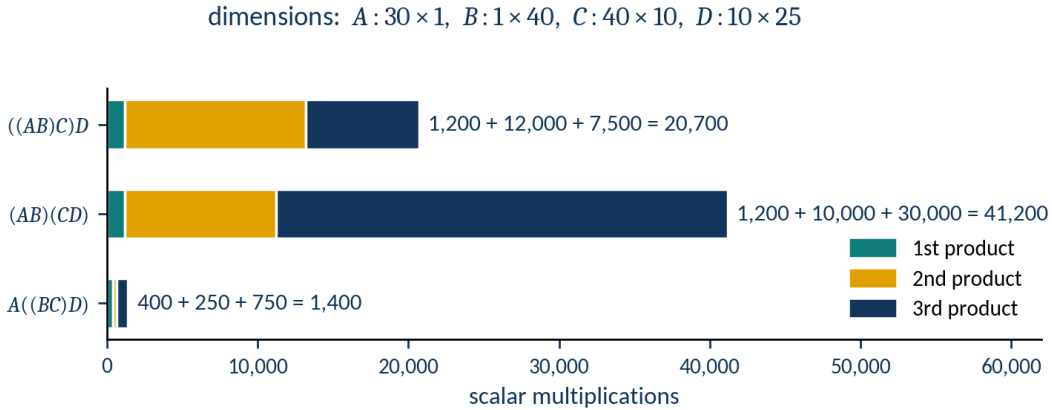


Figure 7.9 A four-matrix example. For A (30 × 1), B (1 × 40), C (40 × 10), D (10 × 25), three of the five possible bracketings cost 20 700, 41 200 and 1 400 respectively; each bar splits the total into its three products. The last line fixes the notation used from here on: we optimally parenthesize $A_1 \times A_2 \times \cdots \times A_n$, where A_i is a $d_{i-1} \times d_i$ matrix.

***Figure 7.9**: $d = \langle 30,1,40,10,25 \rangle$ and A_i is $d_{i-1} \times d_i$. CLRS writes the same sequence as p and the table as m. Book: Section 7.6.3.*

Three Bracketings, Priced

$((AB)C)D$: $1\,200 + 12\,000 + 7\,500 = 20\,700$.

$(AB)(CD)$: $1\,200 + 10\,000 + 30\,000 = 41\,200$.

$A((BC)D)$: $400 + 250 + 750 = 1\,400$ — a factor of twenty-nine cheaper than the worst.

> **WHY THE THIRD WINS.** B is 1×40: using it early keeps every intermediate one row tall. A good bracketing makes the small dimensions do the work.

Every number matches the figure. Book: Section 7.6.3.

Why Not Try Them All?

The outermost multiplication splits the chain at some k, and the two sides bracket independently, which gives Equation (7.8):

$$P(1) = 1, \qquad P(n) = \sum_{k=1}^{n-1} P(k)\, P(n-k) \quad (n \geq 2). \tag{7.8}$$

$P(2) = 1$, $P(3) = 2$, $P(4) = 5$, $P(5) = 14$: the Catalan numbers, $P(n+1) = C_n$ with Equation (7.9), growing like $\Omega(4^n/n^{3/2})$.

$$C_n = \frac{1}{n+1}\binom{2n}{n}, \tag{7.9}$$

At $n = 60$ brute force would outlast the universe. Book: Section 7.6.4.

Brute Force as Ground Truth

Listing 7.2 enumerates and prices every bracketing of the chain $d = \langle 10,20,3,5,30 \rangle$ that we solve by hand next.

Listing 7.2 Brute-force enumeration of every parenthesization, used as ground truth against which the dynamic program is checked. Compile with `g++ -std=c++17 -O2 -o brute brute.cpp`.

```cpp
// brute.cpp - enumerate EVERY parenthesization of a chain and price each one,
// so that the dynamic-programming answer can be checked against the truth.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <climits>
using namespace std;

vector<long long> d;

// Every way to split A_i..A_j, with the cost and the printed grouping.
vector<pair<long long,string>> allWays(int i, int j) {
    if (i == j) return {{0, "A" + to_string(i)}};
    vector<pair<long long,string>> out;
    for (int k = i; k < j; ++k)
        for (auto& L : allWays(i, k))
            for (auto& R : allWays(k + 1, j))
                out.push_back({L.first + R.first + d[i-1]*d[k]*d[j],
                               "(" + L.second + R.second + ")"});
    return out;
}

int main() {
    d = {10, 20, 3, 5, 30};
    int n = (int)d.size() - 1;
    auto ways = allWays(1, n);
    cout << "d = [10, 20, 3, 5, 30]: " << ways.size() << " parenthesizations\n\n";
```

```
    cout << left << setw(22) << "grouping" << right << setw(10) << "cost" << "\n";
    long long best = LLONG_MAX, worst = 0; string bestS, worstS;
    for (auto& w : ways) {
        cout << left << setw(22) << w.second << right << setw(10) << w.first << "\n";
        if (w.first < best)  { best  = w.first; bestS  = w.second; }
        if (w.first > worst) { worst = w.first; worstS = w.second; }
    }
    cout << "\ncheapest : " << bestS  << " at " << best  << "\n";
    cout << "dearest  : " << worstS << " at " << worst << "\n";
    cout << "ratio    : " << fixed << setprecision(2) << (double)worst / best << " times\n";
    return 0;
}
```

```
d = [10, 20, 3, 5, 30]: 5 parenthesizations

grouping                cost
(A1(A2(A3A4)))          8250
(A1((A2A3)A4))          9300
((A1A2)(A3A4))          1950
((A1(A2A3))A4)          2800
(((A1A2)A3)A4)          2250

cheapest : ((A1A2)(A3A4)) at 1950
dearest  : (A1((A2A3)A4)) at 9300
ratio    : 4.77 times
```

Five bracketings, as $P(4) = 5$ predicts. Remember 1 950 *and* $((A_1A_2)(A_3A_4))$*: we will derive both without enumerating anything. Book: Section 7.6.4.*

The Two Key Observations

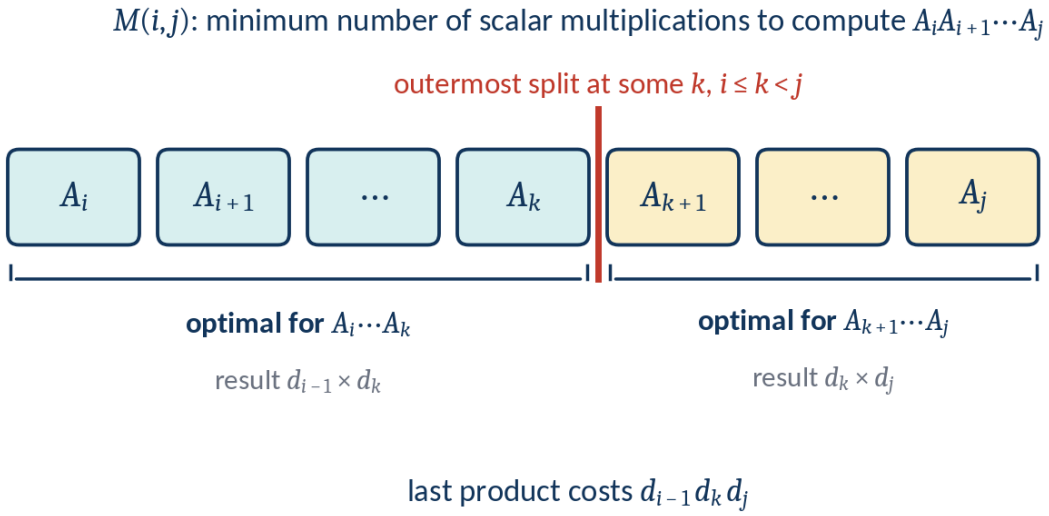


Figure 7.10 The two key observations. First, the outermost bracket must split the chain at *some* position k with $i \le k < j$. Second, whatever k is, the parenthesizations on either side of it must themselves be optimal — optimal substructure, and the justification for the recurrence.

***Figure 7.10**: the outermost bracket splits at some k with $i \le k < j$, and both sides must themselves be optimal. Book: Section 7.6.5.*

The Subproblem and the Recurrence

Let $M(i,j)$ be the minimum cost of $A_i \cdots A_j$ — **contiguous** subchains, because only adjacent matrices can be bracketed. The last product costs $d_{i-1}d_k d_j$ however the halves were bracketed; cut-and-paste shows the halves are optimal. Hence Equation (7.10):

$$M(i,j) = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \left\{ M(i,k) + M(k+1,j) + d_{i-1}\, d_k\, d_j \right\} & \text{if } i < j. \end{cases} \tag{7.10}$$

THREE INDEX TRAPS. (1) $i \le k < j$, not $k \le j$. (2) The right side starts at $k+1$. (3) The triple is d_{i-1}, d_k, d_j — hence $n+1$ dimensions indexed from 0.

In words: try every place the last multiplication could happen; add left cost, right cost and the last product; keep the smallest. Book: Section 7.6.5.

The Matrix-Chain Recurrence

$$M(i,i) = 0$$

$$M(i,j) = \min_{i \le k < j} \{ M(i,k) + M(k+1,j) + d_{i-1}\, d_k\, d_j \}$$

Figure 7.11 The matrix-chain recurrence. The base case is $M(i,i) = 0$; the general case minimizes, over every split point k, the cost of the left subchain plus the cost of the right subchain plus $d_{i-1}d_k d_j$ for the last multiplication.

Figure 7.11*: base case $M(i,i) = 0$; a single matrix needs no multiplication. Book: Section 7.6.5.*

Only $\Theta(n^2)$ Subproblems

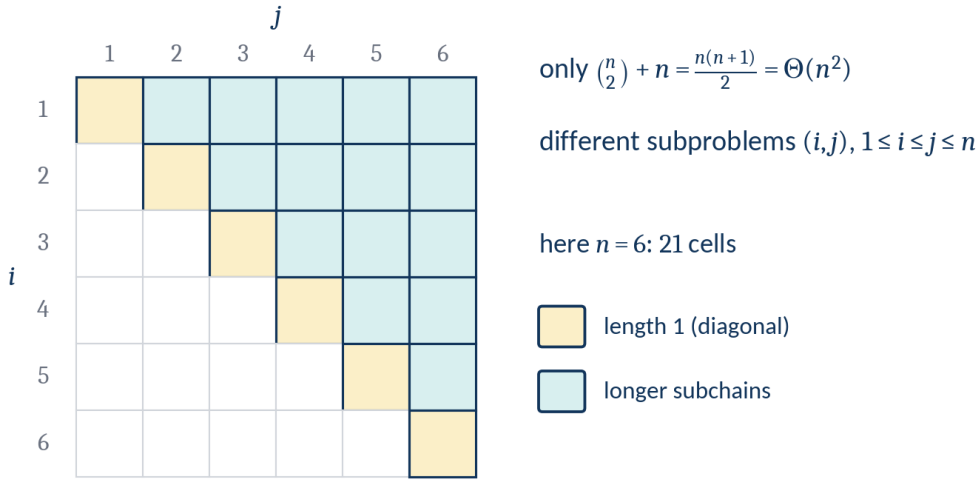


Figure 7.12 The count of the subproblems: there are only $\binom{n}{2} + n = \Theta(n^2)$ different subproblems (i, j) with $1 \le i \le j \le n$. The triangle draws them for $n = 6$: the 21 cells on and above the diagonal. That gap between $\Theta(n^2)$ questions and exponentially many recursive visits is the opening dynamic programming exploits.

***Figure 7.12**: pairs (i, j) with $1 \le i \le j \le n$. For $n = 60$ that is 1 830 questions against about 4.1×10^{32} bracketings. Book: Section 7.6.6.*

Matrix-Chain-Order

Pseudocode 7.5 Matrix-Chain-Order in pseudocode. Lines 1–2 fill the diagonal with zeros. Line 3 is the crucial one: the outer loop runs over the chain **length** l, not over i or j, so that every subchain is solved before any longer subchain that depends on it. Lines 7–11 try every split k and keep both the best cost in M and the winning split in c.

```
MATRIX-CHAIN-ORDER(d[0..n]):
 1.  for i ← 1 to n
 2.      M[i,i] ← 0
 3.  for l ← 2 to n                                  // l = length of the subchain
 4.      for i ← 1 to n − l + 1
 5.          j ← i + l − 1
 6.          M[i,j] ← ∞
 7.          for k ← i to j − 1
 8.              q ← M[i,k] + M[k+1,j] + d[i−1]·d[k]·d[j]
 9.              if q < M[i,j] then
10.                  M[i,j] ← q
11.                  c[i,j] ← k
12.  return M, c
```

Pseudocode 7.5*: fill the diagonal with zeros, then loop over the chain* ***length*** *l, then over i with $j = i + l - 1$, trying every k and recording the winner in c. Book: Section 7.6.7.*

Why Loop over Length

The subproblems number Equation (7.11):

$$\binom{n}{2} + n = \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2} = \Theta(n^2), \tag{7.11}$$

> **THE ORDER.** $M[i,j]$ needs $M[i,k]$ and $M[k+1,j]$, both **shorter** subchains. Process subchains by increasing length and everything a cell needs is ready. Looping over i or j in natural order reads unwritten cells.

The c table costs nothing to keep and is the only way to recover the bracketing, not just its price.

Line 6 initializes $M[i,j]$ to ∞ so the first candidate always wins. Book: Sections 7.6.6 and 7.6.7.

The Table by Hand: $l = 2$ and $l = 3$

$d = \langle 10,20,3,5,30 \rangle$: A_1 is 10×20, A_2 20×3, A_3 3×5, A_4 5×30. Diagonal: all 0.

$l = 2$: $M[1,2] = 10 \cdot 20 \cdot 3 = 600$; $M[2,3] = 20 \cdot 3 \cdot 5 = 300$; $M[3,4] = 3 \cdot 5 \cdot 30 = 450$.

$M[1,3]$: $k = 1$: $0 + 300 + 1\,000 = 1\,300$; $k = 2$: $600 + 0 + 150 = 750$. So 750, $c = 2$.

$M[2,4]$: $k = 2$: $0 + 450 + 1\,800 = 2\,250$; $k = 3$: $300 + 0 + 3\,000 = 3\,300$. So 2 250, $c = 2$.

At $M[1,3]$, paying 600 early collapses the 20 to a 3, so the last product costs only 150. Paying early to shrink a dimension is the theme. Book: Section 7.6.8.

The Table by Hand: $l = 4$

$M[1,4]$: $k = 1$: $0 + 2\,250 + 6\,000 = 8\,250$; $k = 2$: $600 + 450 + 900 = 1\,950$; $k = 3$: $750 + 0 + 1\,500 = 2\,250$. So 1 950, $c[1,4] = 2$ — the brute-force answer. **Table 7.3** and **Table 7.4** are the finished tables.

Table 7.3 The finished M table for $d = \langle 10,20,3,5,30 \rangle$. Only the upper triangle is meaningful, because a subchain must have $i \leq j$. The answer to the original problem is the top-right corner, $M[1,4] = 1950$.

	j=1	j=2	j=3	j=4
$i = 1$	0	600	750	**1950**
$i = 2$	—	0	300	2250
$i = 3$	—	—	0	450
$i = 4$	—	—	—	0

Table 7.4 The finished c table: $c[i, j]$ records the split point k that achieved the minimum in $M[i, j]$. It is the only record of *how* the optimum was reached, and §7.6.9 reads the bracketing out of it.

	j=1	j=2	j=3	j=4
$i = 1$	—	1	2	**2**
$i = 2$	—	—	2	2
$i = 3$	—	—	—	3
$i = 4$	—	—	—	—

The table fills diagonal by diagonal, up and to the right — exactly what looping over l produces. Book: Section 7.6.8.

Value and Subdivisions

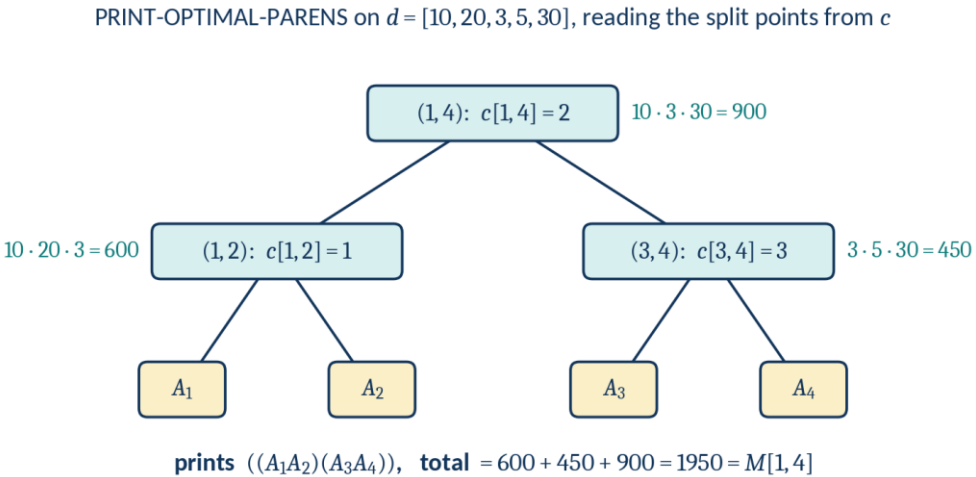


Figure 7.13 After the algorithm runs, $M[1, n]$ holds the optimal value and c holds the optimal subdivisions; a simple recursive procedure **Print-Optimal-Parens**(c, i, j) walks c and prints the bracketing. The tree shows that walk on the example we have just solved by hand, $d = [10,20,3,5,30]$, with the cost of each product beside its call.

***Figure 7.13**: $M[1, n]$ holds the optimal value and c the optimal subdivisions; Print-Optimal-Parens walks c. Book: Section 7.6.9.*

Reading the Bracketing Back Out

Pseudocode 7.6 Print-Optimal-Parens: reading the optimal bracketing out of the c table.

```
string printOptimalParens(int i, int j) {
    if (i == j) return "A" + to_string(i);          // a single matrix prints as itself
    int k = c[i][j];                                 // where the optimum split this chain
    return "(" + printOptimalParens(i, k)
               + printOptimalParens(k + 1, j) + ")";
}
```

Pseudocode 7.6 on the c table: `parens(1,4)` splits at 2 into `parens(1,2)` and `parens(3,4)`; those split at 1 and 3; the result is $((A_1A_2)(A_3A_4))$.

Check from the cost rule alone: $600 + 450 + 10 \cdot 3 \cdot 30 = 600 + 450 + 900 = 1\,950$ ✓ — three independent routes to one number.

Store the choice, not just the value: that is step 4 of the recipe. Book: Section 7.6.9.

The Running Time

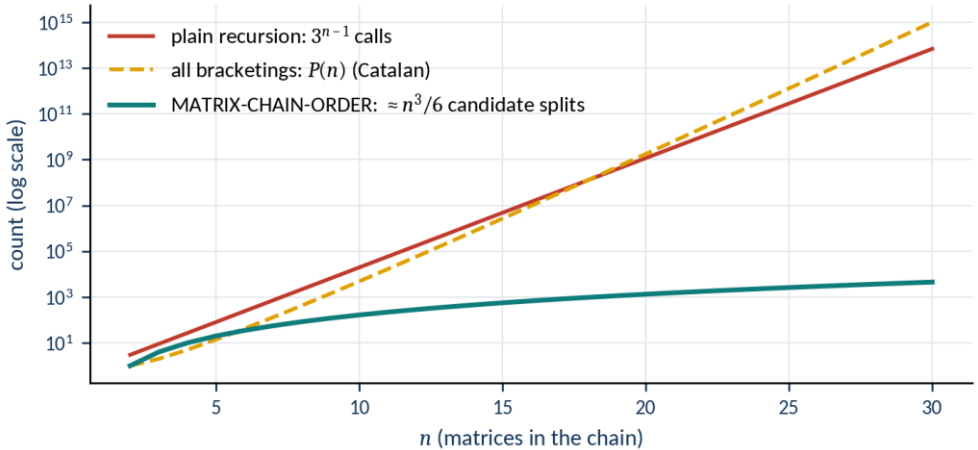


Figure 7.14 The running time of **Matrix-Chain-Order**: easily seen to be $O(n^3)$, and in fact also $\Omega(n^3)$, hence $\Theta(n^3)$. The plot, on a logarithmic scale, sets its candidate-split count against the 3^{n-1} calls of plain recursion and the $P(n)$ bracketings of brute force: *from exponential time to polynomial*, the whole point of the chapter in five words.

***Figure 7.14**: $O(n^3)$ and in fact $\Omega(n^3)$ — from exponential time to polynomial. Book: Section 7.6.10.*

$\Theta(n^3)$, Counted Exactly

For each l the middle loop runs $n - l + 1$ times and the inner loop $l - 1$ times. Substituting $t = l - 1$ and summing gives Equation (7.12):

$$= \frac{3n^2(n-1) - (n-1)n(2n-1)}{6} = \frac{n(n-1)[3n - (2n-1)]}{6} = \frac{n(n-1)(n+1)}{6} = \frac{n^3 - n}{6}. \tag{7.12}$$

For $n = 4$: $(64 - 4)/6 = 10$ — exactly the ten candidate splits of the hand trace ($3 + 2 \cdot 2 + 3$).

Space: two $n \times n$ tables, $\Theta(n^2)$. At $n = 25$: about 1.29×10^{12} bracketings against 2 600 candidate splits.

Listing 7.2 *and the hand-filled table agree; the book's full program prints all ten candidate splits and both tables. Book: Sections 7.6.10 and 7.6.11.*

Lookup-Chain: Keeping the Recursion

Pseudocode 7.7 **Lookup-Chain**: the matrix-chain recurrence written as recursion, with line 1 short-circuiting any subproblem already solved. All entries of M are initialized to ∞ beforehand, and ∞ doubles as the "not yet computed" marker.

```
LOOKUP-CHAIN(d, i, j):
 1.  if M[i,j] < ∞ then
 2.      return M[i,j]                          // solved before: reuse it
 3.  if i = j then
 4.      M[i,j] ← 0
 5.  else for k ← i to j − 1
 6.      q ← LOOKUP-CHAIN(d, i, k) + LOOKUP-CHAIN(d, k+1, j) + d[i−1]·d[k]·d[j]
 7.      if q < M[i,j] then
 8.          M[i,j] ← q
 9.  return M[i,j]
```

***Pseudocode 7.7**: the recurrence as a recursion, with every entry initialized to ∞, which doubles as "not yet computed". Book: Section 7.7.1.*

Memoization against Bottom-Up

Pseudocode 7.8 The first line of **Lookup-Chain**: return a stored answer instead of recomputing it.

```
1  if M[i,j] < ∞ then
2      return M[i,j]
```

Pseudocode 7.8 is the entire difference from an exponential disaster. A sentinel must be **impossible** as a real answer.

Table 7.5 Memoization against bottom-up. Both give the same asymptotic running time; everything that differs is in this table.

	Memoization (top-down)	Bottom-up table
Order of evaluation	Discovered automatically by the recursion	You must work it out and encode it in loops
Code shape	Recurrence, almost transcribed	Loops that may not resemble the recurrence
Subproblems solved	Only those actually reachable	All of them, reachable or not
Constant factor	Higher (calls, frames, scattered access)	Lower (tight loops, sweeping access)
Stack	Depth proportional to recursion depth; can overflow	None
Space saving	Hard — the whole table must persist	Easy — old rows/diagonals can be dropped

Table 7.5, measured: at $n = 17$ plain recursion makes $3^{16} = 43{,}046{,}721$ calls (109 ms), memoization 1 633 (0.005 ms). At $n = 200$ bottom-up is about four times faster than memoized.

When in doubt, write the memoized version first, then convert to bottom-up and check they agree. Book: Sections 7.7.2 and 7.7.3.

The Recipe, Step 1

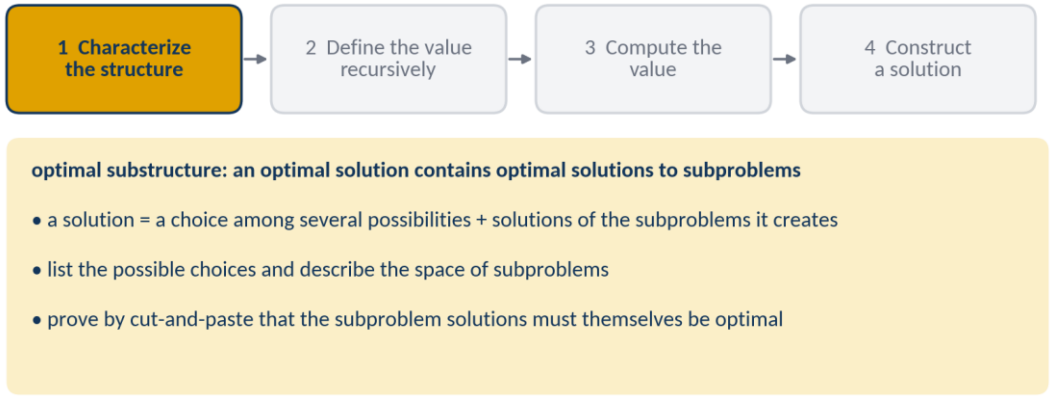


Figure 7.15 Step 1 of the recipe: show optimal substructure. A solution is a choice out of a number of possibilities plus solutions to the subproblems that choice creates; then show, by cut-and-paste, that those subproblem solutions must themselves be optimal.

Figure 7.15*: name the choice; name the subproblems it creates; prove by cut-and-paste that their solutions must be optimal. Book: Section 7.8.1.*

The Recipe, Step 2

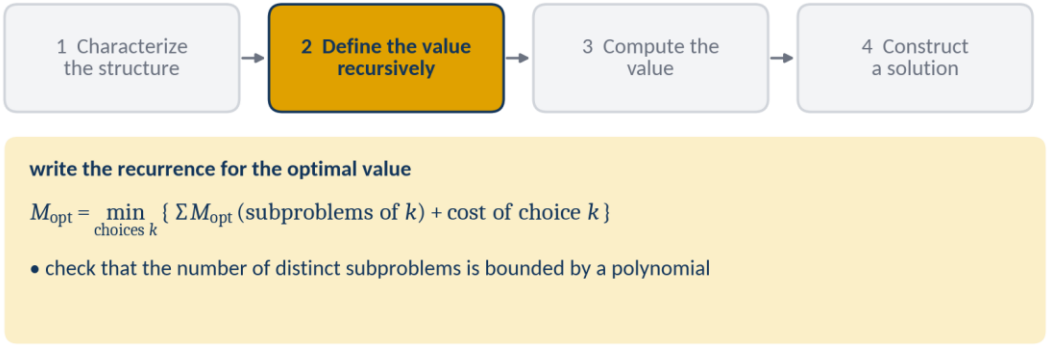


Figure 7.16 Step 2: write the recurrence. Its general form is $M_{opt} = \min_{\text{choices } k}\{(\text{sum of } M_{opt} \text{ of the subproblems produced by } k) + (\text{cost of making choice } k)\}$ — and then show the number of distinct subproblem instances is bounded by a polynomial.

***Figure 7.16**: write the recurrence, and check that the number of distinct subproblems is polynomial. Book: Section 7.8.2.*

The Recipe, Steps 3 and 4

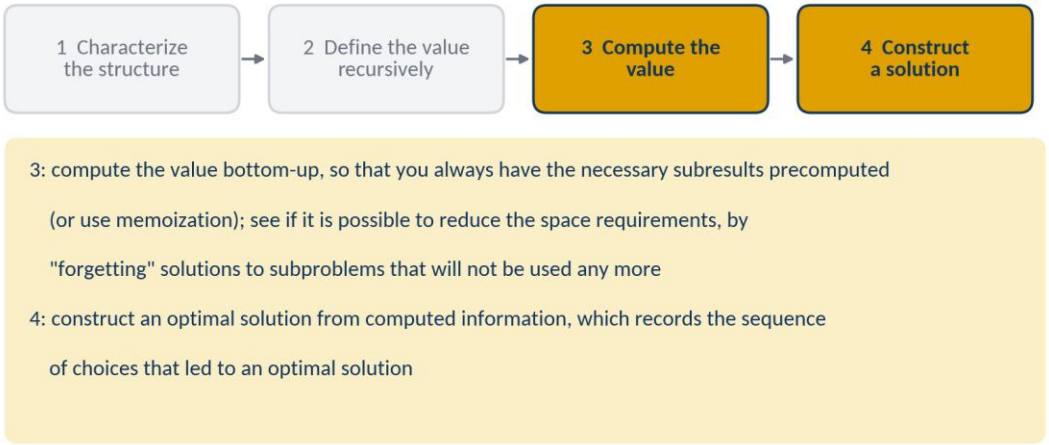


Figure 7.17 Steps 3 and 4: compute the optimal value bottom-up so the needed subresults are always ready (or use memoization), look for a chance to reduce the space by forgetting subproblems that will not be used again, and finally construct an optimal solution from the recorded sequence of choices.

Figure 7.17*: compute bottom-up (or memoize), forget what you no longer need, and construct the solution from the recorded choices. Book: Section 7.8.3.*

The Four Steps, on One Page

Every dynamic program in this book is an instance of the template Equation (7.13):

$$M_{\text{opt}}(\text{problem}) = \min_{\text{all choices } k} \left\{ \sum_{\text{subproblems } s \text{ created by } k} M_{\text{opt}}(s) + \text{cost}(k) \right\}. \tag{7.13}$$

THE FOUR STEPS. 1. Characterize — choice, subproblems, cut-and-paste. **2. Recursively define** — the template, base cases, polynomial count. **3. Compute** — in dependency order, or memoized. **4. Construct** — from recorded choices.

THE MOST COMMON FAILURE. Drawing a table before naming the subproblem. Finish the sentence "the subproblem is: given …, find the best …" first.

Matrix chain fits exactly: choices k, subproblems (i, k) and $(k+1, j)$, cost $d_{i-1}d_k d_j$. Book: Section 7.8.4.

Longest Common Subsequence

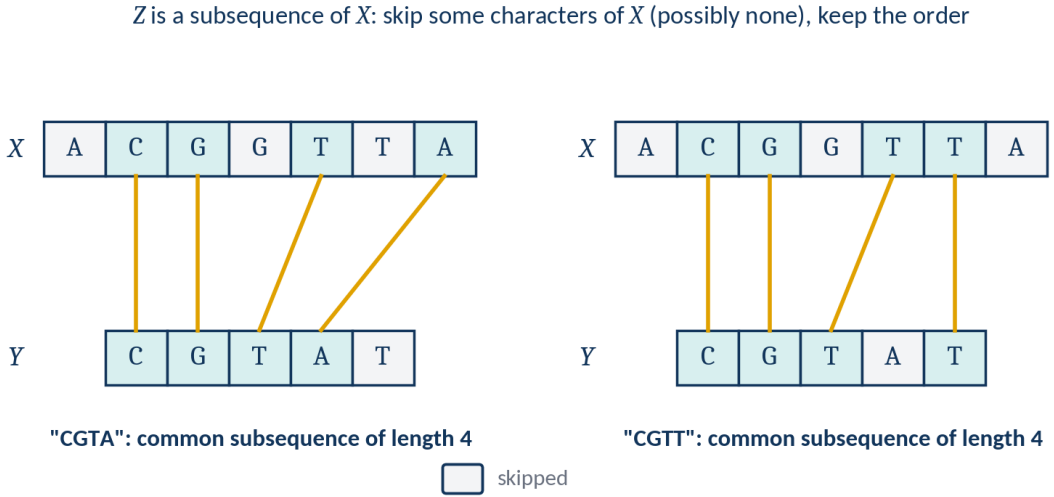


Figure 7.18 The definition: Z is a subsequence of X if Z can be generated by skipping some (possibly none) characters of X. For $X = $ "ACGGTTA" and $Y = $ "CGTAT", LCS(X, Y) is "CGTA" or "CGTT". Solving the problem means finding the *skips*.

***Figure 7.18**: used in DNA comparison, spell checking, `diff`, plagiarism detection — and today's laundry. Book: Sections 7.9.1 and 7.9.2.*

Subsequence, Defined and Checked

DEFINITION. Z is a **subsequence** of X if it is obtained by crossing out some characters and keeping the rest **in order**. An **LCS** of X and Y is a common subsequence of greatest length — *a*, not *the*.

NOT A SUBSTRING. A substring is contiguous; a subsequence need not be. “AGT” is a subsequence of “ACGGTTA” but not a substring.

X = “ACGGTTA”, Y = “CGTAT”: “CGTA” and “CGTT” are common (length 4); “CGTAT” is not. Brute force would test 2^m subsequences — 2^{1000} for $m = 1000$.

We need the dynamic program. Book: Sections 7.9.2 and 7.9.3.

Optimal Substructure of LCS

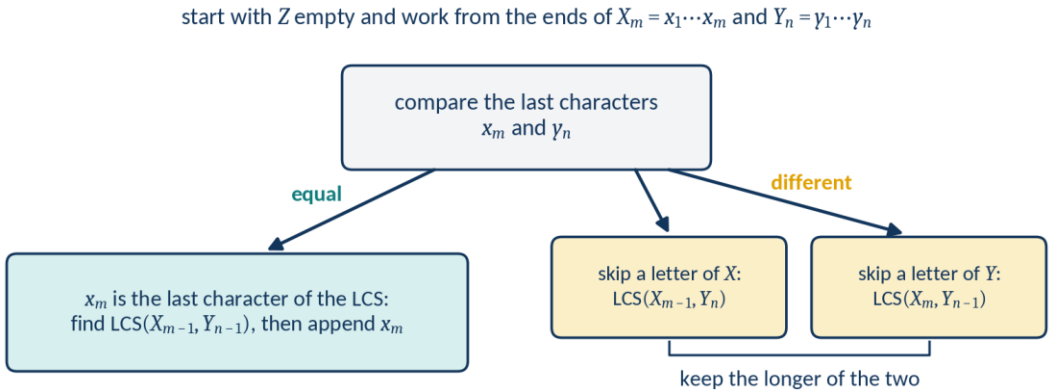


Figure 7.19 The optimal substructure of LCS. Working from the *ends* of X_m and Y_n: if the last characters agree, that character joins the answer and we recurse on both strings shortened by one; if they disagree, we must skip a letter from X or a letter from Y, and we decide which by comparing the two resulting LCS lengths. The justification is the cut-and-paste argument.

Figure 7.19*: work from the ends of the strings; if the last characters agree they join the answer, otherwise skip one of them. Book: Section 7.9.4.*

The Three Cases, Proved

Let X_i be the prefix $x_1 \cdots x_i$, and $Z = z_1 \cdots z_r$ any LCS of X and Y.

(1) If $x_m = y_n$ then $z_r = x_m$ and Z_{r-1} is an LCS of X_{m-1}, Y_{n-1} — otherwise appending x_m, or pasting a longer W, would beat Z.

(2) If $x_m \neq y_n$ and $z_r \neq x_m$, then Z is an LCS of X_{m-1}, Y. **(3)** Symmetrically with Y_{n-1}.

If $x_m \neq y_n$, z_r cannot equal both, so (2) or (3) applies: the cases are exhaustive, and ties explain why several LCSs can exist.

Every proof is a cut-and-paste argument. Book: Section 7.9.4.

The LCS Recurrence

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1],\ c[i-1, j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Figure 7.20 The LCS recurrence: zero if either prefix is empty; one more than the diagonal neighbor when the two characters match; otherwise the larger of the neighbor above and the neighbor to the left.

Figure 7.20*: zero on an empty prefix; diagonal plus one on a match; otherwise the larger of above and left. Book: Section 7.9.5.*

The Recurrence, Written Out

With $c[i, j]$ the LCS length of X_i and Y_j, the recurrence is Equation (7.14); the answer is $c[m, n]$.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max\{c[i, j-1],\ c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases} \tag{7.14}$$

TWO INDEXINGS AT ONCE. The table starts at 0 (empty prefixes); the strings start at 1. In C++ the character at table position (i, j) is `X[i-1]` against `Y[j-1]`.

$(m + 1)(n + 1) = \Theta(mn)$ *subproblems — polynomial, as step 2 requires. Book: Section 7.9.5.*

LCS-Length

Pseudocode 7.9 LCS-Length in pseudocode. Lines 1–4 fill the zero row and zero column. Lines 5–15 sweep the table row by row, filling $c[i, j]$ by the recurrence and recording in $b[i, j]$ which of the three cases was used: `"copy"`, `"skipx"` or `"skipy"`.

```
LCS-LENGTH(X, Y, m, n):
 1.  for i ← 1 to m
 2.      c[i,0] ← 0
 3.  for j ← 0 to n
 4.      c[0,j] ← 0
 5.  for i ← 1 to m
 6.      for j ← 1 to n
 7.          if x_i = y_j then
 8.              c[i,j] ← c[i-1,j-1] + 1
 9.              b[i,j] ← "copy"
10.          else if c[i-1,j] ≥ c[i,j-1] then
11.              c[i,j] ← c[i-1,j]
12.              b[i,j] ← "skipx"
13.          else
14.              c[i,j] ← c[i,j-1]
15.              b[i,j] ← "skipy"
16.  return c, b
```

Pseudocode 7.9*: row by row, recording "copy" (↖), "skipx" (↑) or "skipy" (←) in b. Line 10 compares with ≥, so ties go to skipx. Book: Section 7.9.6.*

The Example to Fill

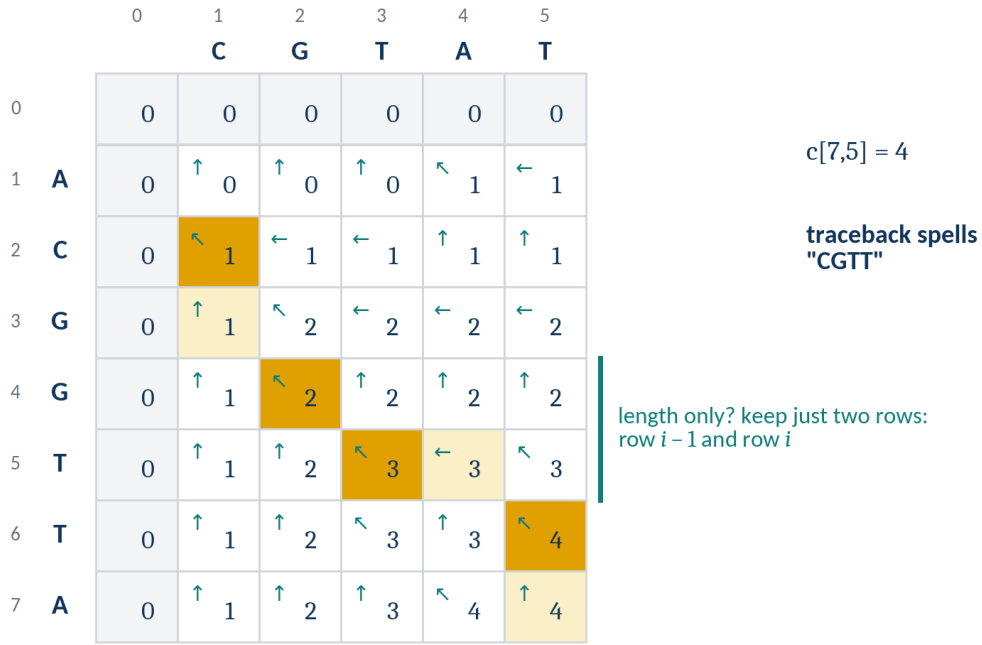


Figure 7.21 The finished tables for $X = $ “ACGGTTA” and $Y = $ “CGTAT”: each cell holds $c[i, j]$ and its arrow $b[i, j]$, and the shaded cells are the traceback that spells “CGTT”. The bar at the side asks the question of §7.9.10: how much space is needed if we do not reconstruct the subsequence itself?

***Figure 7.21**: X = A, C, G, G, T, T, A ($m = 7$) and Y = C, G, T, A, T ($n = 5$). Book: Section 7.9.7.*

The c Table, Cell by Cell

Row 1 (A): only $y_4 =$ A matches, so $c[1,4] = 1$. Row 2 (C): $c[2,1] = 1$ from the match with y_1. Row 3 (G): $c[3,2] = c[2,1] + 1 = 2$. Row 5 (T): $c[5,3] = 3$. Row 6 (T): $c[6,5] = c[5,4] + 1 = 4$. Row 7 (A): $c[7,4] = c[6,3] + 1 = 4$.

Table 7.6 The c table for \$X = \$ "ACGGTTA" and \$Y = \$ "CGTAT", filled cell by cell in §7.9.7. Every entry is the length of an LCS of the two prefixes labeling its row and column. The answer, $c[7,5] = 4$, is the bold cell.

	j=0	j=1 (C)	j=2 (G)	j=3 (T)	j=4 (A)	j=5 (T)
$i = 0$	0	0	0	0	0	0
$i = 1$ **(A)**	0	0	0	0	1	1
$i = 2$ **(C)**	0	1	1	1	1	1
$i = 3$ **(G)**	0	1	2	2	2	2
$i = 4$ **(G)**	0	1	2	2	2	2
$i = 5$ **(T)**	0	1	2	3	3	3
$i = 6$ **(T)**	0	1	2	3	3	4
$i = 7$ **(A)**	0	1	2	3	4	**4**

***Table 7.6**: entries never decrease rightward or downward, and neighbors differ by at most* 1*. The answer is* $c[7,5] = 4$*. Book: Section 7.9.7.*

The Arrows, and the Walk Back

Table 7.7 The b table of arrows for the same instance. ↖ is `"copy"` (characters matched), ↑ is `"skipx"` (value taken from above), ← is `"skipy"` (value taken from the left). Following the arrows backward from the bottom-right corner reconstructs an LCS.

	j=1 (C)	j=2 (G)	j=3 (T)	j=4 (A)	j=5 (T)
$i = 1$ **(A)**	↑	↑	↑	↖	←
$i = 2$ **(C)**	↖	←	←	↑	↑
$i = 3$ **(G)**	↑	↖	←	←	←
$i = 4$ **(G)**	↑	↖	↑	↑	↑
$i = 5$ **(T)**	↑	↑	↖	←	↖
$i = 6$ **(T)**	↑	↑	↖	↑	↖
$i = 7$ **(A)**	↑	↑	↑	↖	↑

From (7,5) in **Table 7.7**: ↑ to (6,5); ↖ emits T; ← to (5,3); ↖ emits T; ↖ emits G; ↑ to (2,1); ↖ emits C; stop at $j = 0$. Emitted backward as T, T, G, C, so LCS(X, Y) = "CGTT". The tie rule ≥ chose it over "CGTA"; both are correct.

The walk takes $O(m+n)$ — negligible beside the $\Theta(mn)$ fill. Book: Sections 7.9.8 and 7.9.9.

$\Theta(mn)$ Time, $\Theta(\min(m,n))$ Space

Time: $m \cdot n$ cells, $O(1)$ each — no minimization loop, so LCS is quadratic where matrix chain is cubic.

Space: $c[i,j]$ reads only row $i-1$ and row i, so two rows suffice; put the shorter string across the columns for $\Theta(\min(m,n))$.

> **THE SPACE RESULT.** The *length* needs only $\Theta(\min(m,n))$ space; the *subsequence* needs the b table (or Hirschberg's refinement). Measured at $m = n = 8000$: 488 MiB and 1 931 ms for the full table, 0.061 MiB and 153 ms for two rows.

Two DNA strings of a million characters: four terabytes for the full table, about eight megabytes for two rows. Book: Sections 7.9.10 and 7.9.11.

Overlapping versus Independent Subproblems

Dynamic programming needs **both** hallmarks: **optimal substructure** (the recurrence exists) and **overlapping subproblems** (few distinct subproblems, visited many times).

Measured ratio of calls to distinct subproblems: Merge Sort 1.00 at every size; naive Fibonacci 7.44, 187.82, 6002; plain matrix chain 2.70, 60.75, 2271.12.

Table 7.8 Divide and conquer against dynamic programming, side by side. The first row is the one that decides which you are looking at; everything else follows from it.

	Divide and conquer	Dynamic programming
Subproblems	Disjoint, all different	Overlapping, few distinct ones
Direction	Top-down only	Top-down with a memo, or bottom-up
Extra memory	None beyond the stack	A table of subproblem answers
Typical shape	$T(n) = aT(n/b) + f(n)$	value(s) = opt over choices
Solved by	Master Theorem (§3.13)	Counting cells × cost per cell
Examples	Merge Sort, binary search, Strassen	Fibonacci, matrix chain, LCS
Failure mode	—	Exponential blow-up if not memoized

***Table 7.8**: the first row decides which you are looking at. A ratio near 1 means leave it alone; a growing ratio means memoize or tabulate. Book: Section 7.10.*

Profitina in Practice: Matching the Slips

In Profitina, near-duplicate page detection uses LCS-style similarity in two-row space; the rewrite "what changed" view is the arrow walk; chained transformations are a short matrix chain.

Profitina Laundry matches handwritten slips to the price list with Equation (7.15), after folding to lower case and removing spaces:

$$\text{similarity}(u, v) = \frac{|\text{LCS}(u, v)|}{\max(|u|, |v|)}, \tag{7.15}$$

`bedcover besar` scores 1.000; `cuci kring expres` 0.882; `cuci kering reg` 0.765. "titip motor sebentar" scores only 0.389, and below 0.600 the app refuses to guess and asks a person.

The app advises; the user decides. No personal field ever enters a similarity computation (UU PDP 27/2022). Book: Sections 7.11 and 7.11.1.

Common Mistakes

Blaming recursion instead of repetition; **filling the table in the wrong order** (loop over chain length).

Getting k's range wrong ($i \leq k < j$); **confusing subsequence with substring**.

A sentinel a real answer could equal; **forgetting the choice table**, then needing the solution.

Off-by-one between string and table indices; **assuming** optimal substructure instead of proving it.

Deep memoized recursion overflowing the stack; reporting the reversed LCS; keeping a table you do not need; memoizing when nothing overlaps.

Twelve mistakes in all, each with its fix. Book: Section 7.12.

Summary: Five Things to Remember

1. Divide and conquer wants disjoint subproblems; when they repeat, a plain recursion is exponential — Fibonacci makes $2F_{n+1} - 1 = \Theta(\phi^n)$ calls.

2. Memoization and bottom-up tabulation are the two forms of dynamic programming; analyze them by cells × cost per cell.

3. Dynamic programming needs optimal substructure (cut-and-paste) and overlapping subproblems (a polynomial count).

4. Matrix chain: $M(i,j) = \min_{i \le k < j}\{M(i,k) + M(k+1,j) + d_{i-1}d_k d_j\}$, filled by length, $\Theta(n^3)$; the answer for $\langle 10,20,3,5,30 \rangle$ is 1 950.

5. LCS: diagonal $+1$ on a match, else the larger neighbor; $\Theta(mn)$ time; "CGTT" by the arrows; two rows for the length alone.

The four-step recipe applies to problems you have never seen. Book: Section 7.13.

Next Time

Chapter 8 — Evaluation and Discussion II. Twenty-five problems drawn from Chapters 5 to 7, worked in full: heaps, partitions, search trees, AVL and red-black trees, matrix chains and LCS.

Before then: Exercises 7.7 (a six-matrix chain) and 7.11 (an LCS table), by hand.

Later, Chapter 9: greedy algorithms — optimal substructure plus the greedy-choice property.

Bring Chapters 5, 6 and 7 with you: every problem next time uses one of them. Book: Sections 7.13.1 and 7.14.

Lecture 8 — Evaluation and Discussion II: Sorting, Search Trees, and Dynamic Programming

Course: Design and Analysis of Algorithms (EF234405)

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*, Chapter 8

Lecturer: Irfan Subakti — yifana@gmail.com

No new theory today: we use Chapters 5, 6 and 7 together on problems large enough that several ideas must cooperate. Book: Chapter 8, Sections 8.1–8.21.

What This Meeting Covers

1. **Sorting** — Problem 8.1, a max-heap built by hand; Problem 8.2, Quicksort's partition, best and worst cases counted exactly.
2. **Search trees** — Problems 8.3 to 8.6: `isBST`, `printDescending`, `max`, `isHeightBalanced`.
3. **AVL maintenance** — Problems 8.7 to 8.10: `insertHB`, the two rebalancers, `deleteHB`.
4. **Red-black trees** — Problems 8.11 to 8.15: four validators and two rotations.
5. **Dynamic programming** — Problem 8.16, a matrix chain; Problem 8.17, SORTED against STORE.
6. **Quick checks, the order-statistic tree** (Problems 8.18 to 8.25) and Profitina Laundry's self-auditing index.

Every problem is stated, analyzed, solved step by step, and checked by a program whose output is printed in the book. Book: Chapter 8.

Where We Stand

Chapter 5: heapsort and Quicksort. Chapter 6: binary search trees and AVL trees. Chapter 7: memoization and dynamic programming. Table 8.1 maps every problem of today to its section and its theory.

Table 8.1 The problems of this chapter, the section that solves each one, and the chapter whose theory it uses. Problem numbers and section numbers are different sequences; the table translates between them.

Problem	Section	What it asks	Theory from
8.1	8.3	Build a max-heap by hand, then heapsort it	Ch. 5
8.2	8.4	Trace Quicksort's partition; best and worst cases	Ch. 5
8.3	8.6	`isBST`: is this binary tree a search tree?	Ch. 6
8.4	8.7	`printDescending`: print a BST largest first, building no list	Ch. 6
8.5	8.8	`max`: the largest key along one path only	Ch. 6
8.6	8.9	`isHeightBalanced`: the AVL condition in one pass	Ch. 6, 7
8.7	8.10	`insertHB`: insertion that keeps the tree balanced	Ch. 6
8.8	8.11	`rebalanceForLeft`: the cases LL and LR	Ch. 6
8.9	8.12	`rebalanceForRight`: the cases RR and RL	Ch. 6
8.10	8.13	`deleteHB`: deletion that keeps the tree balanced	Ch. 6
8.11–8.15	8.14	Red-black trees: four validators and two rotations	Ch. 6
8.16	8.15	Matrix-chain multiplication, the table filled by hand	Ch. 7
8.17	8.16	Longest common subsequence, the table and the walk back	Ch. 7
8.18–8.25	8.18	Further problems on an order-statistic tree	Ch. 5, 6, 7

THE ONE IDEA. Carry information down the recursion, or return it up, so that no piece of work is ever done twice.

Problem numbers and section numbers are different sequences; the table translates between them. Book: Section 8.1.

How to Approach a Multi-Part Problem

Read every part first and draw the dependencies. Problems 8.7 to 8.10 are best solved in the order 8.8, 8.9, 8.7, 8.10 — each routine is then written once.

Before coding, write five things: the exact signature; the base case in words; the invariant or contract in one sentence; the smallest failing example; the complexity claim with its half-sentence justification.

Test tiny cases first — empty, one element, two elements, an absent key — and write down what is not yet handled.

> **THE MIRROR PRINCIPLE.** Reflecting a tree swaps left and right, `<` and `>`, minimum and maximum, and the two rotations. Spot a mirror pair and the work halves.

This is the five-phase framework of Chapter 1, compressed to fit one problem. Book: Section 8.2.

Problem 8.1 — Building a Max-Heap

$A = [5, 13, 2, 25, 7, 17, 20, 8, 4, 11]$: draw the tree, run `BUILD-MAX-HEAP` with every swap, count comparisons and swaps, heapsort the result, and explain the $O(n)$ bound.

Pseudocode 8.1 `MAX-HEAPIFY`: the value at node i sinks until neither child is larger than it.

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```

Pseudocode 8.2 `BUILD-MAX-HEAP`: repair the internal nodes from the bottom up.

```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Pseudocode 8.1 lets one value sink; Pseudocode 8.2 calls it on nodes $\lfloor n/2 \rfloor, \ldots, 1$, counting *down* so both subtrees are heaps first.

Here $n = 10$: internal nodes 1–5, leaves 6–10. The root holds 5 while 25 sits two levels down. Book: Sections 8.3.1–8.3.3.

Problem 8.1 — The Trace, One Call per Row

$i=5$: 11>7, swap. $i=4$: 25 already dominates. $i=3$: 17>2, then 20>17 — compare with the *current champion* — swap with 20. $i=2$: swap 13 with 25. $i=1$: 5 sinks three levels, past 25, 13 and 8.

Table 8.2 `BUILD-MAX-HEAP` on $A=[5,13,2,25,7,17,20,8,4,11]$, one row per call of `MAX-HEAPIFY`. The root's value sinks three levels in the last call.

Call	Value at node i	Comparisons	Swaps	A After the call
`MAX-HEAPIFY(A, 5)`	7	1	1	[5,13,2,25,11,17,20,8,4,7
`MAX-HEAPIFY(A, 4)`	25	2	0	[5,13,2,25,11,17,20,8,4,7
`MAX-HEAPIFY(A, 3)`	2	2	1	[5,13,20,25,11,17,2,8,4,7
`MAX-HEAPIFY(A, 2)`	13	4	1	[5,25,20,13,11,17,2,8,4,7
`MAX-HEAPIFY(A, 1)`	5	6	3	[25,13,20,8,11,17,2,5,4,7
Total		**15**	**6**	

Table 8.2: 15 comparisons and 6 swaps give $[25,13,20,8,11,17,2,5,4,7]$. Check each parent: $25 \geq 13,20$; $13 \geq 8,11$; $20 \geq 17,2$; $8 \geq 5,4$; $11 \geq 7$ ✓.

Check yourself: $[3,9,2,1,4,5]$ *becomes* $[9,4,5,1,3,2]$ *after* 7 *comparisons and* 3 *swaps. Book: Sections 8.3.4 and 8.3.8.*

Problem 8.1 — Heapsort, and Why $O(n)$

Extractions: $[20,13,17,8,11,7,2,5,4|25]$, then $[17,13,7,8,11,4,2,5|20,25]$, then $[13,11,7,8,5,4,2|17,20,25]$; after nine, $[2,4,5,7,8,11,13,17,20,25]$.

A node of height h costs $O(h)$, and at most $\lceil n/2^{h+1} \rceil$ nodes have height h, which gives Equation (8.1):

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(2n) = O(n), \tag{8.1}$$

On a perfect heap the total sinking is exactly Equation (8.2) — measured: $32,767$ ascending keys need $32,752$ swaps.

$$\sum_{\ell=0}^{k-1} 2^{\ell}(k-1-\ell) = 2^k - k - 1 = n - \lg(n+1). \tag{8.2}$$

Common wrong turns: counting the loop upward; comparing the right child with $A[i]$; forgetting to shrink the heap in heapsort. Book: Sections 8.3.5–8.3.9.

Problem 8.2 — Quicksort's Partition

$A = [7,2,9,4,1,8,3,6,5]$: trace Lomuto's partition, then the whole Quicksort; find the slowest and fastest inputs and count their comparisons exactly.

Pseudocode 8.3 Lomuto's `PARTITION`: one left-to-right sweep that grows a region of values $\leq x$.

```
PARTITION(A, p, r)
1  x = A[r]                        // the pivot
2  i = p − 1                       // A[p..i] holds the values ≤ x found so far
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]     // drop the pivot into the gap
8  return i + 1
```

Pseudocode 8.4 `QUICKSORT`: partition, then recurse on the two sides of the pivot.

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

Pseudocode 8.3 keeps four regions — $\leq x$, $> x$, unexamined, pivot; Pseudocode 8.4 never touches the pivot again. Book: Sections 8.4.1 and 8.4.2.

Problem 8.2 — The Partition, Step by Step

Table 8.3 Lomuto's PARTITION on $[7,2,9,4,1,8,3,6,5]$ with pivot $x=5$. Eight comparisons, four exchanges in the loop and one at the end; the pivot finishes at index $q=5$.

j	$A[j]$	$A[j]\le 5$?	i After	Exchange	A After the step
1	7	no	0	—	$[7,2,9,4,1,8,3,6,5]$
2	2	yes	1	$A[1] \leftrightarrow A[2]$	$[2,7,9,4,1,8,3,6,5]$
3	9	no	1	—	$[2,7,9,4,1,8,3,6,5]$
4	4	yes	2	$A[2] \leftrightarrow A[4]$	$[2,4,9,7,1,8,3,6,5]$
5	1	yes	3	$A[3] \leftrightarrow A[5]$	$[2,4,1,7,9,8,3,6,5]$
6	8	no	3	—	$[2,4,1,7,9,8,3,6,5]$
7	3	yes	4	$A[4] \leftrightarrow A[7]$	$[2,4,1,3,9,8,7,6,5]$
8	6	no	4	—	$[2,4,1,3,9,8,7,6,5]$

Table 8.3 (Continued) *Lomuto's PARTITION on [7,2,9,4,1,8,3,6,5] with pivot x=5. Eight comparisons, four exchanges in the loop and one at the end; the pivot finishes at index q=5.*

j	$A[j]$	$A[j]\le 5$?	i After	Exchange	A After the step
line 7				$A[5] \leftrightarrow A[9]$	$[2,4,1,3,5,8,7,6,9]$

Table 8.3: pivot 5 lands at $q=5$ after exactly $r-p=8$ comparisons — a count that does not depend on the data.

The recursion: $[1..4]$ pivot 3 (3 comparisons), $[1..2]$ pivot 1 (1), $[6..9]$ pivot 9 (3), $[6..8]$ pivot 6 (2), $[7..8]$ pivot 8 (1). Total $8+3+1+3+2+1=18$.

Check yourself: partitioning $[3,8,1,6,2,7,5,4]$ *gives* $[3,1,2,4,8,7,5,6]$, $q=4$, *after* 7 *comparisons. Book: Sections 8.4.3, 8.4.4 and 8.4.8.*

Problem 8.2 — Worst and Best, Counted Exactly

Worst: ascending keys make the pivot the maximum every time: $C(n)=C(n-1)+(n-1)=\frac{n(n-1)}{2}$, that is Equation (8.3).

$$T(n)=T(n-1)+\Theta(n)\Rightarrow T(n)=\Theta(n^2). \tag{8.3}$$

Best: median pivots on $n=2^k-1$ give $C(n)=2C\left(\frac{n-1}{2}\right)+(n-1)$, that is Equation (8.4), with the exact count Equation (8.5) proved by induction.

$$T(n)=2T(n/2)+\Theta(n)\Rightarrow T(n)=\Theta(n\lg n), \tag{8.4}$$

$$C(n)=(n+1)\lg(n+1)-2n. \tag{8.5}$$

Measured at $n=1023$: sorted $522,753$ comparisons, best $8,194$, random $10,637$. Randomizing the pivot makes $\Theta(n\lg n)$ the expected cost on every input. Book: Sections 8.4.5–8.4.9.

The Data Type the Tree Problems Use

DEFINITION. A `MyTree` is empty ($\varnothing$) or a triple (v, L, R). Heights follow Equation (8.6), and the balance factor is Equation (8.7) — negative means left-heavy.

$$h(\varnothing) = -1, h((v, L, R)) = 1 + \max(h(L), h(R)). \tag{8.6}$$

$$bf((v, L, R)) = h(R) - h(L), \tag{8.7}$$

Listing 8.1 The `MyTree` type in C++, with the single constructor `node()` that keeps the cached height honest, and a shape printer that writes a whole tree on one line.

```cpp
// mytree.cpp -- the MyTree data type on which every tree problem of this chapter is built.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

struct MyTreeNode;
using MyTree = MyTreeNode*;              // the null pointer IS the empty tree

struct MyTreeNode {
    int    value;                        // the integer stored at this node
    MyTree left, right;                  // the two subtrees
    int    h;                            // cached height; the empty tree has height -1
};

MyTree emptyTree()          { return nullptr; }
bool   isEmpty(MyTree t)    { return t == nullptr; }
int    height(MyTree t)     { return isEmpty(t) ? -1 : t->h; }
int    rootValue(MyTree t)  { return t->value; }
MyTree leftOf(MyTree t)     { return t->left; }
MyTree rightOf(MyTree t)    { return t->right; }

// node(v,l,r): the single constructor. It recomputes the height, so a tree
// assembled only through node() can never hold a stale height.
MyTree node(int v, MyTree l, MyTree r) {
    MyTree t = new MyTreeNode;
    t->value = v; t->left = l; t->right = r;
    t->h = 1 + max(height(l), height(r));
```

```
    return t;
}
MyTree leaf(int v) { return node(v, emptyTree(), emptyTree()); }

int balanceFactor(MyTree t) { return height(t->right) - height(t->left); }

// shape(t) prints value(balanceFactor){left,right}; '.' is the empty tree.
string shape(MyTree t) {
    if (isEmpty(t)) return ".";
    string s = to_string(t->value) + "(" + to_string(balanceFactor(t)) + ")";
    if (isEmpty(t->left) && isEmpty(t->right)) return s;      // a leaf needs no braces
    return s + "{" + shape(t->left) + "," + shape(t->right) + "}";
}

int countNodes(MyTree t) {
    return isEmpty(t) ? 0 : 1 + countNodes(t->left) + countNodes(t->right);
}
```

Listing 8.1 (Continued) *The MyTree type in C++, with the single constructor node() that keeps the cached height honest, and a shape printer that writes a whole tree on one line.*

```
void inorder(MyTree t) {
    if (isEmpty(t)) return;
    inorder(t->left);
    cout << t->value << ' ';
    inorder(t->right);
}

int main() {
    // The eight-node example tree of Chapter 6, assembled by hand.
    MyTree t = node(13,
                    node(10, node(5, leaf(4), leaf(6)), leaf(11)),
                    node(15, emptyTree(), leaf(16)));

    cout << "shape   : " << shape(t) << "\n";
    cout << "nodes   : " << countNodes(t) << "\n";
    cout << "height  : " << height(t) << "\n";
    cout << "inorder : "; inorder(t); cout << "\n";
    cout << "empty   : height " << height(emptyTree())
         << ", isEmpty = " << (isEmpty(emptyTree()) ? "true" : "false") << "\n";
    return 0;
}
```

```
shape   : 13(-1){10(-1){5(0){4(0),6(0)},11(0)},15(1){.,16(0)}}
nodes   : 8
height  : 3
inorder : 4 5 6 10 11 13 15 16
empty   : height -1, isEmpty = true
```

Listing 8.1: `node()` is the only constructor and recomputes the height every time, so a rotation written as a fresh `node(...)` can never leave a stale height. Book: Section 8.5.

Problem 8.3 — `isBST`

Write `bool isBST(MyTree t)`, recursively, with a helper `isBST(t, lowerBound, upperBound)`: is t a BST whose values all lie strictly between the bounds?

The local check — compare each node with its two children — is short, $\Theta(n)$, and **wrong**. Tree: 20 with left 10 $[5, 25]$ and right 30. Every parent–child test passes, but $25 > 20$ sits in 20's left subtree; the inorder walk $5, 10, 25, 20, 30$ is not increasing.

> **THE TRAP.** The local check is *necessary* but not *sufficient*. Three nodes already expose it: 20, left child 10, and 10's right child 25.

The plural in the definition — branches, not children — is the whole question. Book: Sections 8.6.1–8.6.3.

The Definition, Read Carefully

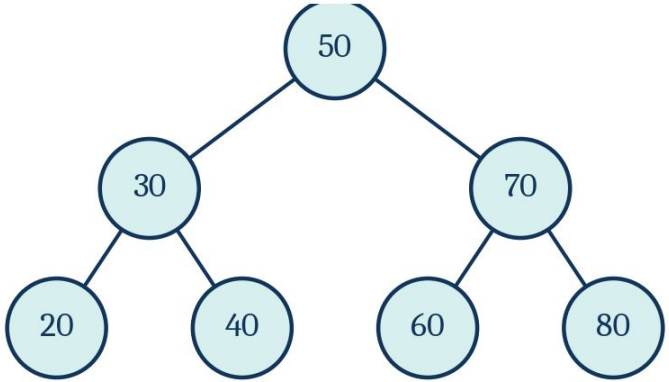


Figure 8.1 A binary search tree, illustrating the definition: “Left branch(es) contains the nodes whose values $<$ the value of the node” — branches, plural, not children.

Figure 8.1: every node anywhere in the left subtree is smaller — not just the left child. Book: Section 8.6.2.

Problem 8.3 — Deriving the Right Answer

Split the property at the root, Equation (8.8); the two quantifiers are range constraints $(-\infty, v)$ and $(v, +\infty)$.

$$\mathrm{BST}(t) \Leftrightarrow (\forall u \in L : u < v) \wedge (\forall u \in R : u > v) \wedge \mathrm{BST}(L) \wedge \mathrm{BST}(R). \tag{8.8}$$

Generalize to $P(t, \ell, u)$: $\ell < v < u$, $P(L, \ell, v)$ and $P(R, v, u)$; $P(\varnothing, \ell, u)$ is true; $BST(t) \Leftrightarrow P(t, -\infty, +\infty)$.

Table 8.4 The correct recursion traced on the trap tree $20\,[10\,[5, 25], 30]$. The bound $u = 20$ reaches node 25 through *two* levels of recursion; that is the information the local check throws away.

Call	Node	ℓ	u	$\ell < v < u$?	Result
1	20	$-\infty$	$+\infty$	$-\infty < 20 <$ ✓	recurse
2	10	$-\infty$	20	$-\infty < 10 < 2($ ✓	recurse
3	5	$-\infty$	10	$-\infty < 5 < 10$ ✓	both subtrees empty → **true**
4	25	10	20	$25 < 20$? **no**	**false**
5	—	—	—	—	call 2 returns **false**
6	—	—	—	—	call 1 returns **false**; node 30 is never visited

Table 8.4: node 25 is rejected by the bound 20 inherited from its grandparent. Every node owns an interval of the key space — the same move as Quicksort's partition. Book: Sections 8.6.4 and 8.6.5.

Problem 8.3 — The Program

Listing 8.2 holds the bounded-interval answer, the local check for contrast, the inorder-walk alternative, and a diagnostic that names the first offending node.

Listing 8.2 Problem 8.3 in full: the bounded-interval answer, the local check that fails, the inorder-walk alternative, and a diagnostic that names the first offending node.

```cpp
// isbst.cpp -- Problem 8.3: isBST(), and the tempting wrong answer.
#include <iostream>
#include <string>
#include <climits>
#include <algorithm>
using namespace std;

struct MyTreeNode;
using MyTree = MyTreeNode*;
struct MyTreeNode { int value; MyTree left, right; int h; };

MyTree emptyTree()       { return nullptr; }
bool   isEmpty(MyTree t) { return t == nullptr; }
int    height(MyTree t)  { return isEmpty(t) ? -1 : t->h; }
MyTree node(int v, MyTree l, MyTree r) {
    MyTree t = new MyTreeNode; t->value = v; t->left = l; t->right = r;
    t->h = 1 + max(height(l), height(r)); return t;
}
MyTree leaf(int v) { return node(v, emptyTree(), emptyTree()); }
string shape(MyTree t) {
    if (isEmpty(t)) return ".";
    string s = to_string(t->value);
    if (isEmpty(t->left) && isEmpty(t->right)) return s;
    return s + "{" + shape(t->left) + "," + shape(t->right) + "}";
}

// ---- The WRONG answer: check only the parent against its two children. ----
bool isBSTLocal(MyTree t) {
```

```
    if (isEmpty(t)) return true;
    if (!isEmpty(t->left)  && t->left->value  >= t->value) return false;
    if (!isEmpty(t->right) && t->right->value <= t->value) return false;
    return isBSTLocal(t->left) && isBSTLocal(t->right);
}

// ---- The right answer: carry an open interval down the recursion. ----
// Every value in the subtree rooted at t must satisfy lowerBound < v < upperBound.
// long long bounds are used so that INT_MIN and INT_MAX are legal tree values.
static bool isBST(MyTree t, long long lowerBound, long long upperBound) {
    if (isEmpty(t)) return true;                                  // base case
    if (t->value <= lowerBound || t->value >= upperBound) return false;
    return isBST(t->left,  lowerBound,   t->value)                // left: (lo, v)
        && isBST(t->right, t->value,     upperBound);             // right: (v, hi)
}
bool isBST(MyTree t) {
    return isBST(t, (long long)INT_MIN - 1, (long long)INT_MAX + 1);
}
```

Listing 8.2 (Continued) *Problem 8.3 in full: the bounded-interval answer, the local check that fails, the inorder-walk alternative, and a diagnostic that names the first offending node.*

```
// ---- Alternative answer: the inorder walk must be strictly increasing. ----
static bool inorderIncreasing(MyTree t, long long& prev) {
    if (isEmpty(t)) return true;
    if (!inorderIncreasing(t->left, prev)) return false;
    if (t->value <= prev) return false;        // the one comparison that matters
    prev = t->value;
    return inorderIncreasing(t->right, prev);
}
bool isBSTByWalk(MyTree t) {
    long long prev = (long long)INT_MIN - 1;
    return inorderIncreasing(t, prev);
}

// ---- Diagnostic version: name the node and the bound it broke. ----
static bool audit(MyTree t, long long lo, long long hi, string& why) {
    if (isEmpty(t)) return true;
    if (t->value <= lo) {
        why = "node " + to_string(t->value) + " must be > " + to_string(lo);
        return false;
    }
    if (t->value >= hi) {
        why = "node " + to_string(t->value) + " must be < " + to_string(hi);
        return false;
    }
    return audit(t->left, lo, t->value, why) && audit(t->right, t->value, hi, why);
}
```

```
void report(const char* name, MyTree t) {
    string why = "-";
    bool ok = audit(t, (long long)INT_MIN - 1, (long long)INT_MAX + 1, why);
    cout << name << "\n";
    cout << "  shape        : " << shape(t) << "\n";
    cout << "  isBSTLocal   : " << (isBSTLocal(t)  ? "true " : "false") << "   (the wrong method)\n";
    cout << "  isBST        : " << (isBST(t)       ? "true " : "false") << "\n";
    cout << "  isBSTByWalk  : " << (isBSTByWalk(t) ? "true " : "false") << "\n";
    cout << "  first fault  : " << (ok ? "none" : why) << "\n\n";
}

int main() {
    // A genuine BST.
    report("good : root 20, left 10{5,15}, right 30{25,35}",
           node(20, node(10, leaf(5), leaf(15)), node(30, leaf(25), leaf(35))));
```

Listing 8.2 (Continued) *Problem 8.3 in full: the bounded-interval answer, the local check that fails, the inorder-walk alternative, and a diagnostic that names the first offending node.*

```
    // The classic counterexample: every parent-child pair is fine, but 25
    // sits in the LEFT subtree of 20 and 25 > 20.
    report("trap : root 20, left 10{5,25}, right 30",
           node(20, node(10, leaf(5), leaf(25)), leaf(30)));

    // A duplicate value. The BST definition used here forbids equal keys.
    report("dup  : root 20, left 20, right 30",
           node(20, leaf(20), leaf(30)));

    // A single node holding INT_MIN: the sentinel test.
    report("edge : the single node INT_MIN", leaf(INT_MIN));

    // The empty tree is a BST by convention.
    report("empty: the empty tree", emptyTree());
    return 0;
}
```

```
good : root 20, left 10{5,15}, right 30{25,35}
  shape        : 20{10{5,15},30{25,35}}
  isBSTLocal   : true    (the wrong method)
  isBST        : true
  isBSTByWalk  : true
  first fault  : none

trap : root 20, left 10{5,25}, right 30
  shape        : 20{10{5,25},30}
  isBSTLocal   : true    (the wrong method)
  isBST        : false
  isBSTByWalk  : false
  first fault  : node 25 must be < 20

dup  : root 20, left 20, right 30
  shape        : 20{20,30}
  isBSTLocal   : false   (the wrong method)
  isBST        : false
  isBSTByWalk  : false
  first fault  : node 20 must be < 20

edge : the single node INT_MIN
  shape        : -2147483648
  isBSTLocal   : true    (the wrong method)
  isBST        : true
  isBSTByWalk  : true
  first fault  : none
```

```
empty: the empty tree
  shape        : .
  isBSTLocal   : true    (the wrong method)
  isBST        : true
  isBSTByWalk  : true
  first fault  : none
```

On the trap tree the local check says true, `isBST` says false, and the diagnostic reports "node 25 must be < 20". Book: Section 8.6.5.

Problem 8.3 — The `int` Pitfall, and a Second Method

Tempting but wrong: seeding `int` bounds with `INT_MIN` and `INT_MAX`. A one-node tree holding `INT_MIN` fails `INT_MIN <= INT_MIN` and is rejected. Repairs: `long long` bounds, `std::optional` bounds, or the inorder walk.

Second method: a tree is a BST iff its inorder walk is strictly increasing. Share `prev` by reference and compare with `<=`.

Not the heap property: a heap's rule is parent against children — genuinely local — so the local check *is* the heap check. The BST rule is global.

Wrong turns, ranked: the local check; non-strict bounds; swapped bounds; no base case; an exposed helper; `int` sentinels. Book: Sections 8.6.6–8.6.9.

Problem 8.4 — `printDescending`, No List

Print a BST largest first, **without making a separate list of its values**. Inorder (left, node, right) is ascending, so its mirror — **right, node, left** — is descending.

$printDescending(\varnothing)$ does nothing; otherwise $printDescending(R)$, print v, $printDescending(L)$. Four lines, $\Theta(n)$ time, $O(h)$ stack.

On the eight-node tree: `16 15 13 11 10 6 5 4`. An iterative version pushes each right spine onto a stack of at most $h+1$ pending nodes — the recursion stack made explicit, not a list.

A traversal is three statements in some order; "descending" means moving one line. Test on $10\,[5, 15]$*: the answer must be* `15 10 5`*. Book: Section 8.7.*

Inorder Successors

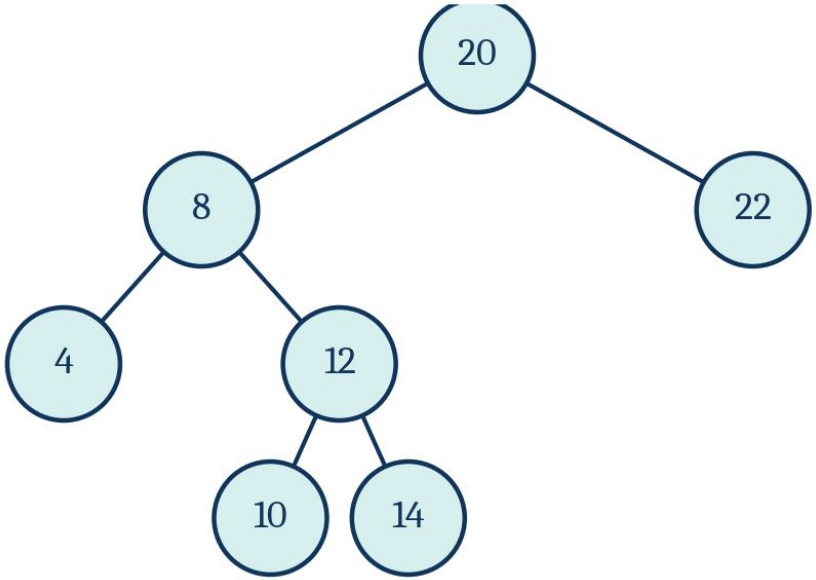


Figure 8.2 The inorder successor. Chaining the successors — 8, 10, 12, 14, 20 — traces the sorted order that the inorder walk produces. Reversing the chain gives descending order.

Figure 8.2: chaining the successors traces the sorted order; reversing the chain gives descending order. Book: Section 8.7.2.

Problem 8.5 — `max` Along One Path

Return the largest value **without visiting all the nodes**. Every left-subtree value is smaller than v, so discard the left subtree unexamined; if the right subtree is non-empty, the answer is its maximum, else v.

The walk follows the **right spine**: at most $h+1$ nodes, so $O(h)$. Measured on a balanced 15-node tree: 4 nodes against 15 for a full scan.

State the bound so that it is true: $O(h)$ always; $O(\lg n)$ only on a balanced tree. On a left-leaning chain of ten, `min` visits all ten.

The empty tree has no maximum: throw, or document a precondition — but never return `INT_MIN`, which is a legal value. Book: Section 8.8.

Problem 8.6 — The AVL Condition

DEFINITION. A tree is **height-balanced** when Equation (8.9) holds, with the empty tree of height -1. The empty tree is height-balanced.

$$\left|bf(x)\right| \le 1 \text{ for every node } x \text{ of } t, \tag{8.9}$$

Write `bool isHeightBalanced(MyTree t)`, aiming for a single pass. The next two pages show one tree of each kind.

The condition is about every node; checking only the root is the local-check mistake again. Book: Sections 8.9.1 and 8.9.2.

An AVL Tree

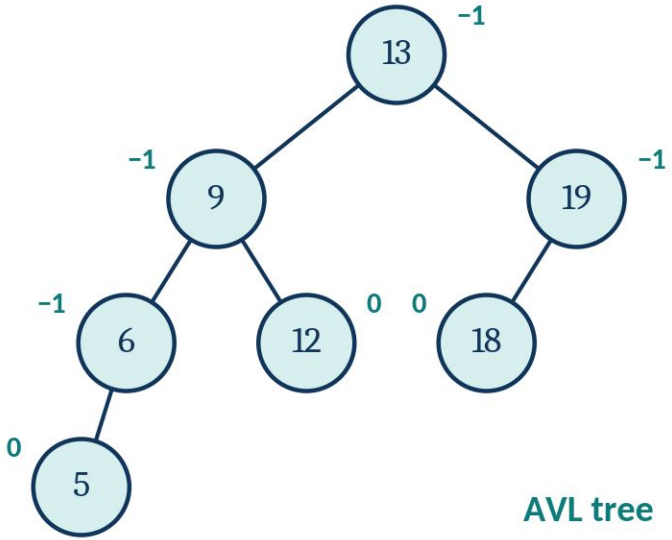


Figure 8.3 A tree that **is** an AVL tree. Every node's two subtrees differ in height by at most one.

Figure 8.3: node 6 *has* $bf = -1$*, node* 9 *has* -1*, node* 19 *has* -1*, the root has* -1*; every leaf has* 0*. Book: Section 8.9.2.*

And One That Is Not

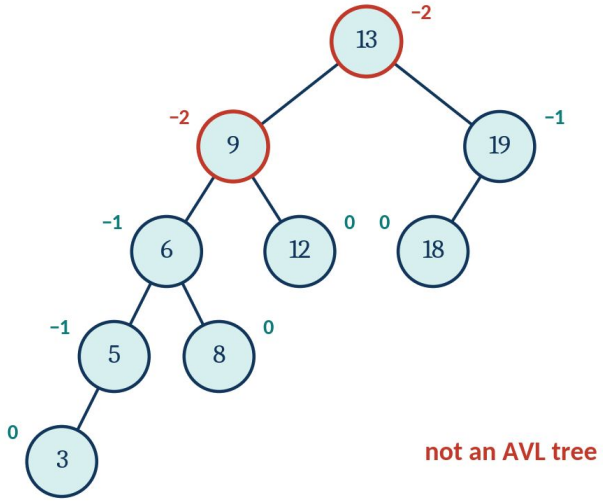


Figure 8.4 A tree that is **not** an AVL tree. Hanging the node 3 below 5 pushes two balance factors, those of 9 and 13, to −2, out of the legal range.

Figure 8.4: with 3 under 5, node 6 has height 2, so node 9 has $bf = 0 - 2 = -2$. Book: Sections 8.9.2 and 6.14.

Problem 8.6 — What the Obvious Answer Costs

Computing both subtree heights at every node is correct but throws each height away. On a perfect tree it inspects $\sum_x size(x)$ nodes, exactly Equation (8.10):

$$W = (n+1)\lg(n+1) - n. \tag{8.10}$$

$n=7$: 17; $n=15$: 49; $n=127$: 769 — measured exactly. That is $\Theta(n \lg n)$, a factor of $\lg n$ above one pass.

> **NOT $\Theta(n^2)$ HERE.** A tree that passes is short and a tree that fails exits early. The genuinely quadratic routine labels *every* node of a chain: $8256 = \frac{128 \cdot 129}{2}$ for $n=128$.

Chapter 7's lesson applies: never recompute what you already know. Book: Section 8.9.3.

Problem 8.6 — One Pass, with a Sentinel

Return the height and use -2 — below every real height — to mean "unbalanced below", which gives Equation (8.11); the tree is balanced iff $ch(t) \neq -2$.

$$\mathrm{ch}(\emptyset) = -1; \qquad \mathrm{ch}((v, L, R)) = \begin{cases} -2 & \text{if } \mathrm{ch}(L) = -2 \\ -2 & \text{if } \mathrm{ch}(R) = -2 \\ -2 & \text{if } |\mathrm{ch}(R) - \mathrm{ch}(L)| > 1 \\ 1 + \max(\mathrm{ch}(L), \mathrm{ch}(R)) & \text{otherwise.} \end{cases} \tag{8.11}$$

Table 8.5 Measured node inspections against the closed forms. The naive balance check matches $(n+1)\lg(n+1)-n$ exactly on perfect trees, the one-pass method matches n, and the height-annotation routine matches $n(n+1)/2$ exactly on chains.

Tree	n	Slow, measured	$(n+1)\lg(n+1)-n$	Fast, measured	Annotate, measured	$n(n+1)/2$
perfect	7	17	17	7	17	—
perfect	15	49	49	15	49	—
perfect	31	129	129	31	129	—
perfect	63	321	321	63	321	—
perfect	127	769	769	127	769	—
chain	16	16	—	16	136	136
chain	64	64	—	64	2080	2080
chain	128	128	—	128	8256	8256

Table 8.5 confirms the counts. The Master Theorem agrees: one pass is Equation (8.12), case 1, $\Theta(n)$; the slow check is Equation (8.13), case 2, $\Theta(n \lg n)$.

$$T(n) = 2T(n/2) + \Theta(1). \tag{8.12}$$

$$T(n) = 2T(n/2) + \Theta(n), \tag{8.13}$$

Choosing -1 as the flag is a classic bug: it is the height of the empty tree. Book: Sections 8.9.4–8.9.7.

Problem 8.7 — `insertHB`

CONTRACT (`rebalanceForLeft(t)`). t is non-empty; both subtrees are balanced; the left may have grown one notch, so $bf(t) \in \{-2, -1, 0, +1\}$. It returns a balanced tree with the same values, in $O(1)$. `rebalanceForRight` mirrors it.

Ordinary insertion rebuilds the path; wrap a rebalancer around each rebuilt node — the side that may have grown — and you have Equation (8.14):

$$\text{insertHB}(n, (v, L, R)) = \begin{cases} \text{rebalanceForLeft}(v, \text{insertHB}(n, L), R) & n < v \\ \text{rebalanceForRight}(v, L, \text{insertHB}(n, R)) & n > v \\ (v, L, R) & n = v \end{cases} \tag{8.14}$$

The signature returns a `MyTree` because a rebalance can change the subtree's root. Cost $O(h) = O(\lg n)$, at most two rotations per insertion. Book: Sections 8.10.1–8.10.3.

A New Key Is Always a Leaf

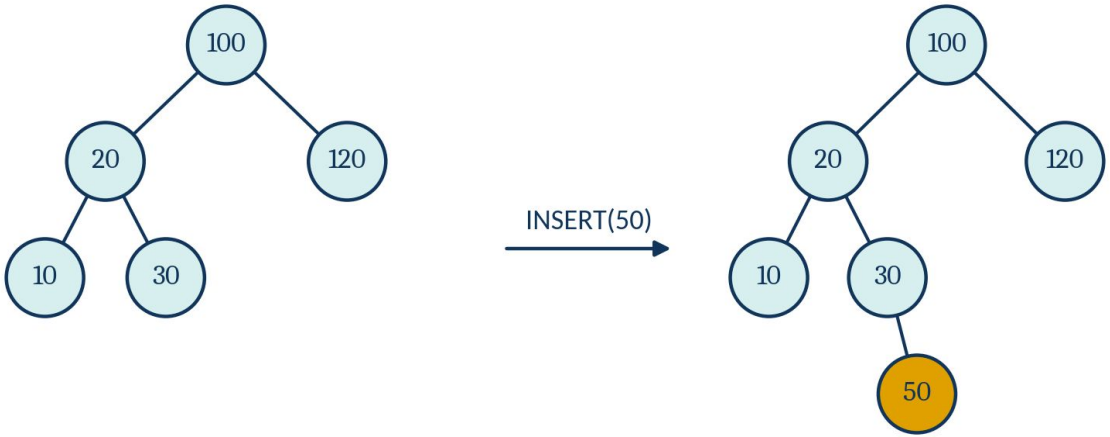


Figure 8.5 Insertion into a plain BST: a new element is always inserted as a leaf. The descent is the search, and the search is the placement.

Figure 8.5: the descent is the search, and the search is the placement. Book: Section 8.10.3.

Inserting 14: No Rotation

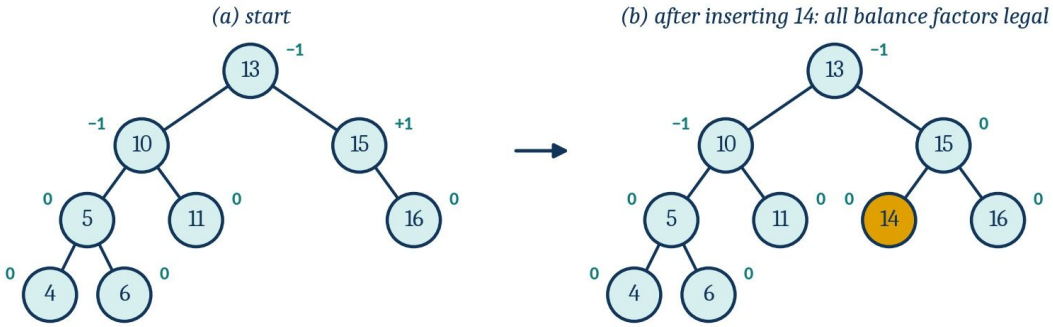


Figure 8.6 Inserting 14 into the eight-node tree. The insertion happens, the balance factors are recomputed up the path, and nothing leaves the legal range — so `insertHB` performs zero rotations. This is the common case.

Figure 8.6: every balance factor on the path stays in $\{-1, 0, +1\}$, so `insertHB` performs zero rotations — the common case. Book: Section 8.10.4.

Problem 8.8 — The One Tool: Rotation

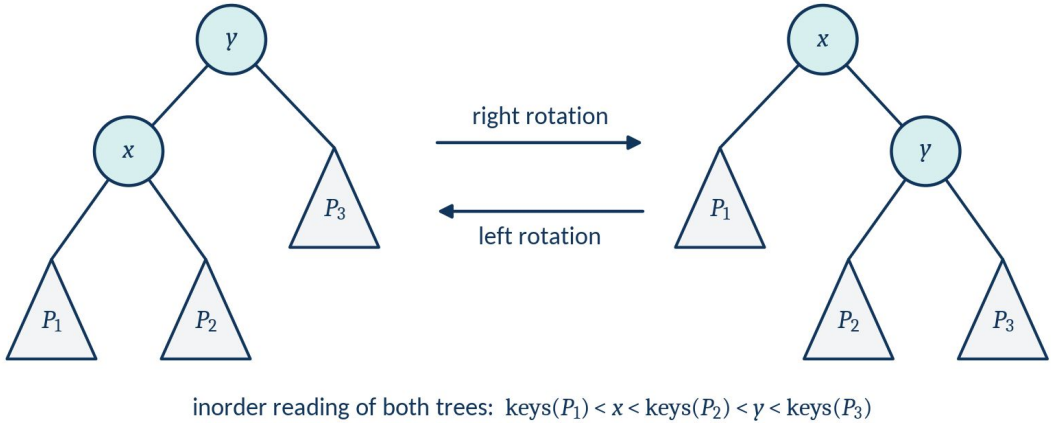


Figure 8.7 The two rotations. In both pictures the inorder reading is P_1, x, P_2, y, P_3, as the line beneath the figure states. A right rotation turns the left picture into the right one; a left rotation is its exact inverse.

Figure 8.7: both shapes read P_1, x, P_2, y, P_3 in order, so both are BSTs holding the same values. Book: Section 8.11.2.

Problem 8.8 — `rebalanceForLeft`

In the functional style a right rotation is one expression, Equation (8.15); `node` fixes the heights in the right order automatically.

$$rotateRight((y,(x,P_1,P_2),P_3)) = (x,P_1,(y,P_2,P_3)), \tag{8.15}$$

With $bf(t) = -2$: **LL** if $bf(L) \le 0$ — one right rotation; **LR** if $bf(L) > 0$ — rotate L left, then t right. That is Equation (8.16):

$$\text{rebalanceForLeft}(t) = \begin{cases} t & \text{bf}(t) \ge -1 \quad \text{(nothing to do)} \\ \text{rotateRight}(t) & \text{bf}(t) = -2 \text{ and } \text{bf}(L) \le 0 \quad \text{(LL)} \\ \text{rotateRight}(v, \text{rotateLeft}(L), R) & \text{bf}(t) = -2 \text{ and } \text{bf}(L) > 0 \quad \text{(LR)} \end{cases} \tag{8.16}$$

Smallest LR case: 10*, left* 5*,* 5*'s right* 7*. A lone right rotation just moves the imbalance to the other side; left-then-right gives* $7\,[5, 10]$*. Book: Sections 8.11.2–8.11.4.*

Case LL

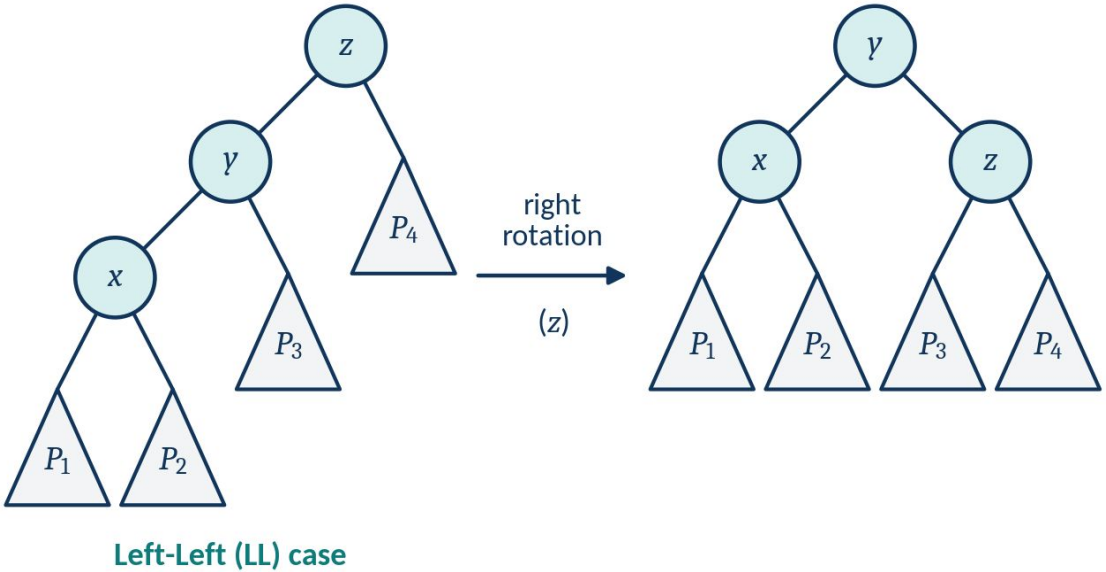


Figure 8.8 The Left-Left case: z is the unbalanced node, y its left child, x its left grandchild. One right rotation about z lifts y to the top. P_1 to P_4 are the four subtrees.

Figure 8.8: with $h(P_4)=k$ and $h(y)=k+2$, one right rotation leaves $bf(z)$ and $bf(y)$ in range whether $h(P_3)$ is k or $k+1$. Book: Section 8.11.3.

Case LR

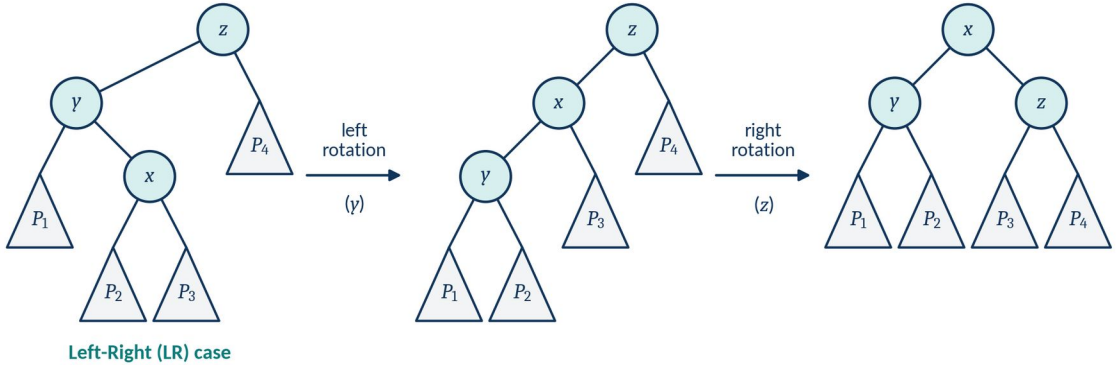


Figure 8.9 The Left-Right case. A single right rotation about z would leave the tree just as unbalanced the other way. First a left rotation about y straightens the zig-zag into a left-left lean; then one right rotation about z finishes the job.

Figure 8.9: the tall part is y's right subtree, which a right rotation about z does not lift; straighten the zig-zag first. Book: Section 8.11.4.

The Subtlest Point: ≤, Not <

After an insertion, $bf(L) = 0$ cannot occur. After a **deletion** it can: 10 with $L = 5\langle 3, 7\rangle$ and $R = 20$; delete 20 and $bf(10) = -2$ while $bf(5) = 0$.

With `<= 0` one right rotation gives $5\langle 3, 10\langle 7, .\rangle\rangle$, balanced. With `< 0` the double rotation does needless work and, in general, leaves ± 2 at the new root.

> **THE ≤ IS NOT DECORATION.** The same helpers serve deletion, and that forces `<= 0`. A `< 0` version passes every insertion test and fails on deletion.

If you are asked one question about `rebalanceForLeft`, it will be this one. Book: Section 8.11.5.

Problem 8.9 — `rebalanceForRight`, the Mirror

Reflect Problem 8.8 symbol by symbol: `>= -1` becomes `<= 1`; `leftOf` becomes `rightOf`; `bf(l) <= 0` becomes `bf(r) >= 0`; `rotateRight` and `rotateLeft` swap. Four changes, not three.

Table 8.6 The four rotation cases, as tested in code. Note that the zero case belongs with the *single* rotation on both sides — ≤ 0 on the left, ≥ 0 on the right — for the reason given in §8.11.5.

bf(t)	bf Of the tall child	Case	Repair
-2	$bf(L) \leq 0$	**LL**	one `rotateRight(t)`
-2	$bf(L) > 0$	**LR**	`rotateLeft(L)`, then `rotateRight(t)`
$+2$	$bf(R) \geq 0$	**RR**	one `rotateLeft(t)`
$+2$	$bf(R) < 0$	**RL**	`rotateRight(R)`, then `rotateLeft(t)`

Table 8.6 is the whole decision table: the zero case goes with the *single* rotation on both sides.

Wrong turns: the outer guard backward; `<` for `<=`; rotating about the wrong node in the double case; assuming the rebalancer always has work to do. Book: Section 8.12.

Case RR

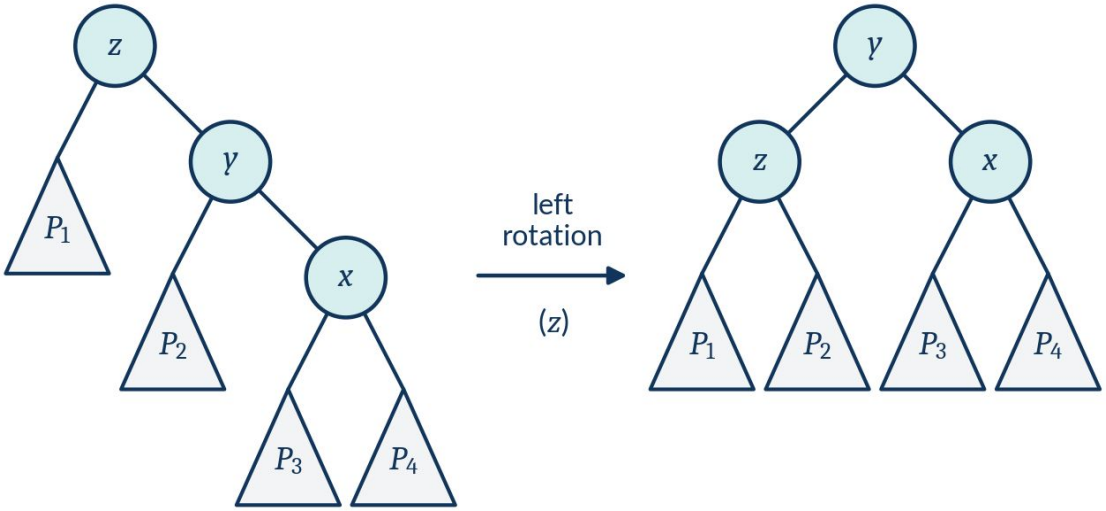


Figure 8.10 The Right-Right case: y is the right child of z and x is the right child of y. One left rotation about z repairs it. This is the exact mirror of Figure 8.8.

Figure 8.10: one left rotation about z*. Book: Section 8.12.3.*

Case RL

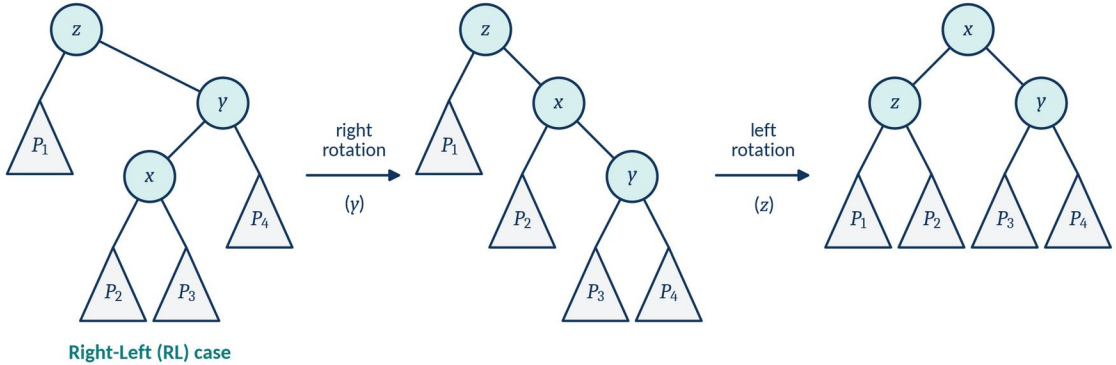


Figure 8.11 The Right-Left case: the zig-zag is right then left. A right rotation about y straightens it, then a left rotation about z finishes. The exact mirror of Figure 8.9.

Figure 8.11: a right rotation about y, then a left rotation about z. Book: Section 8.12.3.

Problem 8.10 — `deleteHB`

At the target: an empty left subtree returns R (cases 1 and 2); an empty right subtree returns L; otherwise replace v by $m = \max(L)$ — Problem 8.5 — and delete m from L.

INSERT LEFT ⇒ `rebalanceForLeft`. DELETE LEFT ⇒ `rebalanceForRight`. Deleting from a side makes the *other* side relatively taller.

$$\text{deleteHB}((v, L, R), x) = \begin{cases} \text{rebalanceForRight}(v, \text{deleteHB}(L, x), R) & x < v \\ \text{rebalanceForLeft}(v, L, \text{deleteHB}(R, x)) & x > v \\ \text{the three cases of §8.13.2} & x = v \end{cases} \tag{8.17}$$

Compared with Equation (8.14), Equation (8.17) has the same shape and the rebalancers exchanged. Insertion rebalances at most once; deletion may cascade up to $\Theta(\lg n)$ times.

Also rebalance after the two-children replacement, and return the tree unchanged when x is absent. Book: Sections 8.13.1–8.13.5.

Deletion Case 1 — a Leaf

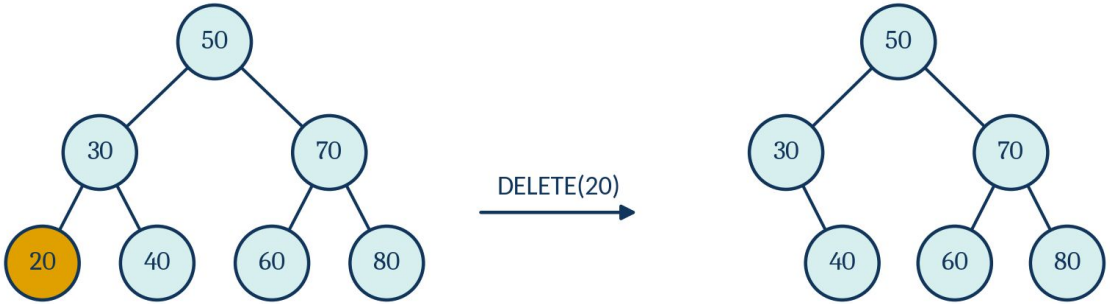


Figure 8.12 The first deletion case: the node to be deleted is a leaf. Simply remove it.

Figure 8.12: simply remove it. Book: Section 8.13.2.

Deletion Case 2 — One Child

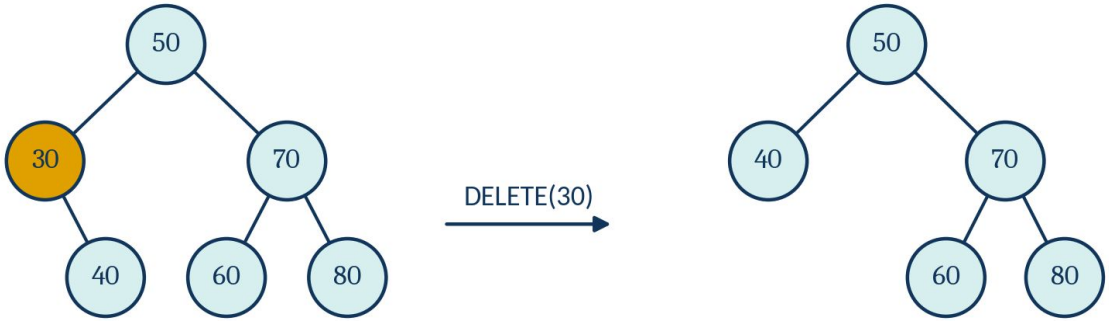


Figure 8.13 The second case: the node has exactly one child. “Copy the child to the node will be deleted, then delete the child” — equivalently, in the functional style, return the child in place of the node.

Figure 8.13: in the functional style, return the child in place of the node. Book: Section 8.13.2.

Deletion Case 3 — Two Children

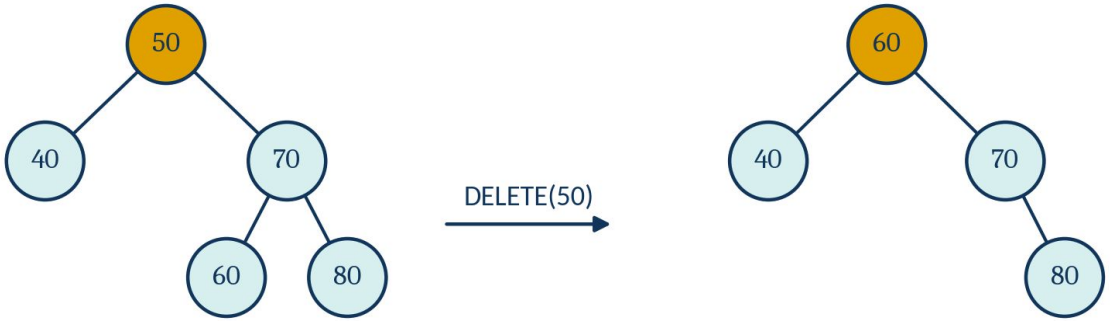


Figure 8.14 The third case: the node has two children. Find its inorder successor (or, symmetrically, its predecessor), copy that value into the node, and delete the successor from the right subtree — a smaller problem of case 1 or case 2.

Figure 8.14: copy the successor (or predecessor) and delete it from its subtree — a case 1 or case 2 problem. Book: Section 8.13.2.

The Worked Deletion — Before

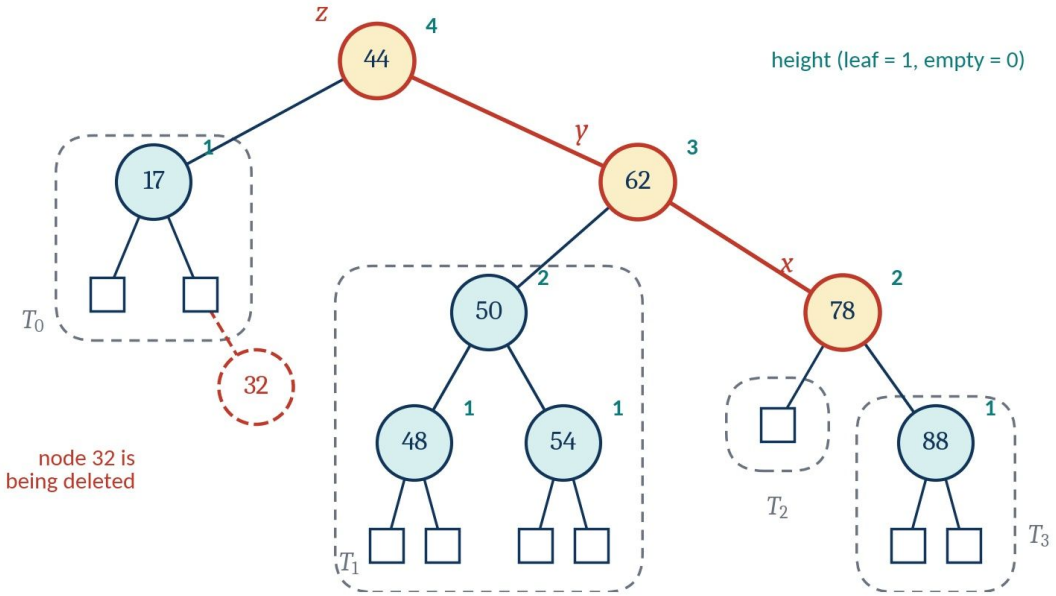


Figure 8.15 Before the deletion: the node 32 is about to be deleted, and the first unbalanced ancestor $z = 44$, its taller child $y = 62$ and y's taller child $x = 78$ are marked. Heights in the figure count nodes, so a leaf is 1 — one more than our convention everywhere.

Figure 8.15: deleting 32 *shortens* 17*'s right side (*`rebalanceForLeft`* does nothing), then* 44*'s left side, so* `rebalanceForRight` *at* 44 *finds* $bf(44) = +2$ *and* $bf(62) = 0 \geq 0$*: the RR row. Book: Section 8.13.5.*

The Worked Deletion — After

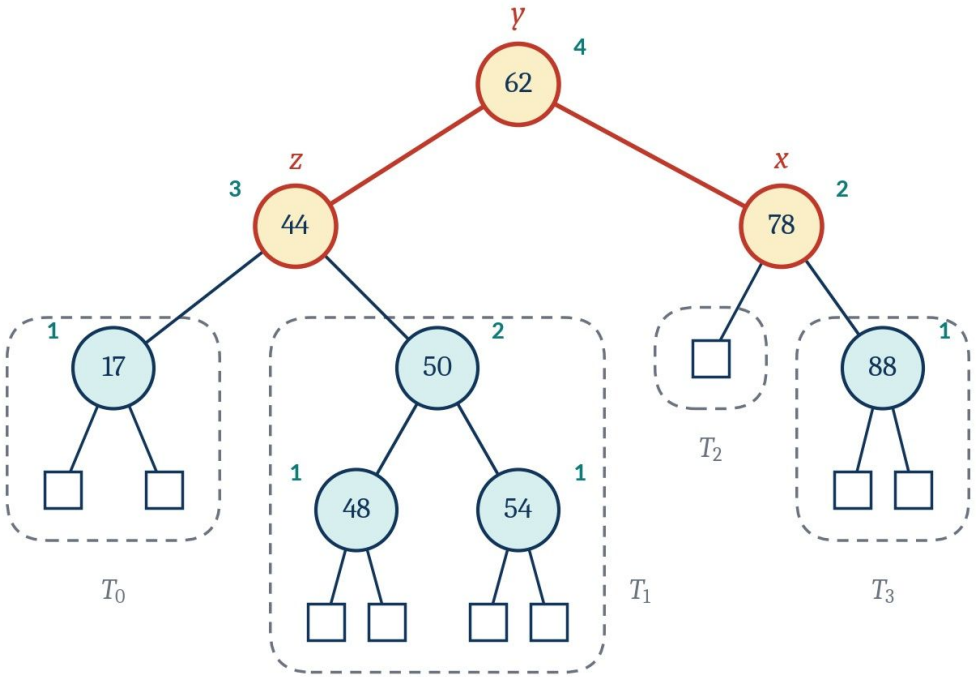


Figure 8.16 After one left rotation about 44. Node 62 is the new root, 44 has dropped to its left holding the subtree rooted at 50, and 78 is unchanged on the right.

Figure 8.16: one left rotation about 44*; the program prints* `62(-1){44(1){17(0),50(0){48(0),54(0)}},78(1){.,88(0)}}`*. Book: Section 8.13.5.*

Problems 8.7 to 8.10, Checked as One Program

Insert 14, 3, 45, 7, 15 and delete 32: rotation counts $0, 1, 1, 2, 2, 1$ — none, LL, RR, LR, RL, RR — and every shape matches the hand-worked trees of Chapter 6.

Delete 62 (two children): the predecessor 54 takes its place, no rotation. Delete 99 (absent): the tree is unchanged.

Stress test: inserting 1 to 1000 in order gives height 9, within the program's check $1.4405 \lg(n+2) - 0.3277 = 14.03$; deleting all thousand empties the tree with 491 rotations.

A correctness argument anyone can check in thirty seconds is worth more than any amount of prose. Book: Sections 8.13.6 and 8.13.7.

Problems 8.11 to 8.15 — Red-Black Trees

Table 8.7 The five red-black problems of this section. The last column says whether the property can be checked by looking at a node and its children alone — the question that decided Problem 8.3.

Problem	Routine	What it must do	Local or global?
8.11	`countBlackNodes(t)`	Count the black nodes, not counting `NIL`	global count, no pruning possible
8.12	`validateProperty4(t)`	Check that no red node has a red child	local: parent and children
8.13	`validateBlackHeight(t)`	Check that every downward path carries the same number of black nodes	global, solved in one pass with a sentinel
8.14	`isValidRBTree(t)`	Check all five red-black properties and the search-tree property	assembly of the others
8.15	`leftRotate(t, x)`, `rightRotate(t, y)`	Rotate in place, repairing all parent pointers	local surgery on six pointers

Table 8.7: property 4 (no red node with a red child) is a parent–child rule, so the **local** check is right this time; property 5 (uniform black-height) uses the sentinel -1, since black-heights are ≥ 0.

`isValidRBTree` needs no test for properties 1 and 3 — a `bool` color and a shared black `NIL` guarantee them.

On the test tree — black root 26*; red* 17 *over black* 14 *and* 21*; black* 41 *over red* 47 *— there are four black nodes and the black-height is* 2*; all* 20 *checks pass. Book: Sections 8.14.1–8.14.6.*

Red-Black Beside AVL

Table 8.8 The AVL problems and the red-black problems side by side. The search-tree check and the rotations are essentially shared; the middle problems differ because the balance invariant differs.

	AVL problems	Red-black problems
Search-tree check	`isBST` (Problem 8.3)	the same `isBST`, identical logic
Counting	`printDescending` (Problem 8.4)	`countBlackNodes` (Problem 8.11)
A local check	`max` (Problem 8.5)	`validateProperty4` (Problem 8.12)
A one-pass sentinel check	`isHeightBalanced` (Problem 8.6)	`validateBlackHeight` (Problem 8.13)
Assembly	`insertHB` (Problem 8.7)	`isValidRBTree` (Problem 8.14)
Rotations	`rebalanceForLeft`, `rebalanceForRight` (8.8, 8.9)	`leftRotate`, `rightRotate` (8.15)
Deletion	`deleteHB` (Problem 8.10)	—
Balance invariant	$\lvert bf(x)\rvert \le 1$	colors, plus a uniform black-height
Style	functional: rebuild and return	imperative: mutate, with `parent` pointers

Table 8.8: the same family. Problem 8.15's imperative rotations have **six** pointers to fix — downward links *and* parent links; forgetting the upward ones gives a tree that prints correctly and walks upward into the wrong place.

The functional rotations of Problem 8.8 need no parent pointers at all, because the new subtree root is returned. Book: Sections 8.14.7 and 8.14.8.

Problem 8.16 — A Matrix Chain by Hand

A_1: 4×10, A_2: 10×3, A_3: 3×12, A_4: 12×20, A_5: 20×7, so $d = \langle 4, 10, 3, 12, 20, 7 \rangle$ and the recurrence is Equation (8.18).

$$M[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{ M[i,k] + M[k+1,j] + d_{i-1}\, d_k\, d_j \} & \text{if } i < j. \end{cases} \tag{8.18}$$

Pseudocode 8.5 `MATRIX-CHAIN-ORDER`: every subchain is solved once, shortest first, and the winning split is recorded.

```
MATRIX-CHAIN-ORDER(d, n)
1  for i = 1 to n
2      M[i, i] = 0
3  for l = 2 to n                          // l is the chain length
4      for i = 1 to n − l + 1
5          j = i + l − 1
6          M[i, j] = ∞
7          for k = i to j − 1
8              q = M[i, k] + M[k + 1, j] + d[i − 1] · d[k] · d[j]
9              if q < M[i, j]
10                 M[i, j] = q
11                 c[i, j] = k
12 return M and c
```

Pseudocode 8.5 fills the table by increasing chain length, so every cell a cell needs is already there. Book: Sections 8.15.1–8.15.3.

Problem 8.16 — Filling the Table

Length 2: $M[1,2]=120$, $M[2,3]=360$, $M[3,4]=720$, $M[4,5]=1680$.

Length 3: $M[1,3]=\min(840,264)=264$; $M[2,4]=\min(1320,2760)=1320$; $M[3,5]=\min(1932,1140)=1140$.

Length 4: $M[1,4]=\min(2120,1080,1224)=1080$; $M[2,5]=\min(1350,2880,2720)=1350$.

Length 5: $M[1,5]=\min(1630,1344,2280,1640)=1344$ at $k=2$.

At $M[1,3]$, multiplying A_1A_2 first shrinks the 10 to a 3, and every later product is cheaper. Book: Sections 8.15.4–8.15.7.

Problem 8.16 — The Tables and the Bracketing

Table 8.9 The cost table M for $d = \langle 4, 10, 3, 12, 20, 7 \rangle$. Each diagonal was filled before the next one; the answer is the top-right corner, $M[1,5] = 1344$.

$M[i,j]$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$i=1$	0	120	264	1 080	**1 344**
$i=2$		0	360	1 320	1 350
$i=3$			0	720	1 140
$i=4$				0	1 680
$i=5$					0

Table 8.10 The split table c for the same chain. The two bold entries are the ones the reconstruction of §8.15.8 reads.

$c[i,j]$	$j=2$	$j=3$	$j=4$	$j=5$
$i=1$	1	2	2	**2**
$i=2$		2	2	2
$i=3$			3	**4**
$i=4$				4

From Table 8.9 and Table 8.10: $c[1,5]=2$, $c[3,5]=4$, so $((A_1A_2)((A_3A_4)A_5))$ at $120+720+420+84=1344$ ✓, against 1784 left to right.

Plain recursion makes $R(n)=3^{n-1}$ calls (81 for $n=5$); memoization makes Equation (8.19) (41 for $n=5$).

$$1+\sum_{\delta=1}^{n-1} 2\delta(n-\delta)=1+\frac{n^3-n}{3}. \tag{8.19}$$

Check yourself: $d=\langle 2,3,6,4,5\rangle$ *costs* 124 *with* $(((A_1A_2)A_3)A_4)$. *Book: Sections 8.15.8–8.15.12.*

Problem 8.17 — SORTED against STORE

Fill c and b with Equation (8.20), ties going up; walk back; decide whether the answer is unique; state the costs.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max\{c[i-1, j],\ c[i, j-1]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases} \tag{8.20}$$

Pseudocode 8.6 `LCS-LENGTH`: the table of lengths and the table of arrows, filled row by row, with ties going up.

```
LCS-LENGTH(X, Y, m, n)
1  for i = 0 to m
2      c[i, 0] = 0
3  for j = 0 to n
4      c[0, j] = 0
5  for i = 1 to m
6      for j = 1 to n
7          if x_i == y_j
8              c[i, j] = c[i − 1, j − 1] + 1;  b[i, j] = "↖"
9          elseif c[i − 1, j] ≥ c[i, j − 1]
10             c[i, j] = c[i − 1, j];          b[i, j] = "↑"
11         else
12             c[i, j] = c[i, j − 1];          b[i, j] = "←"
13 return c and b
```

Pseudocode 8.7 `PRINT-LCS`: follow the arrows back; every diagonal arrow is one character of the answer.

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

Pseudocode 8.6 records ↖, ↑ *or* ←; *Pseudocode 8.7 prints after its recursive call, so the characters come out in order. Book: Sections 8.16.1 and 8.16.2.*

Problem 8.17 — The Table, and the Walk Back

Table 8.11 The tables c and b for \$X = \$ SORTED and \$Y = \$ STORE. The bold cells are the four diagonal arrows met by the walk back of §8.16.4.

	j=0	j=1 (S)	j=2 (T)	j=3 (O)	j=4 (R)	j=5 (E)
$i=0$	0	0	0	0	0	0
$i=1$ (S)	0	**1** ↖	1 ←	1 ←	1 ←	1 ←
$i=2$ (O)	0	1 ↑	1 ↑	**2** ↖	2 ←	2 ←
$i=3$ (R)	0	1 ↑	1 ↑	2 ↑	**3** ↖	3 ←
$i=4$ (T)	0	1 ↑	2 ↖	2 ↑	3 ↑	3 ↑
$i=5$ (E)	0	1 ↑	2 ↑	2 ↑	3 ↑	**4** ↖
$i=6$ (D)	0	1 ↑	2 ↑	2 ↑	3 ↑	4 ↑

Table 8.11: $c[6,5]=4$. From $(6,5)$: ↑; ↖ takes E; ↑; ↖ takes R; ↖ takes O; ←; ↖ takes S. Reversed: **SORE**.

Unique? T comes before O and R in STORE but after them in SORTED, so a common subsequence with T has length at most 3 (STE); D is not in STORE. So SORE is the only one.

$\Theta(mn)$ *for the table (*30 *cells), at most* $m+n$ *steps for the walk (*7 *here). Check yourself: HEAPS and SHAPE give HAP. Book: Sections 8.16.3–8.16.9.*

Quick Checks

Why is `BUILD-MAX-HEAP` $\Theta(n)$? Most nodes are low; $\sum_{h} \lceil n/2^{h+1} \rceil O(h) = O(n)$.

Quicksort's worst case with a last-element pivot? Sorted input: exactly $\frac{n(n-1)}{2}$ comparisons.

The comparison-sort lower bound? $\Omega(n \lg n)$: $n!$ leaves force height $\lg(n!)$.

Rotations per AVL insertion? Per deletion? At most two; up to $\Theta(\lg n)$.

When dynamic programming rather than divide and conquer? Overlapping subproblems plus optimal substructure.

Twenty-five quick checks cover Chapters 1 to 7; anything you miss names the section to reread. Book: Section 8.17.

Problems 8.18 to 8.25 — an Order-Statistic Tree

Every node also stores `h` and `size`, both maintained by the single constructor `node(v, l, r)`. Table 8.12 pairs each problem with the one it extends.

Table 8.12 Problems 8.18 to 8.25 against the problems they extend. The shape is the same; only the invariant changed.

Problem	Extends	What is new
8.18	8.3 `isBST`	two derived fields must also be checked
8.19	8.4 `printDescending`	pruning, not just traversal order
8.20	8.5 `max`	subtree sizes steer the descent
8.21	8.6 `isHeightBalanced`	a *different* balance notion
8.22	8.7 `insertHB`	the `size` field must survive rotations
8.23	8.8 `rebalanceForLeft`	unchanged in structure
8.24	8.9 `rebalanceForRight`	unchanged in structure
8.25	8.10 `deleteHB`	unchanged in structure

Same shapes as Problems 8.3 to 8.10, different invariant: remembering is not enough, understanding is. Book: Section 8.18.1.

The Order-Statistic Tree, Solved

8.18: return the true height and size and compare with the stored fields; the corrupted tree is caught at "node 50 stores size 99 but the truth is 11".

8.19: inorder with two pruning tests, $O(h+k)$. **8.20:** with $\ell = size(\text{left})$, go left if $k \le \ell$, stop if $k = \ell + 1$, else seek $k - \ell - 1$ on the right; `select(6) = 45` after 4 probes.

8.21: an AVL tree need **not** be weight-balanced: a perfect left subtree of 7 nodes against a minimal right subtree of 2 gives $7 > 2 \cdot 2 + 1$.

8.22–8.25: identical to 8.7–8.10, because `node` also fixes `size`. Stress test: 2000 ascending inserts give height 10.

Put the invariant repair in the constructor, and the algorithms stop caring how many invariants there are. Book: Sections 8.18.2–8.18.4.

Profitina in Practice: an Index That Audits Itself

In Profitina, the scan queue is a priority queue (Problem 8.1 at scale), comparing page versions is Problem 8.17, and the due-time index must audit itself with `isBST` and `isHeightBalanced`.

Profitina Laundry keeps pickups in an AVL tree. After a bad in-place edit the audit reads `search-tree=BROKEN, height-balanced=OK` and names the record: “order PL-0431 (minute 500) is filed where only minutes in (-1,20) belong”.

Recovery must not depend on the broken invariant: an inorder walk still reaches every node, so read out, sort by the true key, rebuild with `insertHB`. Billing: Rp 7.000 per kg, two-kilogram minimum, Rp 350.000 for the day.

Do not trust a stored field; recompute it. The platform advises; the customer decides; a complete UU PDP 27/2022 purge needs indexes that can delete. Book: Sections 8.19 and 8.19.1.

Common Mistakes

Reading the problem once — *recursive*, *no separate list*, *one path*, *one pass*, *use the two helpers* are the real content.
Testing only the happy path — the empty tree, one node, two nodes and an absent key find most bugs.
Trusting a stored field; **checking locally what is true globally** (the BST property) or globally what is true locally.
Calling the rebalancer named after the side you recursed into when deleting; saying $O(\lg n)$ when you mean $O(h)$.
Wrong loop bounds or fill order; **keeping the value and throwing away the choice** (M without c, c without b).

Counters cost three lines and turn opinions into evidence. Book: Section 8.20.

Summary: Five Things to Remember

1. Carry information down or return it up, so that no work is done twice: bounds for `isBST`, heights for `isHeightBalanced`, rebuilt subtrees for `insertHB`, tables for DP.

2. Problem 8.1: 15 comparisons, 6 swaps, $O(n)$. Problem 8.2: $\frac{n(n-1)}{2}$ worst, $(n+1)\lg(n+1)-2n$ best.

3. A BST check needs intervals; a heap check does not. `max` walks the right spine in $O(h)$.

4. AVL: one constructor, rotations as expressions, `<= 0` for the single rotation, and the **opposite** rebalancer on deletion.

5. Matrix chain: 1344 with $((A_1 A_2)((A_3 A_4) A_5))$. LCS of SORTED and STORE: SORE.

Every hand trace here is reproduced by a program, and every closed-form count is compared with a measured one. Book: Section 8.21.

Next Time

Chapter 9 — greedy algorithms: activity selection, Huffman coding, and the matroid theory that says when greed is safe.

Before then: Exercises 1, 6, 11, 13 and 14 of Chapter 8, by hand.

Keep asking: *what invariant does my code maintain, and how would I check it?*

A greedy algorithm is correct exactly when a particular exchange invariant survives every step. Book: Sections 8.21.1 and 8.22.

CHAPTER 1

Lecture 9 — Greedy Algorithms

Design and Analysis of Algorithms (EF234405)

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*

Lecturer: Irfan Subakti — yifana@gmail.com

Book: Chapter 9, Greedy Algorithms. Every page of this handout ends with a pointer into the book, which is the primary reference; the handout is the chapter in miniature.

Today's Meeting

1. What a greedy algorithm is — and why a greedy rule is usually *wrong* until it is proved (Sections 9.2–9.3).

2. Activity selection, completely: four rules, optimal substructure, **the exchange argument**, the trace, the running time (Section 9.4).

3. The elements of the greedy strategy, and the two knapsacks that separate greed from dynamic programming (Sections 9.5–9.6).

4. Huffman codes: the tree, the cost $B(T)$, the merge-by-merge trace and the proof (Section 9.7).

5. Matroids — the deepest answer to *when does greed work?* — and scheduling with deadlines (Section 9.8); then the catalog, Profitina Laundry and the common mistakes (Sections 9.9–9.11).

Book: Chapter 9, Sections 9.1 to 9.12.

Where We Left Off

Chapter 7 gave us **dynamic programming**: characterize an optimal solution, define its value recursively, compute in dependency order, reconstruct from recorded choices.

It gave us **optimal substructure** and the **cut-and-paste argument** that proves it (Section 7.5).

And it gave us the bill: matrix-chain at $\Theta(n^3)$ time and $\Theta(n^2)$ space; the longest common subsequence at $\Theta(mn)$ — 488 MiB of table at $m=n=8000$.

Why so expensive? At every cell the dynamic program *does not know which choice is right*, so it tries them all.

> **TODAY'S QUESTION.** *What if the choice were obvious?* If one candidate were visibly right, there would be no table and no minimization — just a sort and a sweep, typically $\Theta(n \lg n)$.

Book: Section 9.1 and Section 9.1.1.

What a Greedy Algorithm Is

> **DEFINITION.** A **greedy algorithm** solves an optimization problem by a sequence of choices. At each decision point it makes the choice **that looks best at that moment**, judged by a fixed rule that examines only the current state — never the solutions to the subproblems the choice will create, never any future choice. It never reconsiders.

"Best at that moment." The rule is *local*: biggest coin that fits; activity that finishes earliest; merge the two rarest symbols.

"Never the subproblems." Dynamic programming *solves the subproblems before making the first choice*; greed *makes its first choice before solving any subproblem*.

"Never reconsiders." No backtracking: that is where the speed comes from, and why a mistake at step 3 can never be undone at step 40.

Book: Section 9.2.1.

Greed Is a Hope, Not a Method

A GREEDY ALGORITHM IS USUALLY WRONG. Take a random optimization problem and a plausible local rule, and the result is, in general, not optimal. Nothing in the *form* of a greedy algorithm makes it correct.

THE PROOF IS THE WHOLE JOB. Writing the algorithm is trivial — sort, then take things in order. Deciding *which* order, and *proving* that order safe, is the entire task. A greedy algorithm *with* a proof is usually the fastest and simplest correct algorithm the problem has.

A dynamic program that tries every split point never gambles; a greedy algorithm gambles **at every step**, and the proof is what turns the gamble into a certainty.

Book: Section 9.2.2; the plan of the chapter is Section 9.2.3.

Making Change: The Greedy Rule

THE COIN-CHANGE PROBLEM. Denominations $d_1, d_2, \ldots, d_k$ (positive integers, unlimited supply) and an amount A: pay exactly A with **as few coins as possible**.

THE GREEDY CHANGE RULE. While money is still owed, hand over the **largest coin that does not exceed the amount still owed**.

Rupiah, owed 78 000: take 50 000 (owed 28 000), 20 000 (owed 8 000), 5 000 (owed 3 000), 2 000 (owed 1 000), 1 000 (owed 0).

Five coins, $50\,000+20\,000+5\,000+2\,000+1\,000=78\,000$, and no four coins can do it.

Book: Sections 9.3.1 and 9.3.2.

The Same Rule, Broken

Coins $\{1, 3, 4\}$, owed $A = 6$. Owed 6: take 4, owed 2. Owed 2: 4 and 3 are too big, take 1; then 1 again. Greed pays $4+1+1$ — **three coins**.

A thoughtful cashier pays $3+3$ — **two coins**. Fifty percent more coins than necessary, on an amount of six.

Coins $\{1, 10, 25\}$, $A = 30$: greed pays $25+1+1+1+1+1$ (six coins); the optimum is $10+10+10$ (three).

> **THE SHAPE OF EVERY GREEDY FAILURE.** The locally attractive choice damages the subproblem it leaves behind by more than the local gain was worth. Taking the 4 left 2, which costs two coins; taking the 3 would have left 3, which costs one.

Book: Section 9.3.3.

Canonical Coin Systems, Audited

A coin system is **canonical** when the greedy rule is optimal for every amount: $\{1,5,10,25\}$ is, $\{1,3,4\}$ is not. Table 9.1 audits six systems for every amount from 1 to 200.

Table 9.1 The same greedy rule on six coin systems, audited by Listing 9.1 against the dynamic-programming optimum for every amount from 1 to 200. The rule did not change; only the input structure did.

Coin system	Canonical?	Smallest counterexample	Greedy / optimal there	Wrong at
$\{1,3,4\}$	no	$A=6$	3 coins vs 2	49 of 200
$\{1,10,25\}$	no	$A=30$	6 coins vs 3	70 of 200
$\{1,5,10,25\}$	yes	—	—	0 of 200
$\{1,2,5,10,20,50,10$	yes	—	—	0 of 200
$\{1,2,4,8,16,32,64\}$	yes	—	—	0 of 200
$\{1,5,10,20,50,100\}$	yes	—	—	0 of 200

THE LESSON IN ONE SENTENCE. The strategy is not the point; the proof is the point. On $\{1,10,25\}$ greed is wrong at **70 of the first 200 amounts** — a failing rule fails systematically, not rarely.

Book: Sections 9.3.4 and 9.3.5.

The Audit Program

Listing 9.1 computes the greedy answer and the true optimum — by the one-dimensional table `best[a] = 1 + min over coins c of best[a-c]` of Chapter 7 — for six systems and every amount up to 200.

Listing 9.1 Auditing the greedy change rule against the true optimum, for six coin systems and every amount up to 200. Compile with `g++ -std=c++17 -O2 -o coins coins.cpp` and run `./coins`.

```cpp
// coins.cpp — Does the greedy "biggest coin that fits" rule give the fewest coins?
// For each coin system we compute, for every amount, the greedy count and the
// true optimum (by dynamic programming), and report the smallest amount where
// the two disagree.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <climits>
using namespace std;

// Greedy: repeatedly take the largest coin that does not exceed what is left.
// Returns the coins used, or an empty vector if greedy gets stuck.
vector<int> greedyChange(vector<int> coin, int amount) {
    sort(coin.rbegin(), coin.rend());           // largest first
    vector<int> used;
    for (int c : coin)
        while (amount >= c) { amount -= c; used.push_back(c); }
    if (amount != 0) return {};                 // greedy stuck: no exact change
    return used;
}

// Optimum by dynamic programming: best[a] = fewest coins summing exactly to a.
// choice[a] remembers one coin on an optimal solution so we can print it.
pair<int, vector<int>> optimalChange(const vector<int>& coin, int amount) {
    const int INF = INT_MAX / 2;
    vector<int> best(amount + 1, INF), choice(amount + 1, -1);
```

```
    best[0] = 0;
    for (int a = 1; a <= amount; ++a)
        for (int c : coin)
            if (c <= a && best[a - c] + 1 < best[a]) {
                best[a] = best[a - c] + 1;
                choice[a] = c;
            }
    if (best[amount] >= INF) return {-1, {}};
    vector<int> used;
    for (int a = amount; a > 0; a -= choice[a]) used.push_back(choice[a]);
    return {best[amount], used};
}
```

Listing 9.1 (Continued) *Auditing the greedy change rule against the true optimum, for six coin systems and every amount up to 200. Compile with g++ -std=c++17 -O2 -o coins coins.cpp and run ./coins.*

```
string show(const vector<int>& v) {
    string s;
    for (size_t i = 0; i < v.size(); ++i) { if (i) s += " + "; s += to_string(v[i]); }
    return s.empty() ? "(none)" : s;
}

void audit(const string& name, vector<int> coin, int limit) {
    cout << "System " << name << " = {";
    for (size_t i = 0; i < coin.size(); ++i) { if (i) cout << ", "; cout << coin[i]; }
    cout << "}\n";
    int firstBad = -1;
    int badCount = 0;
    for (int a = 1; a <= limit; ++a) {
        auto g = greedyChange(coin, a);
        auto o = optimalChange(coin, a);
        if (o.first < 0) continue;              // amount not representable at all
        int gc = g.empty() && a > 0 ? -1 : (int)g.size();
        if (gc != o.first) { ++badCount; if (firstBad < 0) firstBad = a; }
    }
    if (firstBad < 0) {
        cout << "  greedy is optimal for every amount from 1 to " << limit << "\n\n";
        return;
    }
    auto g = greedyChange(coin, firstBad);
    auto o = optimalChange(coin, firstBad);
    cout << "  greedy FAILS.  smallest counterexample: amount = " << firstBad << "\n";
    cout << "    greedy : " << show(g) << "   = " << g.size() << " coins\n";
    cout << "    optimal: " << show(o.second) << "   = " << o.first << " coins\n";
```

```
    cout << "  greedy is wrong for " << badCount << " of the " << limit
         << " amounts tested\n\n";
}

int main() {
    audit("A (textbook trap)",  {1, 3, 4},                 200);
    audit("B (no nickel)",      {1, 10, 25},               200);
    audit("C (US coins)",       {1, 5, 10, 25},            200);
    audit("D (rupiah, x1000)",  {1, 2, 5, 10, 20, 50, 100}, 200);
    audit("E (powers of 2)",    {1, 2, 4, 8, 16, 32, 64},  200);
    audit("F (rupiah minus 2)", {1, 5, 10, 20, 50, 100},   200);
    return 0;
}
```

Its output, which is where the numbers of Table 9.1 come from:

```
System A (textbook trap) = {1, 3, 4}
  greedy FAILS.  smallest counterexample: amount = 6
    greedy : 4 + 1 + 1   = 3 coins
    optimal: 3 + 3   = 2 coins
  greedy is wrong for 49 of the 200 amounts tested

System B (no nickel) = {1, 10, 25}
  greedy FAILS.  smallest counterexample: amount = 30
    greedy : 25 + 1 + 1 + 1 + 1 + 1   = 6 coins
    optimal: 10 + 10 + 10   = 3 coins
  greedy is wrong for 70 of the 200 amounts tested

System C (US coins) = {1, 5, 10, 25}
  greedy is optimal for every amount from 1 to 200

System D (rupiah, x1000) = {1, 2, 5, 10, 20, 50, 100}
  greedy is optimal for every amount from 1 to 200

System E (powers of 2) = {1, 2, 4, 8, 16, 32, 64}
  greedy is optimal for every amount from 1 to 200

System F (rupiah minus 2) = {1, 5, 10, 20, 50, 100}
  greedy is optimal for every amount from 1 to 200
```

Book: Section 9.3.5. Exercise 7.18 is the same phenomenon with laundry bags of 3, 5 and 8 kg.

Activity Selection: The Problem

DEFINITION. $S=\{a_1,\ldots,a_n\}$ are activities wanting one resource. Activity a_i occupies the **half-open** interval $[s_i, f_i)$, $0 \le s_i < f_i < \infty$. Activities a_i and a_j are **compatible** when $s_i \ge f_j$ or $s_j \ge f_i$. Select a **maximum-size** set of mutually compatible activities.

Half-open makes an activity ending at 10:00 compatible with one starting at 10:00 — the resource is handed over, nothing is double-booked.

We maximize the **number** of activities — not busy time, not value. With values attached, the greedy rule of today becomes wrong (Exercise 9.15).

Standing assumption Equation (9.1): the activities arrive sorted by finish time (sort them if not, at $O(n \lg n)$).

$$f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n. \tag{9.1}$$

Book: Section 9.4.1.

The Running Example

Figure 9.1 is the instance for the whole of Section 9.4: $a_1=[1,4)$, $a_2=[3,5)$, $a_3=[0,6)$, ..., $a_{11}=[12,16)$.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Figure 9.1 The eleven activities of the running example, with start times s_i and finish times f_i, already sorted so that $f_1 \le f_2 \le \cdots \le f_{11}$. This one table is the running example for the whole of §9.4.

Book: Section 9.4.2 and Figure 9.1.

Maximal Is Not Maximum

Table 9.2 The instance of Figure 9.1 with the duration of each activity worked out. Activity a_{10} is the monster: twelve time units, nearly the whole day.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16
duration $f_i - s_i$	3	2	6	2	6	4	4	3	4	12	4

Table 9.2 adds the durations. $\{a_3, a_9, a_{11}\}$ is compatible ($8 \geq 6$, $12 \geq 12$, $12 \geq 6$) and nothing can be added to it — yet it has only three activities.

$\{a_1, a_4, a_8, a_{11}\}$ and $\{a_2, a_4, a_9, a_{11}\}$ both have **four**, the most possible — the optimum is **not unique**.

> **MAXIMAL IS NOT MAXIMUM.** *Maximal*: nothing can be added. *Maximum*: nothing legal is bigger. Every greedy algorithm produces maximal for free; the whole difficulty is maximum.

Book: Section 9.4.2.

Four Candidate Greedy Rules

Earliest start, **shortest duration**, **fewest conflicts**, **earliest finish**. On the running example two wrong rules still get four:

```
=== Four greedy rules on the running example ===
  earliest finish time  -> {a1, a4, a8, a11}, size 4
  earliest start time   -> {a3, a7, a11}, size 3
  shortest duration     -> {a2, a4, a8, a11}, size 4
  fewest conflicts      -> {a1, a4, a8, a11}, size 4
```

One instance proves nothing. An exhaustive search finds the smallest counterexamples to the three wrong rules:

```
=== Smallest counterexample for each incorrect rule ===
  earliest start time on an instance with n = 3:
      activity :       a1       a2       a3
      [s, f)   :    [0,8)    [1,4)    [0,1)
      the rule picks 1 activities {a1}
      brute force says the optimum is 2  ->  the rule is WRONG

  shortest duration  on an instance with n = 3:
      activity :       a1       a2       a3
      [s, f)   :    [5,8)    [1,5)    [4,6)
      the rule picks 1 activities {a3}
      brute force says the optimum is 2  ->  the rule is WRONG

  fewest conflicts   on an instance with n = 9:
      activity :       a1       a2       a3       a4       a5       a6       a7       a8       a9
      [s, f)   :    [4,7)   [9,13)    [2,5)    [1,4)    [1,2)   [6,10)    [0,2)   [7,15)  [10,11)
      the rule picks 3 activities {a1, a2, a4}
      brute force says the optimum is 4  ->  the rule is WRONG
```

Book: Section 9.4.3.

Breaking Shortest Duration by Hand

$a_1 = [5, 8)$ (duration 3), $a_2 = [1, 5)$ (duration 4), $a_3 = [4, 6)$ (duration 2). The rule takes the shortest, a_3.

a_1 overlaps a_3 on $[5, 6)$; a_2 overlaps it on $[4, 5)$. Both are discarded: **one** activity.

But $a_2 = [1, 5)$ and $a_1 = [5, 8)$ are compatible ($5 \geq 5$): **two** were available. The short activity sat in the middle and poisoned both neighbors.

> **WHY EARLIEST FINISH IS DIFFERENT.** Choose the activity that leaves the resource free for as many others as possible. The earliest finisher frees it from the earliest possible moment; with sorted input that is simply a_1. Intuition — now we need two proofs: optimal substructure and the greedy-choice property.

Book: Section 9.4.3; the nine-activity fewest-conflicts trace is Exercise 9.4.

Optimal Substructure

The subproblem S_{ij} of Equation (9.2): activities that start after a_i finishes and finish before a_j starts. Sentinels a_0 ($f_0 = 0$) and a_{n+1} ($s_{n+1} = \infty$) make $S_{0,n+1} = S$.

$$S_{ij} = \{a_k \in S : f_i \le s_k \text{ and } f_k \le s_j\}, \tag{9.2}$$

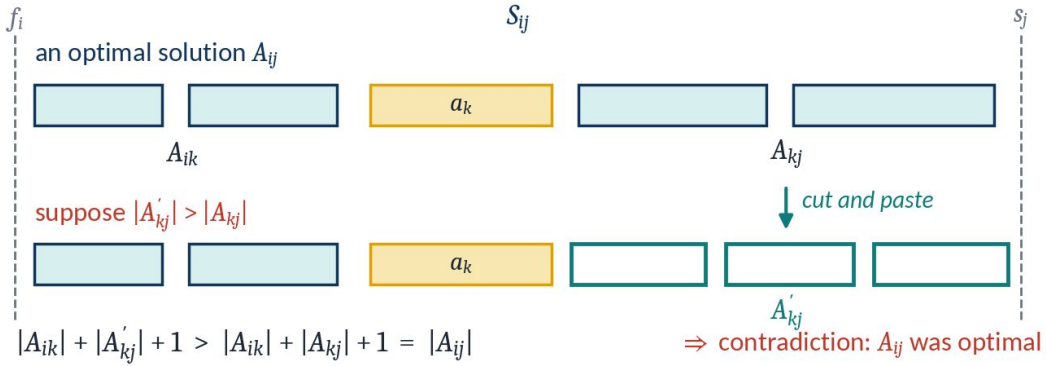
THEOREM 9.1. Let A_{ij} be a maximum-size compatible set in S_{ij} with $a_k \in A_{ij}$, and let $A_{ik} = A_{ij} \cap S_{ik}$, $A_{kj} = A_{ij} \cap S_{kj}$. Then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, the size splits as Equation (9.3), and both pieces are maximum-size in their own subproblems.

$$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1, \tag{9.3}$$

Book: Section 9.4.4, Theorem 9.1.

Cut and Paste, Drawn

Figure 9.2 completes Theorem 9.1 by contradiction and draws the recurrence that follows from it.



recurrence:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \varnothing \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \varnothing \end{cases}$$

Figure 9.2 Completing Theorem 9.1 by contradiction, and drawing the dynamic-programming recurrence for $c[i, j]$ that follows from it. If a better set A'_{kj} existed, substituting it into A_{ij} would give more than $|A_{ij}|$ compatible activities — contradicting the optimality of A_{ij}.

Book: Section 9.4.4 and Figure 9.2.

The Proof, One Line per Step

1–2. Suppose A_{kj} is not maximum: some compatible $A'_{kj} \subseteq S_{kj}$ has $|A'_{kj}| > |A_{kj}|$. **3.** Cut and paste: $B = A_{ik} \cup \{a_k\} \cup A'_{kj}$.

4. Is B legal? Everything in A_{ik} finishes by s_k; everything in A'_{kj} starts at or after f_k; $s_k < f_k$. So all pairs are compatible.

5–7. $|B| = |A_{ik}| + |A'_{kj}| + 1 > |A_{ij}|$, a legal set in S_{ij} bigger than the maximum — contradiction. **8.** Symmetrically for A_{ik}. ∎

The theorem gives the dynamic program Equation (9.4): $\Theta(n^3)$ time and $\Theta(n^2)$ space — the matrix-chain shape again. We are about to throw it away.

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max\limits_{a_k \in S_{ij}} \{\, c[i,k] + c[k,j] + 1 \,\} & \text{if } S_{ij} \neq \emptyset. \end{cases} \tag{9.4}$$

Book: Sections 9.4.4 and 9.4.5.

Only One Subproblem Remains

Figure 9.3 states the reduction. Define the one-index family Equation (9.5): the activities that start after a_k finishes.

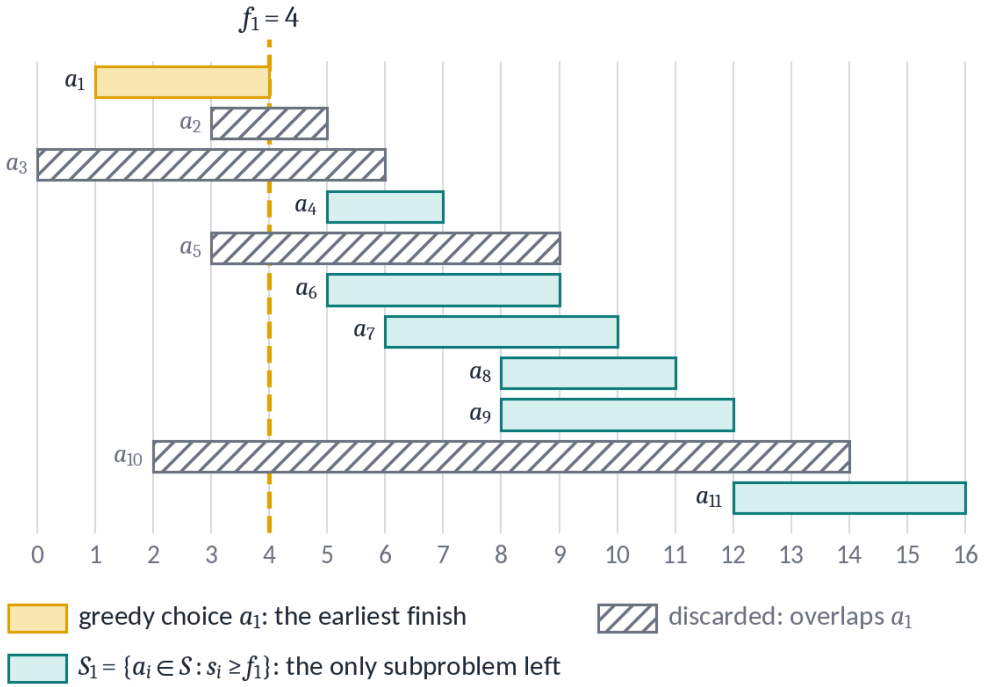


Figure 9.3 The reduction to a single subproblem: once the greedy choice a_1 has been made, the single subproblem S_1 remains, and an optimal solution to the original problem is a_1 together with an optimal solution to S_1.

$$S_k = \left\{ a_i \in S : s_i \geq f_k \right\}, \tag{9.5}$$

Book: Section 9.4.6 and Figure 9.3.

Why Nothing Sneaks In Before a_1

Claim. Every activity compatible with a_1 starts at or after f_1, so after choosing a_1 the remaining problem is exactly S_1.

Proof. If a_i is compatible with a_1, either $s_i \geq f_1$ or $s_1 \geq f_i$. The second gives $f_i \leq s_1 < f_1$ — an activity finishing before a_1, impossible since f_1 is the minimum finish time. ∎

So after the greedy choice there is exactly **one** subproblem — not two as in Theorem 9.1, not n candidates as in Equation (9.4).

Book: Section 9.4.6.

The Greedy-Choice Property

THEOREM 9.2. Let S_k be non-empty and let a_m have the earliest finish time in S_k. Then a_m belongs to **some** maximum-size set of mutually compatible activities of S_k.

Proof (exchange). Let A_k be **any** maximum-size compatible subset of S_k, and a_j its first activity to finish. If $a_j = a_m$, done.

Otherwise **exchange**: $A'_k = (A_k \setminus \{a_j\}) \cup \{a_m\}$. One removed, one added, and $a_m \notin A_k$, so $|A'_k| = |A_k|$ — **no smaller**.

Legal? Any other $a_i \in A_k$ has $f_j \leq s_i$; and $f_m \leq f_j$ because a_m finishes first in all of S_k. So $f_m \leq f_j \leq s_i$: compatible. A'_k is optimal and contains a_m. ∎

Book: Section 9.4.7, Theorem 9.2.

The Exchange Argument as a Template

(1) Take an **arbitrary** optimal solution A — any one, not a convenient one.

(2) Locate the element of A that plays the role the greedy choice wants to play (here: the first activity of A to finish).

(3) Swap the greedy choice in; prove the result is **feasible** and **no worse**.

(4) Conclude that an optimal solution containing the greedy choice exists — the choice is **safe**. Then optimal substructure repeats the argument on what remains.

> **COROLLARY 9.3.** If a_m finishes earliest in S_k and A^* is a maximum-size compatible subset of S_m, then $\{a_m\} \cup A^*$ is a maximum-size compatible subset of S_k — the license to take a_m, recurse once, and glue.

Book: Sections 9.4.7 and 9.4.8, Corollary 9.3.

The Recursive Greedy Algorithm

Figure 9.4 gives **Recursive-Activity-Selector**; the first call is made with $k=0$ and the sentinel $f_0=0$.

```
    RECURSIVE-ACTIVITY-SELECTOR (s, f, k, n)
1   m = k + 1
2   while m ≤ n and s[m] < f[k]          // find the first activity in S_k to finish
3       m = m + 1
4   if m ≤ n
5       return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR (s, f, m, n)
6   else return ∅
```

Figure 9.4 Recursive-Activity-Selector in pseudocode. The **while** loop of line 2 finds the first activity in S_k to finish; the call is made with $k=0$, using the fictitious activity a_0 with $f_0=0$.

Book: Section 9.4.9 and Figure 9.4.

Reading It Line by Line

Line 1 starts at $m = k + 1$. **Lines 2–3** skip every a_m with $s_m < f_k$ — every activity not in S_k.

Line 4 asks whether one was found. **Line 5** takes a_m (Theorem 9.2) and recurses on the one remaining subproblem S_m (Corollary 9.3). **Line 6** returns $\varnothing$.

With $f_0 = 0$ the test $s_m < 0$ is false for every real activity, so the first choice is a_1.

> **IT IS NOT REALLY RECURSION.** The recursive call is a *tail call*: nothing is computed after it except gluing a_m on. That is a loop in disguise, which is why Section 9.4.11 rewrites it as one.

Book: Section 9.4.9.

The Trace, Drawn

Figure 9.5 is the complete trace on the eleven activities of Figure 9.1 — the single most informative picture of the chapter.

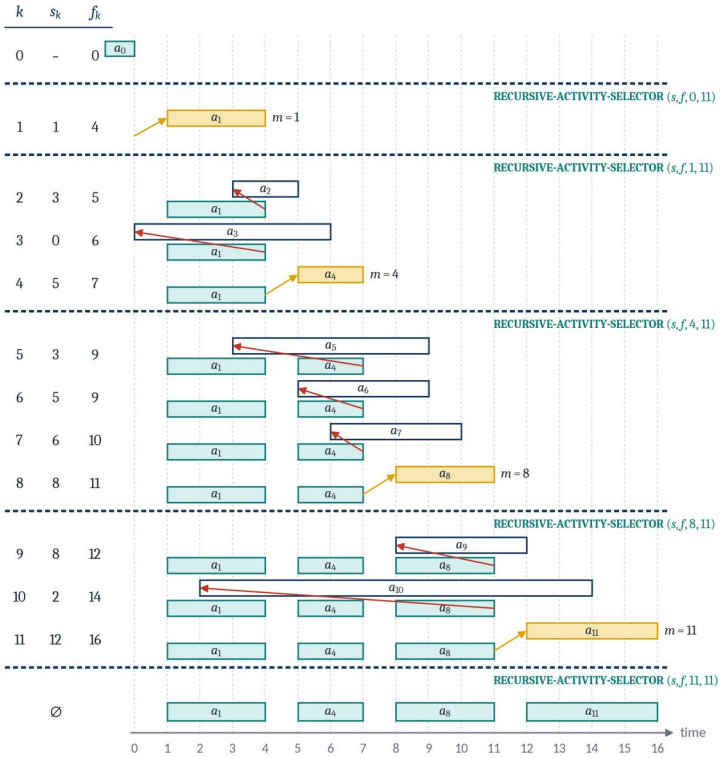

k
s_k
f_k
0 – 0 a_0
RECURSIVE-ACTIVITY-SELECTOR $(s, f, 0, 11)$
1 1 4 a_1 $m = 1$
RECURSIVE-ACTIVITY-SELECTOR $(s, f, 1, 11)$
2 3 5 a_2 a_1
3 0 6 a_3 a_1
4 5 7 a_4 $m = 4$ a_1
RECURSIVE-ACTIVITY-SELECTOR $(s, f, 4, 11)$
5 3 9 a_5 a_1 a_4
6 5 9 a_6 a_1 a_4
7 6 10 a_7 a_1 a_4
8 8 11 a_8 $m = 8$ a_1 a_4
RECURSIVE-ACTIVITY-SELECTOR $(s, f, 8, 11)$
9 8 12 a_9 a_1 a_4 a_8
10 2 14 a_{10} a_1 a_4 a_8
11 12 16 a_{11} $m = 11$ a_1 a_4 a_8
RECURSIVE-ACTIVITY-SELECTOR $(s, f, 11, 11)$
∅ a_1 a_4 a_8 a_{11}
time
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Figure 9.5 The trace of **Recursive-Activity-Selector** on the eleven activities of Figure 9.1. Each row is one iteration of the **while** loop: the upper bar is activity a_k (amber when it is taken), the teal bars are activities already selected, and the arrow marks the comparison s_k against f of the last selection. The heavy dashed lines separate the recursive calls, each labeled with its arguments; the bottom row shows the answer $\{a_1, a_4, a_8, a_{11}\}$ occupying the resource from time 1 to time 16.

Book: Section 9.4.10 and Figure 9.5.

Worked Example 1: The Trace in Words

Call 1 ($k=0$): $m=1$, is $1<0$? No. Take a_1. **Call 2** ($f_1=4$): $m=2$, $3<4$ skip; $m=3$, $0<4$ skip; $m=4$, $5<4$? No — take a_4.

Call 3 ($f_4=7$): a_5 ($3<7$), a_6 ($5<7$), a_7 ($6<7$) skipped; $m=8$, $8<7$? No — take a_8.

Call 4 ($f_8=11$): a_9 ($8<11$), a_{10} ($2<11$) skipped; $m=11$, $12<11$? No — take a_{11}. **Call 5**: $m=12>n$, return $\varnothing$.

```
=== RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11) ===
RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11)  -> m = 1, take a1
  RECURSIVE-ACTIVITY-SELECTOR(s, f, 1, 11)  -> m = 4, take a4
    RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 11)  -> m = 8, take a8
      RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 11)  -> m = 11, take a11
        RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)  -> no activity left, return {}
  result = {a1, a4, a8, a11}, size 4
```

COUNT THE WORK. m took the values 1; 2,3,4; 5,6,7,8; 9,10,11; 12 — every index examined **exactly once**. The recursion does $\Theta(n)$ work in total.

Book: Section 9.4.10.

The Iterative Greedy Algorithm

Figure 9.6 is **Greedy-Activity-Selector**: k holds the most recent selection, so $f[k]$ is when the resource next becomes free.

```
GREEDY-ACTIVITY-SELECTOR (s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

Figure 9.6 Greedy-Activity-Selector, the iterative form of Figure 9.4. The variable k always holds the index of the most recent selection, so $f[k]$ is the time at which the resource next becomes free.

Book: Section 9.4.11 and Figure 9.6.

Invariant and Running Time

LOOP INVARIANT. At the start of each iteration of the **for** loop, A is a maximum-size compatible subset of $\{a_1, \ldots, a_{m-1}\}$ and a_k is its activity with the latest finish time. Initialization ($\{a_1\}$), maintenance (Theorem 9.2 and Corollary 9.3), termination at $m = n+1$.

Both forms are $\Theta(n)$ on sorted input; with the sort, **Theorem 9.4**: $\Theta(n \lg n)$ time and $\Theta(1)$ extra space. Table 9.3 measures it.

Table 9.3 Sorting plus one greedy sweep, measured on random activities. The third column is the elapsed time divided by $n \lg n$; it stays near 4 across a 32-fold range of n, which is what $\Theta(n \lg n)$ looks like on a real machine.

n	Time (ms)	$t/(n \lg n) \times 10^6$	Activities selected
250 000	18.07	4.03	122 426
500 000	36.84	3.89	177 718
1 000 000	91.81	4.61	252 379
2 000 000	163.89	3.91	357 471
4 000 000	373.26	4.25	504 635
8 000 000	762.48	4.16	713 102

Against the dynamic program at $n = 1000$: about ten thousand operations instead of 167 million. **This is what a proof buys you.**

Book: Sections 9.4.11–9.4.13, Theorem 9.4.

The Greedy Strategy in Six Steps

Table 9.4 The six steps of the greedy strategy, mapped onto the work done in §9.4. Nothing was skipped and nothing was added.

Step	Where it was done in §9.4	What it produced
1. optimal substructure	§9.4.4, Theorem 9.1	$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, pieces optimal
2. recursive solution	§9.4.5	$c[i,j] = \max_k \{c[i,k] + c[k,j] + 1\}$
3. only one subproblem remains	§9.4.6	after taking a_1, only S_1 is left
4. the greedy choice is safe	§9.4.7, Theorem 9.2	exchange argument
5. recursive algorithm	§9.4.9	**Recursive-Activity-Selector**
6. iterative algorithm	§9.4.11	**Greedy-Activity-Selector**

Table 9.4 maps the six steps onto the work just done. The short form, three things to **prove**: cast the problem as *choice plus one subproblem*; prove the greedy choice **safe**; prove **optimal substructure** so that choice plus optimal subsolution is optimal.

Book: Section 9.5.1.

Two Ingredients, One Recipe

GREEDY-CHOICE PROPERTY. For every instance, the choice the greedy rule makes is contained in **some** optimal solution.
OPTIMAL SUBSTRUCTURE. An optimal solution contains optimal solutions to its subproblems.

Both are needed, neither implies the other. The 0-1 knapsack has optimal substructure but not the greedy-choice property; the longest simple path has neither.

The recipe. State the objective; list every plausible rule; **try to break each on small instances by brute force**; attempt the exchange argument on the survivors; prove optimal substructure; only then write the code — and keep the brute force as a regression test.

Book: Sections 9.5.2–9.5.4.

The Two Knapsacks

0-1 knapsack: items are taken whole or left. **Fractional knapsack:** any fraction $x_i \in [0,1]$ may be taken. Both have optimal substructure (cut and paste).

The natural greedy rule ranks items by the value density Equation (9.6) and takes the densest first. Table 9.5 is the instance, $W = 50$ kg.

$$\text{density}(i) = \frac{v_i}{w_i}, \tag{9.6}$$

Table 9.5 The three-item knapsack instance, with capacity $W = 50$ kg. The items are already in decreasing order of value per kilogram, so item 1 is the greedy algorithm's first choice.

Item i	Weight w_i (kg)	Value v_i	Density v_i/w_i
1	10	60	6.00
2	20	100	5.00
3	30	120	4.00

Book: Sections 9.6.1 and 9.6.2.

Fractional: Greed Succeeds

Item 1 (6.00/kg), all of it: value 60, room 40. Item 2 (5.00/kg), all: value 160, room 20. Item 3: take $x_3 = 20/30 = 2/3$, value 80. **Total 240**, knapsack exactly full.

Theorem 9.5 (exchange). Let y be any optimal solution and j the first index where it differs from greedy; then $y_j < x_j$. Move weight $\delta = (x_j - y_j) w_j$ onto item j from later items; the value changes by Equation (9.7), which is ≥ 0 because later items are no denser.

$$\text{value}(y') - \text{value}(y) = \delta \cdot \frac{v_j}{w_j} - \sum_{i>j} \delta_i \cdot \frac{v_i}{w_i}, \tag{9.7}$$

WHERE THE PROOF USED DIVISIBILITY. We had to add *exactly* δ and remove *exactly* δ. In the 0-1 world items come in lumps and no lump may be the right size.

Book: Section 9.6.3, Theorem 9.5.

Worked Example 2: 0-1 Greed Fails

Same rule, whole items: item 1 (room 40), item 2 (room 20), item 3 weighs 30 — skip. Greedy: **160**, with 20 kg empty. Table 9.6 prices all eight subsets.

Table 9.6 Every subset of the instance in Table 9.5, priced. The optimum is $\{2,3\}$ at value 220; the greedy rule returns $\{1,2\}$ at value 160.

Subset	Weight (kg)	Value	Fits in 50 kg?
$\{\}$	0	0	yes
$\{1\}$	10	60	yes
$\{2\}$	20	100	yes
$\{1,2\}$	30	160	yes — this is greedy's answer
$\{3\}$	30	120	yes
$\{1,3\}$	40	180	yes
$\{2,3\}$	50	220	yes — **this is the optimum**
$\{1,2,3\}$	60	280	no

Optimum $\{2,3\}$: 50 kg, **220**. Shortfall $220-160=60$, that is $60/220\approx 27.3\ \%$. The optimum *sacrifices the densest item* so that the capacity divides evenly.

Book: Section 9.6.4.

How Badly, How Often

```
=== Stress test: how often is the greedy 0-1 answer wrong? ===
  n =   5: greedy was suboptimal in   4241 of 20000 random instances (21.2%),  mean value/optimum =
0.9789,  worst = 0.4577
  n =  10: greedy was suboptimal in   7033 of 20000 random instances (35.2%),  mean value/optimum =
0.9868,  worst = 0.7821
  n =  20: greedy was suboptimal in   9666 of 20000 random instances (48.3%),  mean value/optimum =
0.9930,  worst = 0.9250

=== Fractional greedy is never wrong: same stress test ===
  n =   5: fractional greedy fell below the 0-1 optimum in 0 of 20000 instances
  n =  10: fractional greedy fell below the 0-1 optimum in 0 of 20000 instances
  n =  20: fractional greedy fell below the 0-1 optimum in 0 of 20000 instances
```

Greedy 0-1 is wrong **often** (48.3% of instances at $n=20$) and usually wrong **by little** (mean ratio 0.993) — but the worst case at $n=5$ was 0.4577. “It works on my test data” is not a proof. Table 9.7 puts the two techniques side by side.

Table 9.7 Greedy against dynamic programming. The decisive row is the second: the greedy-choice property is the extra thing greed demands, and the thing the 0-1 knapsack does not have.

	Greedy	Dynamic programming
Needs optimal substructure	yes	yes
Needs the greedy-choice property	**yes**	no
Needs overlapping subproblems	no	yes (§7.10.1)
When is the first choice made?	**before** any subproblem is solved	**after** the subproblems are solved
How many subproblems after a choice?	exactly one	in general several, and several candidate choices
Direction	top-down, one choice after another	bottom-up, or top-down with memoization
Does it reconsider?	never	implicitly, by taking a max or min over all choices
Typical cost, activity selection	$\Theta(n \lg n)$	$\Theta(n^3)$ time, $\Theta(n^2)$ space
Fractional knapsack	**optimal** (Theorem 9.5)	works but is unnecessary
0-1 knapsack	**wrong** — up to 54% below optimum	optimal, $\Theta(nW)$

Book: Sections 9.6.5 and 9.6.6; the complete program is Section 9.6.7.

Huffman Codes: The Problem

A file of 100 000 characters over a–f. Fixed length needs $\lceil \lg 6 \rceil = 3$ bits each: 300 000 bits. A variable-length code gives frequent characters short codewords. Figure 9.7 is the instance.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 9.7 A character-coding problem. A file of 100 000 characters contains only a–f, with the frequencies shown (in thousands). The fixed-length code costs 300 000 bits; the variable-length code shown costs 224 000.

Book: Sections 9.7.1 and 9.7.2, and Figure 9.7.

Counting the Bits

Table 9.8 The two codes of Figure 9.7, with the cost of each computed character by character. In thousands of bits: 300 for the fixed-length code, 224 for the variable-length one.

Character	Frequency (thousands)	Fixed codeword	Bits	Variable codeword	Bits	Frequency × length
a	45	000	3	0	1	$45 \times 1 = 45$
b	13	001	3	101	3	$13 \times 3 = 39$
c	12	010	3	100	3	$12 \times 3 = 36$
d	16	011	3	111	3	$16 \times 3 = 48$
e	9	100	3	1101	4	$9 \times 4 = 36$
f	5	101	3	1100	4	$5 \times 4 = 20$
total	**100**		**300**			**224**

Table 9.8: $45+39+36+48+36+20=224$ thousand bits against 300 — a saving of 76 000 bits, 25.33 %, or 2.24 bits per character.

Where the saving comes from: “a” shrank from 3 bits to 1, saving 90 000; e and f grew from 3 to 4, costing 14 000; net 76 000.

Book: Section 9.7.2.

Prefix-Free Codes

The code $a=0$, $b=01$, $c=1$ is **ambiguous**: `001` reads as “aac” or as “ab”. The trouble: `0` is a **prefix** of `01`.

DEFINITION. A code is **prefix-free** if no codeword is a prefix of any other codeword.

Then decoding is greedy, left to right, with no look-ahead: `0` → a; `1`, `10`, `101` → b; `1`, `10`, `100` → c; and so on.

Restricting to prefix-free codes costs nothing: every uniquely decodable code has a prefix-free code with the same codeword lengths.

Book: Section 9.7.3.

Codes as Binary Trees

Left edge `0`, right edge `1`, characters at the leaves: a leaf has no descendants, so the code is prefix-free **by construction**. Figure 9.8 draws both codes.

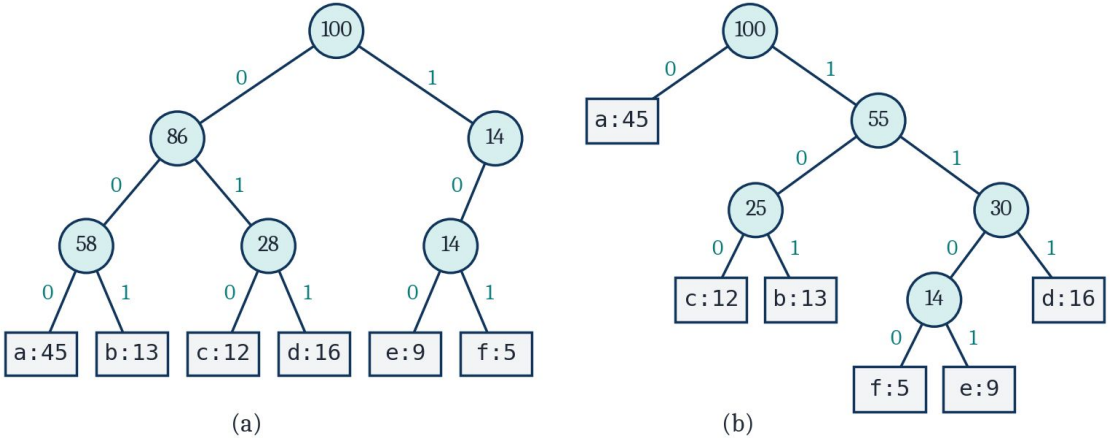


Figure 9.8 Trees for the two coding schemes of Figure 9.7. Each leaf carries a character and its frequency; each internal node carries the sum of the frequencies in its subtree. **(a)** the fixed-length code $a = 000, \ldots, f = 101$; **(b)** the optimal prefix code $a = 0$, $b = 101$, ..., $f = 1100$.

Book: Section 9.7.4 and Figure 9.8.

The Cost of a Tree

Lemma 9.6. An optimal code is a **full** tree: $|C|$ leaves and $|C|-1$ internal nodes (a node with one child could be spliced out, shortening codewords).

With $d_T(c)$ the depth of c's leaf, the cost Equation (9.8) is the total number of bits. Tree (b): $45(1)+13(3)+12(3)+16(3)+9(4)+5(4)=224$.

$$B(T)=\sum_{c\in C} c.freq\cdot d_T(c), \tag{9.8}$$

THE EQUIVALENT FORMULA. $B(T)$ is also the **sum of the internal-node labels**: $100+55+25+30+14=224$. A character at depth d lies under exactly d internal nodes. So the cost of the tree is the total of the merges that built it.

Book: Sections 9.7.4 and 9.7.5, Lemma 9.6.

The Huffman Algorithm

Figure 9.9: Q is a min-priority queue on frequency; each of the $|C|-1$ iterations merges the two least frequent items into a node labeled with their sum.

```
    HUFFMAN (C)
1   n = |C|
2   Q = C
3   for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN (Q)
6       z.right = y = EXTRACT-MIN (Q)
7       z.freq = x.freq + y.freq
8       INSERT (Q, z)
9   return EXTRACT-MIN (Q)          // return the root of the tree
```

Figure 9.9 Huffman(C) in pseudocode. Q is a min-priority queue keyed on frequency; the loop runs $|C|-1$ times, each time replacing the two least frequent items by a new node whose frequency is their sum.

Book: Section 9.7.6 and Figure 9.9.

The Trace, Drawn

Figure 9.10 shows the queue before each merge; panel (f) is tree (b) of Figure 9.8.

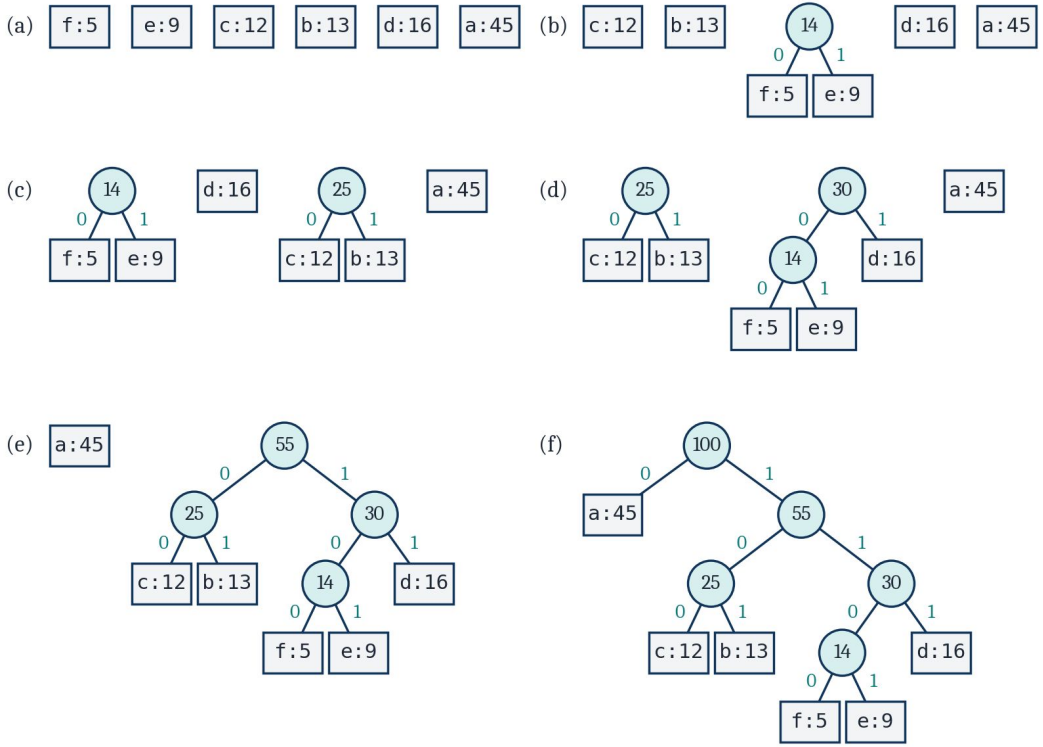


Figure 9.10 The six-panel trace of **Huffman** on the frequencies of Figure 9.7. Each panel shows the queue sorted into increasing order of frequency; at each step the two lowest-frequency trees are merged. Leaves are rectangles, internal nodes circles. **(a)** the initial six leaves; **(b)**–**(e)** the intermediate stages; **(f)** the finished tree.

Book: Section 9.7.7 and Figure 9.10.

Worked Example 3: Five Merges

Table 9.9 The five merges of **Huffman** on the frequencies of Figure 9.7, with the queue after each. The running total of the new labels is the cost $B(T)$ in thousands of bits, and it ends at 224 — the same number Table 9.8 obtained by summing over leaves.

Merge	Items extracted	New label	Queue afterward	Running total
—	—	—	f:5 e:9 c:12 b:13 d:16 a:45	0
1	f:5 and e:9	14	c:12 b:13 (14) d:16 a:45	14
2	c:12 and b:13	25	(14) d:16 (25) a:45	39
3	(14) and d:16	30	(25) (30) a:45	69
4	(25) and (30)	55	a:45 (55)	124
5	a:45 and (55)	100	(100)	**224**

Table 9.9: $f:5+e:9=14$; $c:12+b:13=25$; $(14)+d:16=30$; $(25)+(30)=55$; $a:45+(55)=100$. Five merges for six characters. The running total of the new labels is $14+25+30+55+100=224$ — exactly Equation (9.8) computed over the leaves.

Book: Section 9.7.7.

Why Huffman Is Optimal: The Swap

LEMMA 9.7. If x, y have the lowest frequencies, some optimal tree has x and y as sibling leaves of maximum depth.

Take any optimal T and deepest siblings a, b. Swap x with a: only two depths change, and the cost drops by Equation (9.9).

$$B(T) - B(T') = (a.freq - x.freq)(d_T(a) - d_T(x)). \tag{9.9}$$

Both factors are ≥ 0 (x is rarest, a is deepest), so the swap cannot increase the cost; swap y with b likewise. **A low frequency and a deep position belong together.**

Book: Section 9.7.8, Lemma 9.7.

Substructure, Theorem, Running Time

Lemma 9.8. Replace x, y by one symbol z with $z.freq = x.freq + y.freq$. Any optimal tree T' for the smaller alphabet, with z expanded, is optimal for C, because of Equation (9.10).

$$B(T) = B(T') + x.freq + y.freq. \tag{9.10}$$

Theorem 9.9. Huffman produces an optimal prefix-free code — induction on $|C|$, Lemma 9.7 licensing Lemma 9.8.

Theorem 9.10. With a binary min-heap: $O(n)$ to build, $n-1$ iterations of $O(\lg n)$ — $O(n \lg n)$. Already sorted input: $O(n)$ with two queues (Exercise 9.12).

Book: Sections 9.7.9–9.7.11; the encoder and decoder are Section 9.7.12.

The Encoder at Work

```
=== Part 4: fixed-length against Huffman ===
  6 distinct characters need ceil(lg 6) = 3 bits each
  fixed-length total  = 3 x 100000 = 300000 bits
  Huffman total       = 224000 bits
  saving              = 76000 bits = 25.33%
  average codeword length = 2.24 bits per character

=== Part 5: encode and decode real text ===
  input length      : 126 characters
  distinct symbols  : 20
  codebook (symbol -> codeword), shortest first:
    '_'=101 'a'=111 'r'=001 'b'=0100 'd'=1100 'e'=1001
    'n'=1000 's'=1101 't'=0000 'h'=01101 'i'=00011 'l'=01110
    'u'=01010 'y'=01100 'c'=010111 'g'=000101 'o'=011111 'p'=010110
    'v'=000100 'w'=011110
  fixed-length cost : 1008 bits (8 bits per char)
  Huffman cost      : 489 bits in 62 bytes
  compression       : 51.49% smaller
  round trip        : IDENTICAL
  first 64 bits     : 1110100001111010111111110011101000011111011110100001111010111111
  decoded prefix    : "abracadabra abracadabra a laundry in sur..."

=== Part 6: the one-symbol edge case ===
  1000 copies of 'z': codeword "0", 1000 bits, round trip IDENTICAL
```

A 126-character sentence: 1008 bits plain, 489 bits coded — **51.5% smaller** — and the round trip is identical. Rare symbols (“v”, “w”) get the longest codewords, as Lemma 9.7 demands.

The edge case bites: a file of one repeated symbol gives an empty codeword; the program pairs the lone symbol with a dummy so it costs one bit.

Book: Section 9.7.12.

Matroids: When Greed Is Automatic

Figure 9.11 states the definition: a finite ground set S, a **hereditary** family I of *independent* subsets, and the **exchange property**.

A matroid is a pair $M = (S, \mathcal{I})$ such that

1. S is a finite set — the ground set.
2. $\mathcal{I}$ is a non-empty family of subsets of S, the independent sets, and it is hereditary: $B \in \mathcal{I}$ and $A \subseteq B$ imply $A \in \mathcal{I}$ (so $\emptyset \in \mathcal{I}$).
3. Exchange property: if $A, B \in \mathcal{I}$ and $|A| < |B|$, then some $x \in B - A$ has $A \cup \{x\} \in \mathcal{I}$.

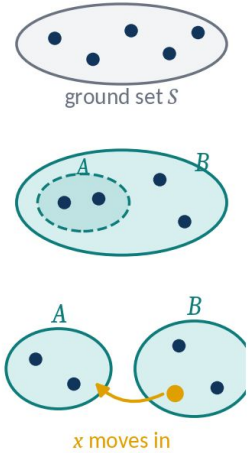


Figure 9.11 The definition of a matroid: a finite ground set S, a hereditary family I of *independent* subsets, and the exchange property.

Book: Sections 9.8.1 and 9.8.2, and Figure 9.11.

The Definition in Plain Words

Hereditary: throwing things away never breaks a legal collection; in particular $\varnothing$ is independent. Read *independent* as "legal".

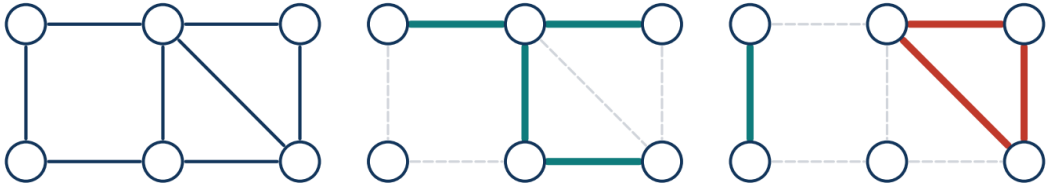
Exchange: if your legal collection is smaller than someone else's, you can steal *some* item of theirs and stay legal.

> **SUFFICIENT, NOT NECESSARY.** Matroid theory covers minimum spanning trees and task scheduling, but **not** activity selection or Huffman coding. Non-example: $A = \{a_3\} = \{[0, 6)\}$, $B = \{a_1, a_4\}$; neither element of B can be added to A.

Book: Sections 9.8.1–9.8.3.

The Graphic Matroid

Figure 9.12: ground set E, a set of edges independent exactly when it is **acyclic**. Exchange holds because a smaller forest has more components, so the bigger forest has an edge joining two of them.



(a) $G = (V, E)$; $S_G = E$ (b) $A \in \mathcal{I}_G$: acyclic, a forest (c) $B \notin \mathcal{I}_G$: B closes a cycle

Figure 9.12 The graphic matroid $M_G = (S_G, I_G)$ of an undirected graph $G = (V, E)$: the ground set is E, and a set of edges is independent exactly when it is acyclic — that is, when it forms a forest. **(a)** A graph G; **(b)** an independent set A of four edges with no cycle; **(c)** a dependent set B, whose red edges close a cycle. M_G is closely related to the minimum-spanning-tree problem, which Chapter 11 covers in detail.

Book: Section 9.8.3 and Figure 9.12.

The Greedy Algorithm on a Matroid

LEMMA 9.12. All maximal independent sets of a matroid have the same size. So in a matroid **maximal = maximum**, and greed's effortless maximality is enough.

Pseudocode 9.1: sort by decreasing weight, keep each element that preserves independence. Its cost is Equation (9.11), where $f(n)$ is the cost of one independence test.

Pseudocode 9.1 Greedy(M, w), the weighted-matroid greedy algorithm. §9.8.8 gives it as a complete compilable program with an abstract independence oracle.

```
GREEDY(M, w)
1  A = {}
2  sort M.S into monotonically decreasing order by weight w
3  for each x in M.S, taken in that order
4      if A union {x} is in M.I          // is the enlarged set independent?
5          A = A union {x}
6  return A
```

$$O(n \lg n + n f(n)). \tag{9.11}$$

Lemma 9.13 (greedy choice, by exchange), **Lemma 9.14** (contraction), **Theorem 9.15**: Greedy is optimal on **every** weighted matroid — the proof is written once, for all of them.

Book: Sections 9.8.4 and 9.8.5, Theorem 9.15.

Unit-Time Tasks with Deadlines

n unit-time tasks; task a_i has deadline d_i and penalty w_i if late. Only the **set** of early tasks matters (early-first form), and by Equation (9.12) we may maximize instead of minimize.

$$\text{minimize} \sum_{\text{late}} w_i \Leftrightarrow \text{maximize} \sum_{\text{early}} w_i. \tag{9.12}$$

A set A is **independent** when all its tasks can be on time. With $N_t(A)$ from Equation (9.13), **Lemma 9.16**: A is independent $\Leftrightarrow N_t(A) \le t$ for all t $\Leftrightarrow$ deadline order makes no task of A late.

$$N_t(A) = \left| \{ a_i \in A : d_i \le t \} \right| \tag{9.13}$$

Theorem 9.17. (S, I) is a matroid — so greed is optimal with no further proof.

Book: Section 9.8.6, Lemma 9.16 and Theorem 9.17.

The Instance

Figure 9.13: $d = \langle 4, 2, 4, 3, 1, 4, 6 \rangle$, $w = \langle 70, 60, 50, 40, 30, 20, 10 \rangle$, total 280, already in decreasing penalty order.

	Task						
a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Figure 9.13 An instance of the problem of scheduling unit-time tasks with deadlines and penalties on a single processor: seven tasks $a_1, \ldots, a_7$ with deadlines d_i and penalties w_i.

Book: Section 9.8.7 and Figure 9.13.

Worked Example 4: Greedy Schedules

Table 9.10 Greedy on the seven tasks of Figure 9.13, one row per element considered in decreasing order of penalty. Early weight 230, penalty paid 50.

Considered	d	w	Independence test	Verdict	A Afterward
a_1	4	70	$N_4 = 1 \le 4$	keep	$\{a_1\}$
a_2	2	60	$N_2 = 1 \le 2$, $N_4 = 2 \le 4$	keep	$\{a_1, a_2\}$
a_3	4	50	$N_4 = 3 \le 4$	keep	$\{a_1, a_2, a_3\}$
a_4	3	40	$N_3 = 2 \le 3$, $N_4 = 4 \le 4$	keep	$\{a_1, a_2, a_3, a_4\}$
a_5	1	30	$N_4 = 5 > 4$	**reject**	$\{a_1, a_2, a_3, a_4\}$
a_6	4	20	$N_4 = 5 > 4$	**reject**	$\{a_1, a_2, a_3, a_4\}$
a_7	6	10	$N_6 = 5 \le 6$	keep	$\{a_1, a_2, a_3, a_4, a_7\}$

Table 9.10: a_5 and a_6 are rejected because five tasks would be due by time 4. Early set $\{a_1, a_2, a_3, a_4, a_7\}$ of weight 230; penalty $30 + 20 = 50$.

Schedule in deadline order: $\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle$. Brute force over $2^7 = 128$ subsets agrees; so do 200 000 random instances.

Book: Section 9.8.7; with an acyclicity oracle the same code is Kruskal's algorithm (Section 9.8.8).

The Greedy Catalog

Table 9.11 The greedy algorithms of this book. The third column is the local rule; the fourth is the proof obligation. Notice how many of the entries in the fourth column say “an exchange argument”.

Algorithm	Problem	The greedy choice	Why it is safe	Chapter
Greedy-Activity-Selector	maximum set of compatible activities	the activity finishing earliest	Theorem 9.2, exchange argument	9 (here)
Fractional-Knapsack	most valuable divisible load	the item of highest value per kg	Theorem 9.5, exchange argument	9 (here)
Huffman	optimal prefix-free code	merge the two rarest symbols	Lemmas 9.7 and 9.8	9 (here)
Greedy(M, w)	maximum-weight independent set	the heaviest element that keeps independence	Theorem 9.15	9 (here)
Task scheduling	least total penalty	the task of highest penalty that keeps independence	Theorem 9.17 + 9.15	9 (here)
Kruskal	minimum spanning tree	the lightest edge that closes no cycle	graphic matroid, Theorem 9.15	11
Prim	minimum spanning tree	the lightest edge leaving the tree built so far	cut property, an exchange argument	11
Dijkstra	single-source shortest paths, non-negative weights	the unvisited vertex of least tentative distance	an exchange argument on paths	13
Topological sort (Kahn)	linear order respecting a DAG	any vertex with no remaining predecessor	every DAG has such a vertex	11

Table 9.11: Kruskal is Greedy on the graphic matroid; Prim and Dijkstra are not matroid algorithms, but their proofs are exchange arguments with the same four moves.

Book: Section 9.9.

Profitina in Practice

Re-scan scheduler — scans compete for workers: activity selection, sort by *finish*, not first come first served.

Crawl budget — pages cannot be half-fetched: a **0-1** knapsack, where greed is wrong; ship the fractional bound beside any greedy answer.

Analytics store — skewed status codes: Huffman compression with a verified round trip.

Profitina Laundry: dryer bookings — earliest finish serves 4, first come first served 3; the status log compresses by 82.74%; the delivery van (0-1) loses 68 of 683 (10.0%) to greed.

Book: Section 9.10 and Section 9.10.1. The app advises; the user decides.

Common Mistakes

1. Calling a greedy algorithm correct without the exchange argument. 2. Testing on one instance — two wrong rules were right on the running example.

3. Sorting by the wrong key (start, duration). **4. Closed intervals** — test `s[m] >= f[k]`, never `>`.

5. The density rule on a 0-1 knapsack. 6. Treating an average as a bound (mean 0.993, worst 0.458).

7. A code that is not prefix-free. 8. The one-symbol Huffman file. 9. Forgetting to store the tree. 10. A max-heap where Huffman needs a min-heap.

11. Assuming a matroid without checking exchange. **12. Minimizing with Greedy** without a valid weight transformation. **13. Maximal for maximum. 14. Adding values to activity selection.**

Book: Section 9.11.

Summary: Five Things to Remember

1. A greedy algorithm is fast and usually wrong; **the proof is the whole job** — the rule never changes, only the structure does (Table 9.1).

2. Two ingredients: **greedy-choice property** (proved by the exchange argument) and **optimal substructure** (cut and paste).

3. Activity selection: **earliest finish**, $\Theta(n \lg n)$; the answer on the running example is $\{a_1, a_4, a_8, a_{11}\}$.

4. Fractional knapsack: greed optimal (240); 0-1: greed 160 against 220. Huffman: merge the two rarest; $B(T) = 224$, the sum of the merges.

5. Matroids: maximal equals maximum, so Greedy is optimal on every weighted matroid — graphic (Kruskal) and task scheduling (penalty 50).

Book: Section 9.12.

Next Time

Chapter 10 — Elementary Graph Algorithms. The input stops being a sequence and becomes a graph: vertices, edges, and three ways to store them.

Breadth-first search (a queue) and **depth-first search** (a stack), both $O(V+E)$, and six problems they solve.

Chapter 11 then returns to greed: Kruskal and Prim — and Section 9.8.8 has already run Kruskal on its graph. The answer, weight 37, is waiting to be checked.

Before the meeting: Exercises 9.1, 9.5, 9.10 and 9.17, and read Sections 10.1 to 10.5.

Book: Section 9.12.1 and Chapter 10.

CHAPTER 1

Lecture 10 — Elementary Graph Algorithms: Breadth-First and Depth-First Search

Design and Analysis of Algorithms (EF234405)

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*

Lecturer: Irfan Subakti — yifana@gmail.com

Book: Chapter 10, Elementary Graph Algorithms. Every page of this handout ends with a pointer into the book, which is the primary reference.

Today's Meeting

1. What a graph is, what graphs are for, and the whole vocabulary — degree, paths, cycles, components, trees (Sections 10.2–10.4).

2. Three ways to store a graph, and what each costs (Section 10.5).

3. **Breadth-first search**: the algorithm, a full trace, the program, $O(V+E)$, and the proof that it finds shortest paths (Sections 10.6–10.7).

4. **Depth-first search**: timestamps, the forest, the parenthesis and white-path theorems, the four edge classes (Section 10.8).

5. Six things the two searches can do, the DAG theorem and topological sort; then Profitina, the common mistakes and the summary (Sections 10.9–10.12).

Book: Chapter 10, Sections 10.1 to 10.12.

Where We Left Off

Nine chapters of **sequences**: sorted arrays, heaps, search trees, dynamic-programming tables — in every case the structure was *ours to choose*.

Chapter 9 changed the style of argument: greedy choices, proved safe by exchange.

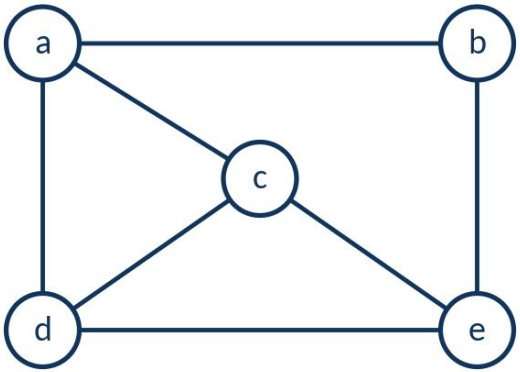
From now on the input **arrives with a structure** we cannot rearrange: a road network, the links of the web, the precedences of a build. The object is the **graph**.

What carries over: asymptotic notation (now with two sizes, V and E), the queue and the stack, loop invariants. What is new: a search can revisit a vertex, so every algorithm carries a **color** to remember what it has seen.

Book: Section 10.1 and Section 10.1.1.

What a Graph Is

DEFINITION. A **graph** $G = (V, E)$ is a set V of **vertices** and a set $E \subseteq V \times V$ of **edges**. Ordered pairs: a **directed** graph; unordered pairs: an **undirected** graph. Figure 10.1 is the example used throughout.



$$V = \{a, b, c, d, e\}$$

$$E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$$

Figure 10.1 The definition of a graph, with the example graph used throughout this chapter: $V = \{a, b, c, d, e\}$ and $E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$. Five vertices, seven edges. Notice that the drawing and the two set-listings carry exactly the same information; the drawing is only easier on the eye.

Book: Sections 10.2.1–10.2.3 and Figure 10.1.

Reading a Graph, Modeling a Problem

Five vertices, seven edges; the missing pairs are (a,e), (b,c), (b,d) — of the $\binom{5}{2}=10$ possible, seven are present. **A drawing is not the graph.**

A **self-loop** is an edge (v,v); a graph with no self-loops and no repeated edges is **simple**; parallel edges make a **multigraph**.

> **MODELING IN THREE QUESTIONS.** *What are the things?* They become V. *When are two things joined?* That rule becomes E. *Does the joining have a direction?* Only then choose an algorithm.

Applications: circuits (Figure 10.2), flight and road networks (Figure 10.3), compilers (an order of tasks), graphics (flood fill is BFS), mazes (DFS), maps, clustering.

Book: Sections 10.2.3, 10.2.4 and 10.3.

A Circuit Is a Graph

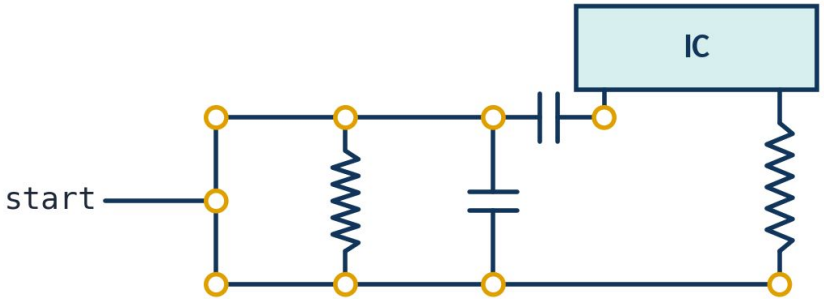


Figure 10.2 An electronic circuit as a graph. The vertices are the solder points where components meet; the edges are the components — resistors, capacitors — that join them. The question "what is the path of least resistance from `start` to the block marked IC?" is a shortest-path question on this graph.

Book: Section 10.3 and Figure 10.2.

A Flight Network Is a Graph

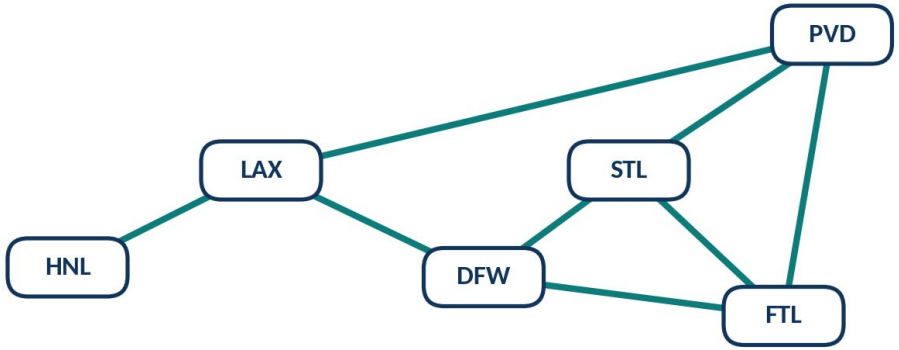


Figure 10.3 A flight network as a graph: the vertices are airports (HNL, LAX, DFW, STL, FTL, PVD), the edges are the routes flown between them. “Can I get from Honolulu to Providence, and with how few changes?” is precisely the question breadth-first search answers in §10.7.

Book: Section 10.3 and Figure 10.3. “Fewest hops” is this chapter; “shortest distance” is Chapter 13.

Degree, Pictured

$Adj[v]$ is the set of **neighbors** of v; the **degree** is $\deg(v) = |Adj[v]|$. Figure 10.4 labels each vertex with its degree.

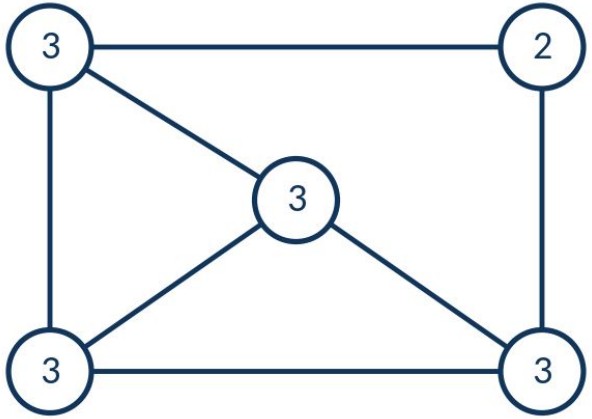


Figure 10.4 Degree. Each vertex of the chapter's graph is labeled with its degree instead of its name. Reading anticlockwise from the top left the labels are $3, 2, 3, 3, 3$; they sum to 14, which is twice the number of edges.

Book: Sections 10.4.1 and 10.4.2, and Figure 10.4.

The Handshake Lemma

$Adj[a]=\{b,c,d\}$, $Adj[b]=\{a,e\}$, $Adj[c]=\{a,d,e\}$, $Adj[d]=\{a,c,e\}$, $Adj[e]=\{b,c,d\}$: degrees $3+2+3+3+3=14=2|E|$.

$$\sum_{v \in V} \deg(v) = 2|E|. \tag{10.1}$$

Proof of Equation (10.1)**:** count the edge-ends twice — by edges ($2|E|$) and by vertices ($\sum_v \deg(v)$). Consequences: the number of odd-degree vertices is even; **all adjacency lists together hold** $2|E|$ **entries**; the average degree is $2|E|/|V|$.

Directed graphs: out-degree $\deg^+$ and in-degree $\deg^-$ each sum to $|E|$, as Equation (10.2) states. In-degree 0 is a **source**, out-degree 0 a **sink**.

$$\sum_{v \in V} \deg^+(v) = |E|, \sum_{v \in V} \deg^-(v) = |E|, \text{so} \sum_{v \in V} (\deg^+(v) + \deg^-(v)) = 2|E|. \tag{10.2}$$

Book: Sections 10.4.3 and 10.4.4.

Paths and Their Length

A **path** is $v_1, \ldots, v_k$ with consecutive vertices adjacent; its **length** is $k-1$ edges; it is **simple** if no vertex repeats. Figure 10.5 shows two.

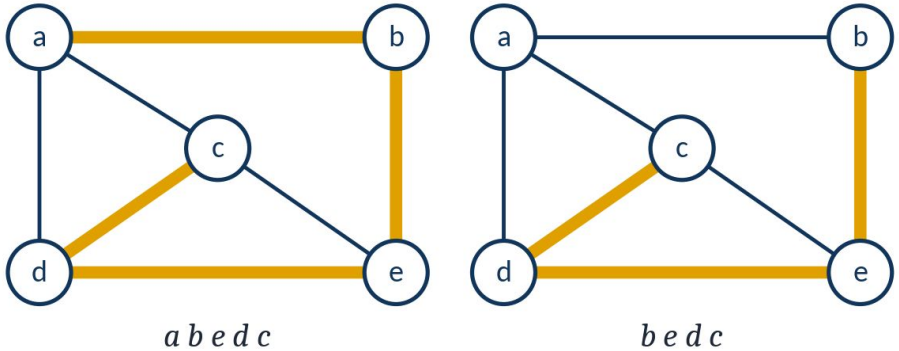


Figure 10.5 Two paths in the chapter's graph, highlighted. On the left, $a\,b\,e\,d\,c$, of length 4; on the right, $b\,e\,d\,c$, of length 3. Both are simple: no vertex is used twice.

Book: Section 10.4.5 and Figure 10.5.

Shortest Paths and Simple Paths

Figure 10.6: $b\,e\,c$ has length 2 and is shortest, since there is no edge (b,c). We write $\delta(u,v)$ for the fewest edges on any path (∞ if unreachable).

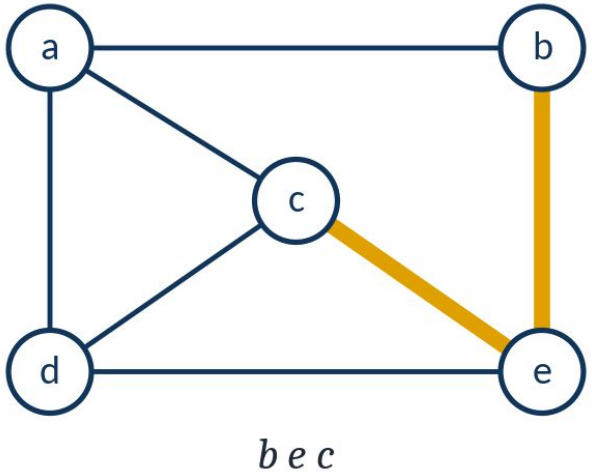


Figure 10.6 A third path, $b\,e\,c$, of length 2. It is the shortest path from b to c in this graph — there is no edge (b,c), so no path of length 1 exists.

KEY IDEA. If a path from u to v exists, a **simple** one exists and is no longer — cut out the loop between two occurrences of a vertex. So no shortest path exceeds $|V|-1$ edges.

Book: Section 10.4.5 and Figure 10.6.

A Cycle

A **cycle** is a simple path whose last vertex equals its first; a graph with none is **acyclic**. Figure 10.7 highlights the triangle $a\,c\,d\,a$.

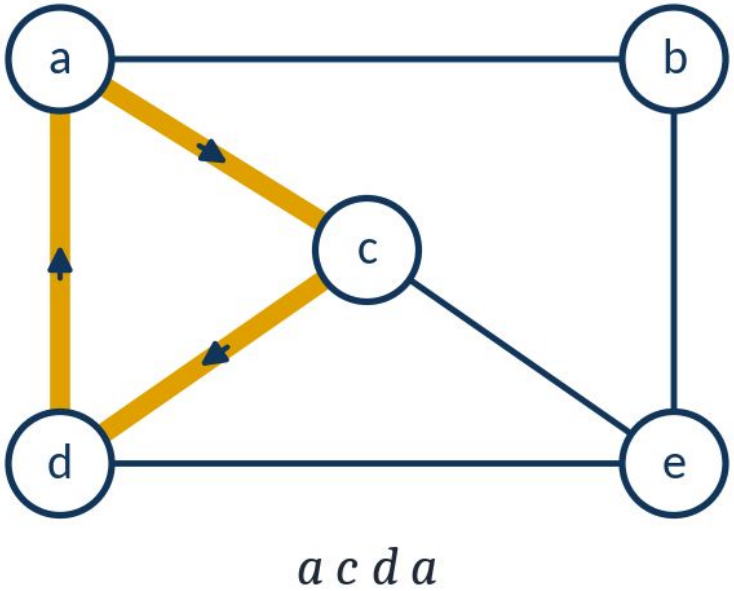


Figure 10.7 A cycle in the chapter's graph, highlighted: $a\,c\,d\,a$. Start at a, walk to c, walk to d, walk back to a — three edges, and you are where you began without having repeated any vertex on the way.

Book: Section 10.4.6 and Figure 10.7. A directed acyclic graph is a DAG (Section 10.9.5).

Connected and Not Connected

An undirected graph is **connected** if every pair of vertices is joined by a path. Figure 10.8 compares the two cases.

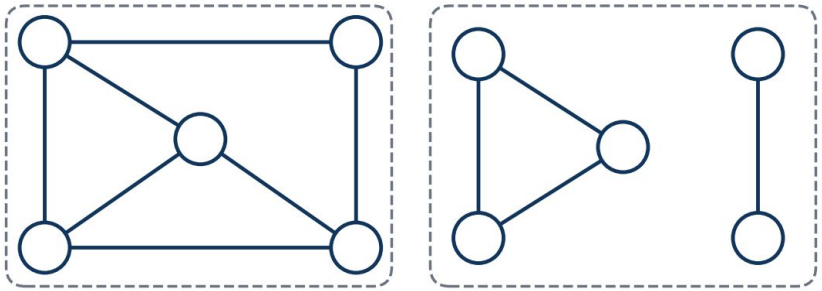


Figure 10.8 Connected and not connected. On the left, every vertex can be reached from every other. On the right the same vertices have been drawn with fewer edges, and the two vertices on the far right are cut off from the triangle on the left: no path joins them, so the graph is disconnected.

Book: Section 10.4.7 and Figure 10.8.

Connected Components

A **component** is a *maximal* connected subgraph. Every vertex lies in exactly one; no edge joins two; so a search started inside one can never leave it. Figure 10.9 has three.

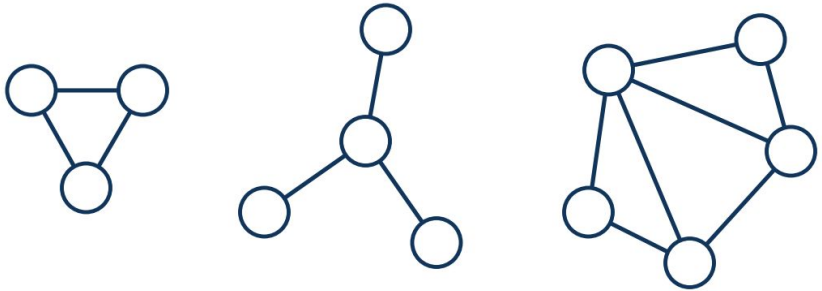


Figure 10.9 A graph with three connected components: a triangle on the left, a star in the middle, and a five-vertex piece on the right. No edge crosses between the three, and inside each one every vertex reaches every other.

Book: Section 10.4.8 and Figure 10.9.

Trees and Forests

A **(free) tree** is connected and acyclic; a **forest** is acyclic. For $|V|$ vertices, any two of *connected*, *acyclic*, $|E|=|V|-1$ imply the third. Figure 10.10 shows both.

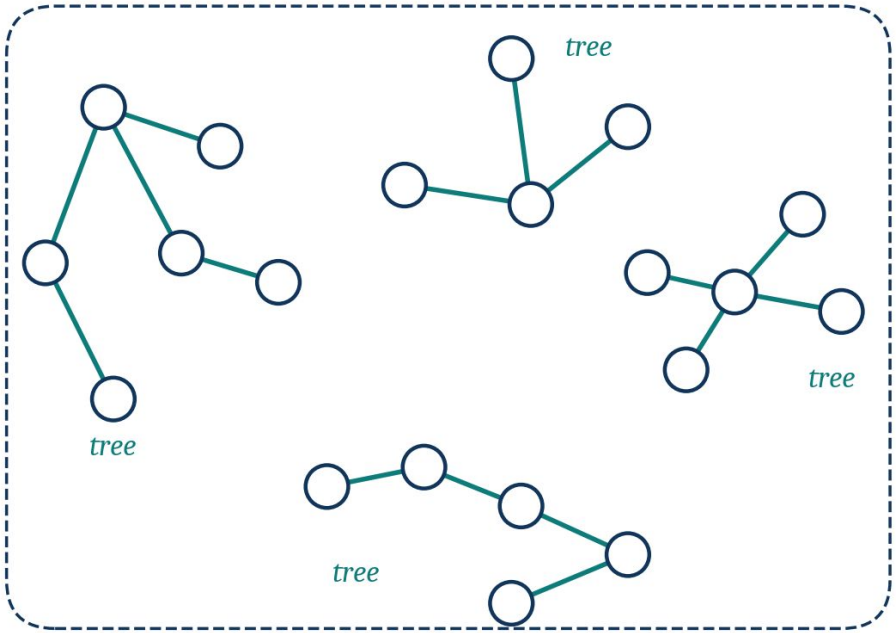


Figure 10.10 Trees and a forest. Each of the four graphs on its own is a *tree*: connected, and with no cycle anywhere in it. All four taken together form a single graph that is acyclic but not connected — a *forest*, with four components.

Book: Section 10.4.11 and Figure 10.10.

More Vocabulary

Strongly connected (directed): every ordered pair joined both ways. The digraph of Section 10.8 has components $\{v, x, y\}, \{u\}, \{w\}, \{z\}$.

Subgraph: a subset of vertices with a subset of the edges among them; **spanning** if it keeps every vertex. Both searches produce spanning subgraphs.

Complete graph K_n: every pair adjacent, with the edge count of Equation (10.3). **Bipartite**: two sides, every edge crossing — exactly when there is **no odd cycle**.

$$|E| = \binom{n}{2} = \frac{n(n-1)}{2}. \tag{10.3}$$

Weighted: $w : E \to R$. **Sparse** ($|E|$ near $|V|$) against **dense** ($|E|$ near $|V|^2$): at 10^6 vertices a matrix has 10^{12} entries. BFS minimizes **edges**, never weights.

Book: Sections 10.4.9–10.4.16; the whole vocabulary is on one page in the book, Section 10.4.16.

Storing a Graph: The Running Example

Every representation is judged on four questions — **Q1** is (u, v) an edge? **Q2** $\deg(v)$? **Q3** list $Adj[v]$? **Q4** list all edges? — and on space. Figure 10.11 is the directed example.

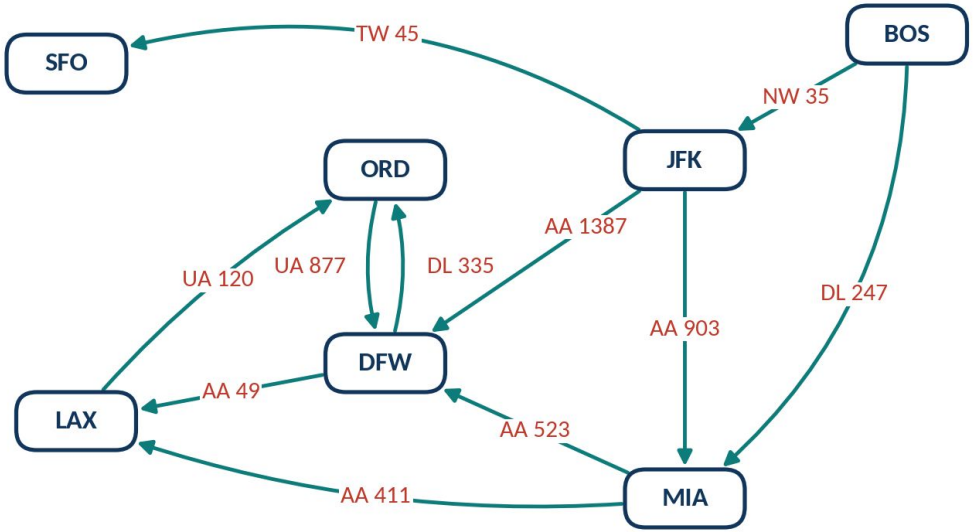


Figure 10.11 The running example for graph storage: a directed graph of flights between seven airports (BOS, JFK, ORD, SFO, LAX, DFW, MIA), each arc labeled with a flight number. There are eleven arcs. Because the arcs are directed, flight AA 49 from DFW to LAX is a different edge from flight DL 335 from DFW to ORD, and neither implies a flight in the reverse direction.

Book: Section 10.5.1 and Figure 10.11.

The Edge List

Figure 10.12: a container of vertices and a container of edges, each edge pointing at its endpoints. Space $\Theta(V+E)$; Q4 in $\Theta(E)$ — but Q1, Q2 and Q3 all scan everything.

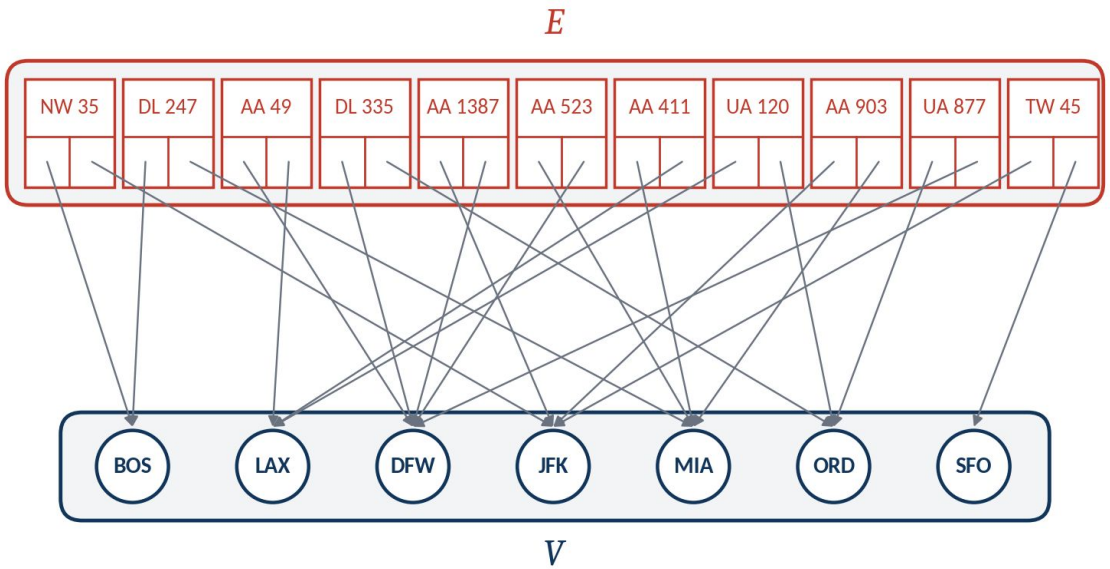


Figure 10.12 The edge list, built on the flight graph of Figure 10.11. The lower container holds the seven vertices; the upper container holds the eleven edges, and every edge object points down at its two endpoints. To find the flights leaving DFW you must inspect every one of the eleven edge objects, because nothing in the structure groups them by airport.

Book: Section 10.5.2 and Figure 10.12. The edge list is exactly what Kruskal's algorithm needs (Chapter 11).

The Adjacency List

Figure 10.13: one list per vertex, $2|E|=14$ entries for an undirected graph. Space $\Theta(V+E)$; Q3 in $\Theta(\deg v)$, optimal. **The default.**

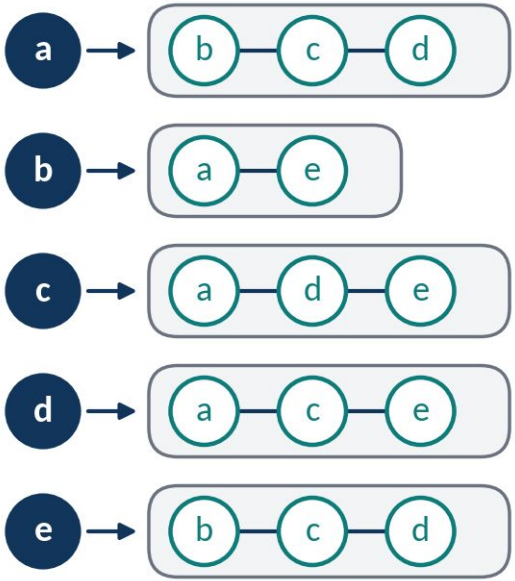
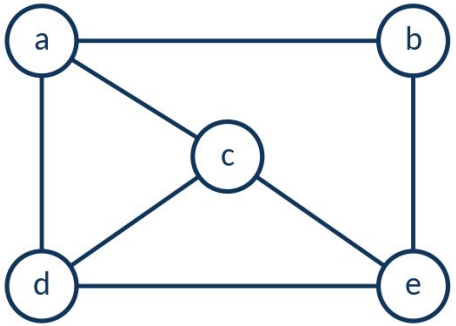


Figure 10.13 The adjacency lists of the chapter's graph. Five lists, one per vertex. Vertex a's list holds b, c, d; vertex b's holds a, e; and so on. The lists together hold 14 entries, which is $2|E|$ — the handshake lemma, made visible.

Book: Section 10.5.3 and Figure 10.13.

The Adjacency Matrix

Figure 10.14: $M[i][j]$ is true exactly when (i, j) is an edge. Symmetric for an undirected graph, row sum = degree. Q1 in $\Theta(1)$ — but space $\Theta(V^2)$ and Q3 in $\Theta(V)$ even for an isolated vertex.



	a	b	c	d	e
a	F	T	T	T	F
b	T	F	F	F	T
c	T	F	F	T	T
d	T	F	T	F	T
e	F	T	T	T	F

Figure 10.14 The adjacency matrix of the chapter's graph, drawn beside the graph itself. Row c reads T, F, F, T, T: vertex c is adjacent to a, d and e but not to b, and the diagonal entry is F because there is no self-loop. The matrix is symmetric about its diagonal because the graph is undirected, and it has 25 entries for a graph with only 7 edges.

Book: Section 10.5.4 and Figure 10.14.

The Three, Side by Side

Table 10.1 The cost of every basic operation in each of the three representations. No representation wins every row, which is why all three survive. The last row is the one that decides this chapter: a search over an adjacency list is linear; the same search over a matrix is quadratic.

Question	Edge list	Adjacency list	Adjacency matrix
Q1: is (u, v) an edge?	$\Theta(E)$	$O(\deg u)$, worst $O(V)$	$\Theta(1)$
Q2: what is $\deg(v)$?	$\Theta(E)$	$\Theta(1)$	$\Theta(V)$
Q3: list $Adj[v]$	$\Theta(E)$	$\Theta(\deg v)$	$\Theta(V)$
Q4: list all edges	$\Theta(E)$	$\Theta(V+E)$	$\Theta(V^2)$
Add an edge	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Delete edge (u, v)	$\Theta(E)$	$O(\deg u + \deg v)$	$\Theta(1)$
Space	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Cost of a full BFS or DFS	—	$\Theta(V+E)$	$\Theta(V^2)$

Table 10.1: no representation wins every row. **Searches, Prim, Dijkstra → adjacency list. Dense graphs, constant-time edge tests, all-pairs algorithms → matrix. Kruskal, reading from a file → edge list.**

Book: Sections 10.5.5–10.5.8; the program that builds all three is Section 10.5.8.

Searching, and Breadth-First Search

> **THREE COLORS. White**: undiscovered. **Gray**: discovered, edges not all examined — the **frontier**. **Black**: finished. Colors move only white → gray → black, so each vertex enters the container at most once.

Explore the place learned about **earliest** → a **queue** → **breadth-first search**. The one learned about **most recently** → a **stack** → **depth-first search**.

BFS is a labyrinth with a string unrolled **one edge at a time**: it computes `color[v]`, the distance `d[v]` and the predecessor `π[v]`. **First touch wins**: a vertex's distance is fixed when it is discovered.

Book: Section 10.6 and Sections 10.7.1–10.7.2.

The BFS Algorithm, Color-Coded

Figure 10.15: amber initializes, teal sets up the source and the queue, grey is the search.

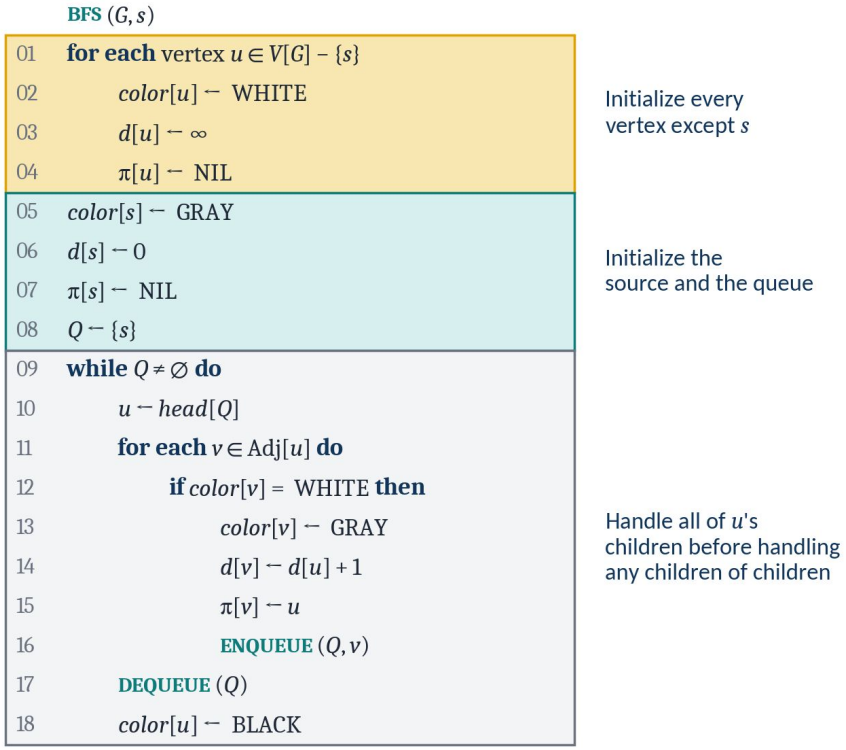


Figure 10.15 Breadth-first search. The amber block (lines 01–04) initializes every vertex other than the source; the teal block (05–08) initializes the source and the queue; the grey block (09–18) is the search itself, and the annotation on the right says what it achieves — *handle all u's children before handling any children of children*, which is the definition of breadth-first.

Book: Section 10.7.3 and Figure 10.15.

BFS in Pseudocode

Pseudocode 10.1 Breadth-first search from a source s, in the notation of this book, with the line numbers of Figure 10.15.

```
BFS(G, s)
 1  for each vertex u in G.V - {s}
 2      color[u] = WHITE
 3      d[u] = INFINITY
 4      pi[u] = NIL
 5  color[s] = GRAY
 6  d[s] = 0
 7  pi[s] = NIL                 // s, not u: a common slip
 8  Q = {s}
 9  while Q is not empty
10      u = head[Q]
11      for each v in Adj[u]
12          if color[v] == WHITE
13              color[v] = GRAY
14              d[v] = d[u] + 1
15              pi[v] = u       // v, not u: a common slip
16              ENQUEUE(Q, v)
17      DEQUEUE(Q)
18      color[u] = BLACK
```

TWO SLIPS TO AVOID. In Pseudocode 10.1, line 07 sets $\pi[s]$ and line 15 sets $\pi[v] \leftarrow u$. Writing $\pi[u] \leftarrow u$ still visits every vertex and computes every distance — and then returns nonsense paths. The most common real bug in BFS code.

Book: Section 10.7.3.

Worked Example 1: The Graph

$V = \{r, s, t, u, v, w, x, y\}$, $E = \{rs, rv, sw, tu, tw, tx, uy, wx, xy\}$ — eight vertices, nine edges, degrees summing to $18 = 2 \times 9$. Source s.

Adjacency order (it fixes the tree, not the distances): $Adj[s] = [w, r]$, $Adj[r] = [s, v]$, $Adj[w] = [s, t, x]$, $Adj[t] = [u, w, x]$, $Adj[x] = [t, w, y]$, $Adj[v] = [r]$, $Adj[u] = [t, y]$, $Adj[y] = [u, x]$.

The next five pages show the states: Figure 10.16, Figure 10.17, Figure 10.18, Figure 10.19 and Figure 10.20. White circle = white, gray = gray, filled = black; the number is d; thick edges are tree edges.

Book: Section 10.7.4.

Initialization and Step 1

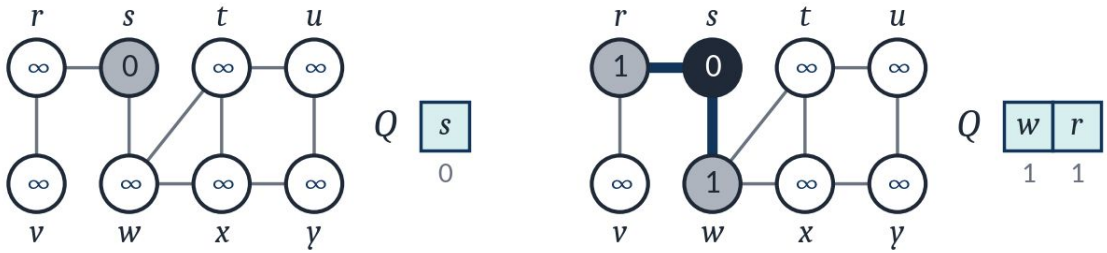


Figure 10.16 The BFS example, the first two states. **Left:** after initialization. Only s is gray and only s has a number, 0; every other vertex shows ∞. The queue holds s. **Right:** after step 1. Both neighbors of s are gray with distance 1, the two tree edges s–w and s–r are drawn thick, s is black, and the queue holds w then r — in that order, which fixes the order of $Adj[s]$ as $[w, r]$.

Book: Section 10.7.4 and Figure 10.16.

Steps 2 and 3

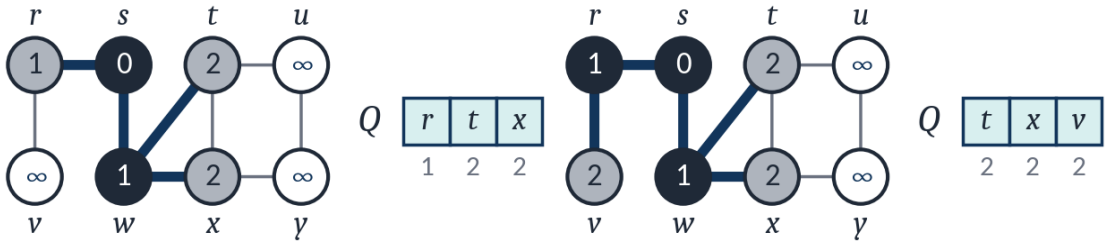


Figure 10.17 Steps 2 and 3. **Left:** w has been dequeued and blackened, discovering t and x at distance 2; the queue is r, t, x. **Right:** r has been dequeued and blackened, discovering v at distance 2; the queue is t, x, v — three vertices all at distance 2, the complete second ring.

Book: Section 10.7.4 and Figure 10.17. After step 3 the queue holds the whole ring at distance 2.

Steps 4 and 5

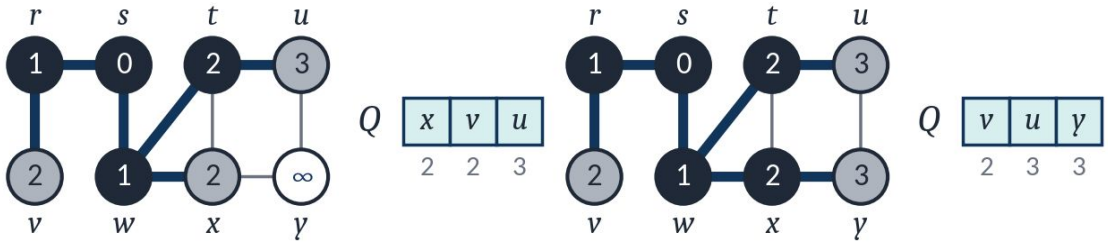


Figure 10.18 Steps 4 and 5. **Left:** t dequeued, discovering u at distance 3; the edge (t, x) was examined and rejected, because x was already gray with $d[x] = 2$. **Right:** x dequeued, discovering y at distance 3; the queue is v, u, y.

Book: Section 10.7.4 and Figure 10.18. At step 4, x is gray: first touch wins.

Steps 6 and 7

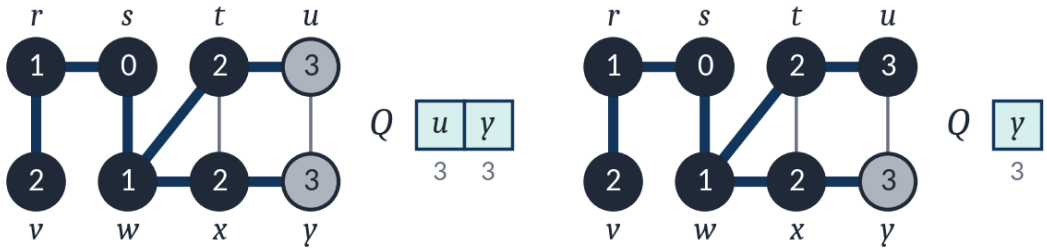


Figure 10.19 Steps 6 and 7. **Left:** v dequeued; its only neighbor r is black, so nothing is discovered and v simply turns black. **Right:** u dequeued; both its neighbors are already seen. The queue is down to y alone.

Book: Section 10.7.4 and Figure 10.19.

The Result

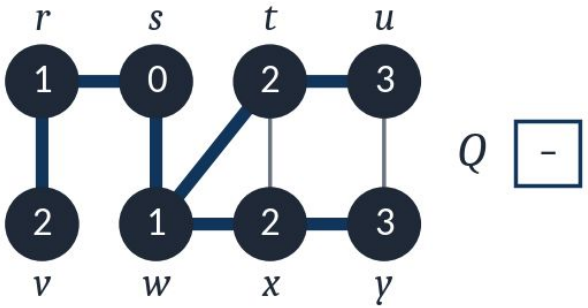


Figure 10.20 The result. Every vertex is black, the queue is empty, each vertex carries its distance from s, and the seven thick edges are the breadth-first tree. There are eight vertices and seven tree edges, exactly as Figure 10.10's arithmetic requires of a tree.

Book: Section 10.7.4 and Figure 10.20.

Reading the Answer Off

Table 10.2 The complete BFS of §10.7.4, one row per turn of the **while** loop. Every vertex is enqueued exactly once and dequeued exactly once, which is why there are eight steps for eight vertices. Note how the distances in the queue never decrease from left to right and never span more than one value — the property proved in Lemma 10.3.

Step	Dequeue	Adj Examined	Discovered, with d	Skipped, and why	Queue after the step
—	—	—	$s{:}\,d=0$	—	$(s:0)$
1	s	w, r	$w:1, r:1$	—	$(w:1, r:1)$
2	w	s, t, x	$t:2, x:2$	s black	$(r:1, t:2, x:2)$
3	r	s, v	$v:2$	s black	$(t:2, x:2, v:2)$
4	t	u, w, x	$u:3$	w black, x gray	$(x:2, v:2, u:3)$
5	x	t, w, y	$y:3$	t, w black	$(v:2, u:3, y:3)$
6	v	r	—	r black	$(u:3, y:3)$
7	u	t, y	—	t black, y gray	$(y:3)$
8	y	u, x	—	both black	$()$

Table 10.3 The output of BFS from s: the distance array d, the predecessor array π, and the shortest path to each vertex read out of π by following predecessors back to s and reversing. Every path's length equals the vertex's d value, as Theorem 10.4 requires.

v	r	s	t	u	v	w	x	y
$d[v]$	1	0	2	3	2	1	2	3
$\pi[v]$	s	NIL	w	t	r	s	w	x
shortest path from s	$s\,r$	s	$s\,w\,t$	$s\,w\,t\,u$	$s\,r\,v$	$s\,w$	$s\,w\,x$	$s\,w\,x\,y$

Table 10.2 is the trace; Table 10.3 the answer. Path to y: $\pi[y]=x$, $\pi[x]=w$, $\pi[w]=s$; reversed, $s\,w\,x\,y$, length $3=d[y]$.

Book: Section 10.7.5.

The Program

Listing 10.1 prints the queue after every step; its queue sequence — (w, r), (r, t, x), (t, x, v), (x, v, u), (v, u, y), (u, y), (y), empty — matches the table and the figures.

Listing 10.1 Breadth-first search in C++, printing the queue at every step. The adjacency lists are written out explicitly rather than built with `addEdge`, so that their *order* matches Figure 10.16 to Figure 10.19 exactly and the printed trace can be compared with Table 10.2 row by row.

```cpp
// Breadth-first search on the graph of Chapter 10, with the queue printed
// at every step, exactly as the figures of Section 10.7.4 show it.
//   V = {r,s,t,u,v,w,x,y}
//   E = {rs, rv, sw, tu, tw, tx, uy, wx, xy}
#include <iostream>
#include <vector>
#include <deque>
#include <string>
#include <algorithm>
#include <climits>
using namespace std;

enum Color { WHITE, GRAY, BLACK };
static const char* CNAME[] = {"white", "gray ", "black"};
static const int INF = INT_MAX;

struct Graph {
    int n;
    vector<vector<int>> adj;
    explicit Graph(int n_) : n(n_), adj(n_) {}
    void addEdge(int u, int v) { adj[u].push_back(v); adj[v].push_back(u); }
};

static const string NAME = "rstuvwxy";
static string qToString(const deque<int>& Q, const vector<int>& d) {
    if (Q.empty()) return "( empty )";
    string s = "(";
    for (size_t i = 0; i < Q.size(); ++i) {
```

```
        if (i) s += " ";
        s += NAME[Q[i]];
        s += ":";
        s += to_string(d[Q[i]]);
    }
    return s + ")";
}

// Returns d[] and pi[]; prints the queue after every enqueue and dequeue.
void bfs(const Graph& G, int s, vector<int>& d, vector<int>& pi, bool trace) {
    vector<Color> color(G.n, WHITE);
    d.assign(G.n, INF);
    pi.assign(G.n, -1);
```

Listing 10.1 (Continued) *Breadth-first search in C++, printing the queue at every step. The adjacency lists are written out explicitly rather than built with addEdge, so that their order matches Figure 10.16 to Figure 10.19 exactly and the printed trace can be compared with Table 10.2 row by row.*

```
    color[s] = GRAY; d[s] = 0; pi[s] = -1;   // the source: gray, distance 0
    deque<int> Q; Q.push_back(s);
    int step = 0;
    if (trace) cout << "  init      enqueue " << NAME[s] << "   Q = " << qToString(Q,d) << "\n";

    while (!Q.empty()) {
        int u = Q.front();                    // u = head[Q]
        Q.pop_front();                        // Dequeue(Q)
        ++step;
        if (trace) cout << "  step " << step << "    dequeue " << NAME[u]
                        << "   Q = " << qToString(Q,d) << "\n";
        for (int v : G.adj[u]) {              // scan Adj[u] once, and only once
            if (color[v] == WHITE) {
                color[v] = GRAY;
                d[v]  = d[u] + 1;             // one edge further than u
                pi[v] = u;
                Q.push_back(v);
                if (trace) cout << "            discover " << NAME[v]
                                << " via " << NAME[u] << ", d=" << d[v]
                                << "  Q = " << qToString(Q,d) << "\n";
            } else if (trace) {
                cout << "            skip " << NAME[v] << " (already "
                     << CNAME[color[v]] << ")\n";
            }
        }
        color[u] = BLACK;                     // u is finished
        if (trace) cout << "           " << NAME[u] << " -> black\n";
    }
```

```
}

vector<int> pathTo(int v, const vector<int>& pi) {
    vector<int> p;
    for (int x = v; x != -1; x = pi[x]) p.push_back(x);
    reverse(p.begin(), p.end());
    return p;
}
```

Listing 10.1 (Continued) *Breadth-first search in C++, printing the queue at every step. The adjacency lists are written out explicitly rather than built with addEdge, so that their order matches Figure 10.16 to Figure 10.19 exactly and the printed trace can be compared with Table 10.2 row by row.*

```
int main() {
    Graph G(8);
    // r=0 s=1 t=2 u=3 v=4 w=5 x=6 y=7 ; the adjacency order used by the figures
    G.adj[1] = {5,0};        // s: w then r  -- the order the figures' queue shows
    G.adj[0] = {1,4};        // r: s, v
    G.adj[5] = {1,2,6};      // w: s, t, x
    G.adj[2] = {3,5,6};      // t: u, w, x
    G.adj[6] = {2,5,7};      // x: t, w, y
    G.adj[4] = {0};          // v: r
    G.adj[3] = {2,7};        // u: t, y
    G.adj[7] = {3,6};        // y: u, x

    vector<int> d, pi;
    cout << "BFS from s\n";
    bfs(G, 1, d, pi, true);

    cout << "\nresult\n  vertex :";
    for (int i = 0; i < 8; ++i) cout << "  " << NAME[i];
    cout << "\n  d      :";
    for (int i = 0; i < 8; ++i) cout << "  " << d[i];
    cout << "\n  pi     :";
    for (int i = 0; i < 8; ++i) cout << "  " << (pi[i] < 0 ? '-' : NAME[pi[i]]);
    cout << "\n";

    cout << "\nbreadth-first tree edges (pi[v], v)\n  ";
    for (int v = 0; v < 8; ++v)
        if (pi[v] >= 0) cout << "(" << NAME[pi[v]] << "," << NAME[v] << ") ";
    cout << "\n";
```

```
    cout << "\nshortest paths read out of pi\n";
    for (int v = 0; v < 8; ++v) {
        cout << "  s to " << NAME[v] << " (" << d[v] << (d[v]==1 ? " edge" : " edges") << "): ";
        for (int x : pathTo(v, pi)) cout << NAME[x];
        cout << "\n";
    }
    return 0;
}
```

Book: Section 10.7.6, where the complete output is printed.

Cost, and the Breadth-First Tree

Each vertex is enqueued at most once (white is tested, then painted gray); each list is scanned once, on dequeue; the lists total $\Theta(E)$ by Equation (10.1). Hence Equation (10.4) — linear in the input, so optimal.

$$T(n) = \underbrace{O(V)}_{\text{initialize}} + \underbrace{O(V)}_{\text{queue operations}} + \underbrace{O(E)}_{\text{scanning the lists}} = O(V+E). \tag{10.4}$$

$O(V+E)$ IS A CLAIM ABOUT ADJACENCY LISTS. Over a matrix, line 11 costs $\Theta(V)$ per vertex and BFS becomes $\Theta(V^2)$ — about 136 times slower at 8 000 vertices.

The predecessor subgraph Equation (10.5) is the **breadth-first tree**: a spanning tree of s's component, $|V_\pi| - 1$ edges, every tree path a shortest path. Here 7 tree edges; (t, x) and (u, y) were rejected.

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}, E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}. \tag{10.5}$$

Book: Sections 10.7.7 and 10.7.8.

Why BFS Finds Shortest Paths (1)

Lemma 10.1 (no algorithm involved): for every edge (u, v), Equation (10.6). One more step cannot take you two steps further from home.

$$\delta(s, v) \leq \delta(s, u) + 1. \tag{10.6}$$

Lemma 10.2: BFS never underestimates, $d[v] \geq \delta(s, v)$ — by induction on the enqueues, using Lemma 10.1.

Lemma 10.3: the queue $\langle v_1, \ldots, v_r \rangle$ satisfies Equation (10.7) — sorted by distance, at most two distinct values. So vertices are dequeued in non-decreasing order of d.

$$d[v_1] \leq d[v_2] \leq \cdots \leq d[v_r] \leq d[v_1] + 1. \tag{10.7}$$

Book: Section 10.7.9, Lemmas 10.1–10.3.

Why BFS Finds Shortest Paths (2)

Theorem 10.4. Suppose some d is wrong; take v with the smallest $\delta(s,v)$ among the wrong ones. By Lemma 10.2, Equation (10.8). Let u precede v on a shortest path; u is right, so Equation (10.9).

$$d[v] > \delta(s,v). \tag{10.8}$$

$$d[v] > \delta(s,v) = \delta(s,u) + 1 = d[u] + 1. \tag{10.9}$$

When u is dequeued, v is white (then $d[v] = d[u]+1$), black ($d[v] \le d[u]$) or gray ($d[v] \le d[u]+1$) — every case contradicts Equation (10.9). ▰

Corollary 10.5: every edge joins levels differing by at most one — the fact bipartiteness testing rests on.

Book: Section 10.7.9, Theorem 10.4 and Corollary 10.5.

Depth-First Search

A labyrinth with a string **and a can of paint**: walk as far as you can, then back up to the last junction with an unexplored corridor. The string is the **route back** — a stack.

> **KEY IDEA.** The gray vertices form a **linear chain**: the path from the root to where you stand. "Is v gray?" means "is v an ancestor of here?" — which is why a gray vertex ahead means a cycle.

DFS takes **no source**: its outer loop starts a new tree at every white vertex, so it explores the whole graph and produces a **forest**. Figure 10.21 gives both procedures.

```
DFS (G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3  time ← 0
4  for each vertex u ∈ V[G]
5       do if color[u] = WHITE
6            then DFS-VISIT (u)

DFS-VISIT (u)
1  color[u] ← GRAY          ▷ White vertex u discovered.
2  d[u] ← time              ▷ Mark with discovery time.
3  time ← time + 1          ▷ Tick global time.
4  for each v ∈ Adj[u]      ▷ Explore all edges (u, v).
5       do if color[v] = WHITE
6            then DFS-VISIT (v)
7  color[u] ← BLACK         ▷ Blacken u; it is finished.
8  f[u] ← time              ▷ Mark with finishing time.
9  time ← time + 1          ▷ Tick global time.
```

Figure 10.21 Depth-first search. The outer procedure DFS(G) whitens every vertex, starts the clock, and then calls DFS-Visit on every vertex that is still white; each such call roots one new tree of the depth-first forest. The inner procedure DFS-Visit(u) is the search proper: paint u gray, stamp its discovery time, recurse into every white neighbor, then paint u black and stamp its finishing time.

Book: Sections 10.8.1 and 10.8.2, and Figure 10.21.

DFS in Pseudocode, and Two Timestamps

Pseudocode 10.2 Depth-first search: the outer procedure starts a new tree at every vertex still white, and DFS-Visit conducts the search, stamping discovery and finishing times.

```
DFS(G)
 1  for each vertex u in G.V
 2      color[u] = WHITE
 3      pi[u] = NIL
 4  time = 0
 5  for each vertex u in G.V
 6      if color[u] == WHITE
 7          DFS-VISIT(u)

DFS-VISIT(u)
 1  color[u] = GRAY              // white vertex u has just been discovered
 2  time = time + 1
 3  d[u] = time                  // stamp the discovery time
 4  for each v in Adj[u]         // explore every edge (u, v)
 5      if color[v] == WHITE
 6          pi[v] = u
 7          DFS-VISIT(v)
 8  color[u] = BLACK             // u is finished
 9  time = time + 1
10  f[u] = time                  // stamp the finishing time
```

Pseudocode 10.2 increments the clock first (as the worked example does). Each vertex gets a **discovery time** $d[u]$ and a **finishing time** $f[u]$, with $d[u] < f[u]$; the clock ends at exactly $2|V|$.

Book: Sections 10.8.2 and 10.8.3.

Worked Example 2: Sixteen Steps

Arcs $u \to v$, $u \to x$, $v \to y$, $w \to y$, $w \to z$, $x \to v$, $y \to x$, $z \to z$; outer loop u, v, w, x, y, z.

u $d=1$; tree $u \to v$ ($d=2$); tree $v \to y$ (3); tree $y \to x$ (4); $x \to v$ finds v **gray**: **back** edge. x finishes at 5, y at 6, v at 7.

$u \to x$: x black and $d[u] < d[x]$ — **forward**. u finishes at 8. New tree at w ($d=9$): $w \to y$ black, $d[w] > d[y]$ — **cross**.

Tree $w \to z$ (10); $z \to z$ finds z gray — **back** (a self-loop needs no special case). z finishes at 11, w at 12 $= 2|V|$. The three drawings are Figure 10.22, Figure 10.23 and Figure 10.24.

Book: Section 10.8.4.

Steps 1 to 6

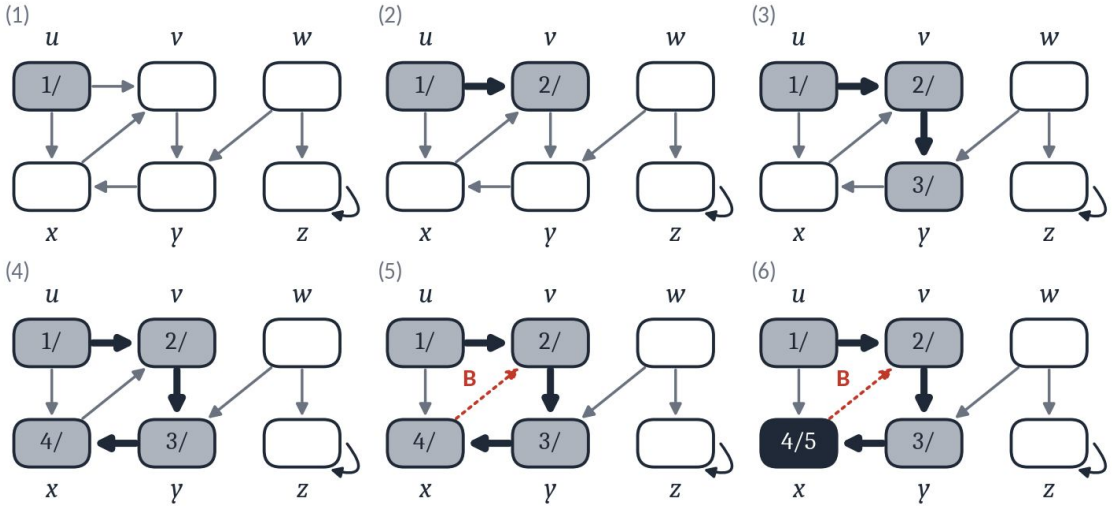


Figure 10.22 Steps 1 to 6 of the DFS example, in reading order: u turns gray with $d=1$; v is discovered as a tree edge with $d=2$; y with $d=3$; x with $d=4$; the edge (x, v) is examined, found to point at a gray vertex, and labeled **B** for back edge, drawn as a dotted arrow; and finally x turns black with $f=5$, shown as $4/5$.

Book: Section 10.8.4 and Figure 10.22.

Steps 7 to 12

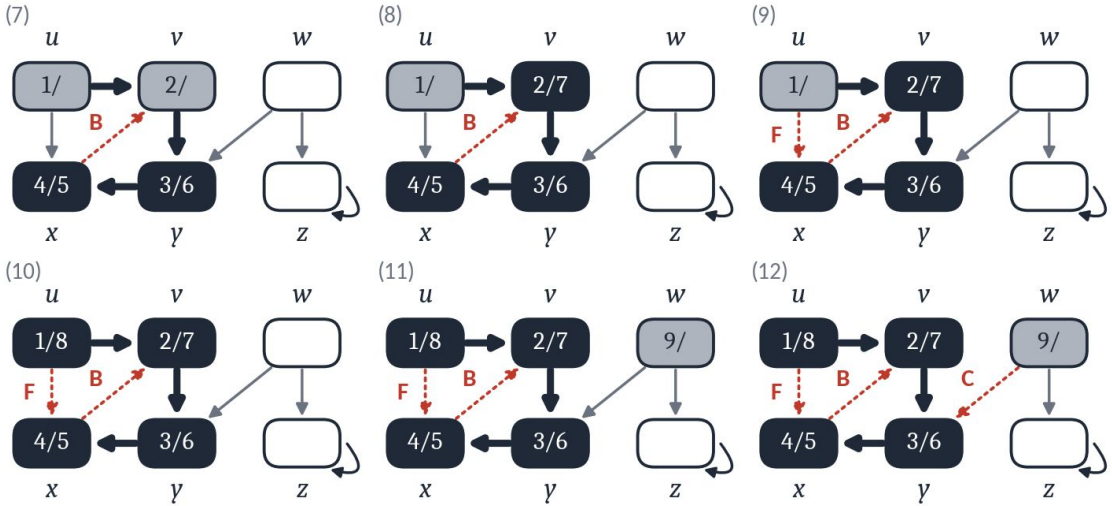


Figure 10.23 Steps 7 to 12. The recursion unwinds, blackening y at $3/6$ and v at $2/7$; back in u's adjacency list the edge (u, x) is examined and labeled **F** for forward edge; u finishes at $1/8$, completing the first tree; the outer loop then roots a second tree at w with $d = 9$; and the edge (w, y) is labeled **C** for cross edge, since y belongs to the first tree.

Book: Section 10.8.4 and Figure 10.23.

Steps 13 to 16

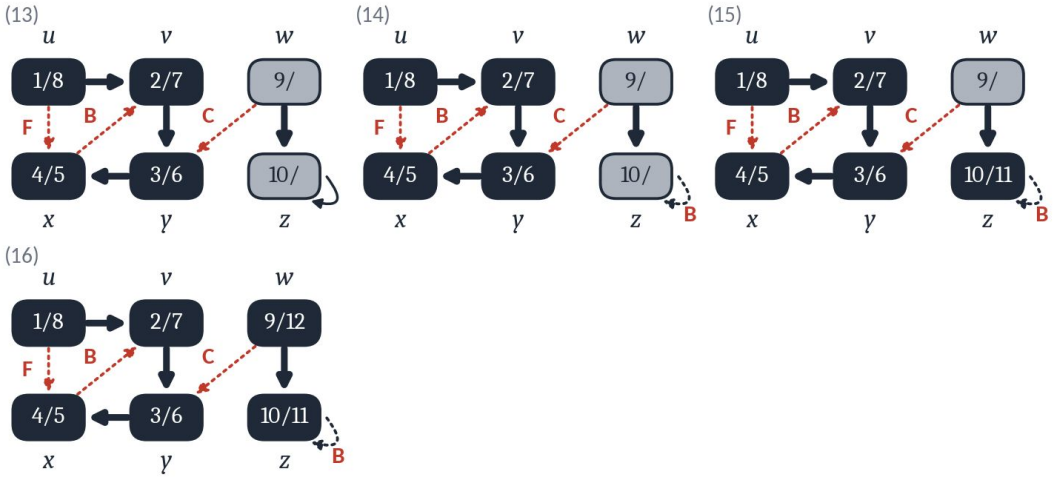


Figure 10.24 Steps 13 to 16. z is discovered from w as a tree edge with $d=10$; the self-loop (z, z) is examined and labeled **B**, because at that instant z is gray; z finishes at $10/11$; and w finishes at $9/12$. Every vertex is now black and the clock has reached $12=2|V|$.

Book: Section 10.8.4 and Figure 10.24.

The Answer

Table 10.4 Timestamps and predecessors after DFS of the graph of §10.8.4. The two roots are u and w, the two vertices with no predecessor. The twelve stamps are the integers 1 to 12, each used once.

v	u	v	w	x	y	z
$d[v]$	1	2	9	4	3	10
$f[v]$	8	7	12	5	6	11
$\pi[v]$	NIL	u	NIL	y	v	w

Table 10.5 Every arc of the example, classified, with the color test that made the decision. All eight arcs appear: three tree edges in the first tree, one in the second, two back edges, one forward, one cross.

Arc	Class	The test that decided it
$u \to v$	tree	v was WHITE
$v \to y$	tree	y was WHITE
$y \to x$	tree	x was WHITE
$x \to v$	back	v was GRAY — an ancestor
$u \to x$	forward	x was BLACK and $d[u] < d[x]$ — a descendant
$w \to y$	cross	y was BLACK and $d[w] > d[y]$ — neither
$w \to z$	tree	z was WHITE
$z \to z$	back	z was GRAY — itself, and gray

Table 10.4 and Table 10.5: two roots, four tree edges $(6 - 2)$, two back, one forward, one cross.

Book: Section 10.8.4.

Forest, Stack, Cost

The depth-first forest Equation (10.10) spans **all** of V: here the trees $u \to v \to y \to x$ and $w \to z$.

$$E_\pi = \{(\pi[v], v) \in E : v \in V \text{ and } \pi[v] \neq \text{NIL}\}. \tag{10.10}$$

Without recursion: a stack of frames $(u, next)$ — the resume position is essential, or returning to a vertex restarts its list. Needed on deep graphs, where recursion overflows the call stack.

Cost Equation (10.11): two $\Theta(V)$ loops, one DFS-Visit per vertex, every list walked once — **aggregate analysis**.

$$T(n) = \underbrace{\Theta(V)}_{\text{two outer loops}} + \underbrace{\Theta(V)}_{|V| \text{ calls, constant overhead each}} + \underbrace{\Theta(E)}_{\text{walking all the lists}} = \Theta(V + E). \tag{10.11}$$

Book: Sections 10.8.5–10.8.7.

A Larger Example, Timestamped

Figure 10.25: thirteen arcs on eight vertices, two trees and all four edge classes.

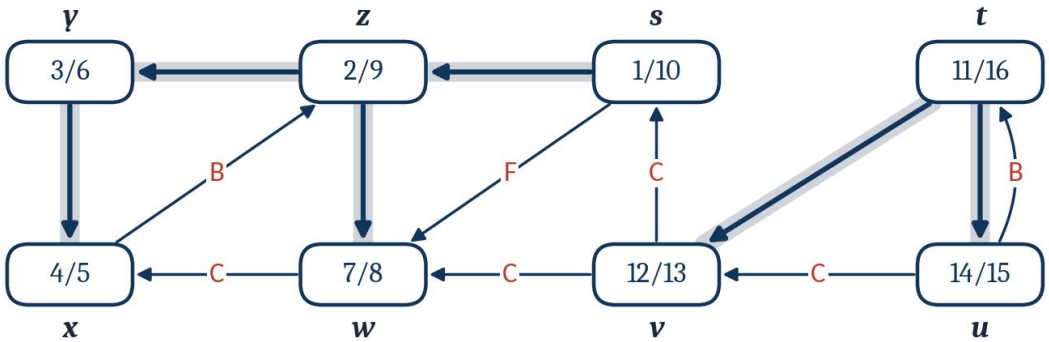


Figure 10.25 The larger DFS example, timestamped. Each vertex box holds d/f: s is $1/10$, z is $2/9$, y is $3/6$, x is $4/5$, w is $7/8$, t is $11/16$, v is $12/13$, u is $14/15$. The thick gray arrows are tree edges; the thin ones are labeled **B** (back), **F** (forward) and **C** (cross). This one figure carries the timestamps, the forest and the complete edge classification at once.

Book: Section 10.8.8 and Figure 10.25.

The Parenthesis Structure

Figure 10.26: one bar per vertex spanning $[d, f]$. First example: $(u(v(y(x\,x)\,y)\,v)\,u)(w(z\,z)\,w)$.

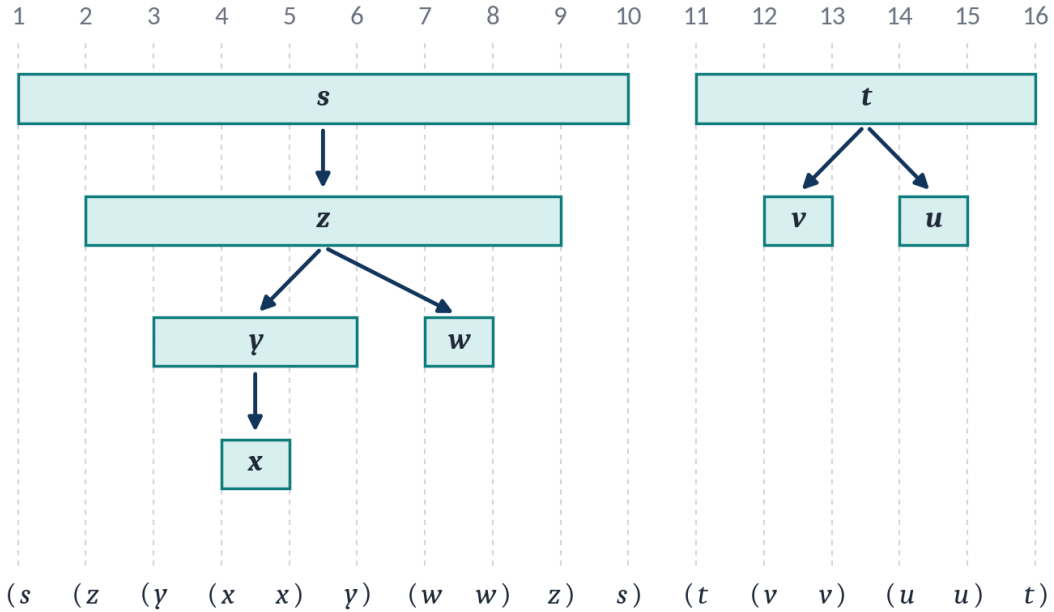


Figure 10.26 The parenthesis structure of Figure 10.25. Above, the depth-first forest redrawn with tree edges pointing downward. Below, the clock from 1 to 16 with one horizontal bar per vertex, spanning $[d, f]$, and the parenthesization written along the bottom: $(s(z(y(xx)y)(ww)z)s)(t(vv)(uu)t)$. Every pair of bars is either nested or disjoint; none overlap partially.

Book: Section 10.8.8 and Figure 10.26.

Parenthesis and White-Path Theorems

THEOREM 10.6 (PARENTHESIS). For any u, v the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are either **disjoint** (neither is a descendant) or **nested** (the inner one is the descendant) — never partially overlapping.

Corollary 10.7: v is a proper descendant of u exactly when Equation (10.12) — tree shape becomes a comparison of four integers.

$$d[u] < d[v] < f[v] < f[u], \tag{10.12}$$

THEOREM 10.8 (WHITE PATH). v becomes a descendant of u if and only if, at time $d[u]$, a path of **white** vertices runs from u to v.

Book: Sections 10.8.8 and 10.8.9, Theorems 10.6 and 10.8.

Classifying the Edges

By the color of v when (u, v) is examined: **tree** (white), **back** (gray), **forward** (black, $d[u] < d[v]$), **cross** (black, $d[u] > d[v]$). Figure 10.27 makes it visual.

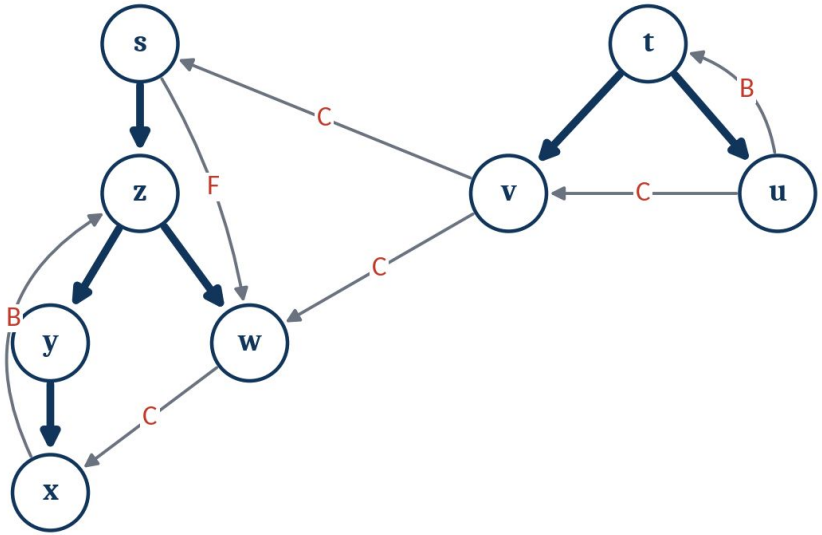


Figure 10.27 The graph of Figure 10.25 redrawn as its depth-first forest, with tree edges thick and downward. Now the classification is visual: $x \to z$ and $u \to t$ curve back **up** to an ancestor (**B**); $s \to w$ points **down** past an intermediate level to a descendant (**F**); $w \to x$, $v \to w$, $v \to s$ and $u \to v$ go **sideways** between branches or between the two trees (**C**).

Book: Section 10.8.10 and Figure 10.27.

The Four Classes, and Undirected Graphs

Table 10.6 The four edge classes: which graphs they can occur in, the color test that identifies them, the timestamp pattern they produce, and what each is good for. In an undirected graph only the first two columns are possible, which is Theorem 10.9.

	Tree	Back	Forward	Cross
Undirected graph	yes	yes	never	never
Directed graph	yes	yes	yes	yes
Color of v when (u, v) examined	white	gray	black	black
Timestamp test	—	$d[v] < d[u] < f[u] < f[v$	$d[u] < d[v] < f[v] < f[u$	$d[v] < f[v] < d[u] < f[u$
What it tells you	the forest	a cycle	nothing much	nothing much

Table 10.6. **Theorem 10.9:** in an **undirected** graph every edge is a tree edge or a back edge — so *any* non-tree edge means a cycle.

Book: Section 10.8.10, Theorem 10.9.

Four Things the Searches Can Do

Components: BFS from each unlabeled vertex; $O(V+E)$ in total, then "connected?" is one comparison. The graph of Figure 10.9: three components of sizes 3, 4, 5.

Cycles: undirected — a non-white neighbor other than the parent (skip the parent **exactly once**); directed — an arc into a **gray** vertex. Testing `visited` instead of gray reports cross and forward edges as cycles.

Bipartiteness: color by the parity of the BFS distance; an edge between two vertices of the same level exhibits an odd cycle.

Shortest paths, unweighted: BFS itself; stop early at the target; many sources start in the queue together; with weights, use Dijkstra (Chapter 13).

Book: Sections 10.9.1–10.9.4.

Directed Acyclic Graphs

A **DAG** is a directed graph with no directed cycle — the model of precedences. Figure 10.28 shows two.

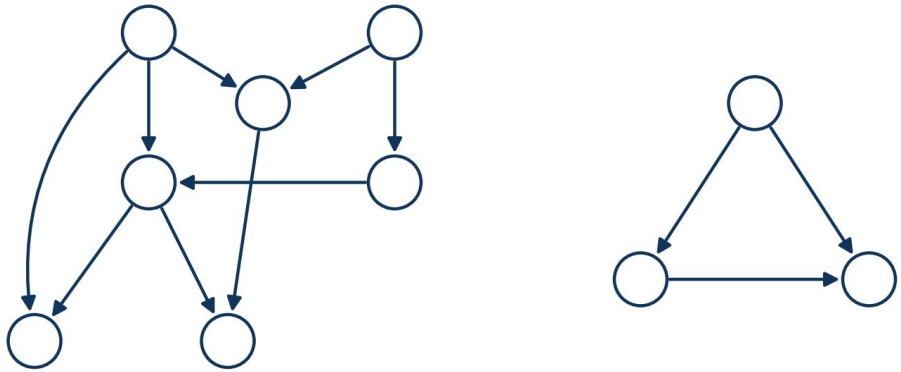


Figure 10.28 Two directed acyclic graphs. In each, follow the arrows from anywhere and you can never return to where you started: there is no directed cycle. A DAG is the natural model of a set of precedences — "this before that" — and an acyclic set of precedences is exactly a satisfiable one.

Book: Section 10.9.5 and Figure 10.28.

The DAG Theorem and Topological Sort

THEOREM 10.10. A directed graph is acyclic **if and only if** a DFS yields no back edge. (Back edge + tree path = cycle; conversely the first-discovered vertex of a cycle sees a white path all the way round.)

Pseudocode 10.3 Topological sort by depth-first search: the vertices in decreasing order of finishing time.

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times f[v] for each vertex v
2  as each vertex is finished, insert it onto the FRONT of a linked list
3  return the linked list of vertices
```

Pseudocode 10.3 lists the vertices in **decreasing finishing time**. **Theorem 10.11:** in a DAG, $f[v] < f[u]$ for every arc (u, v) — a gray v is impossible, a white v finishes inside u's call, a black v has finished already. Cost $O(V+E)$, no sorting.

Book: Sections 10.9.5 and 10.9.6, Theorems 10.10 and 10.11.

Getting Dressed, Timestamped

Figure 10.29: an arrow $x \to y$ means x must be put on before y.

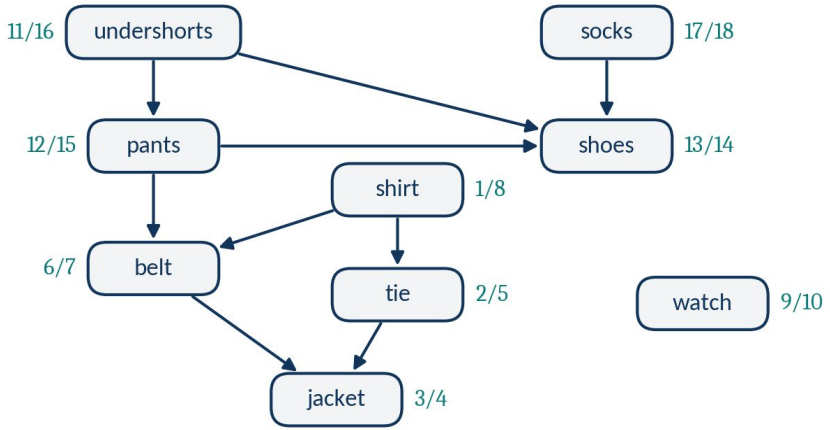


Figure 10.29 The dressing DAG, timestamped by a DFS. Nine garments; an arrow from x to y means x must be put on before y. The stamps are shirt $1/8$, tie $2/5$, jacket $3/4$, belt $6/7$, watch $9/10$, undershorts $11/16$, pants $12/15$, shoes $13/14$, socks $17/18$ — eighteen stamps for nine vertices, as $2|V|$ requires. The watch has no arrows at all: it may be put on at any time.

Book: Section 10.9.6 and Figure 10.29.

Every Arrow Points Right

Figure 10.30: socks, undershorts, pants, shoes, watch, shirt, belt, tie, jacket.

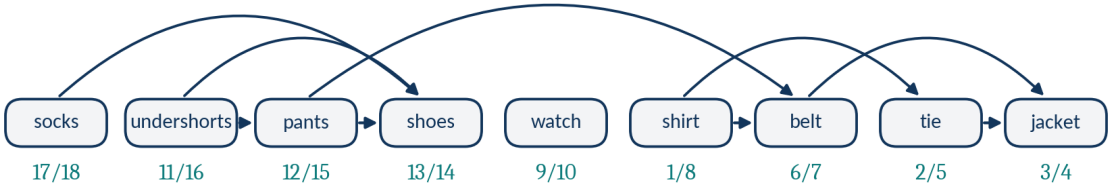


Figure 10.30 The topological order: the same nine vertices laid out left to right in decreasing order of finishing time — socks 18, undershorts 16, pants 15, shoes 14, watch 10, shirt 8, belt 7, tie 5, jacket 4 — and now **every arrow points right**. That is what a topological sort is.

Book: Section 10.9.6 and Figure 10.30. Chapter 11 adds Kahn's algorithm.

BFS or DFS?

Table 10.7 Breadth-first against depth-first, on eleven axes. They cost the same and differ in one line of code; everything else in this table follows from that one line.

	Breadth-first search	Depth-first search
Container	queue (FIFO)	stack (LIFO), usually the call stack
Takes a source?	yes, one per call	no — searches the whole graph
Produces	a tree, of one component	a forest, spanning all of V
Records	$d[v]$ and $\pi[v]$	$d[u]$, $f[u]$ and $\pi[u]$
Cost	$O(V+E)$	$\Theta(V+E)$
Extra space	$O(V)$ for the queue	$O(V)$ for the stack — or the call stack
Finds shortest paths?	**yes**, when edges are unweighted	no
Order of visiting	by increasing distance from the source	as deep as possible, then backtrack
Non-tree edges	only between levels k and $k, k+1$	back, forward and cross
Natural for	fewest hops, nearest thing, flood fill, level-by-level work, bipartiteness	cycles, topological sort, strongly connected components, articulation points, mazes, backtracking
Risk in practice	a large queue on a wide graph	stack overflow on a deep graph

Table 10.7: **distance, hops, nearest, level → BFS. Order, cycles, dependency → DFS.** Measured: doubling the graph doubles the adjacency-list time and quadruples the matrix time (0.778, 3.105, 12.263, 47.902 ms).

Book: Sections 10.9.7 and 10.9.8.

Profitina in Practice

The scan is a BFS over the directed link graph: white = seen in the frontier, gray = fetched, black = scored. BFS reaches the important pages first, and $d[v]$ **is the click depth**, a real GEO signal.

Components find orphan clusters; **cycle detection** guards the signal-dependency DAG; **topological sort** orders the signals; adjacency lists, because the site graph is sparse.

Profitina Laundry: BFS from Keputih puts six zones within two hops; Nginden Jangkungan is four hops away; Dukuh Setro is **unreachable**, not far. Adding *notify* → *invoice* makes the wash process cyclic, and the program names the cycle `pack -> notify -> invoice -> pack`.

Book: Section 10.10 and Section 10.10.1. The product advises; the user decides.

Common Mistakes

1. Not marking vertices as seen — the program never stops. **2. Marking on dequeue** instead of enqueue. **3. BFS on a weighted graph.**

4. `visited` **where three colors are needed** (directed cycles). **5. Forgetting the parent** in undirected cycle detection. **6. Dequeueing with** `vector::erase(begin())`**.**

7. Recursing on a deep graph. 8. Resuming an adjacency list from the start. 9. Adding only one direction of an undirected edge. 10. A $V \times V$ matrix for a sparse graph.

11. Assuming one BFS covers the graph. 12. Treating ∞ as "far". 13. Expecting a unique tree or order. 14. Confusing BFS's d (distance) with DFS's d (time). 15. Claiming $O(V+E)$ over a matrix.

Book: Section 10.11.

Summary: Five Things to Remember

1. A graph is $G=(V,E)$; the **handshake lemma** $\sum_{v} \deg(v)=2|E|$ is the source of every E in the running times.

2. Store it as **adjacency lists** unless you have a reason: $\Theta(V+E)$ space, optimal neighbor listing.

3. BFS = queue: $d[v]=\delta(s,v)$ in $O(V+E)$, proved by Lemmas 10.1–10.3 and Theorem 10.4.

4. DFS = stack: timestamps, the parenthesis and white-path theorems, and four edge classes (only tree and back when undirected).

5. Components, cycles, bipartiteness, shortest hops, the DAG theorem and topological sort — all $O(V+E)$, all a search plus a few lines.

Book: Section 10.12.

Next Time

Chapter 11 — Minimum Spanning Trees, DAGs, and Topological Sort. The DAG theorem proved again, and Kahn's in-degree algorithm beside the DFS method.

Weights arrive: the cheapest spanning tree, the **cut property**, and two greedy algorithms — **Prim** (BFS with a priority queue) and **Kruskal** (a sweep over the edge list).

Before the meeting: Exercises 10.1, 10.6, 10.9 and 10.10, and read Sections 11.1 to 11.4.

Book: Section 10.12.1 and Chapter 11.

Chapter 1

Lecture 11 — Minimum Spanning Trees, DAGs, and Topological Sort

Design and Analysis of Algorithms (EF234405)

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*

Lecturer: Irfan Subakti — yifana@gmail.com

Book: Chapter 11, Minimum Spanning Trees, DAGs, and Topological Sort. Every page of this handout ends with a pointer into the book, which is the primary reference.

Today's Meeting

1. Directed acyclic graphs and the DAG theorem, proved in both directions (Section 11.2).

2. Topological sort two ways — by DFS finishing times and by Kahn's in-degrees — traced on the dressing DAG (Section 11.3).

3. Spanning trees, the minimum-spanning-tree problem, optimal substructure and **the cut property** (Sections 11.4–11.8).

4. Prim–Jarník and **Kruskal**, each traced step by step on the same nine-vertex graph, with the disjoint-set forest that makes Kruskal fast (Sections 11.9–11.10).

5. Prim against Kruskal, ties and uniqueness, Profitina, the common mistakes (Sections 11.11–11.15).

Book: Chapter 11, Sections 11.1 to 11.15.

Where We Left Off

From Chapter 10: graphs, adjacency lists, and depth-first search with its clock — discovery time $d[u]$, finishing time $f[u]$, colors white, gray, black, predecessor $\pi[u]$.

The four edge classes: **tree** (white), **back** (gray, to an ancestor), **forward** and **cross** (black). Today turns on the back edge.

From Chapter 9: the greedy discipline — the **greedy-choice property** and **optimal substructure** must both be proved before greed is allowed.

What is missing: Chapter 10 never *measured* a graph. Attach a weight to each edge and ask for the cheapest set of edges that keeps everything connected.

Book: Section 11.1 and Section 11.1.1.

Directed Acyclic Graphs

A **DAG** is a directed graph with no cycle. **Figure 11.1** shows two; the right one looks like a triangle but its arrows do not run round it in one direction.

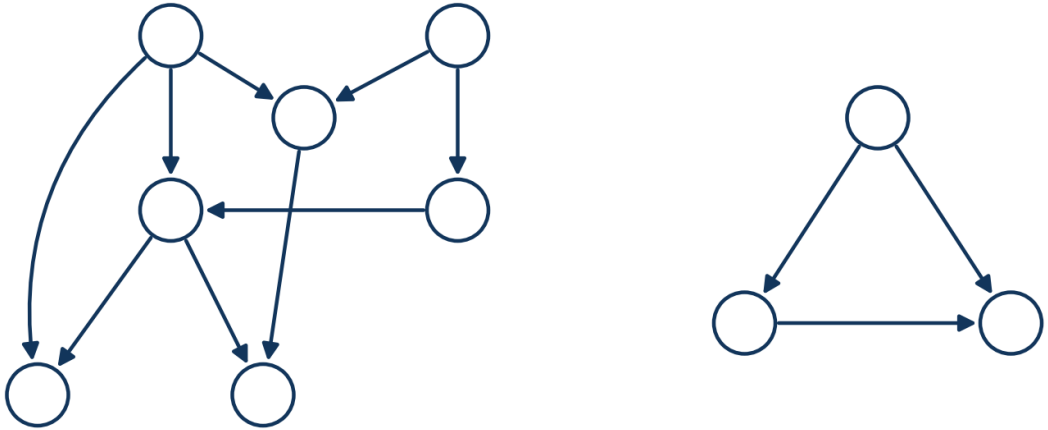


Figure 11.1 Two directed acyclic graphs. **Left:** follow the arrows from anywhere and you always run out of arrows. **Right:** three edges forming a triangle that is still acyclic, because the arrows do not run round it in one direction; reversing the edge from the top vertex to the bottom-right one would close a directed cycle.

Book: Sections 11.2.1 and 11.2.2, and Figure 11.1. Everyday DAGs: build dependencies, course prerequisites, task schedules, parallel code.

The DAG Theorem, Proved

THEOREM 11.1. A directed graph G is acyclic **if and only if** a depth-first search of G yields no back edges. So cycle detection costs $O(V + E)$.

Back edge ⇒ cycle. If (u, v) finds v gray, v is on the recursion stack, an ancestor of u: the tree path $v \rightsquigarrow u$ plus (u, v) closes a cycle.

Cycle ⇒ back edge. Let v be the **first** vertex of the cycle to be discovered and u its predecessor on the cycle. At $d[v]$ the rest of the cycle is white — a white path to u — so u becomes a descendant of v, and when (u, v) is examined v is still **gray**. ■

"FIRST" IS LOAD-BEARING, and the theorem is about **directed** graphs: in an undirected graph the edge you arrived on looks like a back edge.

Book: Sections 11.2.3–11.2.5, Theorem 11.1.

Topological Sort

> **DEFINITION.** A **topological sort** of a DAG is a linear ordering of **all** its vertices such that for every edge (u, v), u comes **before** v — lay them out and every arrow points right.

Almost never unique: unconstrained pairs may go either way. A chain has one order; n isolated vertices have $n!$.

Check the property, never a stored string.

The running example **Figure 11.2**: nine garments, nine edges — undershorts → pants, undershorts → shoes, pants → shoes, pants → belt, shirt → belt, shirt → tie, belt → jacket, tie → jacket, socks → shoes. `watch` is unconstrained.

Book: Sections 11.3.1–11.3.3.

Getting Dressed

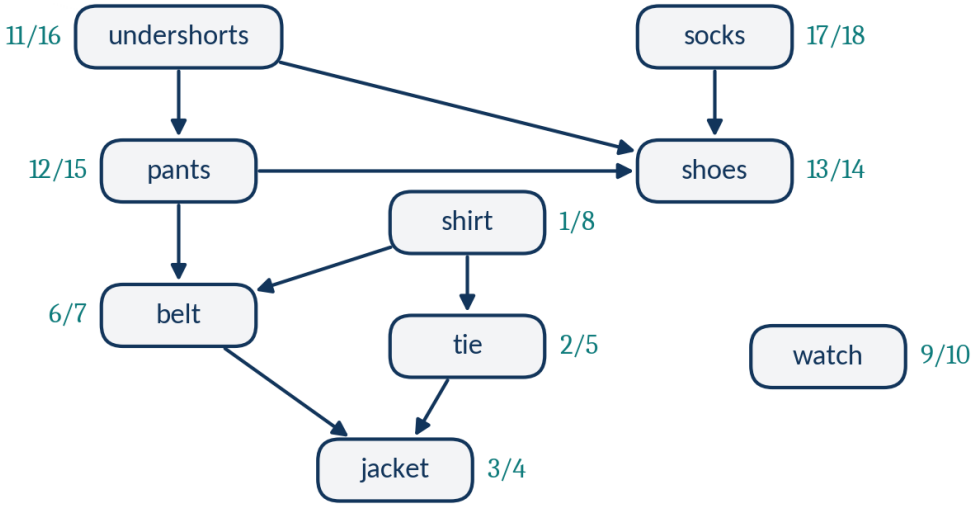


Figure 11.2 The precedence DAG for getting dressed, with the depth-first search timestamps $d[v]/f[v]$ written beside each vertex. Nine vertices, nine edges. `watch` has no edges at all: nothing constrains when you put it on.

Book: Section 11.3.3 and Figure 11.2.

Algorithm 1: Finishing Times

Pseudocode 11.1 Topological sort by depth-first search, recording each vertex at the moment it finishes.

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times f[v] for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

Pseudocode 11.1 runs one DFS and pushes each vertex onto the **front** of a list as it **finishes** — decreasing finishing time, with no sorting at all.

Intuition: a vertex finishes only after everything reachable from it has finished, so "last to finish" is "furthest upstream".

Book: Section 11.3.4.

Worked Example 1: Every Tick of the Clock

Table 11.1 Every tick of the depth-first clock on **Figure 11.2**. The clock advances by exactly one on each discovery and each finishing, so it ends at $2|V| = 18$. Compare the d/f pairs against the numbers written beside the vertices in **Figure 11.2**: they agree exactly.

Clock	Event	State
1	discover `shirt`	d[shirt] $= 1$; `shirt` gray
2	from `shirt`, `tie` is white: discover it	d[tie] $= 2$
3	from `tie`, `jacket` is white: discover it	d[jacket] $= 3$
4	`jacket` has no out-edges: finish it	f[jacket] $= 4$; list = `jacket`
5	back at `tie`, no more edges: finish it	f[tie] $= 5$; list = `tie, jacket`
6	back at `shirt`, `belt` is white: discover it	d[belt] $= 6$
7	from `belt`, `jacket` is **black** — nothing to do; finish `belt`	f[belt] $= 7$; list = `belt, tie, jacket`
8	back at `shirt`, no more edges: finish it	f[shirt] $= 8$
9	outer loop: `watch` is white, discover it	d[watch] $= 9$
10	`watch` has no out-edges: finish it	f[watch] $= 10$
11	outer loop: `undershorts` is white, discover it	d[undershorts] $= 11$
12	from `undershorts`, `pants` is white: discover it	d[pants] $= 12$

Clock	Event	State
13	from `pants`, `shoes` is white: discover it	$d[\text{shoes}] = 13$
14	`shoes` has no out-edges: finish it	$f[\text{shoes}] = 14$
15	back at `pants`, `belt` is black: finish `pants`	$f[\text{pants}] = 15$

Table 11.1 (Continued) *Every tick of the depth-first clock on Figure 11.2. The clock advances by exactly one on each discovery and each finishing, so it ends at 2V=18. Compare the d/f pairs against the numbers written beside the vertices in Figure 11.2: they agree exactly.*

Clock	Event	State
16	back at `undershorts`, `shoes` is black: finish it	$f[\text{undershorts}] = 16$
17	outer loop: `socks` is white, discover it	$d[\text{socks}] = 17$
18	`shoes` is black; finish `socks`	$f[\text{socks}] = 18$

Table 11.1: outer loop shirt, tie, jacket, belt, watch, undershorts, pants, shoes, socks. The clock ends at $2|V| = 18$.

Book: Section 11.3.5.

Every Arrow Points Right

Figure 11.3: finishing times 18,16,15,14,10,8,7,5,4 — socks, undershorts, pants, shoes, watch, shirt, belt, tie, jacket.

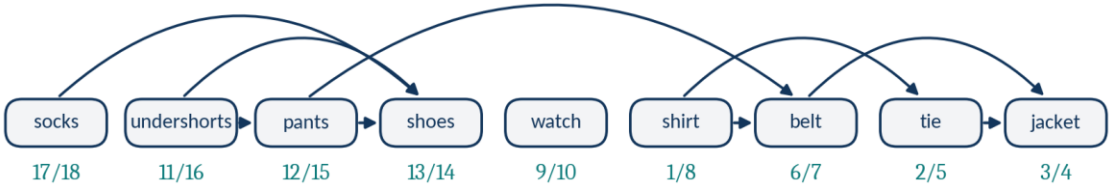


Figure 11.3 The result: the nine vertices of **Figure 11.2** laid out in decreasing order of finishing time. Every arrow now points to the right, which is exactly the definition of a topological sort. The d/f pair is printed under each vertex so the ordering can be checked against **Table 11.1**.

Book: Section 11.3.5 and Figure 11.3.

Why Finishing Times Work

THEOREM 11.2. In a DAG, $f[v] < f[u]$ for every edge (u, v); so decreasing f is a topological sort.

When (u, v) is examined, u is gray. v **gray:** a back edge, impossible in a DAG. v **white:** v becomes a child and finishes inside u's call. v **black:** v has already finished and u has not. ■

Cost $\Theta(V + E)$: one DFS plus $O(1)$ per front insertion — optimal, since the input has size $\Theta(V + E)$.

A TRAP. Insert at the **front** and at **finishing** time. Appending gives the reverse order; recording at discovery gives `shirt, tie, jacket`, and `jacket` has two predecessors not yet listed.

Book: Sections 11.3.6 and 11.3.7, Theorem 11.2; the program is Section 11.3.8.

Algorithm 2: Kahn's In-Degrees

Pseudocode 11.2 Kahn's topological sort: repeatedly emit a vertex with no remaining predecessor; a shortfall at the end proves a cycle.

```
KAHN(G)
 1  compute indeg[v] for every vertex v
 2  R = { v : indeg[v] == 0 }              // R is the "ready" set
 3  L = empty list
 4  while R is not empty
 5      remove some vertex u from R
 6      append u to L
 7      for each v in Adj[u]
 8          indeg[v] = indeg[v] - 1        // u is done, so v waits on one fewer
 9          if indeg[v] == 0
10              add v to R
11  if |L| < |V|
12      report "G has a cycle"
13  return L
```

Table 11.2 Kahn's algorithm on **Figure 11.2**, one removal per line. Nine vertices emitted from nine steps, so no cycle. Compare the result with **Figure 11.3**: the two orders share not one position in common, and both are correct.

Step	Emit	In-degrees that fall to 0	Ready set afterward
1	`shirt`	`tie`	watch, undershorts, socks, tie
2	`tie`	—	watch, undershorts, socks
3	`watch`	—	undershorts, socks
4	`undershorts`	`pants`	socks, pants
5	`pants`	`belt`	socks, belt
6	`belt`	`jacket`	socks, jacket
7	`jacket`	—	socks
8	`socks`	`shoes`	shoes
9	`shoes`	—	(empty)

Pseudocode 11.2 emits any vertex of in-degree 0. In-degrees: shirt 0, tie 1, jacket 2, belt 2, watch 0, undershorts 0, pants 1, shoes 3, socks 0. **Table 11.2** gives shirt, tie, watch, undershorts, pants, belt, jacket, socks, shoes — no position in common with the DFS order, and both are correct.

Book: Sections 11.3.9 and 11.3.10.

Kahn: Correctness, Cost, Choice

Correct: a vertex is appended only when its counter reaches 0, that is, after all its predecessors. **Complete:** if vertices are left over, each has a predecessor left over, and walking backward inside a finite set must repeat — a cycle.

So the test $|L| < |V|$ is an **exact** cycle detector; the stuck set is the diagnosis. Cost $\Theta(V + E)$ (with a heap for tidy output, $O(V \lg V + E)$).

Which to use? DFS: two extra lines if you already have a DFS, reports the offending edge. Kahn: no recursion, full control of the order, parallel waves, and it extends directly to scheduling.

Book: Sections 11.3.11–11.3.14.

Spanning Trees

A **spanning tree** of a connected graph is a subgraph that is a tree and contains **all** the vertices. **Theorem 11.3:** a tree on n vertices has exactly $n - 1$ edges; adding any edge creates exactly one cycle. **Figure 11.4**: 13 vertices, 12 edges.

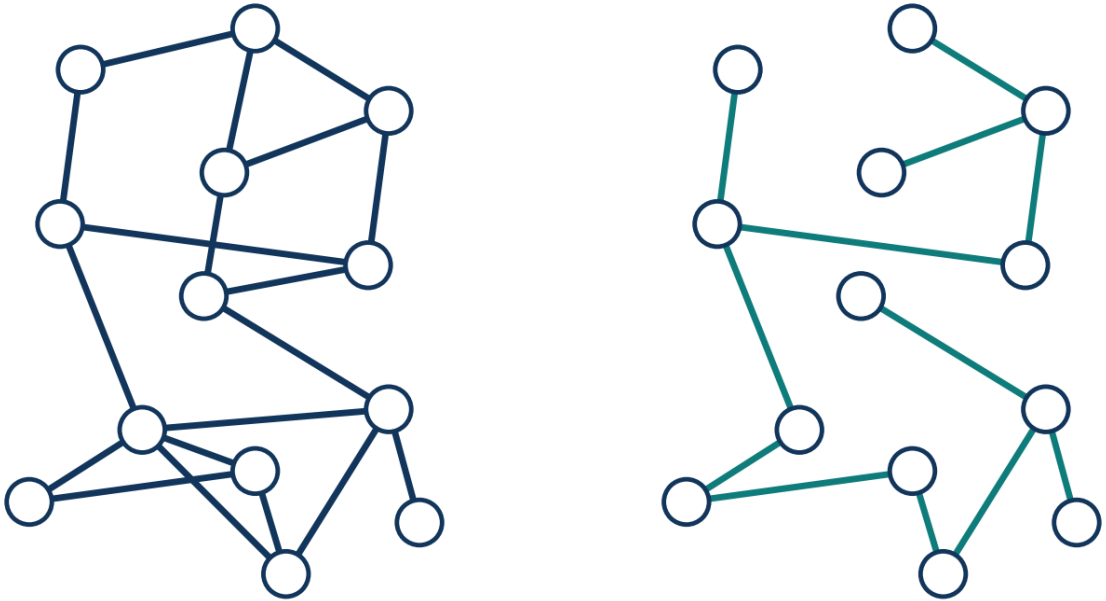


Figure 11.4 A spanning tree. **Left:** the graph G. **Right:** a spanning tree of G — every vertex of G is present, the graph is still connected, and no cycle remains. Count the edges on the right: twelve, for thirteen vertices.

Book: Sections 11.4.1 and 11.4.2 and Figure 11.4. Cayley: K_n has n^{n-2} spanning trees (Section 11.4.3).

The Minimum Spanning Tree Problem

A weight function $w: E \to \mathbb{R}$; the weight of a tree is Equation (11.1). A **minimum spanning tree** has the least weight among all spanning trees.

$$w(T) = \sum_{\{u,v\} \in T} w(u, v). \tag{11.1}$$

Nine villages and piped water: the answer must span, must be acyclic (a cycle contains a pipe you could drop), and must be cheapest. K_9 already has $9^7 = 4\,782\,969$ spanning trees — brute force is hopeless.

> **MST IS NOT SHORTEST PATHS.** The MST minimizes the **total**; it does not minimize the distance between any particular pair (Chapter 13). **Figure 11.5** is a first example.

Book: Sections 11.5.1 and 11.5.2.

A First Weighted Graph

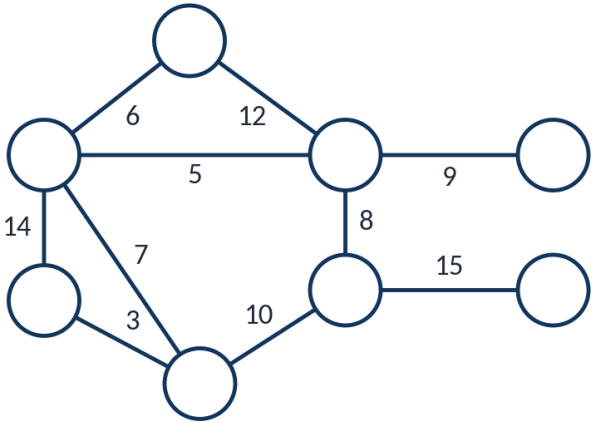


Figure 11.5 A weighted graph: an undirected connected graph with a weight on every edge. Some expensive edges are unavoidable (the 15 is the only way into its vertex); some cheap-looking ones are redundant. Distinguishing the two is the whole problem.

Book: Sections 11.5.3 and 11.5.4 and Figure 11.5. Taking the $|V| - 1$ cheapest edges blindly can close a cycle.

Optimal Substructure: The Split

Remove an edge (u, v) of an MST T (**Figure 11.6**): exactly two pieces remain, T_1 and T_2.

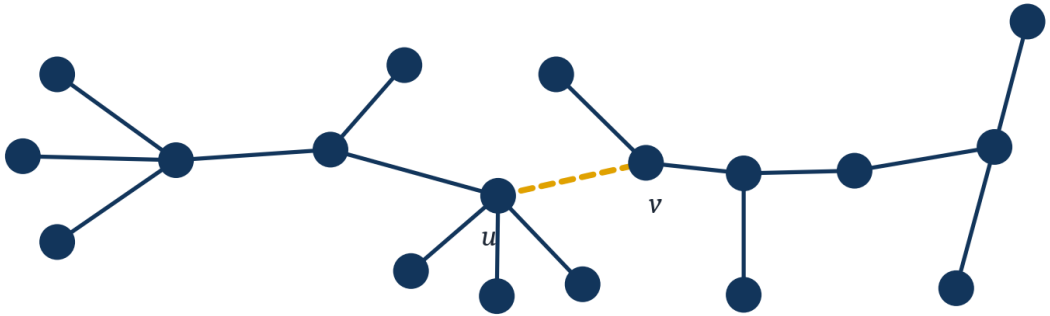


Figure 11.6 A minimum spanning tree T, with one of its edges (u, v) drawn as a dotted line. Removing that edge splits T into two pieces.

Book: Section 11.6.1 and Figure 11.6.

The Two Pieces, Named

Figure 11.7 names them; the weights add up as Equation (11.2).

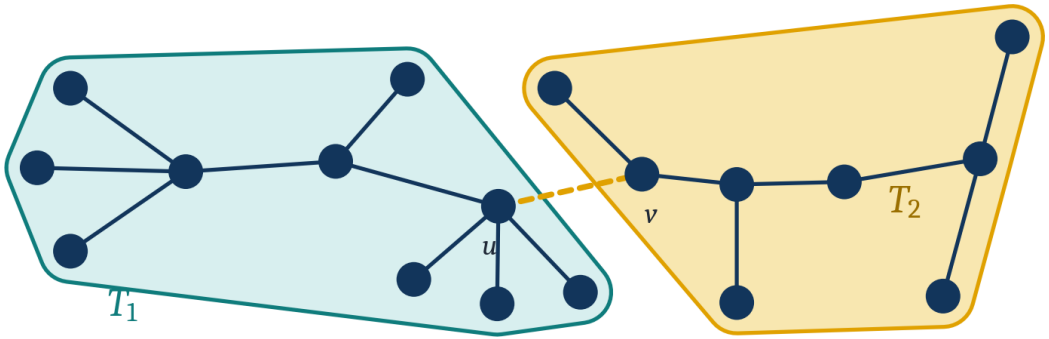


Figure 11.7 Figure 11.6 annotated: the two pieces left after removing the dotted edge (u, v) are named T_1 (left) and T_2 (right). The claim of Theorem 11.4 is that each of them is itself a minimum spanning tree, of the subgraph of G induced by its own vertices.

$$w(T) = w(u, v) + w(T_1) + w(T_2). \tag{11.2}$$

Book: Section 11.6.1 and Figure 11.7.

Theorem 11.4, by Cut and Paste

THEOREM 11.4. T_1 is a minimum spanning tree of the subgraph G_1 induced by its vertices; likewise T_2 of G_2.

Suppose not: some spanning tree $T_1{}'$ of G_1 satisfies Equation (11.3). Glue $T' = T_1{}' \cup \{(u, v)\} \cup T_2$: it spans, is connected, and has $|V| - 1$ edges, so it is a spanning tree.

$$w(T_1{}') < w(T_1). \tag{11.3}$$

Then $w(T') = w(u, v) + w(T_1{}') + w(T_2) < w(T)$, contradicting the minimality of T. ■

What it does not give: it assumes an MST is already in hand. It licenses greed; it does not say **which edge** to take first.

Book: Sections 11.6.2 and 11.6.3, Theorem 11.4.

The Cut Property

DEFINITIONS. A **cut** $(S, V - S)$ splits the vertices into two non-empty parts. An edge **crosses** it if exactly one endpoint is in S. A cut **respects** A if no edge of A crosses it. A minimum-weight crossing edge is a **light edge**.

THEOREM 11.5. For any cut $(S, V - S)$, a minimum-weight edge crossing it belongs to **some** minimum spanning tree. **Figure 11.8** draws the proof.

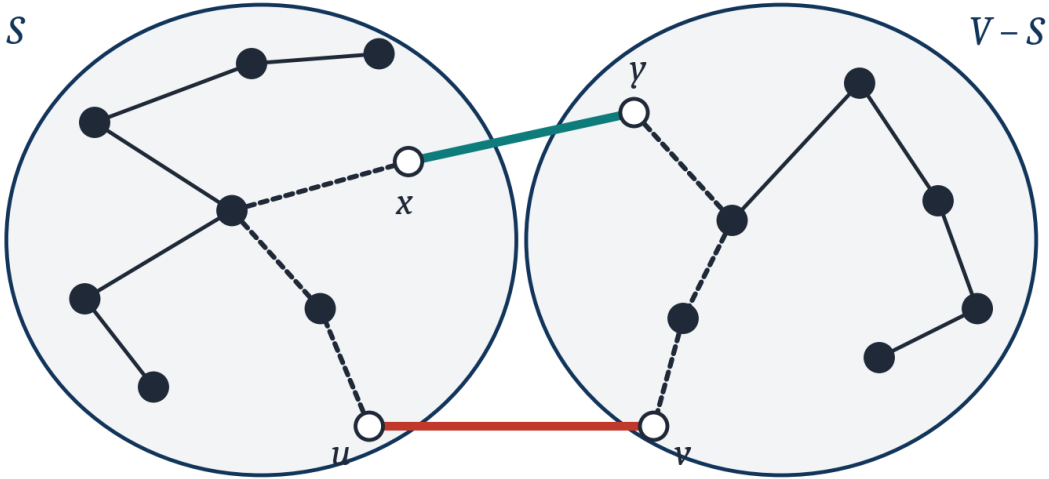


Figure 11.8 The exchange argument that proves Theorem 11.5. S and $V - S$ are the two blobs. The red edge (u, v) at the bottom is the light edge, assumed *not* to be in the tree. The dotted line is the unique path the tree provides from u to v; the teal edge (x, y) is the first edge of that path that crosses the cut. Swapping (x, y) out for (u, v) is the move the proof makes.

Book: Sections 11.7.1–11.7.3, Theorem 11.5, and Figure 11.8.

The Exchange, Step by Step

Let T be an MST without the light edge (u, v). The unique tree path from u to v starts in S and ends outside, so some edge (x, y) of it crosses the cut, and Equation (11.4) holds.

$$w(u, v) \leq w(x, y). \tag{11.4}$$

$T' = (T \setminus \{(x, y)\}) \cup \{(u, v)\}$ is a spanning tree with $w(T') \leq w(T)$, so it is an MST — and it contains (u, v). ■

Corollary 11.6: if A lies in some MST and the cut **respects** A, a light edge is **safe**: $A \cup \{(u, v)\}$ still lies in some MST (the swapped-out edge crosses, so it is not in A).

Worked check on the nine-vertex graph: $S = \{a\}$ gives $\{a, b\}$ (4); $S = \{g, h\}$ gives $\{f, g\}$ (2); $S = \{d\}$ gives $\{c, d\}$ (7) — all in both MSTs.

Book: Sections 11.7.3–11.7.6, Corollary 11.6.

The Generic MST Algorithm

Pseudocode 11.3 The generic minimum-spanning-tree algorithm: grow a set A of edges one safe edge at a time.

```
GENERIC-MST(G, w)
1  A = {}                                  // contains edges that belong to an MST
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A union {(u, v)}
5  return A
```

Pseudocode 11.4 The generic algorithm with "safe" replaced by Corollary 11.6: choose a cut that respects A and take its light edge.

```
MORE-SPECIFIC-MST(G, w)
1    A = {}                                // contains edges that belong to an MST
2    while A does not form a spanning tree
3.1      make a cut (S, V-S) of G that respects A
3.2      take the min-weight edge (u, v) connecting S to V-S
4        A = A union {(u, v)}
5    return A
```

Pseudocode 11.3 adds a **safe** edge each time; the invariant *"A is a subset of some MST"* holds initially, is maintained by the definition of safe, and at $|V| - 1$ edges gives an MST.

Pseudocode 11.4 leaves exactly one decision — **which cut**. Prim: S = the tree so far (vertex-driven, a priority queue). Kruskal: S = one component of the forest (edge-driven, sort once).

Book: Section 11.8.

Prim–Jarník

One growing cloud. For v outside the tree, $key[v]$ is the cheapest edge from v into the tree and $\pi[v]$ its other end; the minimum key is the light edge of the cut.

Pseudocode 11.5 The Prim–Jarník algorithm: $key[v]$ is the weight of the cheapest edge from v into the growing tree, and $\pi[v]$ is its other end.

```
MST-PRIM(G, w, r)
01  Q = G.V                          // Q: the vertices not yet in T
02  for each u in Q
03      key[u] = INFINITY
04  key[r] = 0
05  pi[r] = NIL
06  while Q is not empty
07      u = EXTRACT-MIN(Q)            // u becomes part of T
08      for each v in Adj[u]
09          if v in Q and w(u, v) < key[v]
10              pi[v] = u
11              key[v] = w(u, v)
```

PRIM IS NOT DIJKSTRA. **Pseudocode 11.5** tests $w(u,v) < key[v]$ — one edge, no accumulated length. Writing `key[u] + w(u,v)` gives a shortest-path tree.

Book: Sections 11.9.1–11.9.4; the priority queue offers `BuildPQ`, `ExtractMin` and `DecreaseKey`.

Prim, Step 1: The Root

Figure 11.9 is also the graph for every trace: nine vertices, fourteen edges. Extract a; now $Q = \{b{:}\,4/a,\ h{:}\,8/a,\ \text{rest}\ \infty\}$.

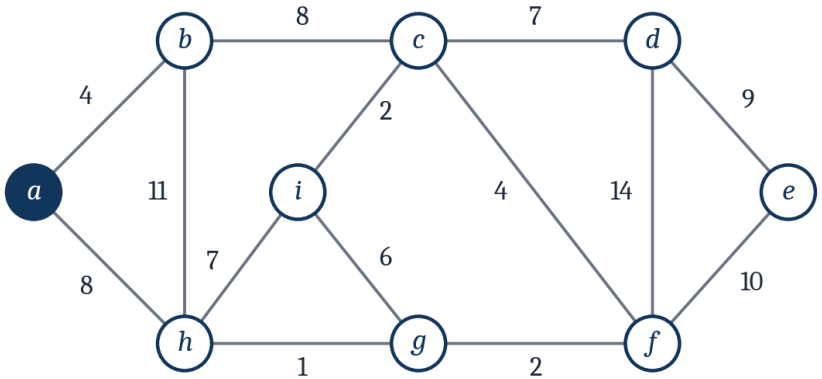


Figure 11.9 Step 1 of the Prim trace: the tree is just the root a. This is also the graph itself — nine vertices, fourteen edges, weights 1 to 14 — and it is the graph used for every trace in the rest of the chapter.

Book: Section 11.9.5 and Figure 11.9.

Prim, Step 2

Figure 11.10: Extract b: 4, commit (a, b), total 4. $key[c] \leftarrow 8/b$; h stays $8/a$ since $11 \not< 8$.

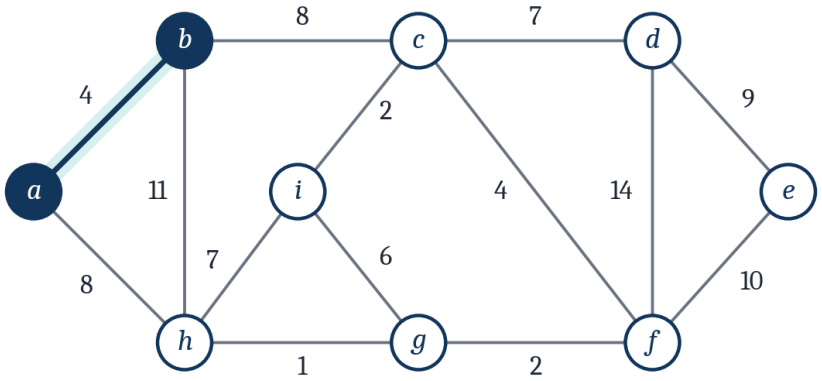


Figure 11.10 Step 2: the light edge for the cut $S = \{a\}$ is (a, b) of weight 4, so b joins the tree.

Book: Section 11.9.5 and Figure 11.10.

Prim, Step 3: A Tie

Figure 11.11: c and h tie at 8; the trace takes c — commit (b, c), total 12. Now i: $2/c$, f: $4/c$, d: $7/c$.

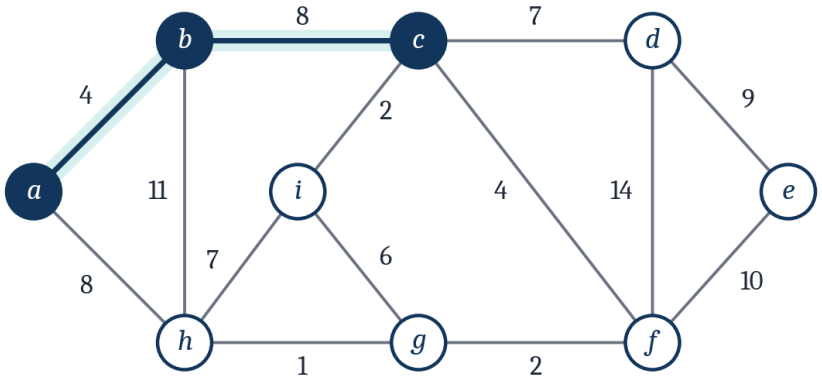


Figure 11.11 Step 3: with $S = \{a, b\}$ the crossing edges are $(b, c) = 8$, $(a, h) = 8$ and $(b, h) = 11$. Two of them tie for lightest; the trace takes (b, c).

Book: Section 11.9.5 and Figure 11.11.

Prim, Step 4

Figure 11.12: Extract i: 2, commit (c, i), total 14. $g \leftarrow 6/i$ and h **improves** from $8/a$ to $7/i$.

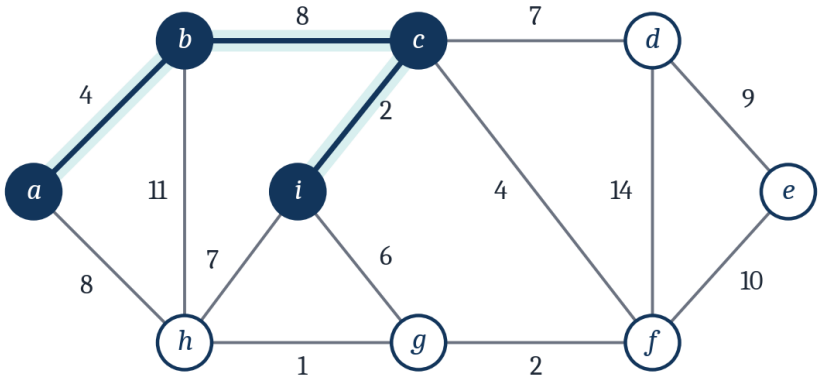


Figure 11.12 Step 4: (c, i) of weight 2 is the cheapest way out of $\{a, b, c\}$, so i joins. This is also the step at which h's key is *improved*, from 8 to 7.

Book: Section 11.9.5 and Figure 11.12.

Prim, Step 5

Figure 11.13: Extract f: 4, commit (c, f), total 18. g improves to $2/f$; $e \leftarrow 10/f$.

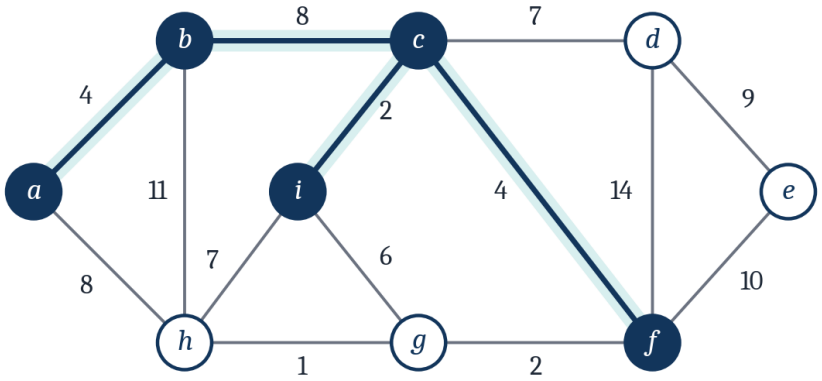


Figure 11.13 Step 5: (c, f) of weight 4 joins f. The tree is now $\{a, b, c, i, f\}$ and g's key drops from 6 to 2.

Book: Section 11.9.5 and Figure 11.13.

Prim, Step 6

Figure 11.14: Extract g: 2, commit (f, g), total 20. h improves again, to $1/g$.

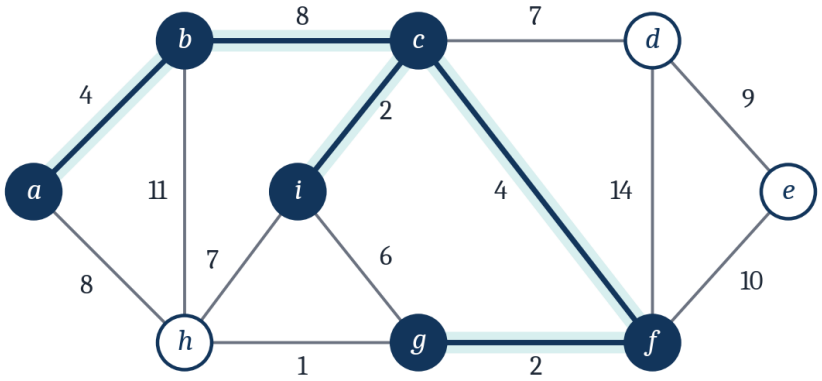


Figure 11.14 Step 6: (f, g) of weight 2 joins g. Vertex h has now had its key lowered three times — 8, then 7, then 1 — which is exactly what `DecreaseKey` is for.

Book: Section 11.9.5 and Figure 11.14.

Prim, Step 7

Figure 11.15: Extract h: 1, commit (g, h), total 21 — the lightest edge of the graph, taken only at step 7.

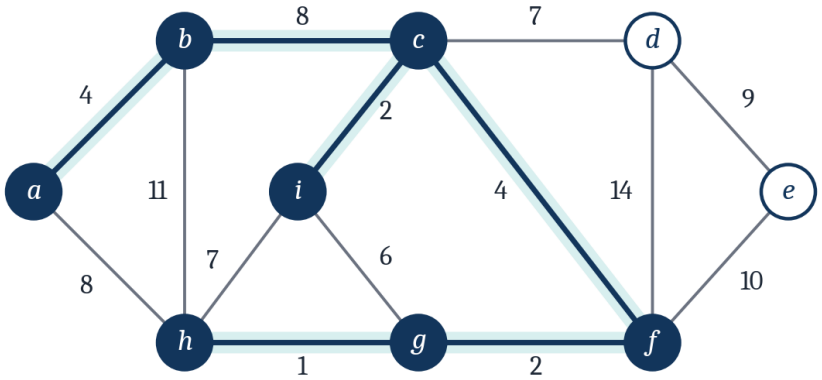


Figure 11.15 Step 7: the cheapest edge in the whole graph, (g, h) of weight 1, is only taken now — at the seventh step. Prim takes edges in the order the growing cloud reaches them, not in order of weight.

Book: Section 11.9.5 and Figure 11.15.

Prim, Step 8

Figure 11.16: Extract d: 7, commit (c, d), total 28; e improves from $10/f$ to $9/d$.

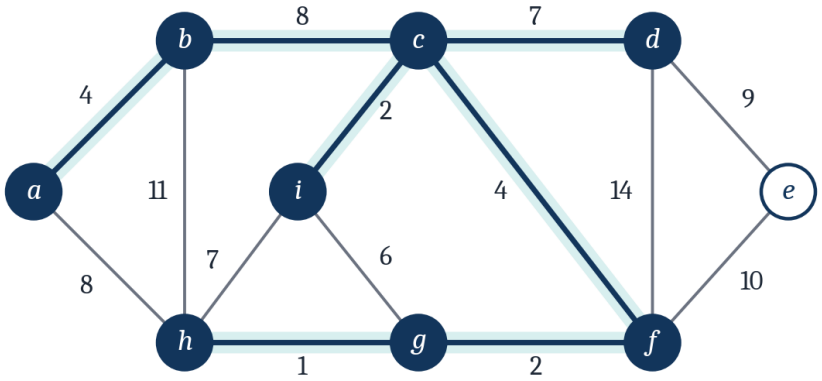


Figure 11.16 Step 8: (c, d) of weight 7 joins d, and e's key improves from 10 to 9.

Book: Section 11.9.5 and Figure 11.16.

Prim, Step 9: The Tree

Figure 11.17: Extract e: 9, commit (d, e), total **37**. Q is empty.

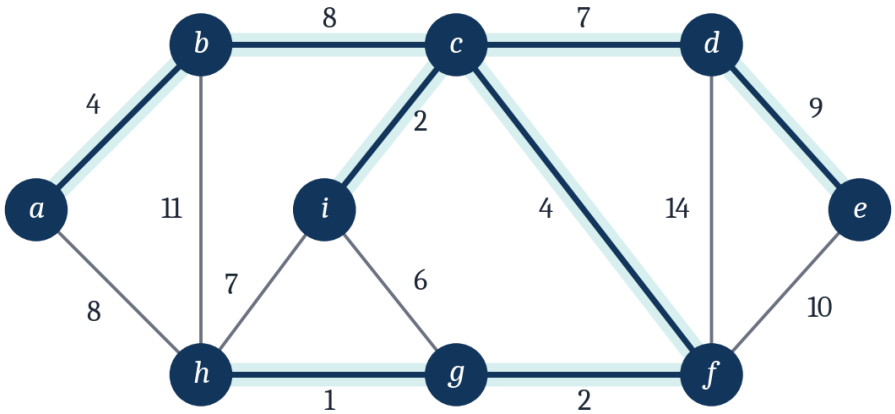


Figure 11.17 Step 9 — the finished picture. All nine vertices are black and the eight highlighted edges form the minimum spanning tree: (a, b), (b, c), (c, i), (c, f), (f, g), (g, h), (c, d), (d, e), of total weight 37.

Book: Section 11.9.5 and Figure 11.17.

Worked Example 2: Prim on One Page

Table 11.3 The complete Prim–Jarník run of **Figure 11.9** through **Figure 11.17**, with the priority queue printed after every step as `vertex:key/π`. Nine extractions, eight edges, total weight 37. The sequence of committed weights is 4, 8, 2, 4, 2, 1, 7, 9 — not sorted, because Prim does not sort.

Step	Extract	*key*	Edge committed	Total	*Q* Afterward
1	a	0	— (root)	0	b:4/a, h:8/a, rest ∞
2	b	4	(a, b)	4	c:8/b, h:8/a, rest ∞
3	c	8	(b, c)	12	i:2/c, f:4/c, d:7/c, h:8/a, e:∞, g:∞
4	i	2	(c, i)	14	f:4/c, g:6/i, d:7/c, h:7/i, e:∞
5	f	4	(c, f)	18	g:2/f, d:7/c, h:7/i, e:10/f
6	g	2	(f, g)	20	h:1/g, d:7/c, e:10/f
7	h	1	(g, h)	21	d:7/c, e:10/f
8	d	7	(c, d)	28	e:9/d
9	e	9	(d, e)	37	(empty)

Table 11.3: committed weights 4, 8, 2, 4, 2, 1, 7, 9 — out of order, because Prim looks only at the boundary of its own cloud.

Book: Section 11.9.5.

Why Prim Is Correct, and Its Cost

Correct: the committed edges form a tree on $S = V \setminus Q$; the cut $(S, V - S)$ respects them; the extracted vertex's edge is light. Corollary 11.6 does the rest.

Cost: $|V|$ extractions and $O(E)$ key changes, Equation (11.5); with a binary heap, Equation (11.6).

$$\text{Time} = |V| \cdot T(\text{ExtractMin}) + O(E) \cdot T(\text{ModifyKey}). \tag{11.5}$$

$$\text{Time} = O(V \lg V) + O(E \lg V) = O((V + E) \lg V) = O(E \lg V), \tag{11.6}$$

Table 11.4 Prim's running time with three priority queues. The Fibonacci-heap row is the reason that structure exists: it makes `DecreaseKey` free in the amortized sense, and `DecreaseKey` is the operation Prim performs most.

Q	T(ExtractMin)	T(DecreaseKey)	Total
unsorted array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(V \lg V + E)$

Table 11.4: array for dense graphs, binary heap for sparse ones, Fibonacci heap for the best bound. `std::priority_queue` has no `DecreaseKey`: the lazy version pushes a new copy and skips stale ones, $O(E \lg E)$.

Book: Sections 11.9.6–11.9.10.

Kruskal and Disjoint Sets

Sort all edges; accept each one whose endpoints lie in **different trees** of the forest. The question "same tree?" is asked $|E|$ times: **disjoint sets** — `MakeSet`, `Union`, `FindSet`.

Attempt 1, **lists** (**Figure 11.18**): `FindSet` $O(1)$; `Union` re-points the smaller list. Each move at least doubles a vertex's set, so the total is Equation (11.7).

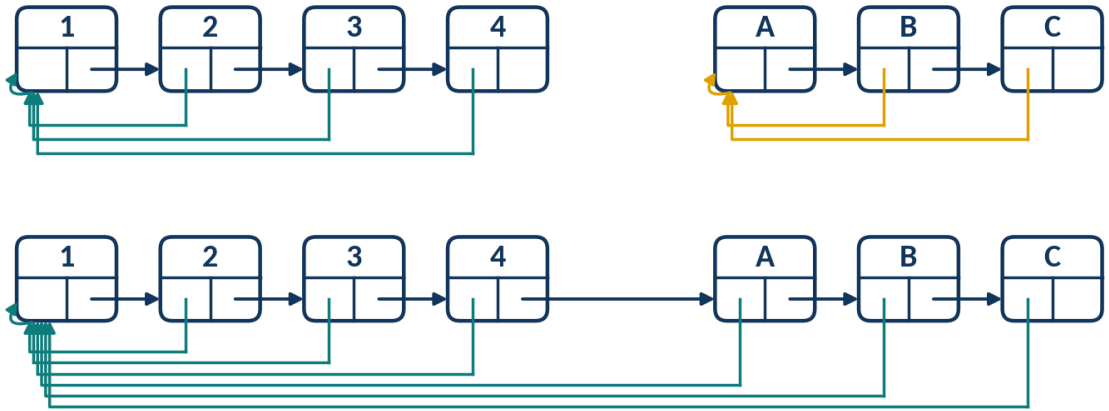


Figure 11.18 Disjoint sets as lists. **Top:** two sets, $\{1,2,3,4\}$ and $\{A, B, C\}$; within each, every element carries a back-pointer to the first element, which is the representative. **Bottom:** the result of uniting them — the shorter list has been appended and every one of its elements re-pointed at the new representative, 1.

$$\sum_{v \in V} t(v) \leq |V| \lg |V| = O(V \lg V). \tag{11.7}$$

Book: Sections 11.10.1–11.10.3 and Figure 11.18.

Trees, Union by Rank, Path Compression

Attempt 2, **trees**: `parent[]`, the root is the representative. Naive unions can build a path of depth $n - 1$.

Pseudocode 11.6 Making a singleton set, and union by rank: the root of smaller rank is hung under the root of larger rank.

```
MAKE-SET(x)
1  parent[x] = x
2  rank[x] = 0

UNION(x, y)
1  rx = FIND-SET(x)
2  ry = FIND-SET(y)
3  if rx == ry
4      return                              // already together
5  if rank[rx] < rank[ry]
6      swap rx and ry
7  parent[ry] = rx
8  if rank[rx] == rank[ry]
9      rank[rx] = rank[rx] + 1
```

Pseudocode 11.7 Find with path compression: on the way back from the root, every node on the path is re-pointed at the root.

```
FIND-SET(x)
1  if parent[x] != x
2      parent[x] = FIND-SET(parent[x])
3  return parent[x]
```

Pseudocode 11.6: hang the smaller rank under the larger. **Lemma 11.7:** rank k means at least 2^k elements, so height $\leq \lg n$. **Pseudocode 11.7**: re-point every node on the path at the root.

Book: Sections 11.10.4–11.10.6, Lemma 11.7.

Worked Example 3: Both Tricks at Work

Table 11.5 A disjoint-set forest with union by rank and path compression, printed after every operation. Changed entries are bold. Two things to notice: a root's rank climbs only when two equal ranks meet, and in this sequence every one of the seven real unions joins equal ranks — to 1 in rows 2–5, to 2 in rows 6–7, to 3 in row 8; and the "no-op" `Union(1,5)` still changes `parent[5]` from 4 to 0, because the two finds it performs compress their paths on the way. After the last sweep of finds, every element points straight at the root and the ranks — which are only ever *upper* bounds on the height — are left stale, which is harmless.

Operation	`parent` (elements 0…7)	`rank`	Sets
after `MakeSet`	0 1 2 3 4 5 6 7	0 0 0 0 0 0 0 0	8
`Union(0,1)`	0 **0** 2 3 4 5 6 7	**1** 0 0 0 0 0 0 0	7
`Union(2,3)`	0 0 2 **2** 4 5 6 7	1 0 **1** 0 0 0 0 0	6
`Union(4,5)`	0 0 2 2 4 **4** 6 7	1 0 1 0 **1** 0 0 0	5
`Union(6,7)`	0 0 2 2 4 4 6 **6**	1 0 1 0 1 0 **1** 0	4
`Union(0,2)`	0 0 **0** 2 4 4 6 6	**2** 0 1 0 1 0 1 0	3
`Union(4,6)`	0 0 0 2 4 4 **4** 6	2 0 1 0 **2** 0 1 0	2
`Union(0,4)`	0 0 0 2 **0** 4 4 6	**3** 0 1 0 2 0 1 0	1
`Union(1,5)` (no-op)	0 0 0 2 0 **0** 4 6	3 0 1 0 2 0 1 0	1

Table 11.5 (Continued) *A disjoint-set forest with union by rank and path compression, printed after every operation. Changed entries are bold. Two things to notice: a root's rank climbs only when two equal ranks meet, and in this sequence every one of the seven real unions joins equal ranks — to 1 in rows 2–5, to 2 in rows 6–7, to 3 in row 8; and the "no-op" Union(1,5) still changes parent[5] from 4 to 0, because the two finds it performs compress their paths on the way. After the last sweep of finds, every element points straight at the root and the ranks — which are only ever upper bounds on the height — are left stale, which is harmless.*

Operation	`parent` (elements 0…7)	`rank`	Sets
after `find(0)`…`find(7)`	**0 0 0 0 0 0 0 0**	3 0 1 0 2 0 1 0	1

Table 11.5: ranks climb only when equal ranks meet; the "no-op" `Union(1,5)` still compresses; after the finds every element points at 0 and the ranks are left stale — harmless.

Book: Section 11.10.6; the program is Section 11.10.8.

Almost Constant

With both tricks, m operations on n elements cost Equation (11.8), where α is the inverse Ackermann function (Tarjan; the proof is amortized analysis, Chapter 14).

$$O(m \cdot \alpha(n)), \tag{11.8}$$

Table 11.6 How the inverse Ackermann function grows. The number of atoms in the observable universe is about 2^{266}, so for every disjoint-set problem that will ever be run on physical hardware, $\alpha(n) \leq 4$.

n	$\alpha(n)$
$0 \leq n \leq 2$	0
$n = 3$	1
$4 \leq n \leq 7$	2
$8 \leq n \leq 2047$	3
$2048 \leq n \leq A_4(1)$, a number far larger than 2^{2048}	4

Table 11.6: $\alpha(n) \leq 4$ for every n ever run on hardware. And $\Omega(\alpha(n))$ is unavoidable for any structure (Fredman and Saks).

Book: Section 11.10.7.

Kruskal's Algorithm

Pseudocode 11.8 Kruskal's algorithm: scan the edges from lightest to heaviest and keep each one that joins two different trees of the forest.

```
MST-KRUSKAL(G, w)
01  A = {}
02  for each vertex v in G.V
03      MAKE-SET(v)
04  sort the edges of G.E into non-decreasing order by weight w
05  for each edge (u, v) in G.E, taken in non-decreasing order by weight
06      if FIND-SET(u) != FIND-SET(v)
07          A = A union {(u, v)}
08          UNION(u, v)
09  return A
```

Pseudocode 11.8 on the nine-vertex graph, sorted: $(g,h)1$, $(c,i)2$, $(f,g)2$, $(a,b)4$, $(c,f)4$, $(g,i)6$, $(c,d)7$, $(h,i)7$, $(a,h)8$, $(b,c)8$, $(d,e)9$, $(e,f)10$, $(b,h)11$, $(d,f)14$.

The next seven pages take the fourteen edges two at a time; an arrow marks the edge under consideration.

Book: Sections 11.10.9 and 11.10.10.

Kruskal, Edges 1 and 2

Figure 11.19: (g, h) 1: accept (total 1). (c, i) 2: accept (total 3). Kruskal starts with the globally cheapest edge.

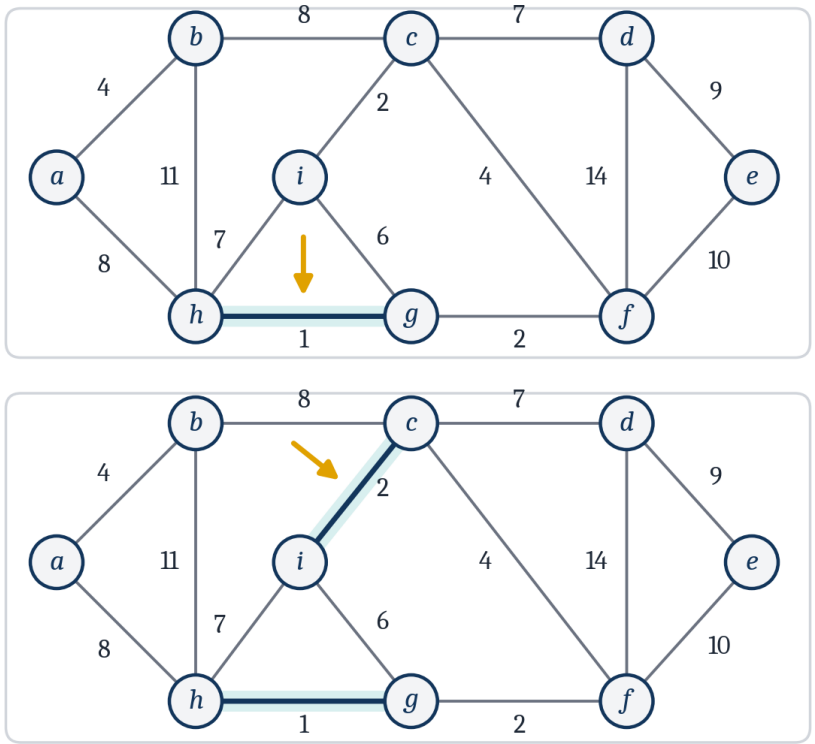


Figure 11.19 The first two Kruskal steps. **Top:** (g, h) of weight 1 — the cheapest edge in the graph — is accepted. **Bottom:** (c, i) of weight 2 is accepted. Unlike Prim, Kruskal starts with the globally cheapest edge, wherever it happens to be.

Book: Section 11.10.10 and Figure 11.19.

Kruskal, Edges 3 and 4

Figure 11.20: (f, g) 2: accept, $\{f, g, h\}$ (total 5). (a, b) 4: accept (total 9). Four trees grow at once.

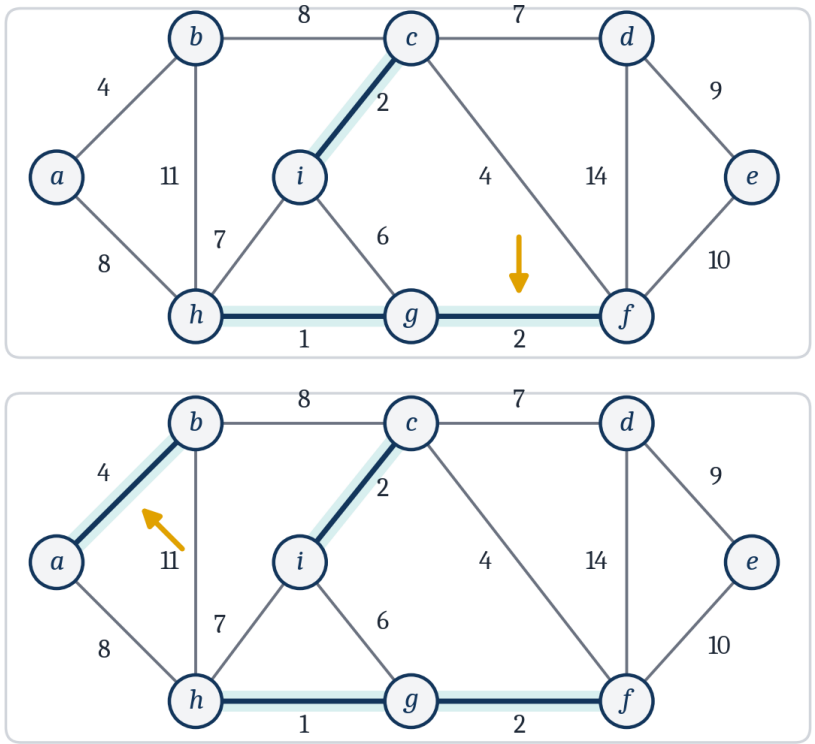


Figure 11.20 Top: (f, g) of weight 2 accepted, merging $\{f\}$ with $\{g, h\}$. **Bottom:** (a, b) of weight 4 accepted. Notice the forest now has four separate trees growing at once — something Prim never does.

Book: Section 11.10.10 and Figure 11.20.

Kruskal, Edges 5 and 6

Figure 11.21: (c, f) 4: accept, $\{c, f, g, h, i\}$ (total 13). (g, i) 6: `Find` gives c for both — **reject**; it would close $g - f - c - i$.

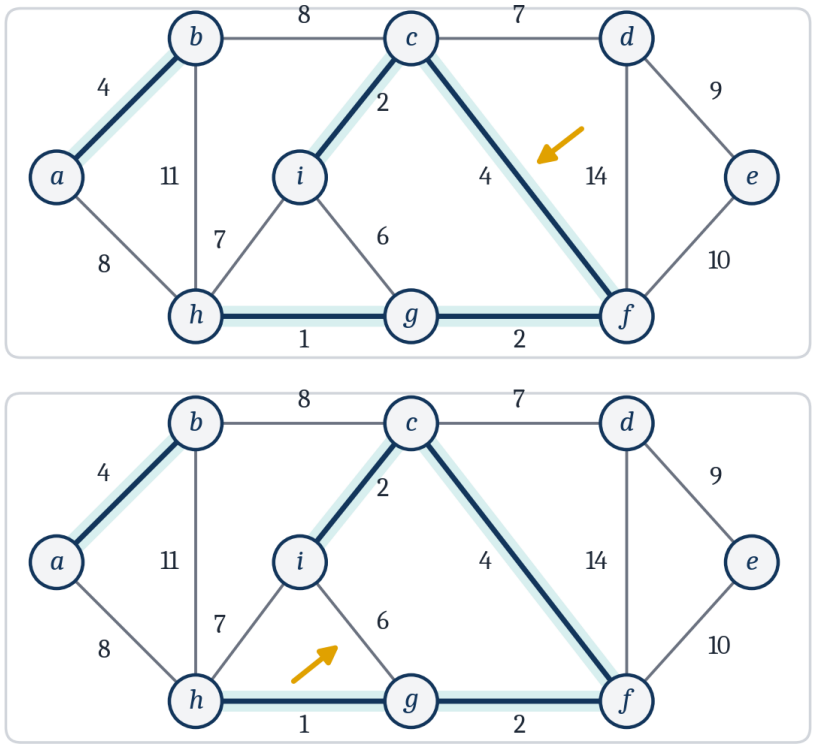


Figure 11.21 Top: (c, f) of weight 4 accepted, merging two trees into $\{c, f, g, h, i\}$. **Bottom:** the first rejection — (g, i) of weight 6 is *not* highlighted, because g and i are already in one tree and the edge would close a cycle.

Book: Section 11.10.10 and Figure 11.21.

Kruskal, Edges 7 and 8

Figure 11.22: (c, d) 7: accept (total 20). (h, i) 7: reject. Equal weights, opposite fates — the partition decides.

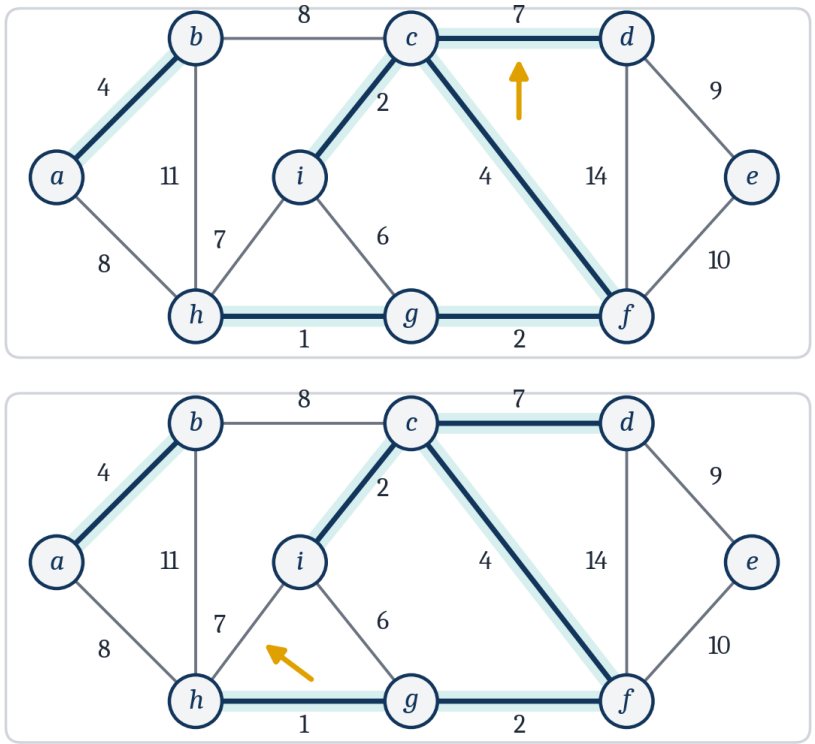


Figure 11.22 Top: (c, d) of weight 7 accepted. **Bottom:** (h, i), also of weight 7, rejected. Two edges of equal weight, opposite fates — the decision has nothing to do with the weight and everything to do with the partition.

Book: Section 11.10.10 and Figure 11.22.

Kruskal, Edges 9 and 10

Figure 11.23: (a, h) 8: accept (total 28). (b, c) 8: reject. **Here Kruskal parts company with Prim**, which took (b, c).

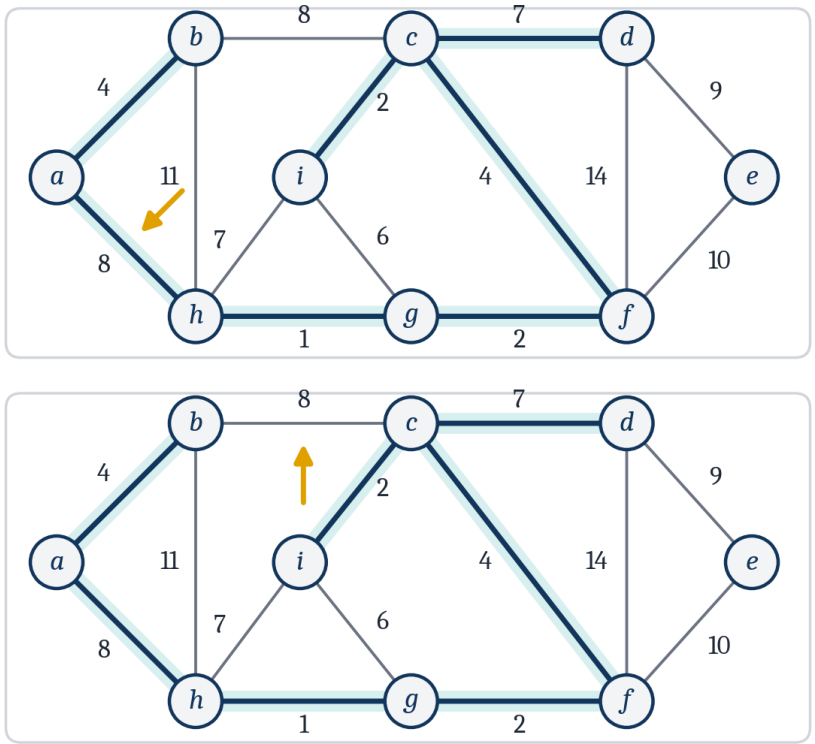


Figure 11.23 Top: (a, h) of weight 8 accepted. **Bottom:** (b, c) of weight 8 rejected. **This is the step at which Kruskal's answer diverges from Prim's:** Prim took (b, c) and never considered (a, h); Kruskal does the opposite. Both trees weigh 37 (see §11.12).

Book: Section 11.10.10 and Figure 11.23.

Kruskal, Edges 11 and 12

Figure 11.24: (d, e) 9: accept — the eighth edge, total **37**. (e, f) 10: reject.

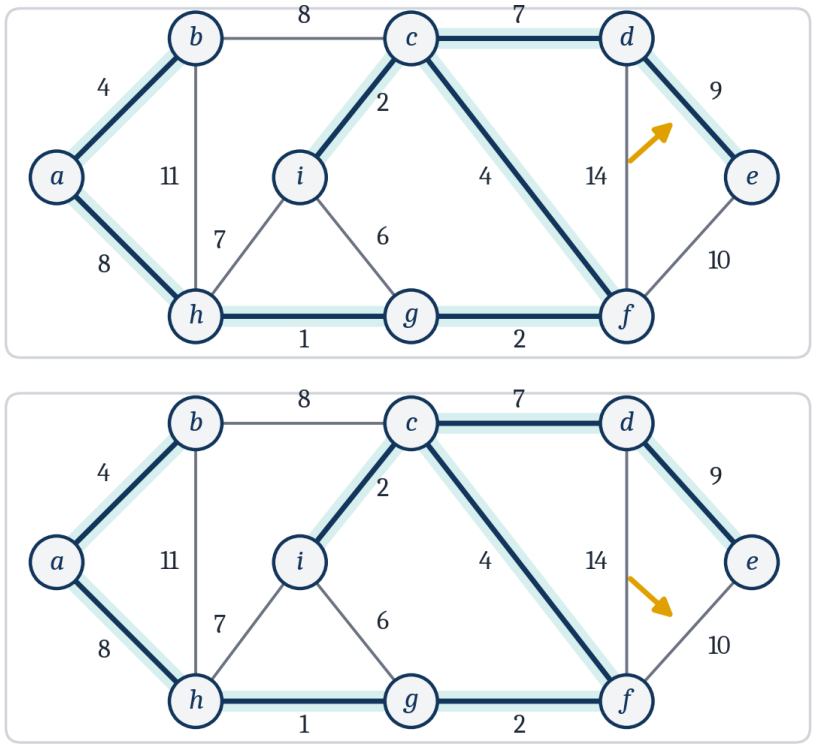


Figure 11.24 Top: (d, e) of weight 9 accepted — the eighth and last edge, bringing the isolated vertex e into the tree. **Bottom:** (e, f) of weight 10 rejected.

Book: Section 11.10.10 and Figure 11.24.

Kruskal, Edges 13 and 14

Figure 11.25: (b, h) 11 and (d, f) 14: reject. A practical implementation stops at $|V| - 1$ accepted edges.

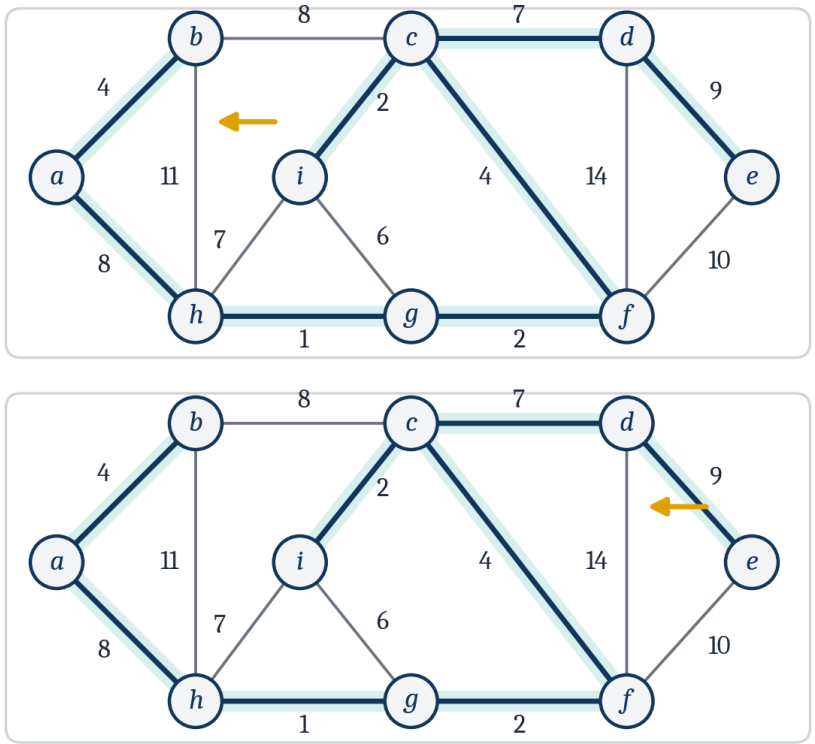


Figure 11.25 The last two panels: (b, h) of weight 11 and (d, f) of weight 14 are both rejected, and the highlighted eight edges are the finished minimum spanning tree of total weight 37 — the same weight as Prim's answer in **Figure 11.17**, by a different set of edges.

Book: Section 11.10.10 and Figure 11.25.

Worked Example 4: Kruskal on One Page

Table 11.7 The complete Kruskal run of **Figure 11.19** through **Figure 11.25**. Fourteen edges examined, eight accepted, six rejected, total weight 37. The `Find` column shows the representatives the disjoint-set forest actually returned; note how c becomes the representative of the growing component and stays so.

#	Edge	w	`Find` of each end	Decision	Total	Forest after
1	(g, h)	1	g, h	accept	1	a,b,c,d,e,f,**gh**,i
2	(c, i)	2	c, i	accept	3	a,b,**ci**,d,e,f,gh
3	(f, g)	2	f, g	accept	5	a,b,ci,d,e,**fgh**
4	(a, b)	4	a, b	accept	9	**ab**,ci,d,e,fgh
5	(c, f)	4	c, g	accept	13	ab,**cfghi**,d,e
6	(g, i)	6	c, c	**reject**	13	ab,cfghi,d,e
7	(c, d)	7	c, d	accept	20	ab,**cdfghi**,e
8	(h, i)	7	c, c	**reject**	20	ab,cdfghi,e

Table 11.7 (Continued) *The complete Kruskal run of Figure 11.19 through Figure 11.25. Fourteen edges examined, eight accepted, six rejected, total weight 37. The Find column shows the representatives the disjoint-set forest actually returned; note how c becomes the representative of the growing component and stays so.*

#	Edge	w	`Find` of each end	Decision	Total	Forest after
9	(a, h)	8	a, c	accept	28	**abcdfghi**,e
10	(b, c)	8	c, c	**reject**	28	abcdfghi,e
11	(d, e)	9	c, e	accept	37	**abcdefghi**
12	(e, f)	10	c, c	**reject**	37	abcdefghi
13	(b, h)	11	c, c	**reject**	37	abcdefghi
14	(d, f)	14	c, c	**reject**	37	abcdefghi

Table 11.7: fourteen edges, eight accepted ($|V| - 1$), six rejected, total 37.

Book: Section 11.10.10.

Why Kruskal Is Correct, and Its Cost

Correct: A is always a forest. For an accepted (u, v) take the cut S = u's component: it respects A, (u, v) crosses it, and any cheaper crossing edge would have been examined earlier and would now be inside a component. So (u, v) is light — **the sort earns its keep.**

Cost Equation (11.9), dominated by the sort; and by Equation (11.10), $\lg E = \Theta(\lg V)$, so Kruskal and heap-Prim cost the same asymptotically.

$$O(V) + \Theta(E \lg E) + O(E\,\alpha(V)) = O(E \lg E). \tag{11.9}$$

$$|V| - 1 \leq |E| \leq \binom{|V|}{2} < |V|^2. \tag{11.10}$$

Book: Sections 11.10.11 and 11.10.12.

The Kruskal Program

Listing 11.1 prints the partition before every decision; every line matches **Table 11.7**, and it ends with one component.

Listing 11.1 Kruskal's algorithm with a proper disjoint-set forest, tracing the partition before every decision. Compile with `g++ -std=c++17 -O2 -o kruskal kruskal.cpp`.

```cpp
// Kruskal's algorithm with a disjoint-set forest (union by rank + path
// compression), on the weighted graph of Section 11.9.5. Every edge considered is
// reported, accepted or rejected, with the components before the decision.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <map>
using namespace std;

class DisjointSets {
public:
    explicit DisjointSets(int n) : parent(n), rank_(n, 0) {
        for (int i = 0; i < n; ++i) parent[i] = i;
    }
    int findSet(int x) {                              // with path compression
        if (parent[x] != x) parent[x] = findSet(parent[x]);
        return parent[x];
    }
    bool unite(int x, int y) {                        // union by rank
        int rx = findSet(x), ry = findSet(y);
        if (rx == ry) return false;
        if (rank_[rx] < rank_[ry]) swap(rx, ry);
        parent[ry] = rx;
        if (rank_[rx] == rank_[ry]) ++rank_[rx];
        return true;
    }
```

```
private:
    vector<int> parent, rank_;
};

struct Edge {
    int u, v, w;
    bool operator<(const Edge& o) const { return w < o.w; }   // by non-decreasing weight
};
```

Listing 11.1 (Continued) *Kruskal's algorithm with a proper disjoint-set forest, tracing the partition before every decision. Compile with g++ -std=c++17 -O2 -o kruskal kruskal.cpp.*

```
// Prints the current partition as a set of blocks, so the trace shows why an
// edge is accepted or rejected.
string partitionOf(DisjointSets& D, const vector<string>& name) {
    map<int, string> blocks;
    for (int v = 0; v < (int)name.size(); ++v) blocks[D.findSet(v)] += name[v];
    string s = "{";
    bool first = true;
    for (auto& kv : blocks) { if (!first) s += ","; s += kv.second; first = false; }
    return s + "}";
}

int main() {
    vector<string> name = { "a","b","c","d","e","f","g","h","i" };
    enum Vertex { Va, Vb, Vc, Vd, Ve, Vf, Vg, Vh, Vi };
    vector<Edge> edges = {
        {Va,Vb,4},  {Va,Vh,8},  {Vb,Vc,8},  {Vb,Vh,11}, {Vc,Vd,7},
        {Vc,Vf,4},  {Vc,Vi,2},  {Vd,Ve,9},  {Vd,Vf,14}, {Ve,Vf,10},
        {Vf,Vg,2},  {Vg,Vh,1},  {Vg,Vi,6},  {Vh,Vi,7}
    };
    const int n = (int)name.size();

    // Line 04: sort the edges of E by non-decreasing weight.
    // stable_sort keeps equal weights in the order they were written above,
    // so the run is reproducible and can be compared with the figures.
    stable_sort(edges.begin(), edges.end());

    cout << "edges in sorted order:\n ";
    for (const Edge& e : edges) cout << " (" << name[e.u] << name[e.v] << ")" << e.w;
```

```
    cout << "\n\n";

    DisjointSets D(n);
    vector<Edge> mst;                                   // the forest being built
    int total = 0, step = 0;
    for (const Edge& e : edges) {
        cout << "forest " << partitionOf(D, name) << "\n";
        int ru = D.findSet(e.u), rv = D.findSet(e.v);
        bool accept = (ru != rv);
        cout << "  edge " << setw(2) << ++step << ": (" << name[e.u] << "," << name[e.v]
             << ") w=" << setw(2) << e.w
             << "  Find(" << name[e.u] << ")=" << name[ru]
             << " Find(" << name[e.v] << ")=" << name[rv]
             << (accept ? "  DIFFERENT -> accept" : "  SAME -> reject");
        if (accept) { D.unite(e.u, e.v); mst.push_back(e); total += e.w;
                      cout << ", total=" << total; }
        cout << "\n";
    }
```

Listing 11.1 (Continued) *Kruskal's algorithm with a proper disjoint-set forest, tracing the partition before every decision. Compile with g++ -std=c++17 -O2 -o kruskal kruskal.cpp.*

```
    cout << "\nMST edges, in the order Kruskal accepted them:\n";
    for (const Edge& e : mst)
        cout << "  (" << name[e.u] << "," << name[e.v] << ") w=" << e.w << "\n";
    cout << "  edge count = " << mst.size() << "  (|V|-1 = " << n - 1 << ")\n";
    cout << "  total weight = " << total << "\n";
    cout << "  components at the end = "
         << [&]{ int c = 0; for (int v = 0; v < n; ++v) if (D.findSet(v) == v) ++c; return c; }()
         << "\n";
    return 0;
}
```

Book: Section 11.10.13.

Prim versus Kruskal

Same weight, 37; different trees — seven shared edges, Prim's (b, c) against Kruskal's (a, h), both of weight 8.

Measured: on 1000 random graphs they never disagree on the weight; sparse graph (200 000 vertices) Kruskal 26.3 ms against Prim 104.9 ms; dense graph (2000 vertices, 10^6 edges) Prim 39.9 ms against Kruskal 58.7 ms. **Table 11.8** is the decision table.

Table 11.8 Choosing between the two algorithms. The first two rows cover most real cases; the rest are the cases where the answer is not obvious.

If...	Use	Because
E is close to V (sparse: road networks, sparse meshes)	**Kruskal**	the sort is over few items and is cache-friendly
E is close to $\mid V \mid^2$ (dense: all-pairs distance matrices)	**Prim**, binary heap	it never sorts the many edges it will never use
$\mid E \mid = \Theta(V^2)$ *and* V is small	**Prim**, unsorted array	$O(V^2)$ with tiny constants, and trivial to write
the edges arrive already sorted, or the weights are small integers	**Kruskal**	drop the sort; the rest is $O(E\,\alpha(V))$, nearly linear
the graph is given as an adjacency matrix	**Prim**, array	scanning a matrix row *is* the `ExtractMin`
the graph does not fit in memory and arrives as a stream of edges	**Kruskal**	it is a single pass over sorted edges; external sorting is a solved problem
you must handle a **disconnected** graph and want a minimum spanning *forest*	**Kruskal**	it needs no change at all; it simply stops short of $\mid V \mid - 1$ edges
you need the tree rooted at a particular vertex, with parents	**Prim**	π is the rooted tree, for free
$\mid E \mid = \Theta(V \lg V)$ or denser and V is in the millions	**Prim**, Fibonacci heap	$O(V \lg V + E)$ is the best known bound

Book: Section 11.11.

Ties, and When the MST Is Unique

Theorem 11.8: distinct weights imply a unique MST. The converse is false (triangle with weights 1, 1, 5).

Criterion: T is the only MST iff deleting any tree edge makes the best spanning tree strictly heavier. **Table 11.9** runs it: only (a, h) can go at no cost — exactly two MSTs.

Table 11.9 The uniqueness test of §11.12.3 applied to the nine-vertex example graph, computed by calling Kruskal from **Listing 11.1** once per tree edge with that edge deleted. Exactly one row has the same weight, so the MST is not unique — and the alternative tree the seventh row produces is $\{(a,b)4, (b,c)8, (c,d)7, (c,f)4, (c,i)2, (d,e)9, (f,g)2, (g,h)1\}$, which is exactly Prim's answer from **Figure 11.17**.

Edge removed	Best weight without it	Conclusion
(g,h) 1	43	heavier — in every MST
(c,i) 2	41	heavier — in every MST
(f,g) 2	41	heavier — in every MST
(a,b) 4	41	heavier — in every MST
(c,f) 4	39	heavier — in every MST
(c,d) 7	40	heavier — in every MST
(a,h) 8	**37**	**same** — another MST exists without it
(d,e) 9	38	heavier — in every MST

FOR TEST SUITES. Never assert a particular edge set; assert $|V| - 1$ edges, connectivity, and the reference **weight**.

Book: Section 11.12, Theorem 11.8.

Profitina in Practice

The scan pipeline is a DAG: checked for cycles at start-up, ordered by **Kahn**, whose ready set is a natural parallel work queue.

Near-duplicate pages: an MST over page distances, cut above a threshold — single-linkage clustering.

Profitina Laundry, question 1: ten branches, eighteen candidate links; nine are laid, **7.9 km** against 22.7 km (65.2% saved); Prim agrees.

Question 2: nine stages topologically sorted; the critical path intake → sort → wash → dry → press → fold → handover takes **108 minutes**, so a faster dryer helps and a second scale does not.

Book: Section 11.13 and Section 11.13.3. The app advises; the user decides.

Common Mistakes

1. MST confused with a shortest-path tree. 2. `key[u] + w(u,v)` **in Prim. 3. Forgetting the** `v in Q` **test.**

4. Not sorting (or sorting descending) in Kruskal. 5. A plain array for the component test. 6. Only one of union by rank and path compression.

7. Asserting a particular MST. 8. Pushing onto the back in DFS topological sort. 9. Recording at discovery time. 10. Assuming a unique topological order.

11. The DAG test on an undirected graph. 12. Prim on a disconnected graph. 13. Overflowing 32-bit weight sums — use `long long`.

Book: Section 11.14.

Summary: Five Things to Remember

1. A directed graph is acyclic iff DFS finds no back edge; topological order = decreasing f, or Kahn's in-degrees — both $\Theta(V + E)$, rarely unique.

2. A spanning tree has $|V| - 1$ edges; the MST has optimal substructure (cut and paste).

3. The cut property: the light edge of any cut respecting A is safe — every MST algorithm is a way to choose the cut.

4. Prim grows one tree with key and π, $O(E \lg V)$; **Kruskal** sorts and merges with a disjoint-set forest, $O(E \lg E)$, set operations $O(\alpha(n))$.

5. Same weight always, same tree only when the MST is unique; distinct weights suffice, but are not necessary.

Book: Section 11.15.

Next Time

Chapter 12 — Evaluation and Discussion III. No new theory: one design project worked end to end, and twenty-three further worked problems on Chapters 9, 10 and 11.

Every queue printed, every timestamp stamped, every union recorded — and a *check yourself* variant after each problem.

Before the meeting: Exercises 11.1, 11.5, 11.6 and 11.16, and reread the cut property (Section 11.7).

Book: Section 11.15.1 and Chapter 12.

Chapter 1

Lecture 12 — Evaluation and Discussion III: Greedy Algorithms and Graphs

Design and Analysis of Algorithms (EF234405)

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*

Lecturer: Irfan Subakti — yifana@gmail.com

Book: Chapter 12, Evaluation and Discussion III: Greedy Algorithms and Graphs. Every page of this handout ends with a pointer into the book, which is the primary reference.

Today's Meeting

1. No new theory. Chapters 9, 10 and 11 consolidated by working problems, with every intermediate value written down (Section 12.1).

2. One design project, worked end to end — Rute Laundry: problem, model, algorithm choice, pseudocode, complexity, hand proof, implementation, evaluation, conclusion (Section 12.2).

3. Worked Problems A–F on greedy algorithms and G–L on searching a graph (Sections 12.3 and 12.4).

4. Worked Problems M–Q on DAGs and topological sort, and R–W on minimum spanning trees (Sections 12.5 and 12.6).

5. Quick self-check drills, Profitina in practice, the common mistakes and the summary (Sections 12.7–12.10). Every problem ends with a *check yourself* variant.

Book: Chapter 12, Sections 12.1 to 12.10.

Where We Stand

Chapter 9: greedy algorithms, the exchange argument, activity selection, the two knapsacks, Huffman codes, matroids.

Chapter 10: graph vocabulary and storage; BFS in rings of distance; DFS with timestamps and four edge classes; components, cycles, bipartiteness.

Chapter 11: DAGs and the DAG theorem; topological sort by finishing time and by Kahn; the cut property; Prim–Jarník and Kruskal with a disjoint-set forest.

> **HOW TO USE TODAY.** Read each problem, cover the solution, and try it for ten minutes before reading on. Naming the tool is half the work — the revision map of Section 12.1.2 names it for every idea.

Book: Sections 12.1.1–12.1.3.

The Yardstick Graph

Figure 12.1: its MST weighs **37**, and two different trees of that weight exist — a thirty-second check for any Prim or Kruskal implementation.

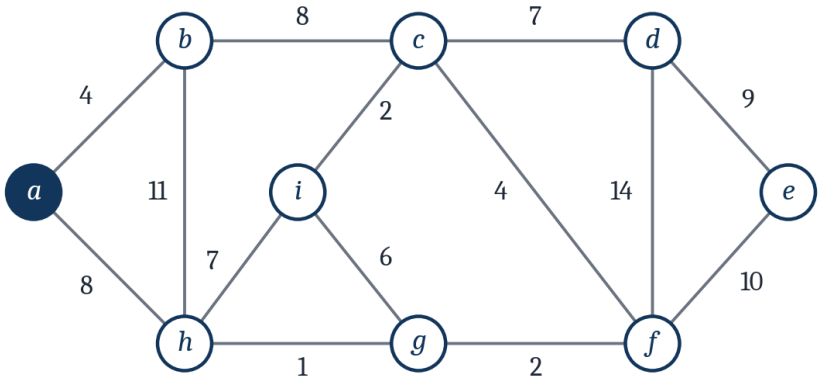


Figure 12.1 The nine-vertex, fourteen-edge weighted graph used for every minimum-spanning-tree trace in Chapter 11. Its minimum spanning tree weighs 37, and two different trees of that weight exist — which is why §11.12 exists.

Book: Section 12.1.2 and Figure 12.1.

A Design Project in Nine Stages

Table 12.1 The nine stages of a design project, the question each one answers, and the evidence each one produces. Stages 1 to 6 happen before any code exists.

Stage	The question it answers	What it must produce
1. Problem	*What exactly is being asked?*	The questions, in plain words, with the inputs and the outputs named
2. Model	*What mathematical object is this?*	The data expressed as a graph, a set, or a sequence, with every symbol defined
3. Algorithm choice	*Which algorithm, and why not the others?*	The algorithm named, and at least one alternative rejected with a reason
4. Pseudocode	*What will the program do, line by line?*	A numbered procedure precise enough to trace by hand
5. Complexity	*What will it cost, before we run it?*	The running time of every phase, predicted in advance
6. Hand proof	*Does the design work on the real data?*	The algorithm traced by hand on the actual input, with every step written down
7. Implementation	*Does it run, and is it the algorithm we designed?*	A complete program whose output can be compared with the hand trace
8. Evaluation	*How do we know it is right, and how do we know it is fast?*	Correctness evidence on many inputs, and measured times compared with the prediction
9. Conclusion	*What did we learn, and what comes next?*	The headline numbers, one real limitation, one specific next step

Table 12.1: **six of the nine stages happen before any code exists**, and stage 8 is not optional.

> **THE LOAD-BEARING TEST.** *If the input grew a thousandfold, would the naive method still work?* If yes, the algorithm is decoration. A sprite in a 5×5 maze fails the test; a pickup route over a city's roads (2^E subsets) passes it.

Book: Sections 12.2.1 and 12.2.2.

Stages 1 and 2: Problem and Model

Rute Laundry, outlet in Sukolilo. **Q1:** which zones get a two-hop pickup, and which cannot be reached? **Q2:** which roads form the cheapest shuttle network? **Q3:** in what order may the stages run, and how fast can a bag be turned round?

Model: an **undirected graph** of 14 zones and the 17 roads of **Table 12.2** (unweighted for Q1, weighted for Q2); a **directed graph** of 10 stages for Q3, which must be acyclic (**Figure 12.2**).

Table 12.2 The seventeen roads of the Rute Laundry instance, with distances in hundreds of meters. Fourteen zones; sixteen of the roads lie in one component and the seventeenth, Bulak–Kenjeran, is an island on its own.

Road	*w*	Road	*w*	Road	*w*
Sukolilo–Menur	8	Nginden–Manyar	6	Mulyosari–Kalijudan	5
Sukolilo–Keputih	6	Klampis–Manyar	3	Kalijudan–DukuhSetro	8
Menur–Gebang	7	Klampis–Mulyosari	13	Manyar–DukuhSetro	9
Menur–Semolowaru	9	Keputih–Kejawan	12	Kejawan–Mulyosari	7
Keputih–Gebang	5	Gebang–Klampis	10	Bulak–Kenjeran	4
Semolowaru–Nginden	4	Semolowaru–Klampis	11		

Book: Sections 12.2.3 and 12.2.4.

Acyclic or Not

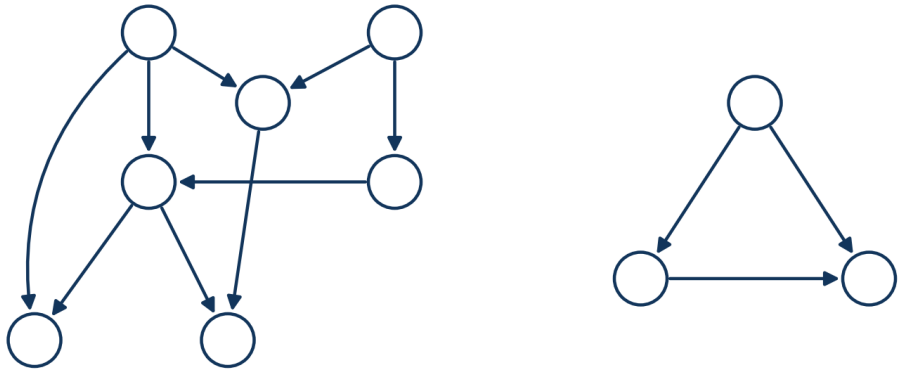


Figure 12.2 Two directed acyclic graphs. On the left, follow the arrows from anywhere and you run out of arrows. On the right, three arrows form a triangle that is still acyclic, because they do not run round it in one direction; reverse the bottom arrow and they would close a directed cycle, and a pipeline specification shaped like that could not be executed in any order at all.

Book: Section 12.2.4 and Figure 12.2.

Stage 3: Algorithms Chosen, and Rejected

Q1: BFS from the outlet, $\Theta(V + E)$. *Rejected:* Dijkstra (a priority queue for unit weights) and all-pairs Floyd–Warshall (one source, not V).

Q2: Kruskal over a disjoint-set forest — the data arrives as an **edge list**. *Rejected:* enumerating spanning trees, since by Cayley's formula Equation (12.1) K_{12} alone has $12^{10} = 61\,917\,364\,224$; and Prim, equally correct but needing an adjacency structure first.

$$\tau(K_n) = n^{n-2}, \tag{12.1}$$

Q3: Kahn, which detects a cyclic specification for free, and in the same pass the critical path by Equation (12.2): a stage finishes its own duration after its latest prerequisite.

$$ef[v] = dur[v] + \max(\{0\} \cup \{\, ef[u] : u \to v \,\}). \tag{12.2}$$

Book: Section 12.2.5.

The Backbone Is a Spanning Tree

Figure 12.3: every vertex present, still connected, no cycle, $|V| - 1$ edges — the shuttle backbone of Q2, chosen as light as possible.

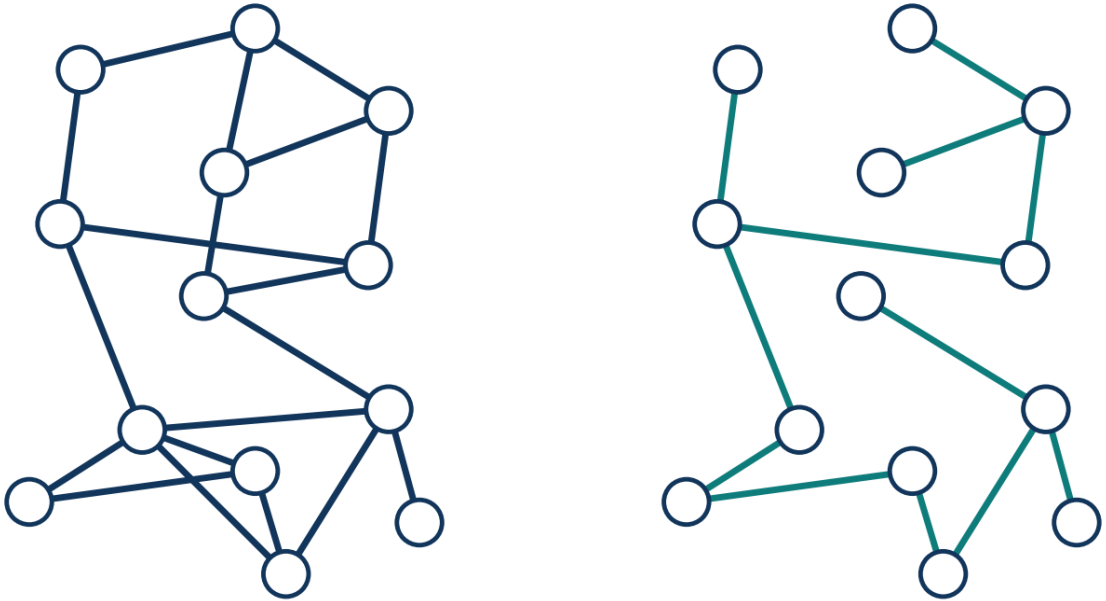


Figure 12.3 A spanning tree. **Left:** a graph. **Right:** a spanning tree of it — every vertex present, still connected, no cycle; thirteen vertices and twelve edges, which is the $|V| - 1$ rule of §11.4.2. The shuttle backbone of question 2 is exactly this object, chosen to be as light as possible.

Book: Section 12.2.5 and Figure 12.3.

Stages 4 and 5: Pseudocode and Cost

Pseudocode 12.1 The Rute Laundry design: breadth-first search and components for the service area, Kruskal over a disjoint-set forest for the backbone, and Kahn's algorithm with the critical path for the pipeline.

```
RUTE-LAUNDRY(zones, roads, stages, precedences, outlet)
 1  d, pi  = BFS(zones, roads, outlet)            // question 1
 2  comp   = COMPONENTS(zones, roads)             // reachability
 3  sort roads by weight, ascending               // question 2
 4  MAKE-SET(z) for every zone z in comp[outlet]
 5  T = empty set
 6  for each road (u, v, w) in weight order
 7      if FIND-SET(u) != FIND-SET(v)
 8          T = T + {(u, v)};  UNION(u, v)
 9  order, ef = KAHN(stages, precedences)         // question 3
10  if |order| < |stages| then report a cyclic specification
11  return d, pi, comp, T, order, max(ef)
```

Pseudocode 12.1 is the whole design in eleven lines. Predicted before any code: adjacency lists, BFS and components $\Theta(n+m)$; sorting $\Theta(m \lg m)$; m finds and $n-1$ unions $O(m\,\alpha(n))$; Kahn $\Theta(s+p)$ — in total Equation (12.3), and **the sort is the bottleneck**.

$$T(n, m, s, p) = \Theta(m \lg m + n + s + p), \tag{12.3}$$

Book: Sections 12.2.6 and 12.2.7.

Stage 6: Q1 Proved by Hand

Table 12.3 The breadth-first rings around the Sukolilo outlet, computed by hand. Six zones lie within the two-hop promise. Two zones are unreachable, which the program must report as a business fact rather than an error.

d	Zones
0	Sukolilo
1	Menur, Keputih
2	Gebang, Semolowaru, Kejawan
3	Nginden, Klampis, Mulyosari
4	Manyar, Kalijudan
5	DukuhSetro
∞	Bulak, Kenjeran

Table 12.3: BFS from Sukolilo, one ring at a time. At Keputih, Gebang is already gray — **first touch wins**. Twelve zones dequeued; Bulak and Kenjeran never touched.

Six zones lie within the two-hop promise; two are **unreachable** — a business fact (a second outlet), not an error.

Book: Section 12.2.8.

Stage 6: Q2 and Q3 Proved by Hand

Table 12.4 Kruskal on the Rute Laundry roads, by hand. Eleven edges accepted for twelve zones — exactly $|V| - 1$ — at a total of 69 hundred meters, or 6.9 km. Five roads are rejected because both their endpoints are already joined.

#	Road	w	Verdict	Running total
1	Klampis–Manyar	3	accept	3
2	Semolowaru–Nginden	4	accept	7
3	Keputih–Gebang	5	accept	12
4	Mulyosari–Kalijudan	5	accept	17
5	Sukolilo–Keputih	6	accept	23
6	Nginden–Manyar	6	accept	29
7	Menur–Gebang	7	accept	36
8	Kejawan–Mulyosari	7	accept	43
9	Sukolilo–Menur	8	**reject** — both ends already in one tree	43
10	Kalijudan–DukuhSetro	8	accept	51
11	Menur–Semolowaru	9	accept	60
12	Manyar–DukuhSetro	9	accept	69

#	Road	w	Verdict	Running total
13–16	Gebang–Klampis 10, Semolowaru–Klampis 11, Keputih–Kejawan 12, Klampis–Mulyosari 13		all reject	69

Table 12.4: eleven roads for twelve zones, **69** hundred meters; all sixteen would cost 123, so the backbone saves $54/123 = 43.9\%$. Row 9 is rejected at weight 8 while row 10, also 8, is accepted — the partition decides.

Q3: Kahn emits intake, weigh, sort, invoice, pretreat, wash, dry, press, fold, pack — ten of ten, so acyclic. By Equation (12.2): pack finishes at $\max(115,9) + 6 = 121$; critical path intake → weigh → sort → pretreat → wash → dry → press → fold → pack, **121 minutes**; invoice has $115 - 9 = 106$ minutes of slack.

Book: Section 12.2.8.

Stage 7: The Implementation

The complete program agrees with the hand traces line for line, including the two infinities, and it reports **two components**; the second, Bulak with Kenjeran, is why the promise must not be advertised city-wide (**Figure 12.4**).

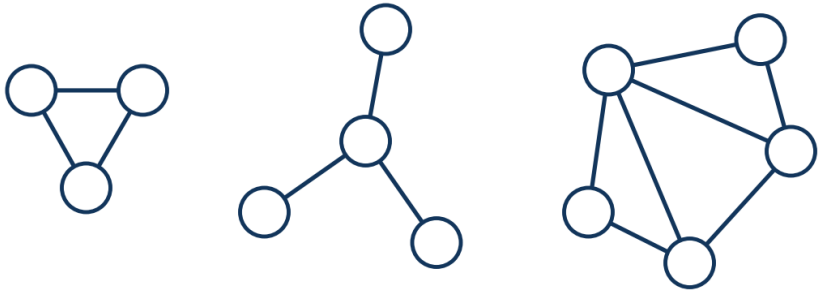


Figure 12.4 A graph with three connected components: a triangle, a star, and a five-vertex piece. No edge crosses between them. The Rute Laundry road network has two such pieces, and the second one — Bulak with Kenjeran — is the reason a two-hop promise must not be advertised for the whole city.

Book: Section 12.2.9 and Figure 12.4. `find` *compresses on the way home; a rejected edge still changes the forest;* `emitted 10 of 10` *is the acyclicity certificate.*

Stage 8: The Evaluation

```
=== Test 1: Kruskal and Prim must agree, on 2000 random graphs ===
mismatches: 0 out of 2000

=== Test 2: the BFS postcondition, on 2000 random graphs ===
edges spanning more than one BFS level: 0

=== Test 3: does it scale as the analysis says? ===
    zones     roads      BFS ms  Kruskal ms   BFS/(V+E) us Kruskal/(ElgE) us
     2000      8000       0.084       0.446         0.0047            0.0043
     4000     16000       0.158       0.921         0.0044            0.0041
     8000     32000       0.355       1.771         0.0049            0.0037
    16000     64000       0.742       3.731         0.0052            0.0037
    32000    128000       1.886       7.174         0.0065            0.0033
```

Test 1 is a cross-check (two wrong programs could agree). **Test 2** is a postcondition, the better evidence: for every edge, Equation (12.4). **Test 3** divides the time by the prediction: Kruskal flat, BFS rising 38% — cache misses, not a broken analysis.

$$|\, d[u] - d[v] \,| \leq 1. \tag{12.4}$$

Book: Section 12.2.10.

Stage 9: The Conclusion

> **CONCLUSION, FOUR SENTENCES.** Three graph problems on one data set, solved by BFS, Kruskal and Kahn at a predicted $\Theta(m \lg m)$. Six zones within two hops, two unreachable, an 11-road backbone of 6.9 km against 12.3 km (43.9% less), a 121-minute critical path with 106 minutes of slack. Limitation: hops are not travel times. Next step: Dijkstra (Chapter 13) and the same harness.

Five pitfalls: the decorative algorithm; the design written last; the missing evaluation; the unread measurement; the unchecked degenerate input.

Check yourself. Sukolilo–Menur drops to 5: it enters, Menur–Gebang (7) leaves, backbone $69 - 7 + 5 = 67$. Invoice takes 110 minutes: turnaround still 121, but now two critical paths and no slack.

Book: Sections 12.2.11 and 12.2.12.

Problem A: Making Change

Pseudocode 12.2 The greedy change-making rule: repeatedly hand over the largest coin that does not exceed the amount still owed.

```
GREEDY-CHANGE(coins, x)          // coins sorted so that c_1 > c_2 > ... > c_k = 1
1  count = 0
2  for i = 1 to k
3      while x >= c_i
4          x = x - c_i
5          count = count + 1
6  return count
```

Tokens $\{1,7,10\}$: **Pseudocode 12.2** pays 14 as $10+1+1+1+1$ — five tokens; $7+7$ needs two. The true minimum comes from the recurrence Equation (12.5).

$$M(0) = 0, \qquad M(x) = 1 + \min_{\substack{c \in \text{coins} \\ c \le x}} M(x-c) \quad (x \ge 1). \tag{12.5}$$

Greed loses at exactly 14, 15, 16, 24, 25, 26 up to 30. **Missing ingredient:** the greedy-choice property — the only optimal payment of 14 contains no 10.

For {1,5,10} **greed is optimal:** an optimal payment has at most four 1s and at most one 5, so fives and ones are worth at most 9, forcing $\lfloor x/10 \rfloor$ tens — exactly greed's choice. **Check yourself:** {1,5,8} fails first at 10 ($8 + 1 + 1$ against $5 + 5$).

Book: Section 12.3.2.

Problem A: The Program

Listing 12.1 prints greed beside the dynamic-programming optimum for every amount from 1 to 30, then audits {1,5,8} (47 losses in 1..100) and {1,5,10,25} (none).

Listing 12.1 Greedy change-making against the dynamic-programming optimum, for every amount from 1 to 30 with the tokens {1,7,10}, followed by two summary audits. Compile with `g++ -std=c++17 -O2 -Wall -o change change.cpp`.

```cpp
// Worked Problem A: making change with the coins {1, 7, 10}.
// For every amount from 1 to 30 it compares the greedy rule ("largest coin
// that fits") with the true optimum computed by dynamic programming, prints
// both coin multisets, and reports the smallest amount on which greed loses.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <climits>
using namespace std;

string showCoins(const vector<int>& c) {
    string s;
    for (size_t i = 0; i < c.size(); ++i) { if (i) s += "+"; s += to_string(c[i]); }
    return s;
}

vector<int> greedy(const vector<int>& coins, int x) {      // coins in increasing order
    vector<int> used;
    for (int k = (int)coins.size() - 1; k >= 0; --k)
        while (x >= coins[k]) { x -= coins[k]; used.push_back(coins[k]); }
    return used;
}

// best[x] = fewest coins that pay x; last[x] = one coin of an optimal payment
void optimal(const vector<int>& coins, int maxX, vector<int>& best, vector<int>& last) {
    best.assign(maxX + 1, INT_MAX); last.assign(maxX + 1, 0);
    best[0] = 0;
```

```
    for (int x = 1; x <= maxX; ++x)
        for (int c : coins)
            if (c <= x && best[x - c] != INT_MAX && best[x - c] + 1 < best[x]) {
                best[x] = best[x - c] + 1; last[x] = c;
            }
}

vector<int> rebuild(const vector<int>& last, int x) {
    vector<int> used;
    while (x > 0) { used.push_back(last[x]); x -= last[x]; }
    return used;
}
```

Listing 12.1 (Continued) *Greedy change-making against the dynamic-programming optimum, for every amount from 1 to 30 with the tokens \{1, 7, 10\}, followed by two summary audits. Compile with g++ -std=c++17 -O2 -Wall -o change change.cpp.*

```
void audit(const vector<int>& coins, int maxX, bool table) {
    vector<int> best, last;
    optimal(coins, maxX, best, last);
    cout << "coins {";
    for (size_t i = 0; i < coins.size(); ++i) cout << (i ? ", " : "") << coins[i];
    cout << "}, amounts 1.." << maxX << "\n";
    if (table) cout << "  x  greedy              #g  optimum             #o\n";
    int firstBad = 0, bad = 0;
    for (int x = 1; x <= maxX; ++x) {
        vector<int> g = greedy(coins, x), o = rebuild(last, x);
        bool lose = g.size() > o.size();
        if (lose) { ++bad; if (!firstBad) firstBad = x; }
        if (table)
            cout << setw(3) << x << "  " << left << setw(18) << showCoins(g) << right
                 << setw(4) << g.size() << "  " << left << setw(18) << showCoins(o)
                 << right << setw(4) << o.size() << (lose ? "   <- greed loses" : "") << "\n";
    }
    cout << "  amounts on which greed loses: " << bad << " of " << maxX;
    if (firstBad) cout << "; the smallest is " << firstBad << "\n";
    else cout << "; greed is optimal on every amount tested\n";
}

int main() {
    audit({1, 7, 10}, 30, true);
    cout << "\n";
    audit({1, 5, 8}, 100, false);                          // the check-yourself system
    audit({1, 5, 10, 25}, 100, false);                     // a canonical system
    return 0;
```

```
}
```

Book: Section 12.3.2, where the complete output is printed. Kozen and Zaks: a first failure lies below the sum of the two largest coins.

Problem B: Activity Selection

Table 12.5 The ten dryer jobs of Worked Problem B, listed in non-decreasing order of finish time.

i	1	2	3	4	5	6	7	8	9	10
s_i	0	2	1	5	4	8	7	10	9	12
f_i	3	4	6	7	9	11	12	13	14	16

Pseudocode 12.3 The earliest-finish-time rule for activity selection, as one sweep over jobs already sorted by finish time.

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)   // jobs sorted so that f_1 <= f_2 <= ... <= f_n
1  A = {a_1}
2  F = f_1
3  for i = 2 to n
4      if s_i >= F
5          A = A + {a_i}
6          F = f_i
7  return A
```

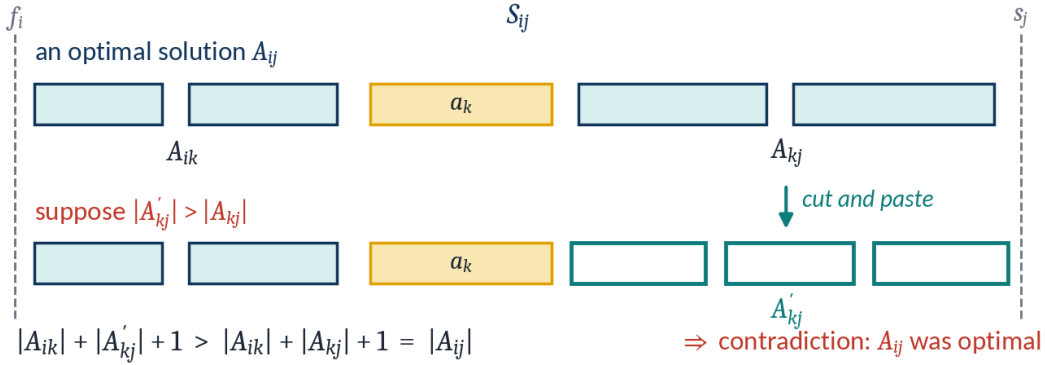
Table 12.5 by **Pseudocode 12.3**, F starting at 0: take a_1 ($F = 3$); skip a_2, a_3; take a_4 ($F = 7$); skip a_5; take a_6 ($F = 11$); skip a_7, a_8, a_9; take a_{10}. $\{a_1, a_4, a_6, a_{10}\}$**, size 4.** Earliest *start* gives $\{a_1, a_5, a_9\}$, size 3.

Book: Section 12.3.3.

Problem B: The Exchange

Take **any** maximum A_k with earliest finisher $a_j \neq a_m$ and form Equation (12.6); every other job starts at or after $f_j \geq f_m$, so it stays compatible and the size is unchanged. **Figure 12.5** draws its cut-and-paste twin.

$$A'_k = (A_k \setminus \{a_j\}) \cup \{a_m\}. \tag{12.6}$$



recurrence:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Figure 12.5 The activity-selection theorem completed by contradiction, together with the dynamic-programming recurrence for $c[i,j]$ that the optimal-substructure half of the argument produces. The substitution drawn at the top — a larger set A'_{kj} pasted in place of A_{kj} — is the cut-and-paste twin of the exchange in Equation (12.6): there one job is swapped for the greedy one and the size is unchanged; here a bigger piece would enlarge the whole, which optimality forbids.

Book: Section 12.3.3 and Figure 12.5. Write "a maximum-size subset": $\{a_2, a_4, a_6, a_{10}\}$ is also optimal.

Problem B: Check Yourself

Latest-start-first on the dryer jobs gives $\{a_2, a_4, a_6, a_{10}\}$ — optimal, by mirroring time. On the eleven activities of **Figure 12.6**, earliest finish gives $\{a_1, a_4, a_8, a_{11}\}$.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Figure 12.6 The eleven activities of §9.4.2 with their start and finish times, already sorted so that $f_1 \leq f_2 \leq \cdots \leq f_{11}$.

Book: Section 12.3.3 and Figure 12.6.

Problem C: Maximum Lateness

Six jobs, $p = \langle 3,2,1,4,3,2 \rangle$, $d = \langle 6,8,9,9,14,15 \rangle$; minimize Equation (12.7).

$$L_{\max} = \max_{1 \le i \le n} (C_i - d_i) \;\rightarrow\; \min. \tag{12.7}$$

Table 12.6 Earliest deadline first on the six jobs. Only J_4 is late, by one unit, so $L_{\max} = 1$.

Order	Job	p_i	C_i	d_i	$L_i = C_i - d_i$
1	J_1	3	3	6	-3
2	J_2	2	5	8	-3
3	J_3	1	6	9	-3
4	J_4	4	10	9	$+1$
5	J_5	3	13	14	-1
6	J_6	2	15	15	0

Table 12.6: earliest deadline first gives $L_{\max} = 1$; shortest processing time first gives 6. **Proof:** swapping an adjacent inverted pair ($d_i > d_j$) cannot raise $L_{\max}$, by Equation (12.8); bubble any optimal schedule into deadline order.

$$L'_i = t + p_i + p_j - d_i \ < \ t + p_i + p_j - d_j = L_j. \tag{12.8}$$

Book: Section 12.3.4. Brute force: optimum 1, attained by 20 of the 720 orders.

Problem D: A Huffman Code

Pseudocode 12.4 Huffman's algorithm: $n - 1$ merges of the two least frequent trees, each merge creating one internal node.

```
HUFFMAN(C)                  // C: the symbols, each with a frequency c.freq
1  n = |C|
2  Q = C                    // a min-priority queue keyed on freq
3  for i = 1 to n - 1
4      allocate a new node z
5      x = EXTRACT-MIN(Q)
6      y = EXTRACT-MIN(Q)
7      z.left = x;  z.right = y
8      z.freq = x.freq + y.freq
9      INSERT(Q, z)
10 return EXTRACT-MIN(Q)    // the root of the tree
```

Table 12.7 The six merges of Huffman's algorithm on the log frequencies. Six merges for seven symbols, as $|C| - 1$ requires. The final node's label, 100, is the total frequency — a useful check.

Merge	Queue before (sorted)	Removed	New node
1	1, 4, 8, 12, 15, 22, 38	Q 1, X 4	5
2	5, 8, 12, 15, 22, 38	5, P 8	13
3	12, 13, 15, 22, 38	F 12, 13	25
4	15, 22, 25, 38	D 15, W 22	37
5	25, 37, 38	25, 37	62
6	38, 62	R 38, 62	100

R 38, W 22, D 15, F 12, P 8, X 4, Q 1 (thousands). Fixed length: $\lceil \lg 7 \rceil = 3$ bits, 300 kbit. **Pseudocode 12.4** makes six merges, **Table 12.7**: 5, 13, 25, 37, 62, 100.

Book: Section 12.3.5.

Problem D: The Codebook and $B(T)$

Table 12.8 The Huffman codebook for the log, with the cost of each symbol. The two rarest symbols, Q and X, get the two longest codewords and differ only in the last bit — they are siblings, which is what the first merge made them.

Symbol	Freq (k)	Codeword	Length	Freq × length
R	38	`0`	1	38
W	22	`111`	3	66
D	15	`110`	3	45
F	12	`100`	3	36
P	8	`1011`	4	32
X	4	`10101`	5	20
Q	1	`10100`	5	5
total	**100**			**242**

Table 12.8; over the leaves, Equation (12.9) gives 242 kbit; over the internal nodes, Equation (12.10) gives the same. Saving $58/300 = 19.33\%$.

$$B(T) = \sum_{c \in C} c\,.\,freq \cdot d_T(c), \tag{12.9}$$

$$B(T) = \sum_{z \text{ internal}} z\,.\,freq = 5 + 13 + 25 + 37 + 62 + 100 = 242 \text{ kbit}. \tag{12.10}$$

Book: Section 12.3.5. Check yourself: `10110100111110` *decodes to PRFWD.*

The Same Process, Drawn

Figure 12.7 is the merge sequence on the six-symbol file of Section 9.7.

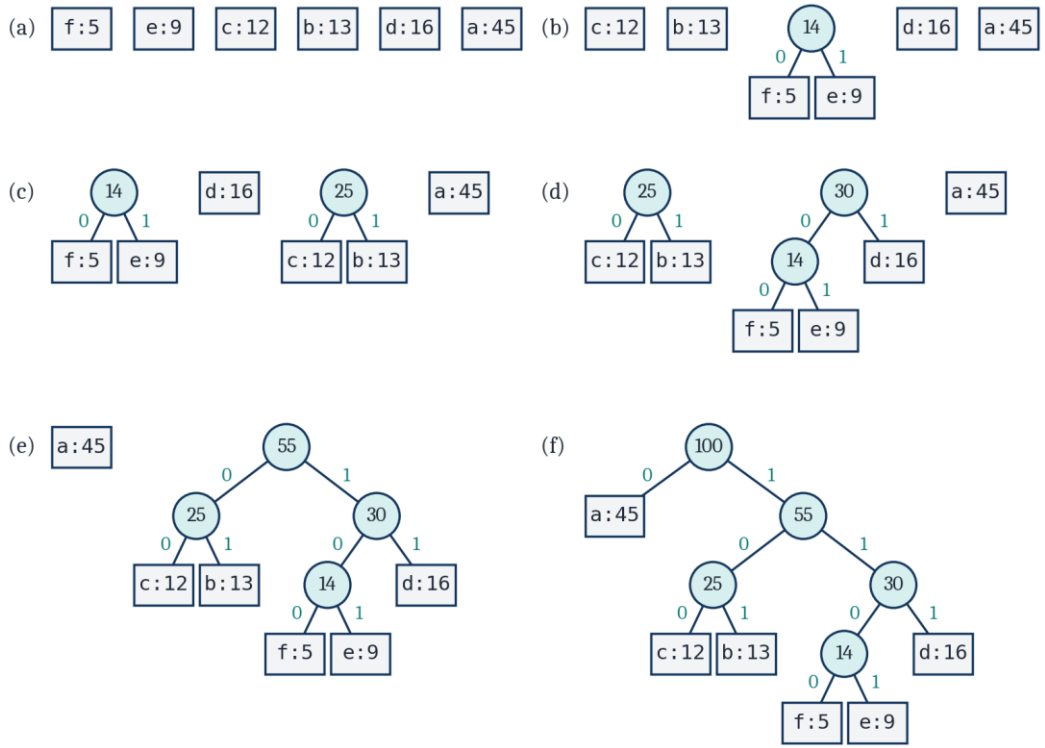


Figure 12.7 A six-panel trace of Huffman's algorithm on the six-symbol data of §9.7. Each panel shows the queue sorted into increasing frequency; at every step the two lowest-frequency trees are merged, leaves drawn as rectangles and internal nodes as circles.

Book: Section 12.3.5 and Figure 12.7.

Fixed Against Variable Length

Figure 12.8: the same calculation on six symbols — 300 000 against 224 000 bits. Huffman is optimal among prefix-free *per-symbol* codes; arithmetic coding can do better.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 12.8 The character-coding problem of §9.7: a 100 000-character file over the alphabet a–f, with a fixed-length code costing 300 000 bits and a variable-length code costing 224 000. The structure of the argument is exactly that of Worked Problem D; only the alphabet has changed.

Book: Section 12.3.5 and Figure 12.8.

Problem E: The Two Knapsacks

Table 12.9 The four items of Worked Problem E.

Item	1	2	3	4
value v_i	7	6	9	9
weight w_i	3	3	5	5

Table 12.9, 10 kg. Densities 2.333, 2.000, 1.800, 1.800. **Fractional:** items 1, 2 whole, 0.8 of item 3: **20.2**. **0-1 greedy:** items 1, 2 — **13**, 4 kg wasted.

0-1 optimum by Equation (12.11), filled into **Table 12.10**: **18**, items 3 and 4. Greed loses $5/18 = 27.8\%$ — the 0-1 problem lacks the greedy-choice property.

$$c[i,w] = \begin{cases} 0 & i = 0 \text{ or } w = 0, \\ c[i-1,w] & w_i > w, \\ \max(c[i-1,w],\ v_i + c[i-1,\ w-w_i]) & \text{otherwise.} \end{cases} \tag{12.11}$$

Table 12.10 The 0-1 knapsack table $c[i, w]$ for Worked Problem E. Fifty-five cells, each a single comparison, so the whole table costs $\Theta(nW) = \Theta(4 \times 10)$. The answer is the bottom-right corner, 18.

$w=$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
$i = 1$	0	0	0	7	7	7	7	7	7	7	7
$i = 2$	0	0	0	7	7	7	13	13	13	13	13
$i = 3$	0	0	0	7	7	9	13	13	16	16	16
$i = 4$	0	0	0	7	7	9	13	13	16	16	**18**

Book: Section 12.3.6. Check yourself (8 kg): fractional 16.6, 0-1 greedy 13, optimum 16.

Problem F: A Matroid at Work

Seven one-hour pressing orders, deadlines $\langle 2,4,1,3,2,4,6 \rangle$, refunds $\langle 55,45,40,35,30,20,10 \rangle$. Independence test Equation (12.12); hereditary and exchange hold (with $A = \{a_1, a_3\}$, $B = \{a_3, a_4, a_5\}$, a_4 can be added, a_5 cannot).

$$A \text{ is independent} \Leftrightarrow N_t(A) \leq t \quad \text{for } t = 0,1, \ldots, n. \tag{12.12}$$

Pseudocode 12.5 The matroid greedy algorithm for unit-time tasks with deadlines and penalties: keep the costliest orders on time whenever independence allows.

```
GREEDY-SCHEDULE(a, d, w, n)
1  sort the orders so that w_1 >= w_2 >= ... >= w_n
2  A = empty set
3  for i = 1 to n
4      if A + {a_i} satisfies N_t <= t for every t
5          A = A + {a_i}
6  schedule A in order of deadline, then the other orders in any order
7  return the schedule, and the refund sum of the orders not in A
```

Table 12.11 The matroid greedy algorithm on the seven orders. Bold counts break the rule $N_t \le t$. Two orders are rejected, and the refund paid is their sum, $30 + 20 = 50$.

Consider	N_1	N_2	N_3	N_4	N_5	N_6	N_7	Verdict	A Afterward
a_1 (55)	0	1	1	1	1	1	1	keep	$\{a_1\}$
a_2 (45)	0	1	1	2	2	2	2	keep	$\{a_1, a_2\}$
a_3 (40)	1	2	2	3	3	3	3	keep	$\{a_1, a_2, a_3\}$
a_4 (35)	1	2	3	4	4	4	4	keep	$\{a_1, a_2, a_3, a_4\}$
a_5 (30)	1	**3**	**4**	**5**	5	5	5	**reject**	unchanged
a_6 (20)	1	2	3	**5**	5	5	5	**reject**	unchanged
a_7 (10)	1	2	3	4	4	5	5	keep	$\{a_1, a_2, a_3, a_4, a_7\}$

Pseudocode 12.5, traced in **Table 12.11**: keep a_1–a_4 and a_7; reject a_5 and a_6. Press a_3, a_1, a_4, a_2, a_7; refund **50** of 235 — confirmed over all 128 subsets.

Book: Section 12.3.7.

The Instance of Section 9.8.7

Figure 12.9 is a different seven-task instance; it also pays 50. **Check yourself:** move a_5's deadline to 5 — only a_6 is late, refund 20.

	Task						
a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Figure 12.9 The instance of §9.8.7 for scheduling unit-time tasks with deadlines and penalties: seven tasks with deadlines d_i and penalties w_i. The greedy matroid algorithm schedules early a set of weight 230 and pays a penalty of 50.

Book: Section 12.3.7 and Figure 12.9; Problems B to E as one program are Section 12.3.8.

Problem G: BFS, Traced

Execute **Figure 12.10** exactly — including the edges to gray vertices, which must be examined and ignored.

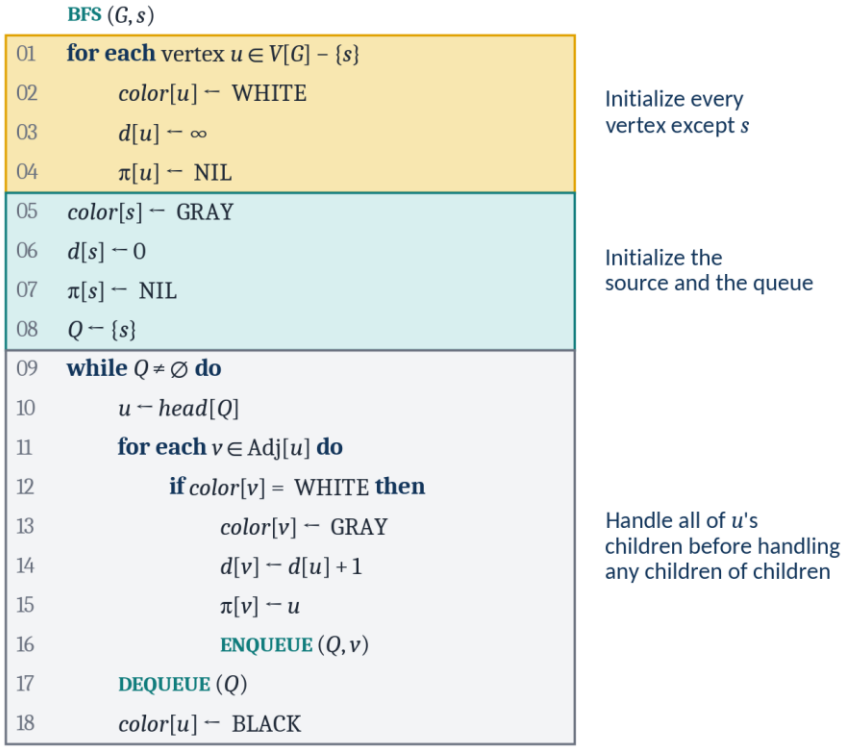


Figure 12.10 Breadth-first search. The amber block initializes every vertex but the source; the teal block initializes the source and the queue; the grey block is the search. The annotation on the right states the invariant in words — *handle all of u's children before any children of children* — and that sentence is the whole difference from depth-first search.

Book: Section 12.4.2 and Figure 12.10.

Problem G: The Trace on G_1

G_1: A–I, edges $AB, AC, BD, BE, CE, CF, DG, EG, EH, FH, GI, HI$; alphabetical lists. Degrees sum to 24, as Equation (12.13) requires.

$$\sum_{v \in V} \deg(v) = 2\,|E|, \tag{12.13}$$

Table 12.12 Breadth-first search of G_1 from A, one row per turn of the **while** loop. Nine vertices, nine steps: every vertex is enqueued once and dequeued once. The distances in the queue never decrease from left to right and never span more than one value — the queue invariant of §10.7.9.

Step	Dequeue	Discovered, with *d*	Skipped, and why	Queue afterward
1	A	B: 1, C: 1	—	B: 1 C: 1
2	B	D: 2, E: 2	A black	C: 1 D: 2 E: 2
3	C	F: 2	A black; E **gray**	D: 2 E: 2 F: 2
4	D	G: 3	B black	E: 2 F: 2 G: 3
5	E	H: 3	B, C black; G **gray**	F: 2 G: 3 H: 3
6	F	—	C black; H gray	G: 3 H: 3
7	G	I: 4	D, E black	H: 3 I: 4
8	H	—	E, F black; I gray	I: 4
9	I	—	G, H black	(empty)

Table 12.13 The output of breadth-first search on G_1: distance and predecessor. Reading π backward from any vertex and reversing gives a shortest path, and its length always equals that vertex's d.

v	A	B	C	D	E	F	G	H	I
$d[v]$	0	1	1	2	2	2	3	3	4
$\pi[v]$	–	A	A	B	B	C	D	E	G

Table 12.12 and **Table 12.13**: a shortest path to I is $A\ B\ D\ G\ I$, length 4; eight tree edges, $|V| - 1$. From E every vertex is within two edges.

Book: Section 12.4.2.

A Finished Search

Figure 12.11: every vertex black, the queue empty, $|V| - 1$ thick tree edges.

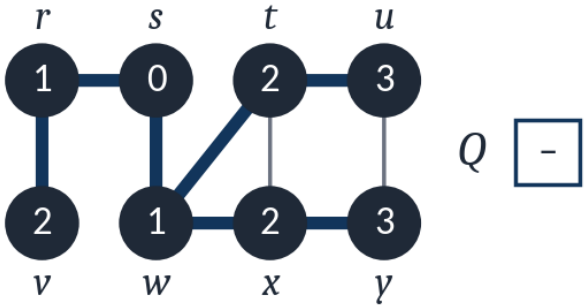


Figure 12.11 A finished breadth-first search: every vertex black, the queue empty, each vertex carrying its distance from s, and the seven thick edges forming the breadth-first tree over eight vertices.

Book: Section 12.4.2 and Figure 12.11. Distances do not depend on list order; trees do.

Problem H: DFS of a Digraph

G_2: $A \to B$, $A \to C$, $A \to D$, $B \to D$, $C \to D$, $C \to E$, $D \to A$, $E \to F$, $F \to E$, $G \to C$, $G \to F$. Apply **Figure 12.12** eleven times.

```
DFS (G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3  time ← 0
4  for each vertex u ∈ V[G]
5       do if color[u] = WHITE
6            then DFS-VISIT (u)

DFS-VISIT (u)
1  color[u] ← GRAY          ▷ White vertex u discovered.
2  d[u] ← time              ▷ Mark with discovery time.
3  time ← time + 1          ▷ Tick global time.
4  for each v ∈ Adj[u]      ▷ Explore all edges (u, v).
5       do if color[v] = WHITE
6            then DFS-VISIT (v)
7  color[u] ← BLACK         ▷ Blacken u; it is finished.
8  f[u] ← time              ▷ Mark with finishing time.
9  time ← time + 1          ▷ Tick global time.
```

Figure 12.12 Depth-first search. The outer procedure whitens every vertex, starts the clock at zero and calls DFS-Visit on every vertex still white — each such call rooting one tree of the forest. The inner procedure paints u gray, stamps $d[u]$, recurses into every white neighbor, then paints u black and stamps $f[u]$.

Book: Section 12.4.3 and Figure 12.12.

Problem H: Stamps and Classes

Table 12.14 Timestamps and predecessors after depth-first search of G_2. The clock stopped at $14 = 2|V|$, and the fourteen stamps are the integers 1 to 14, each used once — the first check to run on any DFS answer.

v	A	B	C	D	E	F	G
d/f	1/12	2/5	6/11	3/4	7/10	8/9	13/14
π	–	A	A	B	C	E	–

Table 12.15 Every arc of G_2 classified, with the test that decided it. Five tree, two back, one forward, three cross — eleven in all. Two back edges mean the graph is not acyclic.

Arc	Color of the far end when examined	Class
$A \to B$	white	tree
$A \to C$	white	tree
$A \to D$	black, $d[A] < d[D]$	**forward**
$B \to D$	white	tree
$C \to D$	black, $d[C] > d[D]$	**cross**
$C \to E$	white	tree
$D \to A$	**gray**	**back**
$E \to F$	white	tree

Table 12.15 (Continued) *Every arc of G2 classified, with the test that decided it. Five tree, two back, one forward, three cross — eleven in all. Two back edges mean the graph is not acyclic.*

Arc	Color of the far end when examined	Class
$F \to E$	**gray**	**back**
$G \to C$	black, $d[G] > d[C]$	cross
$G \to F$	black, $d[G] > d[F]$	cross

Table 12.14 and **Table 12.15**: **gray means back, white means tree, black splits into forward or cross by discovery time.** Two back edges — $A \to B \to D \to A$ and $E \rightleftarrows F$ — so not acyclic.

Book: Section 12.4.3.

Back, Forward, Cross — Seen

Figure 12.13: back edges curve up, forward edges point down past a level, cross edges go sideways.

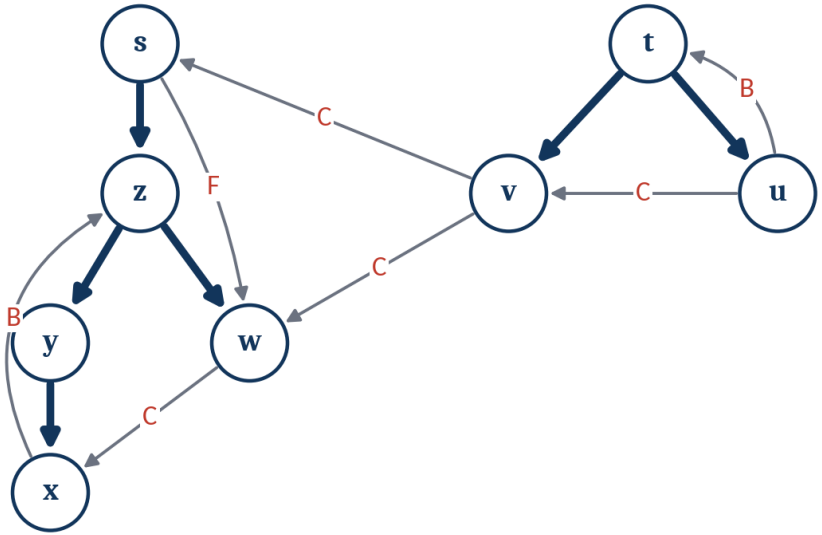


Figure 12.13 A larger example redrawn as its depth-first forest, tree edges thick and downward. Now the classification is geometric: **B** curves back up to an ancestor, **F** points down past an intermediate level to a descendant, and **C** goes sideways between branches or between trees.

Book: Section 12.4.3 and Figure 12.13.

The Parenthesis Structure

G_2: $(A\ (B\ (D\ D)\ B)\ (C\ (E\ (F\ F)\ E)\ C)\ A)\ (G\ G)$. B is [2,5], C is [6,11]: disjoint, so unrelated. **Figure 12.14** shows the bars.

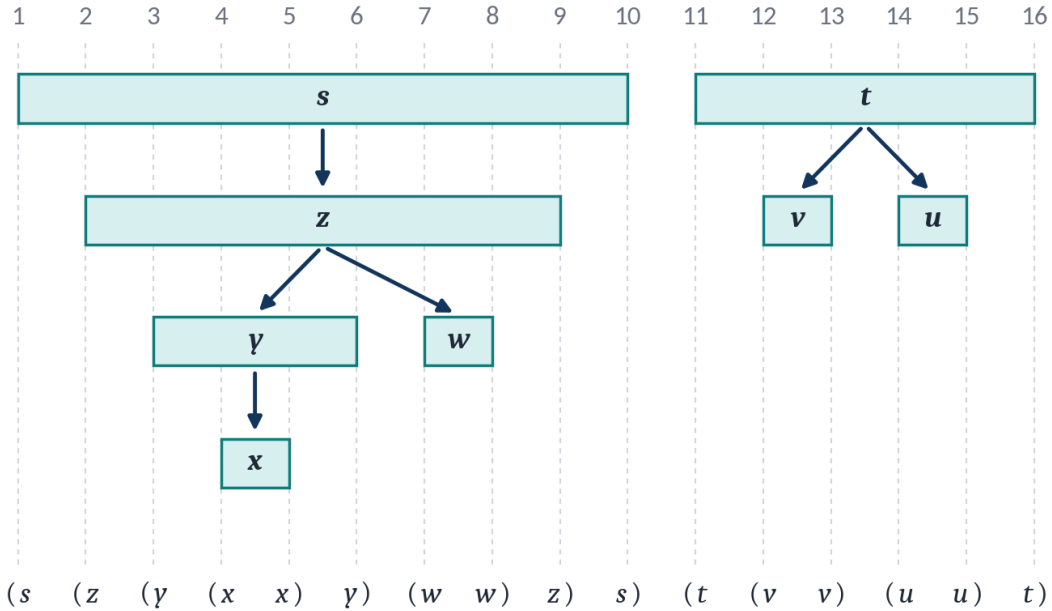


Figure 12.14 The parenthesis structure. Above, the depth-first forest with tree edges pointing downward; below, the clock from 1 to 16 with one bar per vertex spanning $[d, f]$, and the parenthesization written along the bottom. Every pair of bars is nested or disjoint — never partially overlapping.

Book: Section 12.4.3 and Figure 12.14. Check yourself: with $\mathrm{Adj}[A] = [D, C, B]$ *the forward edge vanishes and a fourth cross edge appears.*

The Form to Copy

Figure 12.15: one box per vertex with d/f, thick tree edges, **B**, **F**, **C** on the others — write your own answers this way.

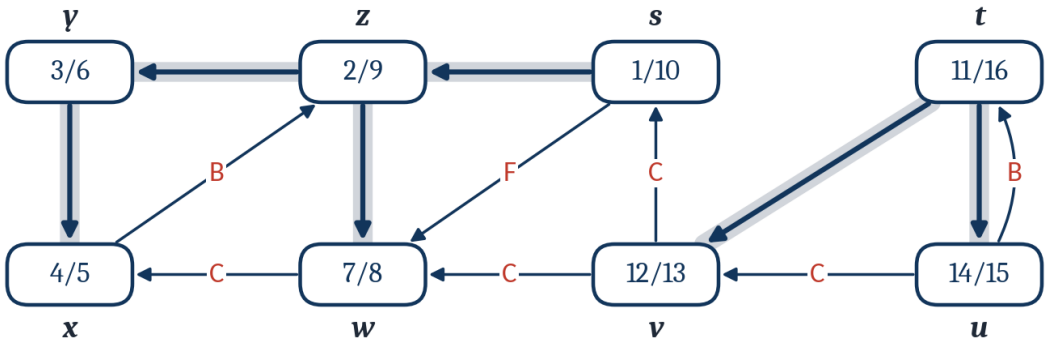


Figure 12.15 An eight-vertex depth-first search, timestamped: s is 1/10, z is 2/9, y is 3/6, x is 4/5, w is 7/8, t is 11/16, v is 12/13, u is 14/15. One figure carrying the timestamps, the forest and the complete edge classification at once — the form to copy when you write out a search of your own.

Book: Section 12.4.3 and Figure 12.15.

Problem I: DFS, Undirected

Every edge sits in two lists. At u with parent p: white → tree; gray and not p → **back**; black → already classified from the other end.

Table 12.16 Depth-first search of the undirected graph G_1 from A. The clock ends at $18 = 2|V|$. The intervals are perfectly nested, because the depth-first tree here is a single path through all nine vertices.

v	A	B	C	D	E	F	G	H	I
d/f	1/18	2/17	6/13	3/16	5/14	7/12	4/15	8/11	9/10
π	–	A	E	B	G	C	D	F	H

Table 12.16: tree $AB, BD, DG, GE, EC, CF, FH, HI$; back EB, CA, HE, IG. No forward or cross edges can occur, and four back edges match the four independent cycles, $12 - 9 + 1$.

Book: Section 12.4.4. Check yourself: from E the back edges are CE, HE, DB, GE — still four.

Problem J: Components

G_3 on 1..12: triangle {1,2,3}, star {4,5,6,7}, five-cycle {8, … ,12}. One BFS per unlabeled vertex, $\Theta(V + E)$ for the whole array in **Table 12.17**.

Table 12.17 The component label of every vertex of G_3. Three components. Once this array exists, connectivity is a single integer comparison, which is why the array and not the search is the answer.

Vertex	1	2	3	4	5	6	7	8	9	10	11	12
component	0	0	0	1	1	1	1	2	2	2	2	2

Queries become comparisons: 1–3 yes, 1–4 no, 5–7 yes, 6–11 no, 8–12 yes. The circuit rank Equation (12.14) gives $11 - 12 + 3 = 2$ independent cycles.

$$r(G) = |E| - |V| + k. \tag{12.14}$$

Book: Section 12.4.5. Adding {3,4} keeps the rank at 2: it is a bridge.

Problems K and L: Colors and Cycles

K. Color G_1 by $d[v]$ mod 2: sides $\{A, D, E, F, I\}$ and $\{B, C, G, H\}$; all twelve edges cross — bipartite. Add EF (both even): not bipartite; certificates $E\ B\ A\ C\ F\ E$ (length 5) or $E\ C\ F\ E$ (length 3).

L, undirected: skip **exactly one** occurrence of the parent; any other non-white neighbor is a cycle. The path is acyclic; the square and G_1 are not.

L, directed: only a **gray** head counts. G_2 has two cycles; $A \to B, A \to C, B \to C$ is a DAG, though C is "seen twice".

> **THE TRAP.** The directed test on an undirected graph reports a cycle in every graph with an edge; the parent rule means nothing in a directed graph.

Book: Sections 12.4.6 and 12.4.7; Problems G, H, J, K and L as one program are Section 12.4.8.

Problems M and N: DAG, Then Order

Table 12.18 The eight subjects of the prerequisite digraph used in Problems M to Q, with their durations and prerequisites. Nine arcs in all: $A \to D$, $B \to D$, $B \to E$, $C \to G$, $D \to E$, $D \to F$, $E \to H$, $F \to H$, $G \to H$.

Code	Subject	Weeks	Prerequisites
A	Programming Basics	3	—
B	Discrete Mathematics	2	—
C	Calculus	4	—
D	Data Structures	5	A, B
E	Algorithms	6	B, D
F	Databases	3	D
G	Statistics	2	C
H	Machine Learning	4	E, F, G

Table 12.19 Depth-first timestamps on the prerequisite DAG. Three roots — A, B and C, the three subjects with no prerequisite — so the forest has three trees and $8 - 3 = 5$ tree edges.

v	A	B	C	D	E	F	G	H
d/f	1/10	11/12	13/16	2/9	3/6	7/8	14/15	4/5

Table 12.18, searched in **Table 12.19**: five tree and four cross arcs, **no back arc** — a DAG. $B \to D$ is cross, not forward: D lies in A's tree.

Decreasing f: C, G, B, A, D, F, E, H; all nine arcs point forward, because Equation (12.15). Sorting by d fails on $B \to A$ searched from A.

$$f[v] < f[u] \qquad \text{for every arc } u \to v \text{ of a DAG,} \tag{12.15}$$

Book: Sections 12.5.1–12.5.3.

The Classic Instance

Figure 12.16: the dressing DAG with its timestamps.

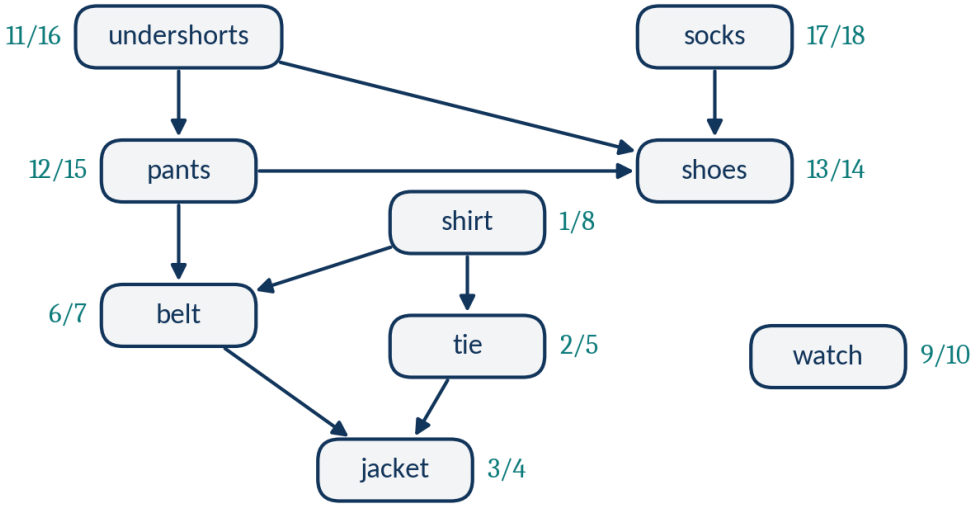


Figure 12.16 The precedence DAG for getting dressed, with the depth-first timestamps beside each vertex: shirt 1/8, tie 2/5, jacket 3/4, belt 6/7, watch 9/10, undershorts 11/16, pants 12/15, shoes 13/14, socks 17/18. Nine vertices, nine arcs, eighteen stamps.

Book: Section 12.5.3 and Figure 12.16.

Its Topological Order

Figure 12.17: decreasing finishing time, every arrow pointing right.

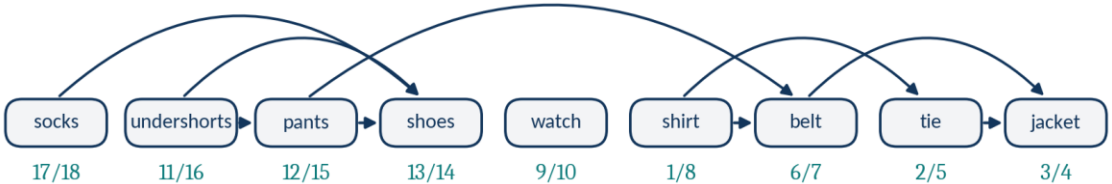


Figure 12.17 The result: the same nine garments laid out left to right in decreasing order of finishing time — socks 18, undershorts 16, pants 15, shoes 14, watch 10, shirt 8, belt 7, tie 5, jacket 4 — and now every arrow points right. That is what a topological sort *is*.

Book: Section 12.5.3 and Figure 12.17.

Problem O: Kahn's Algorithm

Pseudocode 12.6 Kahn's algorithm: repeatedly emit a vertex with no remaining prerequisite, and remove its outgoing arcs.

```
KAHN(G)
 1  for each vertex v:  indeg[v] = 0
 2  for each arc (u, v):  indeg[v] = indeg[v] + 1
 3  Q = an empty FIFO queue;  L = an empty list
 4  for each vertex v with indeg[v] == 0:  ENQUEUE(Q, v)
 5  while Q is not empty
 6      u = DEQUEUE(Q);  append u to L
 7      for each v in Adj[u]
 8          indeg[v] = indeg[v] - 1
 9          if indeg[v] == 0:  ENQUEUE(Q, v)
10  if |L| < |V| then report that G has a cycle
11  return L
```

Table 12.20 Kahn's algorithm on the prerequisite DAG, one row per removal. Changed entries are bold. Eight subjects emitted in eight rounds, so the graph is acyclic and the order is legal.

Round	Emit	In-degrees after (A…H)	Queue afterward
1	A	0 0 0 **1** 2 1 1 3	B C
2	B	0 0 0 **0 1** 1 1 3	C D
3	C	0 0 0 0 1 1 **0** 3	D G
4	D	0 0 0 0 **0 0** 0 3	G E F
5	G	0 0 0 0 0 0 0 **2**	E F
6	E	0 0 0 0 0 0 0 **1**	F
7	F	0 0 0 0 0 0 0 **0**	H
8	H	all zero	(empty)

In-degrees 0,0,0,2,2,1,1,3 (sum $9 = |E|$). **Pseudocode 12.6**, traced in **Table 12.20**: A, B, C, D, G, E, F, H — one position in common with Problem N, and both are legal.

Book: Section 12.5.4. With a min-priority queue the order is alphabetical.

Problems P and Q: Counting, Critical Path

P. 84 of the $8! = 40\,320$ permutations are legal. Without enumerating, memoize Equation (12.16) on subsets: $O(2^n \cdot n)$. The order is unique iff a Hamiltonian path exists: **three** extra arcs are needed ($A \to B$, $E \to C$, $G \to F$).

$$N(\varnothing) = 1, \qquad N(S) = \sum_{\substack{v \in S \\ v \text{ has no prerequisite in } S}} N(S \setminus \{v\}). \tag{12.16}$$

Q. By Equation (12.2) in topological order: A 3, B 2, C 4, D 8, G 6, E 14, F 11, H **18**. Critical path $A \to D \to E \to H$; one at a time would take 29 weeks. Adding $H \to A$: Kahn emits only 3, DFS finds 1 back edge.

Book: Sections 12.5.5 and 12.5.6; Problems M to Q as one program are Section 12.5.7.

Problem R: Kruskal with a Disjoint-Set Trace

Table 12.21 The weighted graph G of Problems R, S, U, V and W: eight vertices and thirteen edges. Two edges, AB and FG, share the weight 3.

Edge	*AB*	*AC*	*BC*	*BD*	*CD*	*CE*	*DE*	*DF*	*EF*	*EG*	*FG*	*FH*	*GH*
w	3	5	1	6	4	9	2	8	7	10	3	12	11

Table 12.22 Kruskal on G, with the disjoint-set forest after every decision. Seven edges accepted, five rejected, one never examined. Row 11 repays study: the edge was *rejected*, yet `parent[G]` changed from `F` to `B`, because the two `Find` calls compressed their paths on the way to the answer.

#	Edge	w	Find(a)	Find(b)	Verdict	parent	rank
1	BC	1	B	C	ACCEPT	ABBDEFGH	01000000
2	DE	2	D	E	ACCEPT	ABBDDFGH	01010000
3	AB	3	A	B	ACCEPT	BBBDDFGH	01010000
4	FG	3	F	G	ACCEPT	BBBDDFFH	01010100
5	CD	4	B	D	ACCEPT	BBBBDFFH	02010100
6	AC	5	B	B	reject	BBBBDFFH	02010100
7	BD	6	B	B	reject	BBBBDFFH	02010100
8	EF	7	B	F	ACCEPT	BBBBBBFH	02010100

Table 12.22 (Continued) *Kruskal on G, with the disjoint-set forest after every decision. Seven edges accepted, five rejected, one never examined. Row 11 repays study: the edge was rejected, yet parent[G] changed from F to B, because the two Find calls compressed their paths on the way to the answer.*

#	Edge	w	Find(a)	Find(b)	Verdict	parent	rank
9	DF	8	B	B	reject	BBBBBBFH	02010100
10	CE	9	B	B	reject	BBBBBBFH	02010100
11	EG	10	B	B	reject	BBBBBBBH	02010100
12	GH	11	B	H	ACCEPT	BBBBBBBB	02010100

Table 12.21, ties alphabetical; **Table 12.22** shows `parent` and `rank` after each decision. Row 11 is rejected, yet `parent[G]` changes — compression. $T = \{BC, DE, AB, FG, CD, EF, GH\}$, $w(T) = 31$; stop at $|V| - 1$.

Book: Sections 12.6.1 and 12.6.2. `rank` *is an upper bound on height, not the height.*

Where Kruskal and Prim Part

Figure 12.18: at weight 8, (a, h) accepted and (b, c) rejected — Prim does the opposite. Both trees weigh 37.

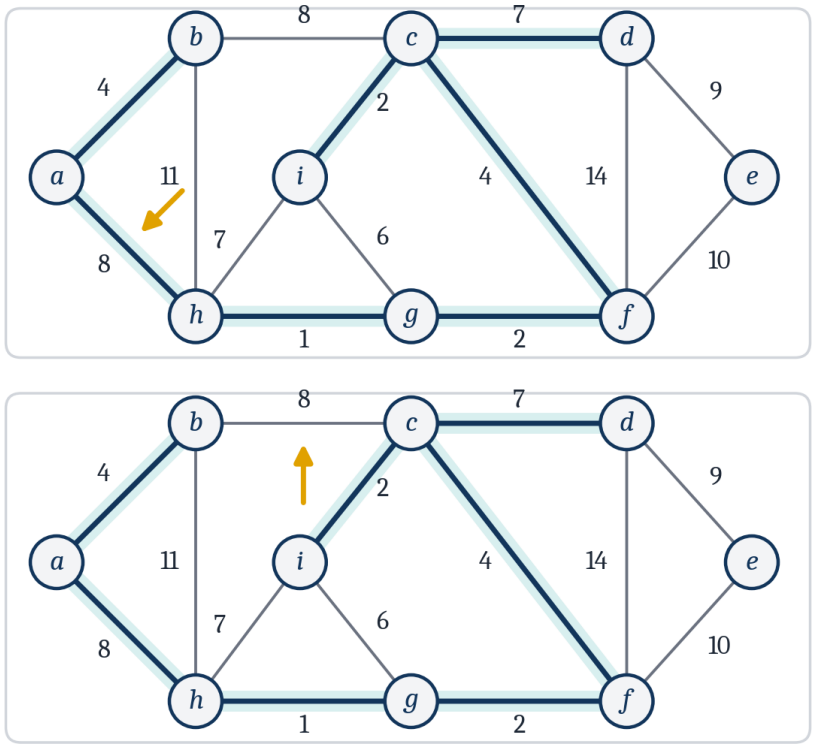


Figure 12.18 Kruskal on the nine-vertex graph of Chapter 11 at weight 8: (a, h) accepted, (b, c) rejected. Prim, on the same graph, does the opposite — it takes (b, c) and never considers (a, h). Both trees weigh 37. Equal weights are where the two algorithms are free to disagree.

Book: Section 12.6.2 and Figure 12.18. Check yourself: with $w(FG) = 13$ the tree weighs 38.

Problem S: Prim with the Key Array

For v outside the tree, Equation (12.17); extract the least key, commit its edge, lower the neighbors' keys.

$$key[v] = \min\{\, w(u,v) : u \in T,\ \{u,v\} \in E \,\}, \tag{12.17}$$

Table 12.23 Prim–Jarník from A, one row per extraction. Eight extractions, seven committed edges, total 31. Every key that falls is a `DecreaseKey`: C from $5/A$ to $1/B$ at step 2, D from $6/B$ to $4/C$ at step 3, E from $9/C$ to $2/D$ at step 4, F from $8/D$ to $7/E$ at step 5, G from $10/E$ to $3/F$ at step 6, and H from $12/F$ to $11/G$ at step 7.

Step	Extract	Edge committed	Total	Waiting afterward, as v : key/π
1	A	(root)	0	$B:3/A$ $C:5/A$ $D:\infty$ $E:\infty$ $F:\infty$ $G:\infty$ $H:\infty$
2	B	$AB(3)$	3	$C:1/B$ $D:6/B$ $E:\infty$ $F:\infty$ $G:\infty$ $H:\infty$
3	C	$BC(1)$	4	$D:4/C$ $E:9/C$ $F:\infty$ $G:\infty$ $H:\infty$
4	D	$CD(4)$	8	$E:2/D$ $F:8/D$ $G:\infty$ $H:\infty$
5	E	$DE(2)$	10	$F:7/E$ $G:10/E$ $H:\infty$
6	F	$EF(7)$	17	$G:3/F$ $H:12/F$
7	G	$FG(3)$	20	$H:11/G$
8	H	$GH(11)$	31	(empty)

Table 12.23: from A the tree grows as a path, weights 3,1,4,2,7,3,11 — the same seven edges as Kruskal, total **31**. From H: the same tree from the other end.

Book: Section 12.6.3.

Problem T: Prim with a Binary Heap

Pseudocode 12.7 Prim's algorithm with a binary heap and lazy `DecreaseKey`: an improved key is pushed as a new entry, and out-of-date entries are discarded when they reach the top.

```
LAZY-PRIM(G, w, r)
 1  for each vertex v:  key[v] = infinity;  pi[v] = NIL;  intree[v] = FALSE
 2  key[r] = 0;  H = a min-heap containing (0, r)
 3  while H is not empty
 4      (k, u) = EXTRACT-MIN(H)
 5      if intree[u] or k != key[u]
 6          continue                          // a stale entry: discard it
 7      intree[u] = TRUE                      // commit edge (pi[u], u)
 8      for each v in Adj[u]
 9          if not intree[v] and w(u, v) < key[v]
10              key[v] = w(u, v);  pi[v] = u
11              INSERT(H, (key[v], v))        // the old entry stays behind
```

Table 12.24 Prim with a binary heap and lazy `DecreaseKey` on the nine-vertex graph, from a. Nine real extractions commit eight edges of total weight 37; four further pops find stale entries — old copies of g, h (twice) and e whose keys had since fallen — and discard them.

Step	Pop	Commit	Total	Keys lowered	Heap afterward
1	a: 0	(root)	0	b: $\infty \to 4$, h: $\infty \to 8$	b: 4 h: 8
2	b: 4	$ab(4)$	4	c: $\infty \to 8$	c: 8 h: 8
3	c: 8	$bc(8)$	12	$d \to 7$, $f \to 4$, $i \to 2$	i: 2 f: 4 d: 7 h: 8
4	i: 2	$ci(2)$	14	$g \to 6$, h: $8 \to 7$	f: 4 g: 6 d: 7 h: 7 h: 8
5	f: 4	$cf(4)$	18	$e \to 10$, g: $6 \to 2$	g: 2 g: 6 d: 7 h: 7 h: 8 e: 10
6	g: 2	$fg(2)$	20	h: $7 \to 1$	h: 1 g: 6 d: 7 h: 7 h: 8 e: 10
7	h: 1	$gh(1)$	21	—	g: 6 d: 7 h: 7 h: 8 e: 10
—	g: 6	stale, discarded	21	—	d: 7 h: 7 h: 8 e: 10
8	d: 7	$cd(7)$	28	e: $10 \to 9$	h: 7 h: 8 e: 9 e: 10
—	h: 7, h: 8	both stale	28	—	e: 9 e: 10

Table 12.24 (Continued) *Prim with a binary heap and lazy DecreaseKey on the nine-vertex graph, from a. Nine real extractions commit eight edges of total weight 37; four further pops find stale entries — old copies of g, h (twice) and e whose keys had since fallen — and discard them.*

Step	Pop	Commit	Total	Keys lowered	Heap afterward
9	e: 9	$de(9)$	37	—	e: 10
—	e: 10	stale, discarded	37	—	(empty)

Pseudocode 12.7 on the nine-vertex graph from a, **Table 12.24**: 13 pushes, 13 pops, **4 stale**; tree weight 37. At most $|E| + 1$ pushes, so Equation (12.18).

$$O(E\lg E) = O(E\lg V), \tag{12.18}$$

Book: Section 12.6.4. From i the weight is again 37, but the tree uses ha instead of bc.

The Tree from a

Figure 12.19: $ab, bc, ci, cf, fg, gh, cd, de$ — weight 37, eight edges for nine vertices.

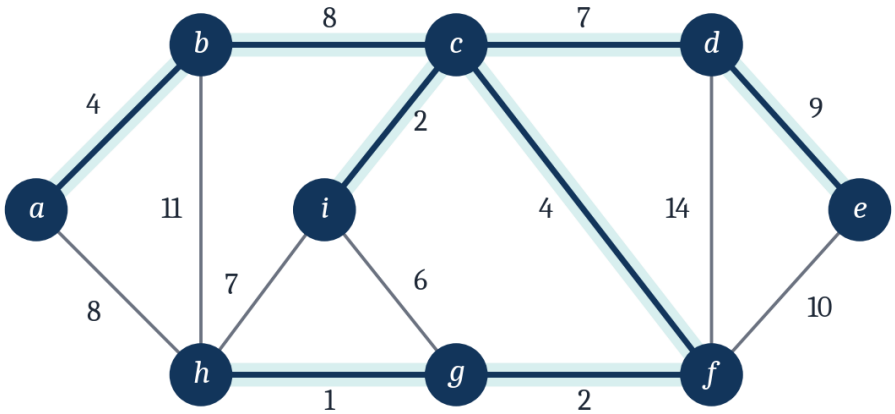


Figure 12.19 The finished minimum spanning tree on the nine-vertex graph: (a, b), (b, c), (c, i), (c, f), (f, g), (g, h), (c, d), (d, e), of total weight 37. Eight edges for nine vertices. Kruskal reaches a different set of eight edges — also weighing 37 — because two edges of weight 8 tie.

Book: Section 12.6.4 and Figure 12.19.

Problem U: Cut and Cycle Properties

Cut property: a light edge of a cut respecting A is safe (**Figure 12.20**). For $S = \{A, B, C\}$ the crossing edges are BD 6, CD 4, CE 9; the light edge CD is in the tree.

Cycle property: a **uniquely** heaviest edge on some cycle is in no MST. **Table 12.25**: every non-tree edge is strictly heaviest on its cycle — so the tree of Problem R is the only MST.

Table 12.25 Every non-tree edge of G, the cycle it closes with the minimum spanning tree, and the heaviest tree edge on that cycle. All six are strictly heaviest, so none can appear in any minimum spanning tree — which is a complete proof that the tree of Problem R is the only one.

Non-tree edge	w	The cycle it closes	Heaviest tree edge on it	Verdict
AC	5	A–B–C–A	3 (AB)	strictly heaviest ⇒ in no MST
BD	6	B–C–D–B	4 (CD)	in no MST
CE	9	C–D–E–C	4 (CD)	in no MST
DF	8	D–E–F–D	7 (EF)	in no MST
EG	10	E–F–G–E	7 (EF)	in no MST
FH	12	F–G–H–F	11 (GH)	in no MST

Book: Section 12.6.5. Check yourself: for $S = \{A, B, C, D\}$ the light edge is DE (2).

The Exchange, Once More

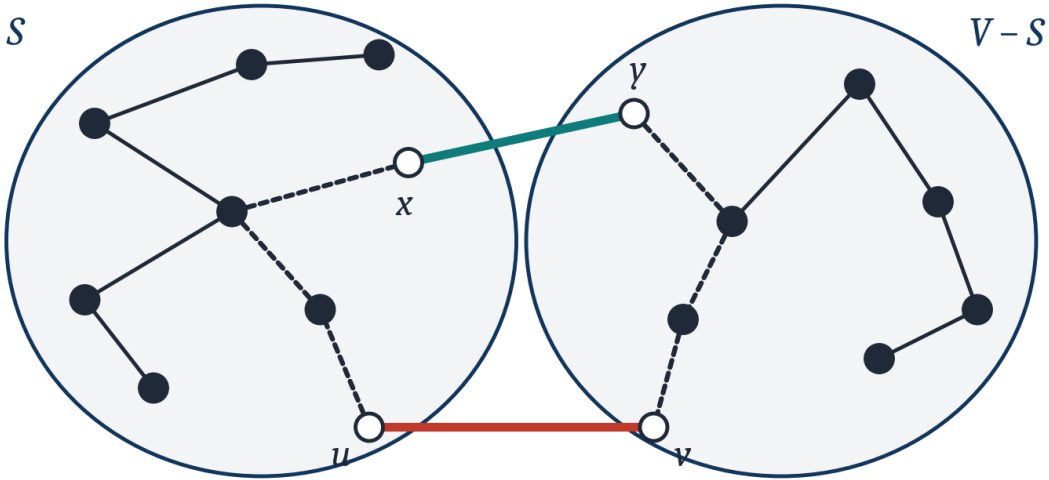


Figure 12.20 The exchange that proves the cut property. S and $V - S$ are the two blobs; the red edge (u, v) at the bottom is the light edge, assumed not to be in the tree; the dotted line is the unique tree path from u to v; and the teal edge (x, y) is the first edge of that path to cross the cut. Swapping (x, y) out for (u, v) is the whole proof.

Book: Section 12.6.5 and Figure 12.20.

Problem V: Union by Rank, Path Compression

Table 12.26 Seven unions under union by rank. The rank of a root climbs only when two equal ranks meet — all seven unions here happen to be such meetings, which is why the structure stays perfectly balanced.

Operation	Effect	parent	rank
`Union(A,B)`	ranks tie at 0; B under A, `rank[A]`$$1	`AACDEFGH`	`10000000`
`Union(C,D)`	ranks tie at 0; D under C	`AACCEFGH`	`10100000`
`Union(E,F)`	ranks tie at 0; F under E	`AACCEEGH`	`10101000`
`Union(G,H)`	ranks tie at 0; H under G	`AACCEEGG`	`10101010`
`Union(A,C)`	ranks tie at 1; C under A, `rank[A]`$$2	`AAACEEGG`	`20101010`
`Union(E,G)`	ranks tie at 1; G under E, `rank[E]`$$2	`AAACEEEG`	`20102010`
`Union(A,E)`	ranks tie at 2; E under A, `rank[A]`$$3	`AAACAEEG`	`30102010`

Table 12.26: every union meets equal ranks, so the tree stays balanced at height 3. After a `Find` on each element, `parent = AAAAAAAA` and the ranks stay stale.

The bound Equation (12.19) caps the height at $\lfloor \lg 8 \rfloor = 3$; a naive chain reaches 7. On 100 000 chained elements: 4 999 950 000 pointer steps without the tricks, 99 999 with them.

$$size(x) \geq 2^{rank(x)} \qquad \text{for every root } x. \tag{12.19}$$

Book: Section 12.6.6. Check yourself: the chaining sequence with union by rank gives height 1.

Lists, for Comparison

Figure 12.21: a list union re-points every element of the shorter list; the forest changes one pointer.

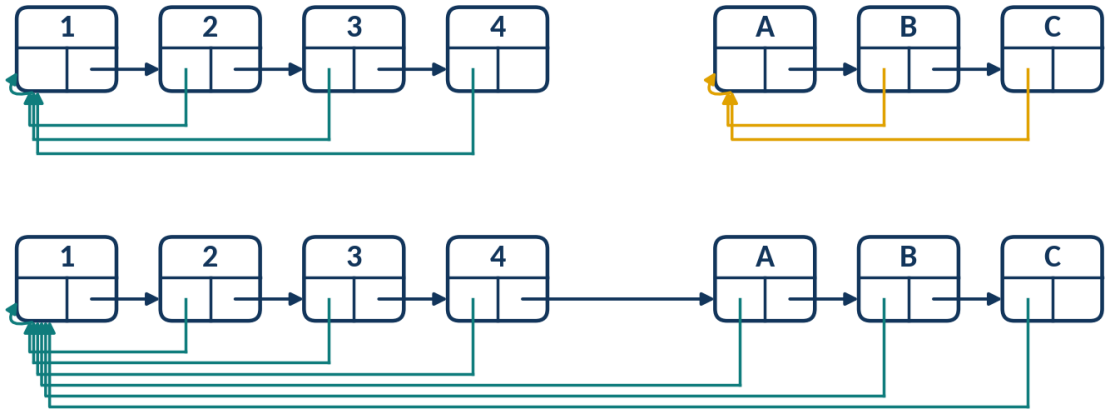


Figure 12.21 Disjoint sets as lists. **Top:** two sets, each element carrying a back-pointer to its representative. **Bottom:** the result of uniting them — the shorter list appended and every one of its elements re-pointed. Compare the cost with **Table 12.26**, where a union changes exactly one pointer.

Book: Section 12.6.6 and Figure 12.21.

Problem W: Is the MST Unique?

Table 12.27 The uniqueness test on G: each tree edge deleted in turn and Kruskal re-run. Every row is strictly heavier than 31, so the minimum spanning tree of G is **unique**.

Edge deleted	w	New best spanning tree	Verdict
BC	1	35	heavier by 4
DE	2	37	heavier by 6
AB	3	33	heavier by 2
FG	3	38	heavier by 7
CD	4	33	heavier by 2
EF	7	32	heavier by 1
GH	11	32	heavier by 1

Table 12.27: every deletion is strictly heavier — **unique**, although AB and FG share weight 3. Distinct weights are sufficient, not necessary.

Lower $w(AC)$ to 3: two MSTs of weight 31, with AB or AC. **Check yourself:** lower $w(FH)$ to 11 instead — again two.

Book: Section 12.6.7; Problems R, S, U, V and W as one program are Section 12.6.8.

Quick Self-Check Drills

Greedy. Two ingredients? Greedy choice and optimal substructure. Maximal or maximum? Greed gives maximal for free. Which knapsack lacks the greedy choice? 0-1.

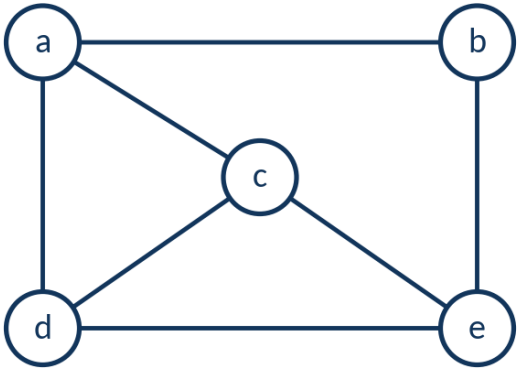
Searching. Handshake lemma? $\sum_v \deg(v) = 2|E|$. Why are BFS distances shortest? Non-decreasing dequeue order. Edge classes in an undirected DFS? Tree and back only.

DAGs. DAG theorem? No back edge. Why decreasing f, not d? $f[v] < f[u]$ for every arc.

MSTs. Safe edge? Keeps A inside some MST. Unique MST? Every non-tree edge strictly heaviest on its cycle. **Figure 12.22** carries the whole vocabulary.

Book: Section 12.7, thirty drills with their section references.

The Vocabulary in One Picture



$$V = \{a, b, c, d, e\}$$

$$E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$$

Figure 12.22 The definition of a graph, with the five-vertex example used throughout Chapter 10: $V = \{a, b, c, d, e\}$ and $E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$. The drawing and the two set-listings carry identical information; the drawing is only easier on the eye. Degrees 3,2,3,3,3 sum to $14 = 2 \times 7$ — drill 11 in one picture.

Book: Section 12.7 and Figure 12.22.

Profitina in Practice

Crawl = BFS (click depth $d[v]$; $d[v] = \infty$ means orphan). **Near-duplicates = MST with heavy edges cut** (**Figure 12.23**). **Remediation plan = DAG**, topologically sorted. **Re-scans** = earliest deadline first.

Profitina Laundry's website audit: `/promo-lama` is an orphan; three near-duplicate clusters (threshold 30); the remediation takes **16 hours** on the critical path against 25 one task at a time.

Book: Section 12.8 and Section 12.8.1. The app advises; the user decides.

A Similarity Graph Has This Shape

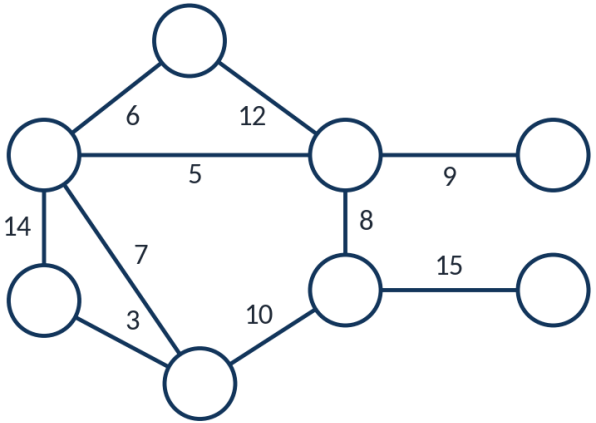


Figure 12.23 A weighted graph from Chapter 11. Some expensive edges are unavoidable — the 15 is the only way into its vertex — and some cheap-looking ones are redundant. A page-similarity graph has exactly this character, and the same algorithm separates the two kinds of edge.

Book: Section 12.8 and Figure 12.23.

Common Mistakes

Design: a decorative algorithm; the design written after the program; no evaluation; timings printed but not read.

Greedy: believing a rule because it worked once; "*the* optimal solution"; reading the matroid exchange as "every element works"; an unchecked Huffman code.

Searching: only the end result of a trace; the directed test on an undirected graph; forward or cross edges in an undirected DFS; "forward" because the target comes later.

DAGs and MSTs: sorting by discovery time; stale heap entries not skipped; equal weights taken to mean non-unique; Prim and Kruskal expected to return the same edges; `rank` read as height.

Book: Section 12.9.

Summary: Five Things to Remember

1. A design project runs in nine stages, six of them before code; the load-bearing test decides whether the problem deserves an algorithm.

2. Rute Laundry: 6 of 14 zones within two hops, 2 unreachable; a 6.9 km backbone (43.9% saved); a 121-minute critical path — all proved by hand first.

3. Evaluation answers *is it right, does it scale, what does it buy* — a postcondition beats a cross-check; divide time by the predicted cost.

4. Greedy rules need an exchange proof or a counterexample; traces need every intermediate value.

5. Searches, topological orders and MSTs are rarely unique — check properties and weights, not strings and edge sets.

Book: Section 12.10.

Next Time

Chapter 13 — Single-Source Shortest Paths. Weights become travel times and the question becomes the cheapest **path**.

BFS when every edge costs the same; **Dijkstra** for non-negative costs; **Bellman–Ford** for negative ones; and on a DAG one topological sweep — the sweep that produced today's critical paths.

Before the meeting: Exercises 12.3, 12.5, 12.9 and 12.12, and read Sections 13.1 to 13.4.

Book: Section 12.10.1 and Chapter 13.

Lecture 13 — Single-Source Shortest Paths

Design and Analysis of Algorithms (EF234405)

Chapter 13 of the book, in miniature, for the classroom.

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*.

Lecturer: Irfan Subakti — yifana@gmail.com

Today we stop counting edges and start adding them up: three algorithms, one primitive, and one assumption that, when it is broken, breaks everything silently. Book: Chapter 13, opening paragraph.

Agenda

1. The problem, stated precisely. The weight of a path, the number $\delta(u, v)$, and the four variants of the question.

2. Two structural facts. Optimal substructure and the triangle inequality; every algorithm today exploits both.

3. Negative weights. A negative *edge* is harmless; a negative *cycle* destroys the problem.

4. Relaxation, the one primitive, and the five properties that make it safe.

5. Dijkstra, traced panel by panel, proved correct and timed; **6. Bellman–Ford**, traced pass by pass, with its negative-cycle detector.

7. DAGs in linear time; **8. recovering the route**; which algorithm for which graph; Profitina Laundry.

The order of today's pages follows the book exactly, and the exercises at the end of the chapter are the homework. Book: Sections 13.2 to 13.12.

Where We Left Off

From Chapter 10, the graph. A directed graph $G=(V,E)$; a path $p=\langle v_1, v_2, \ldots, v_k \rangle$ with $(v_i, v_{i+1}) \in E$; adjacency lists $Adj[u]$. Inside asymptotic notation we drop the bars: $O(V+E)$ means $O(|V|+|E|)$.

From Chapter 10, breadth-first search. It finds the fewest *number of edges* from s to every vertex in $O(V+E)$: today's problem in the special case where every edge counts the same.

From Chapter 11, weights. A weight function $w : E \to R$. Note R, not $R^{\geq 0}$: negative numbers are allowed, and Section 13.5 charges us for that.

From Chapter 11, the priority queue (`Insert`, `ExtractMin`, `DecreaseKey`, as an unsorted array, a binary heap or a Fibonacci heap) and **topological sort**, which Section 13.9 uses directly.

> **A minimum spanning tree is not a shortest-path tree.** Chapter 11 measured a *whole network*: the cheapest connected subgraph, with no starting point. Today we measure a *journey*: the cheapest routes from one particular vertex. They are usually different trees (Exercises 13.4 and 13.5).

Book: Section 13.1.

From Hop Counts to Total Weight

DEFINITION. The **weight of a path** $p=\langle v_1, v_2, \ldots, v_k \rangle$ is the sum of the weights of its edges:

$$w(p)=\sum_{i=1}^{k-1} w(v_i, v_{i+1}). \tag{13.1}$$

In Equation (13.1) the index i runs to $k-1$, one value per **edge**, not per vertex. A one-vertex path has no edges, so its sum is empty and $w(p)=0$; that convention makes $\delta(s,s)=0$ come out right.

DEFINITION. The **shortest-path weight** from u to v is

$$\delta(u,v) = \begin{cases} \min\{\, w(p) : p \text{ is a path from } u \text{ to } v \,\} & \text{if such a path exists,} \\ \infty & \text{otherwise.} \end{cases} \tag{13.2}$$

A **shortest path** is any path p from u to v with $w(p)=\delta(u,v)$.

Three traps in Equation (13.2). δ is a *number*, a shortest path is a *sequence*. It is **a** shortest path, never **the**: there may be several of equal weight. And $\delta(u,v)=\infty$ is not a failure; it is the answer *there is no route*.

EXAMPLE. In the Dijkstra example graph, $p=\langle s,x,u,v \rangle$ has $w(p)=5+3+1=9$, while $q=\langle s,u,v \rangle$ has $w(q)=10+1=11$. The cheaper path uses *more* edges: fewer edges and less cost are different things.

In code, ∞ is `const long long INF = LLONG_MAX / 4;` *and every relaxation is guarded by* `d[u] < INF`*, which is why none of today's programs wrap around. Book: Sections 13.2.1 to 13.2.3.*

$\delta(s,v)$ the Honest Way

Before any algorithm, compute $\delta(s,v)$ on the Dijkstra example graph by *listing every simple path from s to v and adding up*. The edges are $w(s,u)=10$, $w(s,x)=5$, $w(u,v)=1$, $w(u,x)=2$, $w(x,u)=3$, $w(x,v)=9$, $w(x,y)=2$, $w(v,y)=4$, $w(y,v)=6$, $w(y,s)=7$.

Table 13.1 Every simple path from s to v in the Dijkstra example graph, weighed one edge at a time; the minimum, 9, is $\delta(s,v)$, achieved by $\langle s,x,u,v\rangle$.

Path from s to v	Edge weights	Total
$\langle s,u,v\rangle$	10+1	11
$\langle s,x,v\rangle$	5+9	14
$\langle s,x,u,v\rangle$	5+3+1	9
$\langle s,x,y,v\rangle$	5+2+6	13
$\langle s,u,x,v\rangle$	10+2+9	21
$\langle s,u,x,y,v\rangle$	10+2+2+6	20

Why exactly six rows in Table 13.1? From s the only ways out are to u and to x; following every branch until it reaches v or runs out of unvisited vertices gives precisely these six simple routes.

Correct, and hopeless in general: a graph on n vertices can have more than $(n-2)!$ simple paths between one pair. On thirty vertices this lists more paths than a grain of sand has atoms.

Keep the number 9 *in mind: Dijkstra will produce it in five extractions. Book: Section 13.2.4.*

Four Problems, One Algorithm

Table 13.2 The four shortest-path problems. Three of the four reduce to the second, which is why the second is the one we solve.

Variant	What is fixed	What is asked	How we solve it
Single-pair	source s, target t	$\delta(s,t)$ and one path	solve single-source from s, stop early if you like
Single-source	source s	$\delta(s,v)$ for all v	this chapter: Dijkstra, Bellman–Ford, DAG relaxation
Single-destination	target t	$\delta(u,t)$ for all u	reverse every edge, then solve single-source from t
All-pairs	nothing	$\delta(u,v)$ for all u,v	Chapter 15: Floyd–Warshall, Johnson; or $\lvert V\rvert$ runs of this chapter

Single-pair is not easier. Every known method is a single-source method allowed to stop when the target leaves the queue, and in the worst case the target leaves last. No asymptotically better method is known.

Single-destination is a mirror. Build the transpose G^T with $E^T = \{(v,u) : (u,v) \in E\}$, keep the weights, run from t. Reversing costs $\Theta(V+E)$.

All-pairs is $\lvert V\rvert$ single-source runs, or the subject of Chapter 15. So Table 13.2 says: three of the four variants either reduce to single-source or are built out of it.

Book: Sections 13.2.5 and 13.2.6.

Where Shortest Paths Turn Up

Network routing. OSPF is *link-state*: every router builds a map and runs **Dijkstra**. RIP is *distance-vector*: routers exchange distance tables with their neighbors, which is **Bellman–Ford** run distributed, one pass per exchange round.

Robot motion planning. Discretize the floor; weight each edge by time or energy. The weights are naturally non-negative, so Dijkstra applies.

Maps and traffic. Weights are travel times, re-weighted through the day, and the query is single-pair. Production systems use A* and contraction hierarchies, whose correctness is argued from the properties of Section 13.6.

Scheduling with precedences. Tasks form a DAG, weights are durations, and the *longest* path is the critical path (Section 13.9.7).

Currency arbitrage. Weight each exchange edge by $-\ln(\text{rate})$; a negative cycle is a sequence of trades that returns more money than it started with. Bellman–Ford's cycle detector *is* an arbitrage detector.

The general theory, shortest paths as the simplest minimum-cost flow, is in Ahuja, Magnanti and Orlin; a gentler treatment is Kleinberg and Tardos. Book: Section 13.2.7.

Optimal Substructure

THEOREM 13.1. *Subpaths of shortest paths are shortest paths.* If $p=\langle v_1,\dots,v_k\rangle$ is a shortest path from v_1 to v_k, then for any $1\le i\le j\le k$ the subpath $p_{ij}=\langle v_i,\dots,v_j\rangle$ is a shortest path from v_i to v_j.

Proof, by cut and paste. Split p at the two chosen vertices:

$$w(p)=w(p_{1i})+w(p_{ij})+w(p_{jk}). \tag{13.3}$$

If some p'_{ij} had $w(p'_{ij})<w(p_{ij})$, cut out p_{ij} and paste in p'_{ij} (legal, the endpoints match). By Equation (13.3) the new path weighs $w(p_{1i})+w(p'_{ij})+w(p_{jk})<w(p)$, contradicting the minimality of p. ∎

The splice step deserves suspicion. It works here because any two paths sharing an endpoint concatenate. It fails for longest *simple* paths, where the splice may revisit a vertex; that is why that problem is NP-hard and this one is not.

What it buys us. The license to build shortest paths incrementally. Every algorithm today answers one question: *in what order should I consider the vertices, so that when I use $\delta(s,u)$ it is already known?* Dijkstra: *in increasing order of distance.* Bellman–Ford: *I don't know, so try everything $|V|-1$ times.* The DAG algorithm: *in topological order.*

If the cheapest way from Depot to Keputih passes through ITS, then its Depot–ITS part is the cheapest way from Depot to ITS. Book: Section 13.3.

The Triangle Inequality

THEOREM 13.2 (vertex form). For any three vertices u, x, v,

$$\delta(u,v) \le \delta(u,x) + \delta(x,v). \tag{13.4}$$

Proof of Equation (13.4)**.** If either term on the right is ∞, it holds trivially. Otherwise let p_1 be a shortest $u \rightsquigarrow x$ and p_2 a shortest $x \rightsquigarrow v$; their concatenation q is a path from u to v with $w(q) = \delta(u,x) + \delta(x,v)$. A minimum over a set is at most any member, so $\delta(u,v) \le w(q)$. ∎

COROLLARY 13.3 (edge form). For every edge $(u,v) \in E$ and every source s,

$$\delta(s,v) \le \delta(s,u) + w(u,v). \tag{13.5}$$

Proof of Equation (13.5)*:* Theorem 13.2 on (s,u,v) gives $\delta(s,v) \le \delta(s,u) + \delta(u,v)$, and the single edge $\langle u,v \rangle$ is a path, so $\delta(u,v) \le w(u,v)$. ∎ In words: the true distance to v can never be beaten by reaching u optimally and taking one more edge.

All of Section 13.6 is built on the edge form; the algorithms try to make their estimates satisfy it with equality. Book: Sections 13.4.1 to 13.4.3.

The Triangle Inequality, Drawn

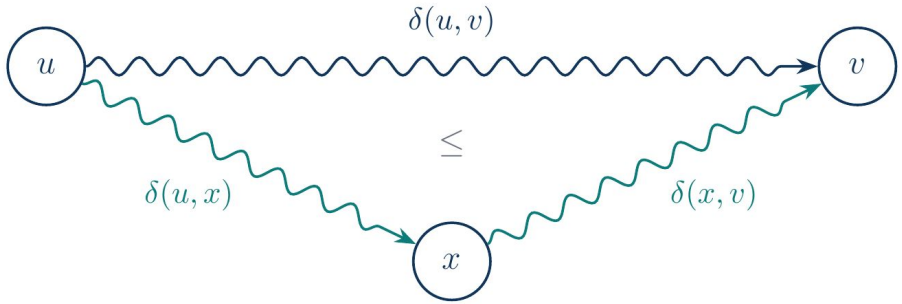


Figure 13.1 The triangle inequality: the upper wavy arrow is a shortest path from u to v, the lower pair a shortest path from u to x followed by one from x to v. The lower route is *a* path from u to v, so it cannot be cheaper than the best one.

In Figure 13.1 the wavy lines are the standard notation for "some path, we do not care which". The one-line proof: the best route $u \rightsquigarrow v$ is no longer than any other route, in particular the one that is forced through x.

Book: Section 13.4.2.

The Edge Form, Checked on Ten Edges

Table 13.3 The edge form of the triangle inequality checked on every edge of the Dijkstra example graph, with $\delta(s,s)=0$, $\delta(s,x)=5$, $\delta(s,y)=7$, $\delta(s,u)=8$, $\delta(s,v)=9$.

Edge (u,v)	$w(u,v)$	$\delta(s,u)+w(u,v)$	$\delta(s,v)$	Holds?
(s,u)	10	$0+10=10$	8	$8 \le 10$ yes
(s,x)	5	$0+5=5$	5	equality
(u,v)	1	$8+1=9$	9	equality
(u,x)	2	$8+2=10$	5	yes
(x,u)	3	$5+3=8$	8	equality
(x,v)	9	$5+9=14$	9	yes
(x,y)	2	$5+2=7$	7	equality
(v,y)	4	$9+4=13$	7	yes
(y,v)	6	$7+6=13$	9	yes
(y,s)	7	$7+7=14$	0	yes

The rows of Table 13.3 showing *equality* — (s, x), (x, u), (u, v), (x, y) — are precisely the four edges of the shortest-path tree, as Section 13.6.9 explains. Reaching that equality is the whole job.

Book: Section 13.4.4.

A Negative Edge; A Negative Cycle

A negative edge is not exotic. A courier leg ending at a partner drop-point that pays a rebate has negative net cost. Theorems 13.1 and 13.2 were proved *without* assuming $w \geq 0$: the proofs only ever add and compare.

> **WARNING.** If a negative-weight cycle is reachable from s and v is reachable from that cycle, then $\delta(s, v)$ *does not exist* as a finite number. By convention we write $\delta(s, v) = -\infty$.

Why. With x on a cycle c, a path $p : s \rightsquigarrow x$ and a path $q : x \rightsquigarrow v$, the walk that goes round c exactly k times weighs $w(p) + k\,w(c) + w(q)$. Since $w(c) < 0$, this falls below any number you name: there is no minimum.

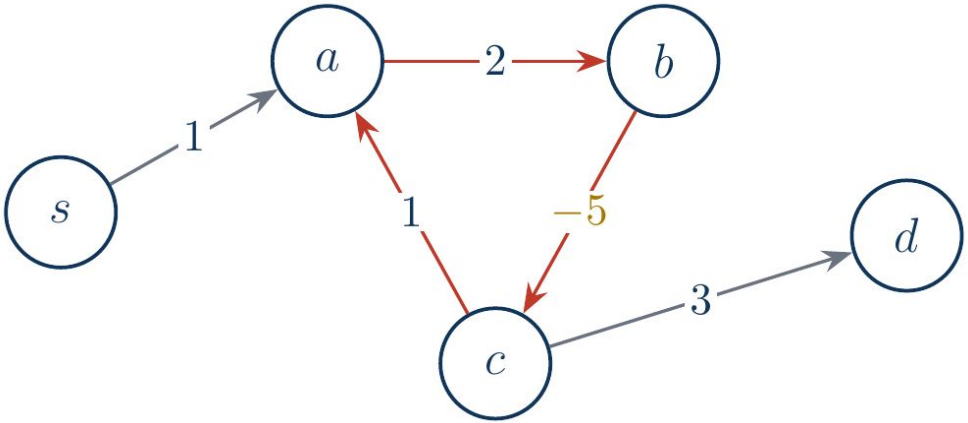


Figure 13.2 A graph with a reachable negative cycle. Going round the red cycle $a \to b \to c \to a$ costs $2+(-5)+1=-2$, so each extra lap makes the "path" two cheaper; every vertex reachable from the cycle has $\delta(s,\cdot)=-\infty$.

In Figure 13.2, $\delta(s,a)=\delta(s,b)=\delta(s,c)=\delta(s,d)=-\infty$ while $\delta(s,s)=0$. An **unreachable** negative cycle does no harm, and a vertex not reachable *from* the cycle keeps its finite distance. The problem is not hard here; it is *ill-posed*.

The Bellman–Ford example graph has three negative edges and a perfectly well-defined answer. Book: Sections 13.5.1 and 13.5.2.

Shortest Paths Have No Cycles

THEOREM 13.4. If no negative-weight cycle is reachable from s, then for every reachable v there is a shortest path from s to v that is **simple**, and therefore has at most $|V|-1$ edges.

Proof. Let p be a shortest path that repeats a vertex x, so p has the shape $s \rightsquigarrow x \overset{c}{\rightsquigarrow} x \rightsquigarrow v$. Excising c leaves p' with $w(p')=w(p)-w(c)$.

c is reachable from s, so by hypothesis $w(c)\geq 0$, giving $w(p')\leq w(p)$; and p is shortest, so $w(p')\geq w(p)$. Hence the shorter path is *also* shortest. Each excision removes at least one edge, so the process ends in a simple shortest path, with at most $|V|-1$ edges. ∎

Remember this one. *Any shortest path can be no longer than $n-1$ edges.* It is the reason Bellman–Ford runs exactly $|V|-1$ passes, and the reason its $|V|$-th pass is a *detector* rather than useful work.

EXAMPLE. Both example graphs have five vertices, so shortest paths from s have at most four edges. In the Dijkstra graph the longest is $\langle s, x, u, v\rangle$, three edges; in the Bellman–Ford graph it is $\langle s, y, x, t, z\rangle$, exactly four. That is why all four passes are needed there.

Book: Section 13.5.3.

Which Algorithm Survives What

Table 13.4 What each of the three algorithms does in each weight regime. The word to notice is *silently*.

Situation	Dijkstra	Bellman–Ford	DAG relaxation
All weights ≥ 0	correct	correct	correct (if acyclic)
Some weights < 0, no negative cycle	**wrong, silently**	correct	correct (if acyclic)
Negative cycle reachable from s	wrong, silently	detects it, returns `false`	cannot arise: a DAG has no cycles at all
Negative cycle not reachable from s	wrong if any negative edge is reachable	correct for reachable part	cannot arise

The word to notice in Table 13.4 is **silently**: Dijkstra does not crash on a negative edge and does not complain; it returns a plausible-looking wrong number. Section 13.7.3 builds the smallest such graph.

Row 2 is the row that decides your program. Row 3 decides whether the question has an answer at all.

A useful habit: check `w >= 0` for every edge before running any Dijkstra you ship; it turns a wrong answer into a loud failure. Book: Section 13.5.4.

Relaxation: the Two Arrays

DEFINITION. For each vertex v we keep two values. $d[v]$, the **shortest-path estimate**, is always an *upper bound* on $\delta(s, v)$: the weight of the best route from s to v found so far. $\pi[v]$, the **predecessor**, is the vertex from which we arrived at v along that route, or `NIL`.

The numbers d answer *how far*; the predecessors π answer *which way*, and Section 13.10 turns them into a list of vertices.

WARNING. $d[v]$ is an *estimate*. During the run it can be too large; it is never too small. Code that treats $d[v]$ as the answer halfway through the loop is the commonest bug in the whole subject.

KEY IDEA. Relaxation is the **only** way d and π change after initialization. Dijkstra, Bellman–Ford and the DAG algorithm differ only in **the order in which they relax edges, and how many times**, so everything proved about d and π applies to all three at once.

Book: Section 13.6.1.

Initialize, Then Relax

Pseudocode 13.1 Initialization shared by every single-source algorithm: all estimates infinite, no predecessors, the source at distance zero.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v in V[G]
2      d[v] <- infinity
3      pi[v] <- NIL
4  d[s] <- 0
```

Pseudocode 13.2 Relaxing the edge (u, v): one comparison, and two assignments when the route through u is strictly better.

```
RELAX(u, v, w)
1  if d[v] > d[u] + w(u,v)
2      d[v]  <- d[u] + w(u,v)
3      pi[v] <- u
```

Pseudocode 13.1 costs $\Theta(V)$ and says: *I know one route, the empty route from s to itself, and it costs nothing; about everything else I know nothing at all.* It is the first lines of all three algorithms.

Pseudocode 13.2 is one comparison and two assignments. Line 1 fires only when the new value is **smaller**, so $d[v]$ only ever falls, like a stretched spring allowed to settle.

These same three lines are the highlighted "relaxing edges" lines of Dijkstra's pseudocode: the only lines that change the answer. Book: Sections 13.6.2 and 13.6.3.

The Two Outcomes of a Relaxation

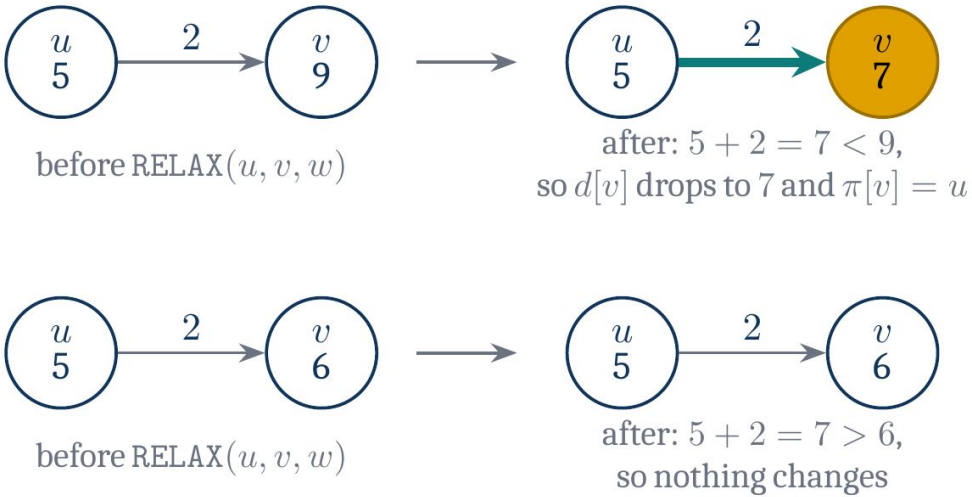


Figure 13.3 The two outcomes of `RELAX(u,v,w)` with $w(u,v)=2$. Top: $d[u]+w=5+2=7<d[v]=9$, so $d[v]$ drops to 7 and $\pi[v]$ becomes u. Bottom: $d[u]+w=7$ is not less than $d[v]=6$, so nothing happens.

Top of Figure 13.3**.** We knew a route to v costing 9 and a route to u costing 5. One more edge of weight 2 gives 7, which is better: take it, and remember we arrived via u.

Bottom. The route through u would cost 7; we already have one costing 6. Keep it and leave $\pi[v]$ alone.

A relaxation that changes nothing is not wasted work: it confirms that the current estimate already beats this route. Book: Section 13.6.4.

Upper-Bound and No-Path

LEMMA 13.5 (Upper-bound property). After initialization and after *any* sequence of relaxations, $d[v] \geq \delta(s,v)$ for every v. Once $d[v]$ reaches $\delta(s,v)$ it never changes again.

Proof, by induction on the number of relaxations. *Base:* $d[v] = \infty \geq \delta(s,v)$ for $v \neq s$, and $d[s] = 0 = \delta(s,s)$ because no cycle through s is negative. *Step:* only $d[v]$ changes, to $d[u] + w(u,v) \geq \delta(s,u) + w(u,v) \geq \delta(s,v)$, by the hypothesis and then by Equation (13.5). ∎

The freezing half. A later change would strictly decrease $d[v]$ below $\delta(s,v)$, which is impossible. **The estimate is never optimistic, and once it is right it stays right.**

LEMMA 13.6 (No-path property). If there is no path from s to v, then $d[v] = \delta(s,v) = \infty$ at all times.

For $d[v]$ to become finite, some relaxation of (u,v) must have found $d[u]$ finite; chasing that back ends at s, and the chain *is* a path from s to v. In code, this lemma is the guard `d[u] < INF`.

Book: Sections 13.6.5 and 13.6.6.

Convergence and Path-Relaxation

LEMMA 13.7 (Convergence property). Suppose $s \rightsquigarrow u \to v$ is a shortest path, so $\delta(s,v)=\delta(s,u)+w(u,v)$. If at some moment $d[u]=\delta(s,u)$, then at any time *after* (u,v) is relaxed, $d[v]=\delta(s,v)$.

Proof. If $d[v]$ already equalled $\delta(s,v)$, Lemma 13.5 froze it. Otherwise $d[v]>\delta(s,v)=\delta(s,u)+w(u,v)=d[u]+w(u,v)$, so the test fires and sets $d[v]$ to exactly $\delta(s,v)$. ∎ This is the workhorse lemma of every correctness proof today.

LEMMA 13.8 (Path-relaxation property). Let $p=\langle v_0,\ldots,v_k\rangle$ be a shortest path from $s=v_0$. If the edges of p are relaxed **in order**, with any number of other relaxations interleaved anywhere, then afterward $d[v_k]=\delta(s,v_k)$ for ever.

Proof sketch. Induction on i: by optimal substructure $\delta(s,v_i)=\delta(s,v_{i-1})+w(v_{i-1},v_i)$, so Lemma 13.7 fires at each edge in turn. **The interleaving is free**: other relaxations only lower estimates, never below δ. That is what makes Bellman–Ford work.

Only the relative order matters. We need (v_0,v_1) before (v_1,v_2) before (v_2,v_3), not consecutively, and we never need to know what p is.

Book: Sections 13.6.7 and 13.6.8.

The Predecessor Subgraph Is a Tree

DEFINITION. The **predecessor subgraph** $G_\pi = (V_\pi, E_\pi)$ is defined by

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}, \quad E_\pi = \{(\pi[v], v) : v \in V_\pi \setminus \{s\}\}. \tag{13.6}$$

A **shortest-path tree** rooted at s contains exactly the vertices reachable from s, is a rooted tree with root s, and its unique path from s to each v is a shortest path in G.

In words, Equation (13.6) keeps every vertex that has a predecessor, plus s, and draws one edge into each such vertex from its predecessor.

LEMMA 13.9 (Predecessor-subgraph property). If no negative cycle is reachable from s, and at some moment $d[v] = \delta(s, v)$ for every v, then G_π is a shortest-path tree rooted at s.

Sketch. *Acyclic:* summing $d[c_i] \geq d[c_{i-1}] + w(c_{i-1}, c_i)$ round a cycle of G_π cancels the d-terms and forces $w(\text{cycle}) \leq 0$; negative contradicts the hypothesis, and zero is ruled out in Exercise 13.10. *Right vertex set:* $\pi[v] \neq$ NIL exactly when v is reachable. *Right paths:* walking back along π telescopes to $\delta(s, v)$. ∎

This lemma is the license for Section 13.10: follow π backward and you really get a shortest path, once the algorithm has terminated. Book: Section 13.6.9.

The Five Properties on One Page

Table 13.5 The five properties of relaxation, proved once and used by all three algorithms.

Property	Statement	What it licenses
Upper-bound (13.5)	$d[v] \geq \delta(s, v)$ always; once equal, frozen	never trusting d too early; stopping when nothing changes
No-path (13.6)	no route $\Rightarrow d[v] = \delta(s, v) = \infty$	the `d[u] < INF` guard in every program
Convergence (13.7)	$d[u] = \delta(s, u)$ and (u, v) on a shortest path, relax $(u, v) \Rightarrow d[v] = \delta(s, v)$	the induction in every correctness proof
Path-relaxation (13.8)	relax a shortest path's edges in order $\Rightarrow$ its end is solved	Bellman–Ford's $\lvert V \rvert - 1$ passes; the DAG algorithm's single pass
Predecessor-subgraph (13.9)	when all $d = \delta$, G_π is a shortest-path tree	recovering the route from π

Table 13.5 is the page to photograph. No algorithm today needs a correctness argument that does not reduce to this table.

When asked to prove an algorithm correct, name the property you invoke at each step: "by the convergence property", never "obviously".

Book: Section 13.6.10.

Dijkstra: the Idea

Non-negative edge weights. Published by Edsger Dijkstra in 1959 and reportedly designed in about twenty minutes in an Amsterdam café. The restriction is real, and Section 13.7.3 shows where it bites.

Greedy, like Prim. Prim's key for v is $\min\{w(u,v)\}$, the cost of *one edge*; Dijkstra's key is $\min\{d[u]+w(u,v) : u \in S\}$, the cost of *the whole journey*. Same shape, different key.

Like BFS. If every weight equals 1, "closest unsolved vertex" is "fewest edges away" and the priority queue becomes the FIFO queue of Chapter 10.

The method. Keep a set S of *solved* vertices and a priority queue Q keyed by $d[v]$. Repeatedly extract the closest unsolved vertex, add it to S, and relax every edge out of it.

> **KEY IDEA.** Dijkstra processes vertices in increasing order of their true distance from s. Once a vertex leaves Q, its d value is **final**. That is safe only because no edge weight is negative.

Book: Section 13.7.1.

Dijkstra's Pseudocode

```
DIJKSTRA(G, w, s)
 1  for each v ∈ V
 2      do d[v] ← ∞
 3  d[s] ← 0
 4  S ← ∅        ▷ set of solved vertices
 5  Q ← V
 6  while Q ≠ ∅
 7      do u ← EXTRACT-MIN(Q)
 8          S ← S ∪ {u}
 9          for each v ∈ Adj[u]
10              do if d[v] > d[u] + w(u, v)
11                  then d[v] ← d[u] + w(u, v)
```

relaxing edges

Figure 13.4 Dijkstra's algorithm: lines 1–3 initialize, lines 4–5 create the empty solved set S and the queue Q, and lines 6–8 repeatedly declare the closest unsolved vertex solved. The highlighted lines 9–11 are RELAX, applied to every edge out of that vertex.

In Figure 13.4: **lines 1–3** set $d[v] \leftarrow \infty$, then $d[s] \leftarrow 0$; **line 4** $S \leftarrow \varnothing$; **line 5** $Q \leftarrow V$, keyed by d. **Lines 6–8**: while Q is non-empty, extract the minimum u and add it to S; the body runs $|V|$ times and the first extraction is s.

Lines 9–11 relax every edge out of u. *Setting $d[v]$ updates Q (needs `Decrease-Key`).* The figure never sets π; every implementation adds $\pi[v] \leftarrow u$ beside line 11.

Book: Section 13.7.2.

Why the Weights Must Be Non-Negative

Dijkstra's greed rests on one assumption: **taking an extra edge can never make a journey cheaper.** If some $w<0$, a long detour can end with a large rebate.

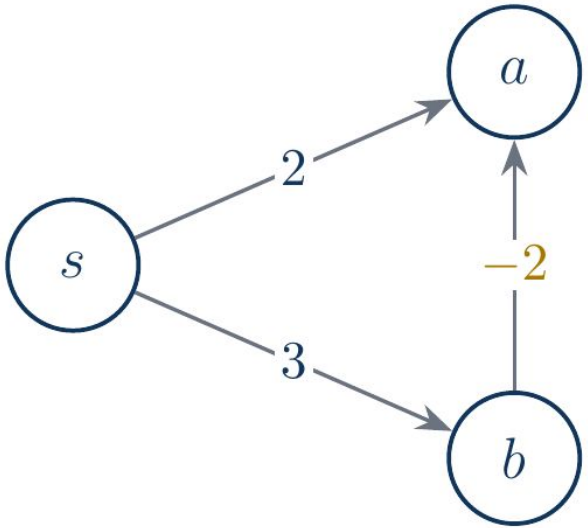


Figure 13.5 A graph on which Dijkstra silently returns the wrong answer. There is no cycle, so no negative cycle, and the true distances are $\delta(s,a)=1$ and $\delta(s,b)=3$. Dijkstra finalizes a at 2, before it has ever looked at the edge (b,a), and never revisits it.

The trace of Figure 13.5**.** (1) Extract s at 0; relax to $d[a]=2$, $d[b]=3$. (2) Extract a at 2, since $2<3$: a **is final.** (3) Extract b at 3; relaxing (b, a) offers $3+(-2)=1<2$, but a has left the queue. Reported 2; truth 1.

Adding a constant c to every weight does not fix it: a k-edge path gains $k\,c$, so paths with many edges are penalized more and the cheapest path can change (Exercise 13.6).

The program of Section 13.7.3 prints Dijkstra's 2 beside Bellman–Ford's correct 1: no crash, no warning, no error code. Book: Section 13.7.3.

Panel (a) — After Initialization

Vertices s, u, v, x, y; ten directed edges: $w(s,u)=10$, $w(s,x)=5$, $w(u,v)=1$, $w(u,x)=2$, $w(x,u)=3$, $w(x,v)=9$, $w(x,y)=2$, $w(v,y)=4$, $w(y,v)=6$, $w(y,s)=7$. **White** = still in Q; **amber** = just extracted; **navy** = in S; bold arrows are the edges of G_π.

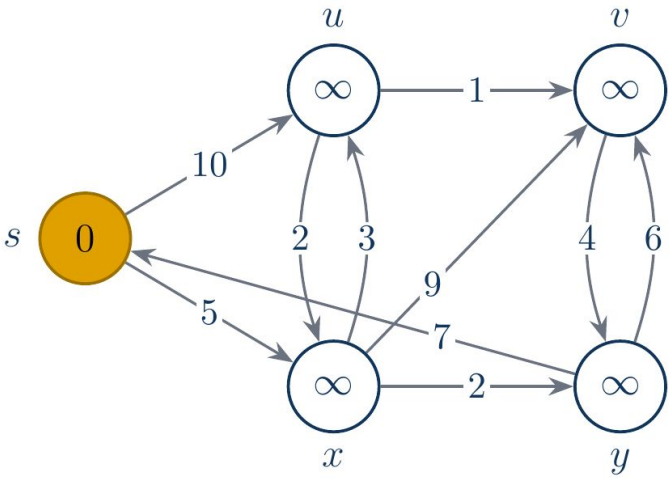


Figure 13.6 The Dijkstra example, panel (a): the graph immediately after `INITIALIZE-SINGLE-SOURCE`. $d[s]=0$ (amber, about to be extracted) and every other estimate is ∞. $Q=\{s=0, u=\infty, v=\infty, x=\infty, y=\infty\}$.

The priority queue, which Figure 13.6 implies but does not draw: $Q=\{s=0, u=\infty, v=\infty, x=\infty, y=\infty\}$ and $S=\varnothing$.

Book: Section 13.7.4.

Step 1 — Extract s, $d=0$

(s,u), $w=10$: $\infty>0+10$, so $d[u]\leftarrow 10$, $\pi[u]\leftarrow s$. (s,x), $w=5$: $\infty>0+5$, so $d[x]\leftarrow 5$, $\pi[x]\leftarrow s$.

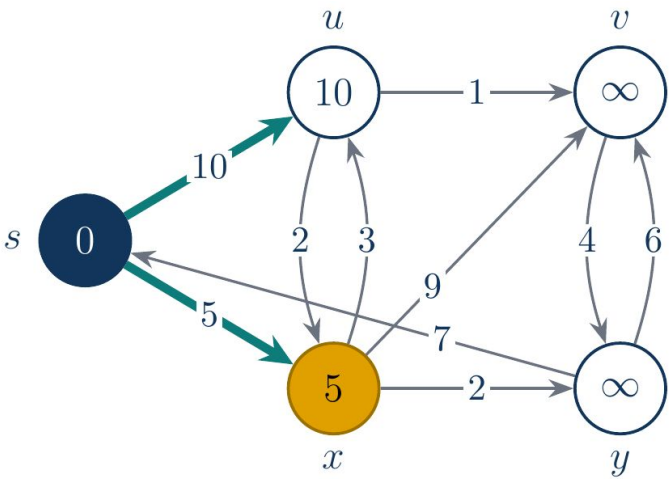


Figure 13.7 Panel (b): s is navy (solved), the two edges out of it have been relaxed, and x — amber — is the next vertex to be extracted because $5<10$. Note that u already has an estimate of 10; it will be improved before u is ever extracted.

Now $Q=\{x=5, u=10, v=\infty, y=\infty\}$ and $S=\{s\}$, the state of Figure 13.7.

Book: Section 13.7.4, Step 1.

Step 2 — Extract x, $d=5$

(x,u), $w=3$: $10>5+3=8$, so $d[u]\leftarrow 8$, $\pi[u]\leftarrow x$ — **the first `Decrease-Key`**. (x,v), $w=9$: $d[v]\leftarrow 14$, $\pi[v]\leftarrow x$. (x,y), $w=2$: $d[y]\leftarrow 7$, $\pi[y]\leftarrow x$.

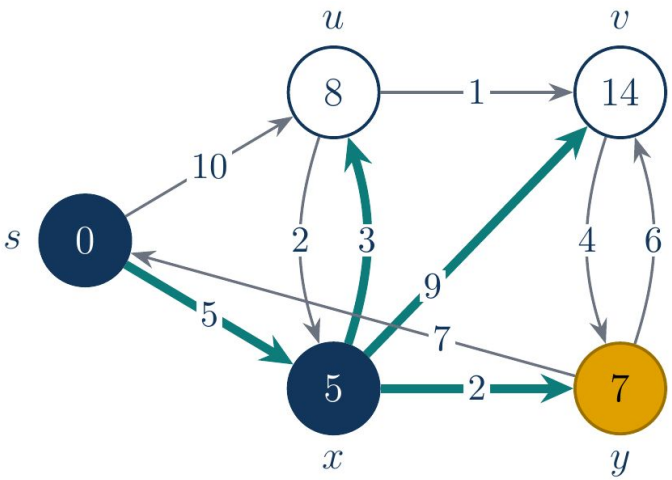


Figure 13.8 Panel (c): s and x are navy. The estimate for u has dropped from 10 to 8 and its predecessor edge has swung from (s,u) to (x,u). y is amber: at 7 it is the smallest key left.

Now $Q=\{y=7,u=8,v=14\}$ and $S=\{s,x\}$: Figure 13.8.

Book: Section 13.7.4, Step 2.

Step 3 — Extract y, $d=7$

(y, v), $w=6$: $14>7+6=13$, so $d[v] \leftarrow 13$, $\pi[v] \leftarrow y$ — another `Decrease-Key`. (y, s), $w=7$: $7+7=14>0$, so nothing happens.

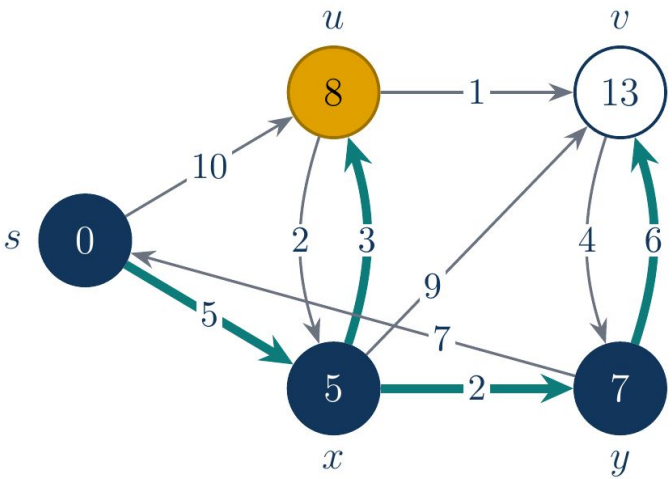


Figure 13.9 Panel (d): s, x and y are navy. v's estimate has improved from 14 to 13 and its bold predecessor arrow has swung from x to y. u, amber at 8, is next.

Now $Q=\{u=8, v=13\}$ and $S=\{s, x, y\}$: Figure 13.9.

Book: Section 13.7.4, Step 3.

Step 4 — Extract u, $d=8$

(u,v), $w=1$: $13>8+1=9$, so $d[v]\leftarrow 9$, $\pi[v]\leftarrow u$ — a third `Decrease-Key`; v has gone $\infty\to 14\to 13\to 9$. (u,x), $w=2$: $8+2=10>5$, nothing.

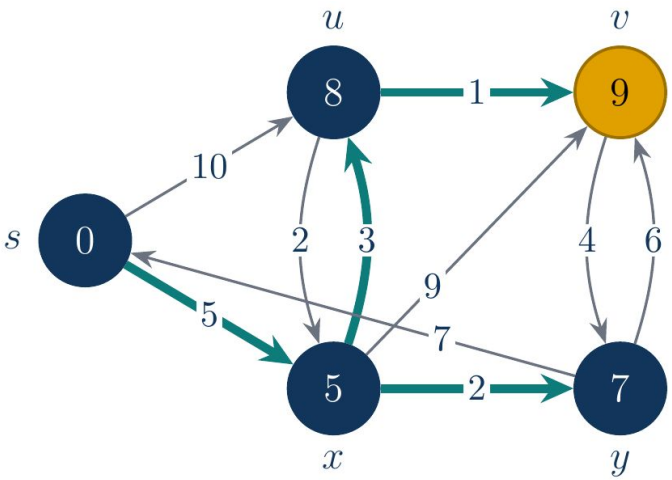


Figure 13.10 Panel (e): four vertices are navy, v is amber at 9, and its predecessor arrow now comes from u. Every edge in the graph has been relaxed at least once except the two leaving v.

Now $Q=\{v=9\}$ and $S=\{s,x,y,u\}$: Figure 13.10.

Book: Section 13.7.4, Step 4.

Step 5 — Extract v, $d=9$

(v, y), $w=4$: $9+4=13>7$, nothing happens. Q is empty; the algorithm halts.

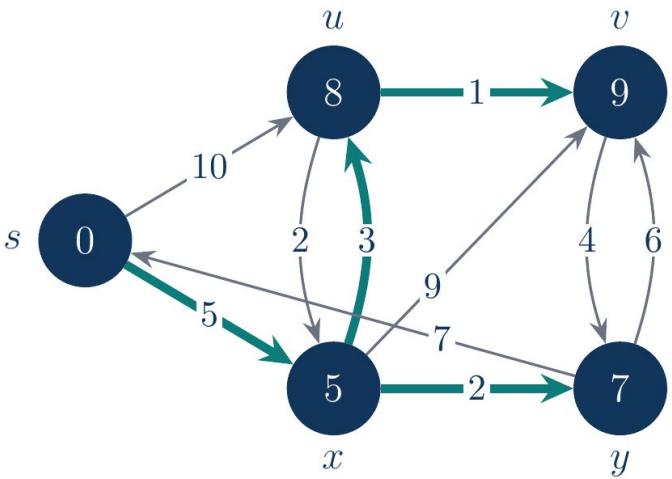


Figure 13.11 Panel (f): every vertex navy, the algorithm finished. The four bold edges (s, x), (x, u), (u, v) and (x, y) are the predecessor subgraph G_π, the shortest-path tree rooted at s.

In Figure 13.11, reading a route is a matter of walking the bold arrows backward from any vertex to s. $d[v]=9$, the number that Table 13.1 found by listing six paths.

Book: Section 13.7.4, Step 5.

The Whole Run in Two Tables

Table 13.6 The complete run of Dijkstra on the example graph, one row per iteration of the `while` loop.

Step	Extracted u	$d[u]$	Edges relaxed (result)	Q After the step	S After
0	—	—	(initialize)	$s=0, u=\infty, v=\infty, x=$	$\varnothing$
1	s	0	$(s,u)\to 10; (s,x)\to 5$	$x=5, u=10, v=\infty, y=$	$\{s\}$
2	x	5	$(x,u)\to 8; (x,v)\to 14; (x,y)\to 7$	$y=7, u=8, v=14$	$\{s,x\}$
3	y	7	$(y,v)\to 13; (y,s)$ no change	$u=8, v=13$	$\{s,x,y\}$
4	u	8	$(u,v)\to 9; (u,x)$ no change	$v=9$	$\{s,x,y,u$
5	v	9	(v,y) no change	empty	all

Table 13.7 The output of Dijkstra on the example graph: distance, predecessor, and the route found by walking π backward.

v	s	u	v	x	y
$d[v]=\delta(s,v)$	0	8	9	5	7
$\pi[v]$	NIL	x	u	s	x
shortest path	s	s,x,u	s,x,u,v	s,x	s,x,y

In Table 13.6 the keys in Q only ever fall, while the extracted keys $0, 5, 7, 8, 9$ never fall. Table 13.7 agrees with Table 13.1.

Book: Section 13.7.5.

The Program

The standard `priority_queue` has no `DecreaseKey`, so Listing 13.1 uses **lazy deletion**: when $d[v]$ falls, push a second copy of v with the smaller key and discard stale copies on extraction. The queue holds up to $|E|$ entries, so the cost is $O(E \lg E) = O(E \lg V)$.

Listing 13.1 Dijkstra's algorithm with a binary-heap priority queue on the example graph, printing the queue at every step. Compile with `g++ -std=c++17 -O2 -o dijkstra dijkstra.cpp`.

```cpp
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>

using namespace std;

const long long INF = LLONG_MAX / 4;          // "infinity" we can still add to safely

struct Edge { int to; long long w; };

// The Dijkstra example graph: vertices s,u,v,x,y.
const vector<string> NAME = {"s", "u", "v", "x", "y"};
enum { S = 0, U = 1, V = 2, X = 3, Y = 4 };

// Print the priority queue without destroying it: we copy it and drain the copy.
// Stale entries (a vertex already in S, or a key that a later DecreaseKey beat)
// are skipped, so what is printed is the logical contents of Q.
string queueContents(priority_queue<pair<long long,int>,
                                    vector<pair<long long,int>>,
                                    greater<pair<long long,int>>> q,
                     const vector<bool>& done, const vector<long long>& d) {
    vector<pair<long long,int>> live;
    vector<bool> seen(done.size(), false);
    while (!q.empty()) {
```

```
        auto [key, v] = q.top(); q.pop();
        if (done[v] || seen[v] || key > d[v]) continue;
        seen[v] = true;
        live.push_back({key, v});
    }
    sort(live.begin(), live.end());
    string out = "{";
    for (size_t i = 0; i < live.size(); ++i) {
        if (i) out += ", ";
        out += NAME[live[i].second] + "=" +
               (live[i].first >= INF ? string("INF") : to_string(live[i].first));
    }
    return out + "}";
}
```

Listing 13.1 (Continued) *Dijkstra's algorithm with a binary-heap priority queue on the example graph, printing the queue at every step. Compile with g++ -std=c++17 -O2 -o dijkstra dijkstra.cpp.*

```
int main() {
    int n = 5;
    vector<vector<Edge>> adj(n);
    auto add = [&](int a, int b, long long w) { adj[a].push_back({b, w}); };
    add(S, U, 10); add(S, X, 5);
    add(U, V, 1);  add(U, X, 2);
    add(X, U, 3);  add(X, V, 9); add(X, Y, 2);
    add(V, Y, 4);
    add(Y, V, 6);  add(Y, S, 7);

    vector<long long> d(n, INF);
    vector<int> pi(n, -1);
    vector<bool> done(n, false);                    // done[v] == "v is in S"
    d[S] = 0;

    priority_queue<pair<long long,int>, vector<pair<long long,int>>,
                   greater<pair<long long,int>>> Q;
    for (int v = 0; v < n; ++v) Q.push({d[v], v});

    cout << "initial Q = " << queueContents(Q, done, d) << "\n\n";

    int step = 0;
    while (!Q.empty()) {
        auto [key, u] = Q.top(); Q.pop();
        if (done[u]) continue;                      // a stale copy: ignore it
        if (key > d[u]) continue;                   // superseded by a smaller key
        done[u] = true;
        cout << "step " << ++step << ": EXTRACT-MIN gives " << NAME[u]
```

```
            << " with d=" << (d[u] >= INF ? string("INF") : to_string(d[u])) << "\n";
        if (d[u] >= INF) { cout << "    unreachable; no edges relaxed\n\n"; continue; }
        for (const Edge& e : adj[u]) {
            long long cand = d[u] + e.w;
            cout << "    relax (" << NAME[u] << "," << NAME[e.to] << ") w=" << e.w
                 << ": d[" << NAME[e.to] << "]="
                 << (d[e.to] >= INF ? string("INF") : to_string(d[e.to]))
                 << " vs " << d[u] << "+" << e.w << "=" << cand << " -> ";
            if (cand < d[e.to]) {
                d[e.to] = cand; pi[e.to] = u;
                Q.push({cand, e.to});
                cout << "improved, d[" << NAME[e.to] << "]=" << cand
                     << ", pi[" << NAME[e.to] << "]=" << NAME[u] << "\n";
            } else {
                cout << "no change\n";
            }
        }
        cout << "    Q = " << queueContents(Q, done, d) << "\n\n";
```

Listing 13.1 (Continued) *Dijkstra's algorithm with a binary-heap priority queue on the example graph, printing the queue at every step. Compile with g++ -std=c++17 -O2 -o dijkstra dijkstra.cpp.*

```
    }

    cout << "vertex |  d  | pi | shortest path from s\n";
    cout << "-------+-----+----+----------------------\n";
    for (int v = 0; v < n; ++v) {
        vector<string> path;
        for (int t = v; t != -1; t = pi[t]) path.push_back(NAME[t]);
        reverse(path.begin(), path.end());
        string p;
        for (size_t i = 0; i < path.size(); ++i) { if (i) p += " -> "; p += path[i]; }
        cout << "   " << NAME[v] << "   |" << setw(4) << d[v] << " |  "
             << (pi[v] == -1 ? string("-") : NAME[pi[v]]) << " | " << p << "\n";
    }
    return 0;
}
```

Book: Section 13.7.6.

What It Prints

```
initial Q = {s=0, u=INF, v=INF, x=INF, y=INF}

step 1: EXTRACT-MIN gives s with d=0
    relax (s,u) w=10: d[u]=INF vs 0+10=10 -> improved, d[u]=10, pi[u]=s
    relax (s,x) w=5: d[x]=INF vs 0+5=5 -> improved, d[x]=5, pi[x]=s
    Q = {x=5, u=10, v=INF, y=INF}

step 2: EXTRACT-MIN gives x with d=5
    relax (x,u) w=3: d[u]=10 vs 5+3=8 -> improved, d[u]=8, pi[u]=x
    relax (x,v) w=9: d[v]=INF vs 5+9=14 -> improved, d[v]=14, pi[v]=x
    relax (x,y) w=2: d[y]=INF vs 5+2=7 -> improved, d[y]=7, pi[y]=x
    Q = {y=7, u=8, v=14}

step 3: EXTRACT-MIN gives y with d=7
    relax (y,v) w=6: d[v]=14 vs 7+6=13 -> improved, d[v]=13, pi[v]=y
    relax (y,s) w=7: d[s]=0 vs 7+7=14 -> no change
    Q = {u=8, v=13}

step 4: EXTRACT-MIN gives u with d=8
    relax (u,v) w=1: d[v]=13 vs 8+1=9 -> improved, d[v]=9, pi[v]=u
    relax (u,x) w=2: d[x]=5 vs 8+2=10 -> no change
    Q = {v=9}

step 5: EXTRACT-MIN gives v with d=9
    relax (v,y) w=4: d[y]=7 vs 9+4=13 -> no change
    Q = {}

vertex |  d  | pi | shortest path from s
```

```
-------+-----+----+----------------------
   s   |   0 |  - | s
   u   |   8 |  x | s -> x -> u
   v   |   9 |  u | s -> x -> u -> v
   x   |   5 |  s | s -> x
   y   |   7 |  x | s -> x -> y
```

Compare it line for line with the six panels and with Table 13.6: the same three `Decrease-Key` moments and the same three relaxations that change nothing.

Book: Section 13.7.7.

Correctness: the Claim and the Picture

THEOREM 13.10. Run Dijkstra with $w(u,v) \geq 0$ for every edge. At the top of each iteration of the `while` loop, $d[v] = \delta(s,v)$ for every $v \in S$. Hence at termination $d[v] = \delta(s,v)$ for every v, and G_π is a shortest-path tree rooted at s.

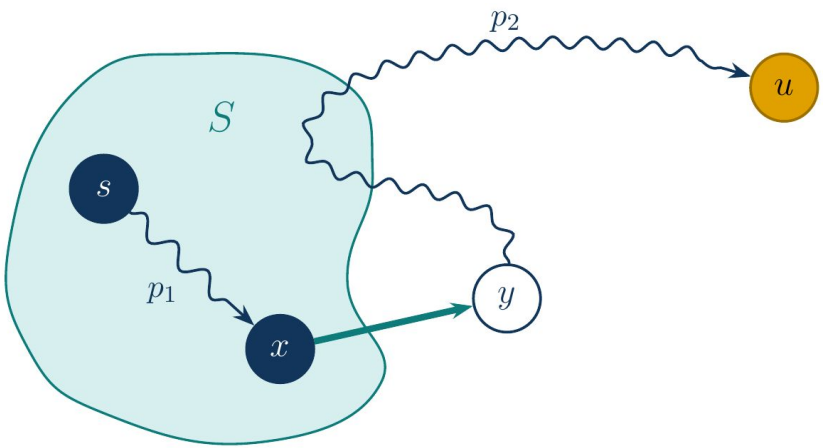


Figure 13.12 The picture behind Dijkstra's correctness proof: the light-teal region is the solved set S, and u is the first vertex allegedly extracted with a wrong estimate. The true shortest path leaves S at the edge (x, y), so it reads $s \overset{p_1}{\rightsquigarrow} x \to y \overset{p_2}{\rightsquigarrow} u$.

The proof is by contradiction and runs inside Figure 13.12: *whenever u is added to S, $d[u] = \delta(s,u)$, and equality is maintained thereafter.*

Book: Section 13.7.8.

The Proof, in Six Steps

Setup. Let u be the **first** vertex extracted with $d[u] \neq \delta(s,u)$; by Lemma 13.5,

$$d[u] > \delta(s,u). \tag{13.7}$$

1. $u \neq s$, since $d[s] = 0 = \delta(s,s)$. **2.** u is reachable, else $d[u] = \infty = \delta(s,u)$, against Equation (13.7).

3. Decompose at the boundary of S: y is the first vertex of the shortest path p not in S, and $x \in S$ its predecessor on p.

4. $d[y] = \delta(s,y)$**:** x was extracted earlier, so $d[x] = \delta(s,x)$; (x,y) was relaxed then; by optimal substructure $\delta(s,y) = \delta(s,x) + w(x,y)$; Lemma 13.7 applies.

5. $\delta(s,u) = \delta(s,y) + w(p_2)$ and every edge of p_2 is non-negative, so

$$\delta(s,y) \leq \delta(s,u). \tag{13.8}$$

> **Step 5 is the only step that uses $w \geq 0$.** If p_2 could weigh less than zero, the true route to u could pass through a vertex *further away* than u itself, exactly as in Figure 13.5.

6. Contradiction. By step 4, Equation (13.8) and Equation (13.7): $d[y] = \delta(s,y) \leq \delta(s,u) < d[u]$. Both were in Q and the algorithm extracted the **minimum**, which needs $d[u] \leq d[y]$. ∎

Book: Section 13.7.8.

Extracted Keys Never Fall

COROLLARY 13.11. The sequence of keys returned by successive `Extract-Min` calls in Dijkstra is non-decreasing.

Proof, two consecutive extractions u then u'. When u is extracted, every key in Q is $\geq d[u]$. A relaxation of (u, v) sets $d[v] \leftarrow d[u] + w(u, v) \geq d[u]$, because $w \geq 0$. So every key is still $\geq d[u]$, and $d[u'] \geq d[u]$. ∎

In our trace the extracted keys were $0, 5, 7, 8, 9$: monotone, as promised.

The early exit. For the single-pair problem you may stop the moment t is extracted: every later key is at least as large, so no later relaxation can lower $d[t]$.

With a negative edge a relaxation can plant a key below $d[u]$, and extraction goes backward, as in the Profitina Laundry cost graph of Section 13.12.2. Book: Section 13.7.8.

Running Time

Count the queue operations: $|V|$ `Insert`, $|V|$ `Extract-Min` (one per iteration), and at most $|E|$ `Decrease-Key` (each edge is examined once, when its tail is extracted):

$$T = |V| \cdot T_{Extract\text{-}Min} + |E| \cdot T_{Decrease\text{-}Key}. \tag{13.9}$$

Table 13.8 Dijkstra's running time for three priority-queue implementations, obtained by substituting each queue's costs into the operation count.

Q Implemented as	T(Extract-Min)	T(Decrease-Key)	Total
unsorted array	$\Theta(V)$	$\Theta(1)$	$\Theta(V^2)$
binary heap	$\Theta(\lg V)$	$\Theta(\lg V)$	$O((V+E)\lg V) = O(E$
Fibonacci heap	$O(\lg V)$ amortized	$\Theta(1)$ amortized	$O(V \lg V + E)$

Table 13.8 substitutes into Equation (13.9). **Array** for a *dense* graph: no heap overhead, and you cannot beat $\Theta(V^2)$ with $\Theta(V^2)$ edges to read. **Binary heap** for a *sparse* graph. **Fibonacci heap**: the best bound, but large constants; use it only after measuring.

Measured (Section 13.7.10): on sparse graphs ($m = 8n$) the heap is four to twelve times faster; on dense graphs ($m = n^2/4$) the two are close. Three implementations agreed on 2000 random graphs.

The crossover between array and heap is at roughly $E = \Theta(V^2 / \lg V)$. Book: Sections 13.7.9 and 13.7.10.

BFS, Prim and Dijkstra

Table 13.9 Breadth-first search, Prim and Dijkstra compared: the same skeleton with three different keys.

	BFS (Ch. 10)	Prim–Jarník (Ch. 11)	Dijkstra (Ch. 13)
Question answered	fewest edges from s	cheapest spanning tree	cheapest route from s
Frontier structure	FIFO queue	priority queue	priority queue
Key of v	(none; order of arrival)	$\min\{w(u,v) : u \in \text{tree}\}$	$\min\{d[u] + w(u,v) : u \in S\}$
Update rule	enqueue if white	$key[v] \leftarrow w(u,v)$	$d[v] \leftarrow d[u] + w(u,v)$
Needs $w \geq 0$?	no weights at all	no — any weights allowed	**yes**
Graph direction	either	undirected	directed (or either)
Output	BFS tree + hop counts	minimum spanning tree	shortest-path tree + distances
Time (binary heap)	$O(V+E)$	$O(E \lg V)$	$O(E \lg V)$

The line of Table 13.9 that separates Prim from Dijkstra is the key: Prim measures **one edge**, Dijkstra measures **the whole journey**.

That is also why Prim tolerates negative weights and Dijkstra does not: Prim never *adds* two weights, so it never needs the triangle inequality. Same skeleton, different bones.

Book: Section 13.7.11.

Bellman–Ford: Why and How

Dijkstra's algorithm doesn't work when there are negative edges, because *we cannot be greedy on the assumption that the lengths of paths will only increase in the future.* Bellman (1958), Ford (1956) and Moore (1957) found the same cure: **abandon greed**, relax *every* edge $|V| - 1$ times, and let Lemma 13.8 do the rest.

```
BELLMAN-FORD(G, w, s)
01  for each v ∈ V[G]
02      d[v] ← ∞
03      π[v] ← NIL
04  d[s] ← 0
05  for i ← 1 to |V[G]| − 1 do
06      for each edge (u, v) ∈ E[G] do
07          RELAX(u, v, w)
08  for each edge (u, v) ∈ E[G] do
09      if d[v] > d[u] + w(u, v) then return false
10  return true
```

Figure 13.13 Bellman–Ford: lines 01–04 initialize, lines 05–07 make $|V| - 1$ passes that each relax every edge, and lines 08–09 are the detector, returning `false` if any edge can still be relaxed. Line 10 returns `true` when the answer is trustworthy.

Figure 13.13 has no queue and no ordering, because the algorithm makes no decision at all. Lines 01–04 are exactly `INITIALIZE-SINGLE-SOURCE`.

In exchange for being slower it is correct with negative edges and can detect a negative cycle instead of returning nonsense. Book: Sections 13.8.1 and 13.8.2.

Why $|V|-1$ Passes Are Enough

THEOREM 13.12. With no reachable negative cycle, after the i-th pass $d[v]=\delta(s,v)$ for every v whose shortest path uses at most i edges. Hence after $|V|-1$ passes, $d[v]=\delta(s,v)$ for every v.

Proof. By Theorem 13.4 there is a *simple* shortest path $\langle v_0,\dots,v_k\rangle$ with $k\le|V|-1$. Pass i relaxes every edge, so in particular (v_{i-1},v_i). After k passes the edges are relaxed in order, and Lemma 13.8 gives $d[v_k]=\delta(s,v_k)$. ∎

Edge-count form (Theorem 13.12′). Let $\delta_i(s,u)$ be the cheapest weight over paths of *at most i edges*. The best such path either uses $\le i-1$ edges, or ends with an edge (z,u):

$$\delta_i(s,u)=\min\Big(\delta_{i-1}(s,u),\ \min_{(z,u)\in E}\big(\delta_{i-1}(s,z)+w(z,u)\big)\Big). \tag{13.10}$$

Pass i relaxes every edge, so it catches both cases of Equation (13.10): after pass i, $\delta(s,u)\le d[u]\le\delta_i(s,u)$. Since $\delta_{n-1}=\delta$ (Theorem 13.4), the two ends meet.

Why an inequality? A single pass can carry an estimate along several edges at once when the edge list is in a lucky order; $|V|-1$ is the worst case. Book: Section 13.8.3.

Bellman–Ford: Panel (a)

Vertices s, t, x, y, z; ten edges, **three negative**, relaxed in this order: $w(t,x)=5$, $w(t,y)=8$, $w(t,z)=-4$, $w(x,t)=-2$, $w(y,x)=-3$, $w(y,z)=9$, $w(z,x)=7$, $w(z,s)=2$, $w(s,t)=6$, $w(s,y)=7$. The edges out of s come **last**: deliberately the worst case.

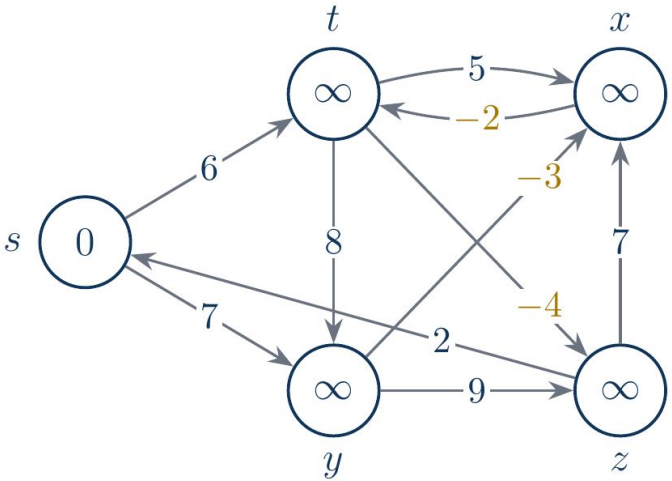


Figure 13.14 The Bellman–Ford example, panel (a): initialization. Three of the ten edges carry negative weights, which is exactly why Dijkstra may not be used here. There is no negative *cycle*: $t \to z \to s \to t$ costs $-4+2+6=4>0$, and $t \to x \to t$ costs $5+(-2)=3>0$.

Check the two cycles named in Figure 13.14 before starting: a negative edge is fine; a negative cycle would make the question meaningless.

Book: Section 13.8.4.

Pass 1 — Only s's Edges Bite

The first eight edges do nothing: their tails are still at ∞. Then (s,t), $w=6$: $d[t] \leftarrow 6$, $\pi[t] \leftarrow s$; and (s,y), $w=7$: $d[y] \leftarrow 7$, $\pi[y] \leftarrow s$.

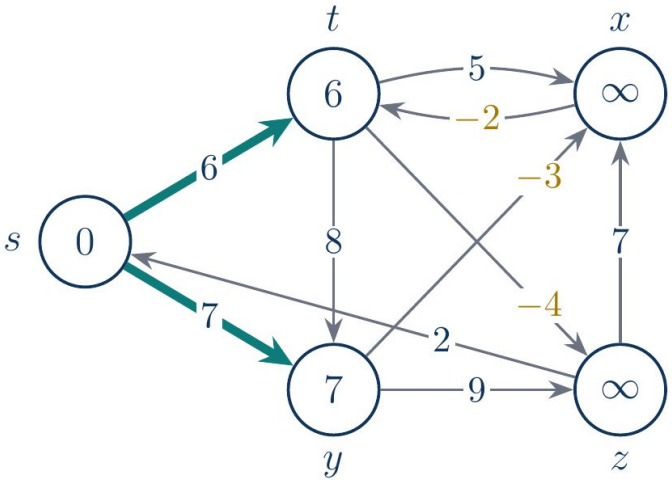


Figure 13.15 Panel (b): after pass 1. Only the two edges leaving s have had any effect, because they are the only ones whose tail had a finite estimate when they were examined.

After pass 1: $d=(s:0, t:6, x:\infty, y:7, z:\infty)$, as in Figure 13.15.

Book: Section 13.8.4, Pass 1.

Pass 2 — Two Negative Edges Pay

(t,x): $d[x] \leftarrow 6+5=11$. (t,y): $14>7$, no. (t,z): $d[z] \leftarrow 6-4=2$. (x,t): $11-2=9>6$, no. (y,x): $11>7-3=4$, so $d[x] \leftarrow 4$, $\pi[x] \leftarrow y$: **improved twice in one pass**. The rest change nothing.

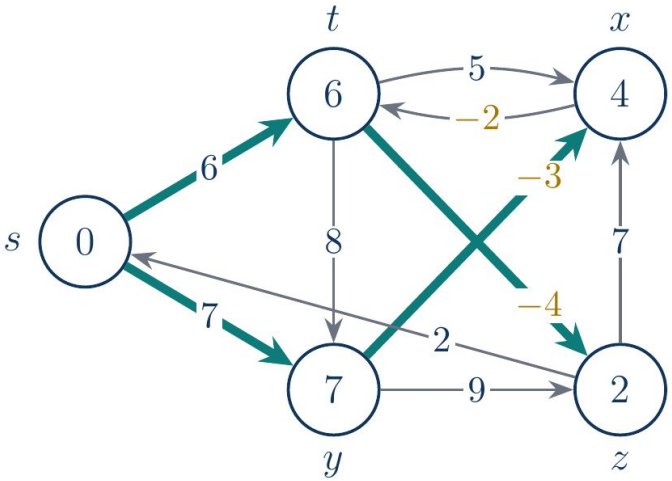


Figure 13.16 Panel (c): after pass 2. Every vertex now has a finite estimate. $d[x]$ settled at 4 via y and the negative edge (y,x), and $d[z]$ at 2 via t and the negative edge (t,z). Two of the five estimates are still wrong.

After pass 2: $d=(0,6,4,7,2)$, as in Figure 13.16.

Book: Section 13.8.4, Pass 2.

Pass 3 — One Edge, Three Hops Deep

Exactly one relaxation succeeds: (x,t), $w=-2$: $6>4-2=2$, so $d[t]\leftarrow 2$, $\pi[t]\leftarrow x$. The route $s\to y\to x\to t$ costs $7+(-3)+(-2)=2$ and beats the direct edge of weight 6.

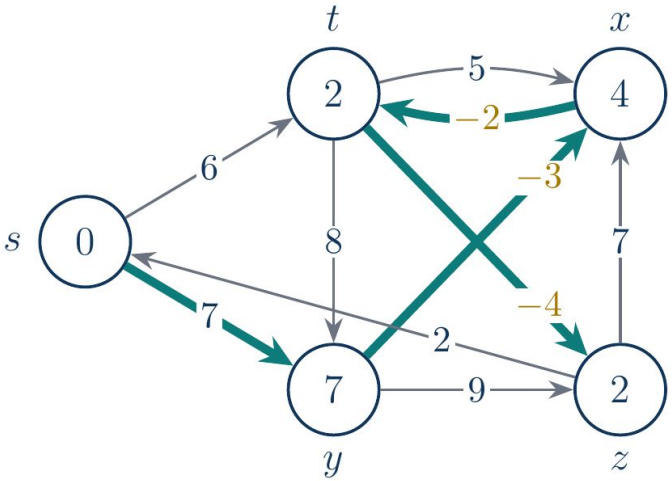


Figure 13.17 Panel (d): after pass 3. The estimate for t has fallen from 6 to 2, because the route $s\to y\to x\to t$ costs $7+(-3)+(-2)=2$, beating the direct edge of weight 6. Its bold predecessor arrow has swung from s to x.

After pass 3: $d=(0,2,4,7,2)$. The improvement in Figure 13.17 was invisible until pass 3 because it needs three edges.

Book: Section 13.8.4, Pass 3.

Pass 4 — The Final Answer

Again exactly one relaxation succeeds: (t, z), $w = -4$: $2 > 2 - 4 = -2$, so $d[z] \leftarrow -2$; $\pi[z]$ stays t.

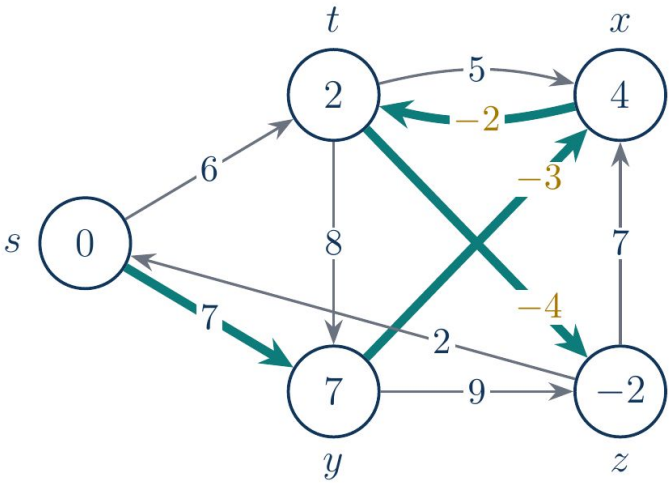


Figure 13.18 Panel (e), after pass 4: $\delta(s, z) = -2$ by the four-edge route $s \to y \to x \to t \to z$, of weight $7 + (-3) + (-2) + (-4)$. Four edges is the most a simple path in a five-vertex graph can have, so all $|V| - 1 = 4$ passes were needed.

The detector pass on Figure 13.18: (y, x) gives $4 = d[x]$, (x, t) gives $2 = d[t]$, (t, z) gives $-2 = d[z]$, (z, s) gives $0 = d[s]$. Nothing improves; the algorithm returns `true`.

Book: Section 13.8.4, Pass 4 and the detector pass.

The Run, Pass by Pass

Table 13.10 The complete run of Bellman–Ford on the example graph, one row per pass over all ten edges; only the value at the end of each pass is shown.

After	$d[s]$	$d[t]$	$d[x]$	$d[y]$	$d[z]$	What changed
init	0	∞	∞	∞	∞	—
pass 1	0	6	∞	7	∞	$(s,t),(s,y)$
pass 2	0	6	4	7	2	$(t,x)\to 11,(t,z)\to 2,(y,x)\to 4$
pass 3	0	2	4	7	2	$(x,t)\to 2$
pass 4	0	2	4	7	-2	$(t,z)\to -2$
detect	0	2	4	7	-2	nothing — returns `true`

Table 13.11 Bellman–Ford's output on the example graph. Three of the four non-trivial routes use y and the negative edge (y,x).

v	s	t	x	y	z
$\delta(s,v)$	0	2	4	7	-2
$\pi[v]$	NIL	x	y	s	t
shortest path	s	s,y,x,t	s,y,x	s,y	s,y,x,t,z

In Table 13.10, $d[x]$ went to 11 and then to 4 *within pass 2*. Table 13.11 is a structure that a greedy method which finalizes vertices early can easily miss.

Book: Section 13.8.5.

The Program

Listing 13.2 prints every pass. Its second half lowers $w(z,s)$ from 2 to -8, creating the cycle $s \to t \to z \to s$ of weight $6+(-4)+(-8)=-6$, and the detector fires.

Listing 13.2 Bellman–Ford with negative-cycle detection on the example graph, printing every pass; the second half of `main` creates a negative cycle and shows the detector firing. Compile with `g++ -std=c++17 -O2 -o bellman bellman.cpp`.

```cpp
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <algorithm>
#include <climits>

using namespace std;

const long long INF = LLONG_MAX / 4;

struct Edge { int u, v; long long w; };

struct Graph {
    vector<string> name;
    vector<Edge>   edges;
    int n() const { return (int)name.size(); }
    int id(const string& s) const {
        return (int)(find(name.begin(), name.end(), s) - name.begin());
    }
};

string show(long long x) { return x >= INF ? "INF" : to_string(x); }

void printRow(const string& tag, const Graph& g, const vector<long long>& d) {
    cout << setw(9) << left << tag << right;
    for (int v = 0; v < g.n(); ++v)
        cout << setw(6) << (g.name[v] + "=" + show(d[v]));
```

```
    cout << "\n";
}

// Returns true when no negative cycle is reachable from s.
// d and pi are filled in either way.
bool bellmanFord(const Graph& g, int s, vector<long long>& d, vector<int>& pi,
                 bool verbose) {
    d.assign(g.n(), INF);
    pi.assign(g.n(), -1);
    d[s] = 0;
    if (verbose) printRow("init", g, d);
```

Listing 13.2 (Continued) *Bellman–Ford with negative-cycle detection on the example graph, printing every pass; the second half of main creates a negative cycle and shows the detector firing. Compile with g++ -std=c++17 -O2 -o bellman bellman.cpp.*

```
    for (int pass = 1; pass <= g.n() - 1; ++pass) {
        bool changed = false;
        for (const Edge& e : g.edges) {
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) {   // RELAX(u,v,w)
                d[e.v] = d[e.u] + e.w;
                pi[e.v] = e.u;
                changed = true;
                if (verbose)
                    cout << "   pass " << pass << ": relax ("
                         << g.name[e.u] << "," << g.name[e.v] << ") w=" << e.w
                         << " -> d[" << g.name[e.v] << "]=" << d[e.v] << "\n";
            }
        }
        if (verbose) printRow("pass " + to_string(pass), g, d);
        if (!changed) {                    // nothing moved: later passes cannot move anything
            if (verbose) cout << "   (no change in this pass; the rest are skipped)\n";
            break;
        }
    }

    for (const Edge& e : g.edges)          // the |V|-th pass: the detector
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) {
            if (verbose)
                cout << "   edge (" << g.name[e.u] << "," << g.name[e.v]
                     << ") still relaxes: " << d[e.u] << "+" << e.w
                     << " < " << d[e.v] << "  => NEGATIVE CYCLE\n";
            return false;
        }
```

```
    return true;
}

int main() {
    // The Bellman-Ford example graph, edges in the order used in the trace.
    Graph g;
    g.name = {"s", "t", "x", "y", "z"};
    auto E = [&](const char* a, const char* b, long long w) {
        g.edges.push_back({g.id(a), g.id(b), w});
    };
    E("t","x", 5); E("t","y", 8); E("t","z",-4);
    E("x","t",-2);
    E("y","x",-3); E("y","z", 9);
    E("z","x", 7); E("z","s", 2);
    E("s","t", 6); E("s","y", 7);
```

Listing 13.2 (Continued) *Bellman–Ford with negative-cycle detection on the example graph, printing every pass; the second half of main creates a negative cycle and shows the detector firing. Compile with g++ -std=c++17 -O2 -o bellman bellman.cpp.*

```
    cout << "=== the example graph ===\n";
    vector<long long> d; vector<int> pi;
    bool ok = bellmanFord(g, g.id("s"), d, pi, true);
    cout << "returns " << (ok ? "true" : "false") << "\n\n";

    cout << "vertex |  d  | pi | shortest path from s\n";
    cout << "-------+-----+----+---------------------\n";
    for (int v = 0; v < g.n(); ++v) {
        vector<string> path;
        for (int t = v; t != -1; t = pi[t]) path.push_back(g.name[t]);
        reverse(path.begin(), path.end());
        string p;
        for (size_t i = 0; i < path.size(); ++i) { if (i) p += " -> "; p += path[i]; }
        cout << "   " << g.name[v] << "   |" << setw(4) << d[v] << " |  "
             << (pi[v] == -1 ? string("-") : g.name[pi[v]]) << " | " << p << "\n";
    }

    // Same graph, but w(z,s) is lowered from 2 to -8. Now s->t->z->s costs
    // 6 + (-4) + (-8) = -6 < 0: a negative cycle.
    cout << "\n=== the same graph with w(z,s) = -8 ===\n";
    for (Edge& e : g.edges)
        if (e.u == g.id("z") && e.v == g.id("s")) e.w = -8;
    ok = bellmanFord(g, g.id("s"), d, pi, false);
    cout << "BELLMAN-FORD returns " << (ok ? "true" : "false") << "\n";
    return 0;
}
```

The guard `d[e.u] < INF` is Lemma 13.6 in code, and `if (!changed) break;` is a free early exit; the detector pass still runs afterward. Book: Section 13.8.6.

What It Prints

```
=== the example graph ===
init        s=0 t=INF x=INF y=INF z=INF
   pass 1: relax (s,t) w=6 -> d[t]=6
   pass 1: relax (s,y) w=7 -> d[y]=7
pass 1      s=0   t=6 x=INF   y=7 z=INF
   pass 2: relax (t,x) w=5 -> d[x]=11
   pass 2: relax (t,z) w=-4 -> d[z]=2
   pass 2: relax (y,x) w=-3 -> d[x]=4
pass 2      s=0   t=6   x=4   y=7   z=2
   pass 3: relax (x,t) w=-2 -> d[t]=2
pass 3      s=0   t=2   x=4   y=7   z=2
   pass 4: relax (t,z) w=-4 -> d[z]=-2
pass 4      s=0   t=2   x=4   y=7  z=-2
returns true

vertex |  d  | pi | shortest path from s
-------+-----+----+----------------------
   s   |   0 |  - | s
   t   |   2 |  x | s -> y -> x -> t
   x   |   4 |  y | s -> y -> x
   y   |   7 |  s | s -> y
   z   |  -2 |  t | s -> y -> x -> t -> z

=== the same graph with w(z,s) = -8 ===
BELLMAN-FORD returns false
```

Every number matches Table 13.10 and the five panels, including the transient $d[x]=11$ inside pass 2.

Book: Section 13.8.7.

The Detector, and Why It Works

THEOREM 13.13. After $|V|-1$ passes, Bellman–Ford returns `false` **if and only if** the graph contains a negative-weight cycle reachable from s.

No cycle ⇒ `true`. After $|V|-1$ passes $d=\delta$, so by Equation (13.5) $d[v]\le d[u]+w(u,v)$ for every edge. No edge fires.

A cycle ⇒ `false`. If it returned `true`, then for every edge with $d[u]$ finite

$$d[v]\le d[u]+w(u,v). \tag{13.11}$$

Sum Equation (13.11) round a reachable negative cycle c: the two d-sums run over the same vertices and cancel, leaving $0\le w(c)<0$. Absurd. ∎

`false` means "a negative cycle is reachable from s", not "the graph has one somewhere". To test the whole graph, add a vertex with a zero-weight edge to every other vertex and run from it (Exercise 13.14). To find the cycle, walk $|V|$ steps back along π and then round (Exercise 13.15).

Running time.

$$T_{\mathrm{BF}} = \underbrace{\Theta(V)}_{\text{init}} + \underbrace{(|V|-1)\,|E|}_{\text{passes}} + \underbrace{|E|}_{\text{detector}} = \Theta(VE). \tag{13.12}$$

By Equation (13.12): sparse gives $\Theta(V^2)$, dense gives $\Theta(V^3)$, the price of Floyd–Warshall for *all* pairs.

The queue-based refinement known as SPFA is often much faster in practice, but its worst case is still $\Theta(V E)$: never quote a better bound. Book: Sections 13.8.8 and 13.8.9.

Shortest Paths in a DAG

Finding shortest paths in DAGs is much easier, because it is easy to find an order in which to do the relaxations — topological sorting! By the time we reach u, every edge that could improve $d[u]$ has been relaxed.

```
DAG-SHORTEST-PATHS(G, w, s)
01  for each v ∈ V[G]
02      d[v] ← ∞
03  d[s] ← 0
04  topologically sort the vertices of V[G]
05  for each vertex u, taken in topological order, do
06      for each vertex v ∈ Adj[u] do
07          RELAX(u, v, w)
```

Figure 13.19 `DAG-Shortest-Paths`: lines 01–03 initialize, line 04 sorts the vertices topologically, and lines 05–07 relax each vertex's outgoing edges exactly once, in that order. There is no queue and no repetition.

In Figure 13.19, line 04 replaces Dijkstra's priority queue, and the $|V|-1$ passes of Figure 13.13 have vanished.

THEOREM 13.14. On a DAG the algorithm ends with $d[v]=\delta(s,v)$ for every v. *Proof:* in a topological order the vertices of any path appear in path order, so a shortest path's edges are relaxed in order and Lemma 13.8 applies. **No assumption about signs is needed.** ▰

Running time $\Theta(V+E)$, linear in the input: no lg *factor because there is no queue, no factor of $|V|$ because there is no repetition. Book: Sections 13.9.1 to 13.9.3 and 13.9.6.*

A DAG Worked by Hand

Take the Bellman–Ford graph of Figure 13.14 and delete its three *backward* edges (x, t), (z, x) and (z, s). Kahn's sort gives the order s, t, y, x, z.

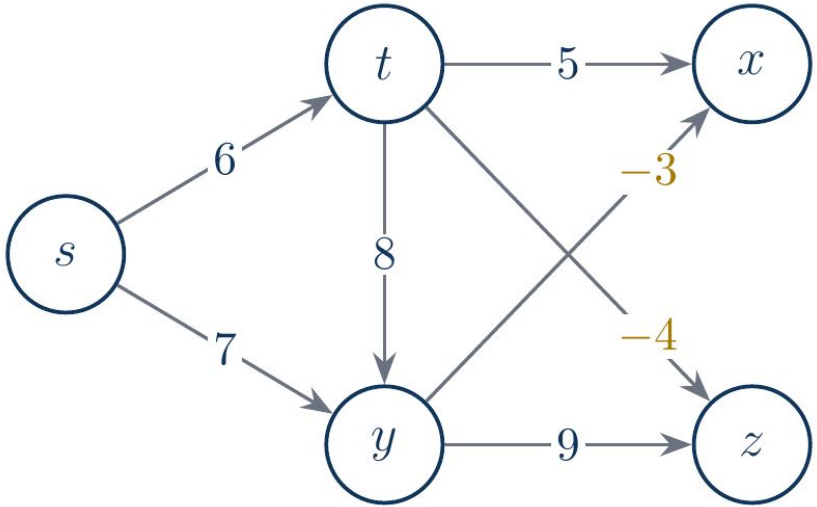


Figure 13.20 The Bellman–Ford example graph with its three backward edges removed: $s \to t(6)$, $s \to y(7)$, $t \to x(5)$, $t \to y(8)$, $t \to z(-4)$, $y \to x(-3)$, $y \to z(9)$. The graph is a DAG, and it still contains two negative edges.

Table 13.12 `DAG-Shortest-Paths` on the DAG example, one row per vertex in topological order.

Vertex taken	$d[s]$	$d[t]$	$d[x]$	$d[y]$	$d[z]$	Edges relaxed
(init)	0	∞	∞	∞	∞	—
s	0	6	∞	7	∞	$(s,t) \to 6, (s,y) \to 7$
t	0	6	11	7	2	$(t,x) \to 11$, (t,y) no, $(t,z) \to 2$
y	0	6	4	7	2	$(y,x) \to 4$, (y,z) no
x	0	6	4	7	2	none
z	0	6	4	7	2	none

Table 13.12 on Figure 13.20: **seven** relaxations, against forty for Bellman–Ford in Table 13.10. The answers differ ($\delta(s,t)=6$ here, 2 there) because deleting (x,t) removed the cheap route: two graphs, not two answers.

Book: Section 13.9.4.

The DAG Program, and Longest Paths

Listing 13.3 `DAG-Shortest-Paths`: Kahn's topological sort followed by a single relaxation pass; the test `order.size() != n` is a free cycle detector. Compile with `g++ -std=c++17 -O2 -o dag dag.cpp`.

```cpp
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>

using namespace std;

const long long INF = LLONG_MAX / 4;

struct Edge { int to; long long w; };

int main() {
    // The Bellman-Ford example graph with its three backward edges
    // (x,t), (z,x) and (z,s) deleted. What is left is a DAG.
    vector<string> name = {"s", "t", "x", "y", "z"};
    int n = (int)name.size();
    auto id = [&](const string& s) {
        return (int)(find(name.begin(), name.end(), s) - name.begin());
    };
    vector<vector<Edge>> adj(n);
    auto E = [&](const char* a, const char* b, long long w) {
        adj[id(a)].push_back({id(b), w});
    };
    E("s","t", 6); E("s","y", 7);
    E("t","x", 5); E("t","y", 8); E("t","z",-4);
```

```
    E("y","x",-3); E("y","z", 9);

    // --- step 1: topological sort (Kahn's in-degree algorithm, Chapter 11) ---
    vector<int> indeg(n, 0);
    for (int u = 0; u < n; ++u) for (const Edge& e : adj[u]) ++indeg[e.to];
    queue<int> ready;
    for (int v = 0; v < n; ++v) if (indeg[v] == 0) ready.push(v);
    vector<int> order;
    while (!ready.empty()) {
        int u = ready.front(); ready.pop();
        order.push_back(u);
        for (const Edge& e : adj[u]) if (--indeg[e.to] == 0) ready.push(e.to);
    }
    if ((int)order.size() != n) { cout << "not a DAG\n"; return 1; }
```

Listing 13.3 (Continued) *DAG-Shortest-Paths: Kahn's topological sort followed by a single relaxation pass; the test order.size() != n is a free cycle detector. Compile with g++ -std=c++17 -O2 -o dag dag.cpp.*

```
    cout << "topological order:";
    for (int v : order) cout << " " << name[v];
    cout << "\n\n";

    // --- step 2: one relaxation pass, vertices in topological order ---
    vector<long long> d(n, INF);
    vector<int> pi(n, -1);
    int s = id("s");
    d[s] = 0;

    for (int u : order) {
        cout << "take " << name[u] << " (d="
             << (d[u] >= INF ? string("INF") : to_string(d[u])) << ")\n";
        if (d[u] >= INF) { cout << "    unreachable; skip its edges\n"; continue; }
        for (const Edge& e : adj[u]) {
            long long cand = d[u] + e.w;
            cout << "    relax (" << name[u] << "," << name[e.to] << ") w=" << e.w
                 << ": " << d[u] << (e.w < 0 ? "+(" + to_string(e.w) + ")"
                                             : "+" + to_string(e.w))
                 << "=" << cand << " vs d["
                 << name[e.to] << "]="
                 << (d[e.to] >= INF ? string("INF") : to_string(d[e.to])) << " -> ";
            if (cand < d[e.to]) { d[e.to] = cand; pi[e.to] = u; cout << "improved\n"; }
            else                { cout << "no change\n"; }
        }
    }

    cout << "\nvertex |  d  | pi | shortest path from s\n";
```

```
    cout <<   "-------+-----+----+--------------------\n";
    for (int v = 0; v < n; ++v) {
        vector<string> path;
        for (int t = v; t != -1; t = pi[t]) path.push_back(name[t]);
        reverse(path.begin(), path.end());
        string p;
        for (size_t i = 0; i < path.size(); ++i) { if (i) p += " -> "; p += path[i]; }
        cout << "   " << name[v] << "   |" << setw(4) << d[v] << " |  "
             << (pi[v] == -1 ? string("-") : name[pi[v]]) << " | " << p << "\n";
    }
    return 0;
}
```

```
topological order: s t y x z

take s (d=0)
    relax (s,t) w=6: 0+6=6 vs d[t]=INF -> improved
    relax (s,y) w=7: 0+7=7 vs d[y]=INF -> improved
take t (d=6)
    relax (t,x) w=5: 6+5=11 vs d[x]=INF -> improved
    relax (t,y) w=8: 6+8=14 vs d[y]=7 -> no change
    relax (t,z) w=-4: 6+(-4)=2 vs d[z]=INF -> improved
take y (d=7)
    relax (y,x) w=-3: 7+(-3)=4 vs d[x]=11 -> improved
    relax (y,z) w=9: 7+9=16 vs d[z]=2 -> no change
take x (d=4)
take z (d=2)

vertex |  d  | pi | shortest path from s
-------+-----+----+----------------------
   s   |   0 |  - | s
   t   |   6 |  s | s -> t
   x   |   4 |  y | s -> y -> x
   y   |   7 |  s | s -> y
   z   |   2 |  t | s -> t -> z
```

Listing 13.3 refuses to answer on a graph where its assumption fails. **Longest paths come for nothing**: negate every weight, run the same loop, negate the answer. That is the **critical path method** of project scheduling.

Book: Sections 13.9.5 and 13.9.7.

Recovering the Route from π

Pseudocode 13.3 Printing a shortest path from s to v by recursing on the predecessor array.

```
PRINT-PATH(G, s, v)
1  if v = s
2      print s
3  else if pi[v] = NIL
4      print "no path from s to v exists"
5  else
6      PRINT-PATH(G, s, pi[v])
7      print v
```

Pseudocode 13.3 prints in the right order without an explicit reversal: line 6 prints everything before v, then line 7 prints v. The recursion is as deep as the path, up to $|V|-1$, so an iterative version is safer on a large graph.

On the Dijkstra graph it prints `s`, `s -> x -> u`, `s -> x -> u -> v`, `s -> x`, `s -> x -> y`: the bottom row of Table 13.7.

Three traps. Reading π before termination (mid-run it described a real path of weight 13); never setting π at all; and walking π after Bellman–Ford returned `false`, when π may contain a cycle, so guard the walk with a step count.

Book: Section 13.10.

Which Algorithm for Which Graph

Table 13.13 Which single-source algorithm to use. The first two questions are about correctness and admit no compromise; only the third is about speed.

	Dijkstra	Bellman–Ford	DAG relaxation	BFS (Ch. 10)
Graph must be	any digraph	any digraph	**acyclic**	any graph
Weights allowed	$w \geq 0$ only	any real w	any real w	none (all equal)
Negative cycle	undetected, wrong answer	**detected**, returns `false`	impossible	impossible
Time, sparse ($E = \Theta(V)$)	$O(V \lg V)$	$\Theta(V^2)$	$\Theta(V)$	$\Theta(V)$
Time, dense ($E = \Theta(V^2)$)	$\Theta(V^2)$ with an array	$\Theta(V^3)$	$\Theta(V^2)$	$\Theta(V^2)$
Extra structure needed	priority queue	none	topological sort	FIFO queue
Typical use	road networks, OSPF routing	currency arbitrage, RIP routing	project scheduling, pipelines	hop counts, unweighted mazes

The procedure behind Table 13.13**.** 1. All weights equal? **BFS.** 2. Acyclic? **DAG relaxation**, negative weights and all. 3. All weights ≥ 0? **Dijkstra**: heap if sparse, array if dense. 4. Otherwise **Bellman–Ford**, *and check its return value*. 5. All pairs? **Chapter 15.**

400,000 junctions, 1,000,000 roads, travel minutes: Dijkstra with a heap. *5,000 build tasks with durations*: DAG relaxation. *40 currencies at* $-\ln(\text{rate})$: Bellman–Ford, run precisely to see whether it returns `false`.

Book: Section 13.11.

Profitina in Practice

Inside the GEO platform. A scan's crawl frontier pops the cheapest next page by a cost accumulated along the link path: Dijkstra's priority queue, with non-negative weights. The scan pipeline is a DAG, and its minimum duration is a longest path.

Profitina Laundry, Question 1. Seven stops and eleven one-way legs weighted by driving minutes: Dijkstra. Depot → Dharmahusada directly takes 11 minutes; via Kertajaya it takes $7+3=10$.

Question 2. The ITS partner pays a rebate, so the leg (ITS, Keputih) costs -15 thousand rupiah: Bellman–Ford. Keputih costs 19 to reach, while ITS, *on the way to it*, costs 34: a cost that falls as you go further.

Question 3. Raise the rebate to -80 and the loop Depot → Kertajaya → Manyar → ITS → Keputih → Depot costs $14+8+12-80+30=-16$: the detector fires.

The algorithm advises; a person decides. Book: Section 13.12.

Common Mistakes

1. Dijkstra on a graph with a negative edge: the program does not complain (Figure 13.5). **2. Adding a constant to every weight**: a k-edge path gains $k\,c$. **3. Confusing a negative edge with a negative cycle** (Figure 13.2).

4. Running Bellman–Ford for $|V|$ or $|E|$ passes; Theorem 13.4 fixes it at $|V|-1$. **5. Omitting the detector pass**, which costs one part in $|V|$ of the total.

6. Relaxing out of an unreached vertex: `INF + (-4) < INF`. **7. Overflow from a too-large sentinel**: `INT_MAX + w` wraps negative. Use `LLONG_MAX/4` and guard first.

8. Updating d without π. 9. Reading π before termination. 10. Faking `DecreaseKey` wrongly: push a second copy *and discard the stale one.*

11. Assuming the shortest-path tree is the MST. 12. The DAG algorithm on a cyclic graph (Listing 13.3 refuses). **13. Quoting $O(E)$ for SPFA. 14. Treating $\delta(s,v)=\infty$ as an error**: it means *unreachable.*

Book: Section 13.13.

Summary — Five Things

1. **The problem.** $\delta(s, v)$ is the minimum of $w(p)$ over all paths. Of the four variants we solve **single-source**, because the others reduce to it or are built from it.
2. **Two structural facts.** *Optimal substructure*, by cut and paste; *the triangle inequality* $\delta(s, v) \le \delta(s, u) + w(u, v)$ for every edge.
3. **Negative edges are harmless; negative cycles are fatal.** Without a reachable negative cycle, shortest paths may be taken simple, so at most $|V| - 1$ edges.
4. **Relaxation is the only primitive**, made safe by five properties: upper-bound, no-path, convergence, path-relaxation, predecessor-subgraph.
5. **Three algorithms.** *Dijkstra*, $w \ge 0$, $O(E \lg V)$ with a heap. *Bellman–Ford*, any weights, $\Theta(V E)$, with a detector. *DAG relaxation*, $\Theta(V + E)$. The route is read backward off π once the algorithm has finished.

Book: Section 13.14.

Next Time

Chapter 14 — Amortized Analysis. We step away from graphs to pay a debt: the "amortized $\Theta(1)$" in the Fibonacci-heap row of Table 13.8 has now been used without proof.

Three methods, *aggregate*, *accounting* and *potential*, applied to the stack with `Multipop`, the binary counter and the dynamic table; and the point that catches everybody: **amortized is not average-case.**

Chapter 15 — All-Pairs Shortest Paths answers the fourth row of Table 13.2: Floyd–Warshall, matrix multiplication with min and $+$, and Johnson's reweighting, which makes Dijkstra legal on graphs with negative edges.

Homework. Exercises 13.1–13.16; start with 13.7 (Dijkstra by hand from source x) and 13.13 (labeling every vertex $+\infty$, finite or $-\infty$).

Cormen et al., Introduction to Algorithms, 4th ed., Chapter 22, covers the same ground with the same drawing conventions. Book: Sections 13.14.1 and 13.15.

Lecture 14 — Amortized Analysis

Design and Analysis of Algorithms (EF234405)

Chapter 14 of the book, in miniature, for the classroom.

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*.

Lecturer: Irfan Subakti — yifana@gmail.com

Today: structures whose operations are sometimes cheap and occasionally catastrophic, and a bound on the whole sequence that is far better than adding up worst cases. Book: Chapter 14, opening paragraph.

Agenda

1. **The question.** Why "worst single operation times n" can be off by a factor of n, and what *amortized cost* means.
2. **Not average-case.** No probability anywhere; the bound holds for every sequence, including an adversary's.
3. **Three methods** — aggregate, accounting, potential — each applied to the stack with `Multipop` and the binary counter, plus Exercises 17.1-3, 17.2-2 and 17.3-6 worked in full.
4. **Dynamic tables.** Doubling on insert, and why halving at half-full thrashes while halving at quarter-full does not.
5. **Union–find**, the promise from Chapter 11 kept: what $\alpha(n)$ is and why it never exceeds 4. Then choosing a method, Profitina, mistakes, summary.

The pages follow the book in order; the exercises at the end of the chapter are the homework. Book: Sections 14.2 to 14.10.

Where We Left Off

A debt from Chapter 11. The disjoint-set forest with union by rank and path compression was quoted at $O(m\,\alpha(n))$ for m operations, with the remark that you can only bound *the average over the whole sequence*. Section 14.8 pays that debt.

A second IOU. The Fibonacci-heap `DecreaseKey` was listed at "$O(1)$ amortized" in the priority-queue tables of Chapters 11 and 13. By the end of today you will know exactly what that row asserts.

What you need. A stack, an array, O, Θ, Ω from Chapter 3, and the summation sign. Everything else is built today.

> **A change of habit.** Throughout today, n is the **number of operations in the sequence**, not the number of elements. When we need the number of elements we call it s (items on a stack) or num (items in a table).

Book: Section 14.1.

A Bound That Is True, and Useless

A stack with three operations. `Push(S,x)` and `Pop(S)` each cost 1. `Multipop(S,k)` removes the top k items, or all of them if there are fewer; with s items it costs $\min(s,k)$.

The first analysis everyone does. The stack never holds more than n items, so one operation costs at most n; n operations cost at most $n \times n = n^2$. **Every word is true**, and the true answer is $O(n)$.

Sequence one. Push $n-1$ items, then `Multipop(S,n)`: $(n-1)+(n-1)=2n-2$. **Sequence two.** Push $n/2$, then $n/2$ calls of `Multipop(S,n)`: the first empties the stack, the rest cost 0; total n.

> **Where the crude bound goes wrong.** It treats every operation as if it could be the worst one *at the same time*. A `Multipop` that removes a million items needs a million earlier `Push`es from the same budget: **the expensive operations are paid for by the cheap ones that came before.**

For $n = 1,000,000$ the crude bound says a trillion steps; the truth is under two million. Book: Sections 14.2.1 and 14.2.2.

Three Costs, One Inequality

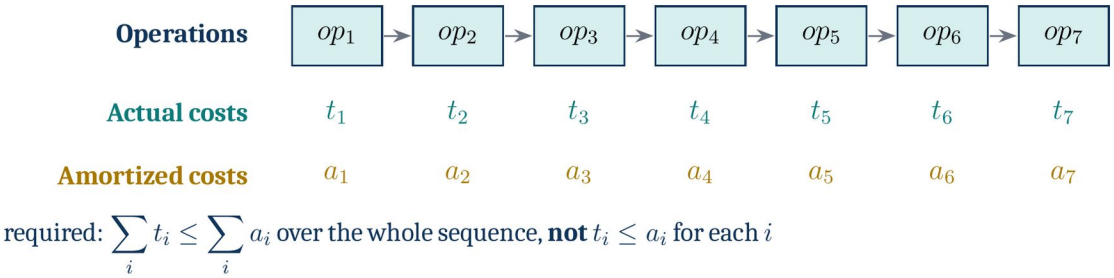


Figure 14.1 Operations $o\,p_1, \ldots, o\,p_7$ with their **actual costs** t_i and the **amortized costs** a_i we assign to them. The requirement is $\sum t_i \le \sum a_i$ over the whole sequence, and emphatically **not** $t_i \le a_i$ for each i.

In Figure 14.1 the middle row is the **actual cost** (written t_i there, c_i from now on); the bottom row is the **amortized cost** a_i, a number *we invent*, subject to one rule:

$$\sum_{i=1}^{n} t_i \le \sum_{i=1}^{n} a_i \text{ over the entire sequence, starting from } o\,p_1. \tag{14.1}$$

Not required: $t_i \le a_i$ for each i. One operation may blow straight through its allowance; only the running total of Equation (14.1) must hold. Some operations save up; later ones spend the savings.

"Amortize" comes from finance: spreading one large payment over many small installments. Book: Section 14.2.3.

The Definition

DEFINITION. Let c_i be the **actual cost** of the i-th of n operations. Numbers $a_1, \ldots, a_n$ are valid **amortized costs** if

$$\sum_{i=1}^{k} c_i \le \sum_{i=1}^{k} a_i \text{ for every } k = 1, 2, \ldots, n. \tag{14.2}$$

When every $a_i \le \hat{c}$, the operation has **amortized cost** $\hat{c}$, and any n operations cost at most $n\,\hat{c}$.

Equation (14.2) asks for the inequality at **every prefix** k, not only at $k = n$. That is what the accounting and potential methods deliver, and it means you may stop the program at any moment and the bound still holds.

THE QUESTION OF THIS CHAPTER. *Given a sequence of n operations, what is the worst-case total cost of the whole sequence?* The total divided by n is the **amortized cost per operation**.

Book: Section 14.2.3.

Amortized Is Not Average-Case

Average-case analysis assumes a probability distribution over inputs; its answer is an *expectation*. Randomized quicksort's $O(n \lg n)$ is an expectation, and some runs take $\Theta(n^2)$.

Amortized analysis assumes **nothing**: no distribution, no randomness, no typical input. It is a **worst-case bound on the total**, valid for every sequence, including one chosen by an adversary who has read your source code.

> **The misconception to kill.** "Amortized $O(1)$" does *not* mean "$O(1)$ most of the time". It means **no sequence of n operations costs more than $O(n)$ in total.** What it does not tell you is that any *particular* operation is fast.

The test. *Could an adversary who knows my entire implementation defeat this bound?* For an expected bound, often yes. For an amortized bound, never.

The book says this twice on purpose; it lands differently after you have seen the aggregate method work. Book: Sections 14.2.4 and 14.4.

The Three Methods

Three ways to find an amortized cost

1. **Aggregate method.** Bound the total cost $T(n)$ of any n operations; every operation is charged $T(n)/n$.
2. **Accounting method.** Give operations different charges; overcharge some so that they prepay, as stored credit, for others.
3. **Potential method.** Store the prepayment as the "potential energy" Φ of the data structure itself: amortized cost = actual cost + change in potential.

Figure 14.2 The three common techniques: the **aggregate method** divides the total cost $T(n)$ by n; the **accounting method** overcharges some operations so that they prepay for others; the **potential method** stores the prepayment as the "potential energy" of the structure itself.

Aggregate: compute the total $T(n)$ by a counting argument and give every operation $T(n)/n$. **Accounting:** charge each operation type a fixed amount and store the surplus as **credit** on specific objects. **Potential:** store the prepayment as one number Φ computed from the whole structure.

All three, as Figure 14.2 suggests, give the same answers on today's examples; they differ in how much work they are and how much they tell you.

Book: Section 14.2.5.

The Aggregate Method: the Stack

Two steps. (1) Show that any n operations take $T(n)$ in the worst case. (2) Declare the amortized cost of each operation to be $T(n)/n$. Its one weakness: every operation gets the same number.

$O(1)$ each

```
PUSH(S, x)
1  top[S] ← top[S] + 1
2  S[top[S]] ← x
```

```
POP(S)
1  if STACK-EMPTY(S)
2       then error "underflow"
3       else top[S] ← top[S] − 1
4       return S[top[S] + 1]
```

cost = number of loop iterations = $\min(s, k)$

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k ≠ 0
2       do POP(S)
3            k ← k − 1
```

Figure 14.3 The stack operations: `Push` and `Pop` each touch one array slot and one index, so both are $O(1)$. `Multipop(S,k)` calls `Pop` until the stack is empty or k pops have been done, so its cost is the number of times round the loop.

In Figure 14.3, `Multipop` loops while the stack is non-empty *and* $k \neq 0$, so with s items it runs $\min(s,k)$ times: cost $O(\min(s,k))$, **not** $O(1)$.

Book: Sections 14.3.1 and 14.3.2.

Popped at Most Once

Claim. Starting from an empty stack, the number of removals never exceeds the number of pushes. **Proof.** An item enters only by one `Push`; it leaves only by a `Pop`, and a `Multipop` iteration *is* a `Pop`; once removed it is gone. So each item is popped **at most once** after its push. ∎

The total. With p pushes and q removals, $q \le p \le n$, so

$$T(n) = p \times 1 + q \times 1 \le n + n = 2n, \quad \frac{T(n)}{n} \le 2 = O(1).$$

> **"Each element is popped at most once after it is pushed."** That sentence is the whole analysis. Its general shape: *find a unit of work charged to an object, and show each object is charged a bounded number of times over the whole sequence.*

All three operations have amortized cost $O(1)$, including `Multipop`, whose actual cost can be $\Theta(n)$. That is not a contradiction; it is the definition doing its job.

The bookkeeping inside `Multipop` turns $2n$ into cn for a small constant c; the clean figure is 2. Book: Section 14.3.2.

The Stack, in C++

Listing 14.1 counts every item leaving the stack on three sequences: the adversary's, a trivially cheap one, and a pseudo-random mix.

Listing 14.1 A stack with `Multipop`, instrumented. Compile with `g++ -std=c++17 -O2 -o multipop multipop.cpp`.

```cpp
// A stack with PUSH, POP and MULTIPOP, instrumented so that every
// elementary stack slot touched is counted.  The program then reports
// the measured amortized cost -- total work divided by the number of
// operations -- for three different sequences.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

class CountingStack {
public:
    long long work = 0;        // elementary slot writes/reads, i.e. actual cost
    long long nPush = 0, nPop = 0, nMultipop = 0, nOps = 0;

    void push(int x) { data.push_back(x); ++work; ++nPush; ++nOps; }

    // One POP as an operation of its own.
    void pop() { ++nPop; ++nOps; popOne(); }

    // MULTIPOP(S,k): remove the top min(s,k) items.  Counted as ONE
    // operation, but its actual cost is the number of items it removes.
    int multipop(int k) {
        ++nMultipop; ++nOps;
        int removed = 0;
        while (!data.empty() && k != 0) { popOne(); --k; ++removed; }
        return removed;
    }
```

```
    size_t size() const { return data.size(); }
    bool empty() const { return data.empty(); }

private:
    vector<int> data;
    // The single elementary removal.  MULTIPOP is built out of these,
    // so `work` counts exactly one unit per item that leaves the stack.
    void popOne() { if (!data.empty()) { data.pop_back(); ++work; } }
};
```

Listing 14.1 (Continued) *A stack with Multipop, instrumented. Compile with g++ -std=c++17 -O2 -o multipop multipop.cpp.*

```
void report(const string& name, const CountingStack& S) {
    cout << left << setw(30) << name
         << " ops=" << setw(8) << S.nOps
         << " (push " << setw(7) << S.nPush
         << " pop " << setw(7) << S.nPop
         << " multipop " << setw(7) << S.nMultipop << ")"
         << "  actual cost=" << setw(9) << S.work
         << "  cost/op=" << fixed << setprecision(6)
         << (double)S.work / (double)S.nOps << "\n";
}

int main() {
    const int n = 100000;

    // Sequence A: the adversary's favorite.  Push n/2 items, then ask
    // MULTIPOP to empty the whole stack in one operation.  Repeat.
    {
        CountingStack S;
        while (S.nOps + 1 < n) {
            for (int i = 0; i < n / 2 && S.nOps + 1 < n; ++i) S.push(i);
            S.multipop(n);                          // one very expensive operation
        }
        report("A: fill, then MULTIPOP(n)", S);
    }

    // Sequence B: push one, multipop one, forever -- every MULTIPOP is cheap.
    {
        CountingStack S;
```

```
        while (S.nOps + 2 <= n) { S.push(1); S.multipop(n); }
        report("B: push/MULTIPOP alternating", S);
    }

    // Sequence C: a pseudo-random mix, deterministic so the run repeats.
    {
        CountingStack S;
        unsigned long long seed = 12345;
        auto rnd = [&seed]() { seed = seed * 6364136223846793005ULL + 1442695040888963407ULL;
                               return (unsigned)(seed >> 33); };
        while (S.nOps < n) {
            unsigned r = rnd() % 100;
            if (r < 60) S.push((int)r);
            else if (r < 90) { if (!S.empty()) S.pop(); else S.push(0); }
            else S.multipop((int)(rnd() % 1000));
        }
        report("C: random 60/30/10 mix", S);
    }
```

Listing 14.1 (Continued) *A stack with Multipop, instrumented. Compile with g++ -std=c++17 -O2 -o multipop multipop.cpp.*

```
    cout << "\nThe bound proved in the text is: actual cost < 2n, i.e. cost/op < 2.\n";
    cout << "With n = " << n << " that ceiling is " << 2 * n << ".\n";
    return 0;
}
```

Book: Section 14.3.2, “Measuring it”.

The Stack, Measured

```
A: fill, then MULTIPOP(n)      ops=100000   (push 99998   pop 0       multipop 2      )  actual
cost=199996     cost/op=1.999960
B: push/MULTIPOP alternating   ops=100000   (push 50000   pop 0       multipop 50000  )  actual
cost=100000     cost/op=1.000000
C: random 60/30/10 mix         ops=100000   (push 64514   pop 25192   multipop 10294  )  actual
cost=129027     cost/op=1.290270

The bound proved in the text is: actual cost < 2n, i.e. cost/op < 2.
With n = 100000 that ceiling is 200000.
```

Sequence A empties the stack in one gigantic `Multipop` and reaches $199,996$ against the ceiling $200,000$: four units short of the proved bound, and no further.

Sequence C, an ordinary mix, costs 1.29 per operation. No case reaches 2, and none is anywhere near the $n = 100,000$ per operation that the $O(n^2)$ bound would allow.

Book: Section 14.3.2, "Measuring it".

The k-Bit Binary Counter

An array $A[0..k-1]$ of bits, $A[0]$ least significant, representing $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. To add one: turn trailing 1s into 0s and set the first 0, exactly as 1999+1 on paper.

```
INCREMENT(A)
1  i ← 0
2  while i < length[A] and A[i] = 1
3      do A[i] ← 0
4         i ← i + 1
5  if i < length[A]
6      then A[i] ← 1
```

$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	
0	0	0	0	0	1	1	1	$= 7$
0	0	0	0	1	0	0	0	$= 8$

bit array $A[0..k-1]$, $length[A] = k$;
$A[0]$ is the least significant bit

Figure 14.4 `Increment(A)`: lines 2–4 walk up from $A[0]$ turning every trailing 1 into a 0, and line 6 sets the first 0 reached. If every bit was 1, the counter silently wraps to zero, exactly as a k-bit machine register does.

Trace of Figure 14.4, $k=4$, from 0111 (seven): reset $A[0]$, $A[1]$, $A[2]$; stop at $A[3]=0$; set $A[3]$. Result 1000 (eight), four bits written. **The cost of one `Increment` is the number of bits it changes.**

Book: Section 14.3.3.

The 8-Bit Table

Counter value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 14.5 An 8-bit counter after each of 16 increments; the shaded cells in a row are the bits that the next increment flips, so their number is that increment's actual cost. The running total after 16 increments is 31, under $2 \times 16 = 32$.

In Figure 14.5, $A[0]$ is shaded in *every* row: it flips n times in n increments. The wide rows ($7 \to 8$ shades four cells, $15 \to 16$ five) are rare, and rarer the wider they get.

The crude bound says $O(nk)$. The correction is one observation: *not all bits change every time* `Increment` *is called.*

Book: Section 14.3.3, "The 8-bit table".

How Often Does Bit i Flip?

$A[0]$ flips on every increment: n times. $A[1]$ only when $A[0]$ was 1, every second increment: $\lfloor n/2 \rfloor$. $A[2]$ every fourth: $\lfloor n/4 \rfloor$.

In general, for $i = 0, 1, \ldots, \lfloor \lg n \rfloor$, bit $A[i]$ changes $\lfloor n/2^i \rfloor$ times, because it flips exactly when the low i bits of the value being incremented are all 1, one value in every block of 2^i. Higher bits never move.

The total over n increments:

$$T(n) = \sum_{i=0}^{\lfloor \lg n \rfloor} \lfloor \frac{n}{2^i} \rfloor. \tag{14.3}$$

We write $\lg$ *for* $\log_2$*, as Chapter 3 established. Book: Section 14.3.3, "How often does bit* i *flip?".*

Adding It Up

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor \;<\; n \sum_{i=0}^{\infty} \frac{1}{2^i} \;=\; 2n$$

Figure 14.6 The aggregate analysis of the counter in one display: drop the floor, extend the sum to infinity, factor out n, and sum the geometric series to 2.

Figure 14.6 bounds Equation (14.3): $\lfloor x \rfloor \le x$; extending to infinity adds positive terms; pull out n; and $1+1/2+1/4+\cdots=2$ (let S be the sum; $1/2\,S = S-1$, so $S=2$). Hence $T(n) < 2n$ and the amortized cost is $< 2 = O(1)$.

> **In plain language.** To flip k bits you must first have set those k bits, one at a time, over the previous $2^k - 1$ increments. Fewer than two bits flip per increment on average, **however large k is**: a 64-bit and an 8-bit counter cost the same.

Measured over a million increments, the total is $1,999,993$, seven below $2n$. Book: Section 14.3.3, "Adding it up" and "Measuring it".

Exercise 17.1-3

EXERCISE 17.1-3. The i-th of n operations costs i if i is an exact power of 2, and 1 otherwise. Use the aggregate method to find the amortized cost per operation.

- A sequence of n operations is performed on a data structure.
- The cost of the i-th operation is i if i is an exact power of 2, and 1 otherwise.
- Use the **aggregate method** to determine the amortized cost per operation.

Operation i	**Cost** c_i
1	$1 = 2^0$
2	$2 = 2^1$
3	1
4	$4 = 2^2$
5	1
Total	**9**

Figure 14.7 The first five operations of Exercise 17.1-3 cost $1, 2, 1, 4, 1$: the powers of two $1, 2, 4$ charge their own index and every other operation charges 1. The total is 9, or 1.8 per operation.

As Figure 14.7 shows, the cost function is

$$c_i = \begin{cases} i & \text{if } i = 2^j \text{ for some integer } j \geq 0, \\ 1 & \text{otherwise.} \end{cases} \tag{14.4}$$

Split Equation (14.4) into a baseline of 1 per operation and the extras $2^j - 1$ of the powers of two, $j = 0, \ldots, \lfloor \lg n \rfloor$:

$$\text{Total} = \sum_{j=0}^{\lfloor \lg n \rfloor} \left(2^j - 1\right) + \sum_{i=1}^{n} 1. \tag{14.5}$$

The doubling table of Section 14.7 behaves almost exactly like this caricature. Book: Section 14.3.4.

Exercise 17.1-3, Solved

$$
\begin{aligned}
\textit{Total} &= \sum_{j=0}^{\lfloor \lg n \rfloor} \left(2^j - 1\right) + \sum_{i=1}^{n} 1 && \text{powers of two pay } 2^j \text{ instead of } 1 \\
&= \left(2^{\lfloor \lg n \rfloor + 1} - 1\right) - \left(\lfloor \lg n \rfloor + 1\right) + n && \text{geometric series: } \sum_{j=0}^{m} x^j = \frac{x^{m+1} - 1}{x - 1} \\
&= 2^{\lfloor \lg n \rfloor + 1} + n - \lfloor \lg n \rfloor - 2 \\
&\le 2n + n - \lfloor \lg n \rfloor - 2 && \text{since } 2^{\lfloor \lg n \rfloor} \le 2^{\lg n} = n \\
&= 3n - \lfloor \lg n \rfloor - 2.
\end{aligned}
$$

The total cost of n operations is $O(n)$, so the amortized cost per operation is $O(1)$.

Figure 14.8 The complete solution to Exercise 17.1-3, from the split sum down to $\text{Total} \le 3n - \lfloor \lg n \rfloor - 2$, with the geometric-series formula and the logarithm identity it needs.

Geometric series (multiply by x and subtract):

$$\sum_{k=0}^{m} x^k = \frac{x^{m+1} - 1}{x - 1} \left(x \neq 1\right). \tag{14.6}$$

With $x=2$, Equation (14.6) turns Equation (14.5) into the exact total

$$\text{Total} = 2^{\lfloor \lg n \rfloor + 1} + n - \lfloor \lg n \rfloor - 2. \tag{14.7}$$

In Equation (14.7), $2^{\lfloor \lg n \rfloor + 1} \le 2^{\lg n + 1} = 2n$, so $\text{Total} \le 3n - \lfloor \lg n \rfloor - 2 \le 3n - 2$, as Figure 14.8 concludes: amortized cost $< 3 = O(1)$.

The bound is tight exactly when n is a power of two, and the program of Section 14.3.4 confirms the exact formula at every checkpoint. Book: Section 14.3.4.

What the Aggregate Method Cannot Do

One number for all operations. On the stack, `Push`, `Pop` and `Multipop` all come out at 2, though a `Push` obviously costs exactly 1.

It needs a global counting argument. One existed for the stack and the counter; for a splay tree or a Fibonacci heap none is known, and the method simply fails.

It says nothing about *when* the cost is incurred. Amortized bounds are about throughput, not latency: a real-time deadline needs a different guarantee.

> **Why 3 and not worse, in Exercise 17.1-3.** The extras $1, 2, 4, \ldots$ double, so their sum is under twice the largest, less than $2n$; one unit each for n operations makes $3n$. Costs that double give a constant amortized cost.

Book: Sections 14.3.4 and 14.3.5.

The Accounting Method

Rules. Each operation is handed an allowance $\hat{c}_i$ that we choose. It spends its actual cost c_i; a surplus is deposited as **credit** on a specific object; a deficit is paid from credit deposited earlier.

THE CREDIT INVARIANT. For every $k = 1, 2, \ldots, n$,

$$C_k = \sum_{i=1}^{k} \hat{c}_i - \sum_{i=1}^{k} c_i \geq 0. \tag{14.8}$$

The structure never goes into debt.

Rearranged, Equation (14.8) is exactly Equation (14.2). It must hold at **every** k, because the sequence can stop anywhere.

The trap. "The charges are obviously big enough" proves nothing. You must **locate** the credit: say which object holds each unit, so that the balance is visibly a count and cannot be negative.

Think of each elementary step as costing one rupiah. Book: Sections 14.5.1 and 14.5.2.

The Stack, by Accounting

Operation	Actual cost	Amortized cost (charge)
`push`	1	2
`pop`	1	0
`multipop`	$\min(s, k)$	0

- Every charge is a constant, so the amortized cost of each of the three operations is $O(1)$.
- The credit equals the number of items on the stack, which is never negative; hence n operations cost $O(n)$ in total.

Figure 14.9 Accounting charges for the stack: `Push` (actual cost 1) is charged 2, while `Pop` (cost 1) and `Multipop` (cost $\min(s, k)$) are charged 0. All three charges are constants, so n operations cost $O(n)$.

Credit rule for Figure 14.9: every item on the stack carries exactly 1 rupiah. `Push` spends 1 and leaves 1 on the new item. `Pop` and each iteration of `Multipop` spend the rupiah carried by the item being removed.

Why the credit is never negative. The total credit equals the number of items on the stack, and a count cannot be negative. ∎

> *Each item pays, at the moment it is pushed, for both its arrival and its eventual departure.* Maximum charge 2, so n operations cost at most $2n$: the same answer as the aggregate method.

Book: Section 14.5.3.

The Counter, by Accounting

Charges. Setting a bit to 1: **2**. Resetting a bit to 0: **0**.

Credit rule: every bit that is currently 1 carries exactly 1 rupiah. The one set-to-1 spends 1 and deposits 1 on the bit it set. Each of the t resets is paid by the rupiah sitting on that 1 bit.

So every `Increment` is charged at most 2, however many bits it flips. The increment from $2^{20} - 1$ to 2^{20} flips twenty-one bits and is still charged 2.

Never negative: the credit equals the number of 1 bits. **Overflow:** all k resets are paid by the k rupiah on the k one-bits, so the credit simply drops to zero. ▰

Instrumented, the credit column equals the item count and the ones count at every step, and its lowest value over 200,000 stack operations and 1,000,000 increments is 0. Book: Section 14.5.4.

Exercise 17.2-2

Redo Exercise 17.1-3 by the accounting method. Charge **every** operation 3. Cheap operations deposit 2; an operation of cost i withdraws $i-3$.

Charge every operation 3. The credit left after the i-th operation, written Total_i, is

$$\mathit{Total}_i = \sum_{j=1}^{i} 3 \;-\; \mathit{ActualCost}_i = 3i - \left(2^{\lfloor \lg i \rfloor + 1} + i - \lfloor \lg i \rfloor - 2\right).$$

Case 1: $i = 2^k$ with $k \geq 0$. Then $2^{\lfloor \lg i \rfloor + 1} = 2i$ and

$$\mathit{Total}_i = 3i - (2i + i - k - 2) = k + 2 \;\geq\; 2.$$

Case 2: $i = 2^k + j$ with $k \geq 0$ and $0 \leq j < 2^k$. Then $2^{\lfloor \lg i \rfloor + 1} = 2^{k+1} = 2(i - j)$ and

$$\mathit{Total}_i = 3i - \left(2(i - j) + i - k - 2\right) = 2j + k + 2 \;\geq\; 2.$$

In both cases the credit is non-negative, so the total amortized cost $3n$ is an upper bound on the total actual cost: n operations cost $O(n)$, and the amortized cost per operation is $O(1)$.

Figure 14.10 The accounting-method solution to Exercise 17.2-2: every operation is charged 3, and the credit after operation i is $3i$ minus the actual cost so far, split into two cases by where i sits among the powers of two.

Figure 14.10: the credit after i operations is $3i - \text{ActualCost}_i$, and Equation (14.7) with i for n gives ActualCost_i. So

$$\text{Total}_i = 3i - \left(2^{\lfloor \lg i \rfloor + 1} + i - \lfloor \lg i \rfloor - 2\right).$$

Book: Section 14.5.5.

The Two Cases

Case 1: $i = 2^k$. Then $2^{\lfloor \lg i \rfloor + 1} = 2i$ and $\text{Total}_i = 3i - (3i - k - 2) = k + 2 \geq 2$. This is the tightest moment, right after the most expensive operation so far.

Case 2: $i = 2^k + j$, $0 \leq j < 2^k$. Then $2^{\lfloor \lg i \rfloor + 1} = 2(i - j)$ and

$$\text{Total}_i = 3i - (3i - 2j - k - 2) = 2j + k + 2.$$

(14.9)

Case 1 is $j = 0$ in Equation (14.9). In both cases the credit is at least 2, so the charge of 3 is legitimate: n operations cost at most $3n = O(n)$, amortized $O(1)$ each.

The aggregate method gave $3n - \lfloor \lg n \rfloor - 2$; the accounting method gives the slightly weaker $3n$ with a much shorter, per-operation argument. Book: Section 14.5.5, the two cases and the conclusion.

Exercise 17.2-2, Checked

Listing 14.2 Exercise 17.2-2: the credit balance tracked and checked against $2j+k+2$. Compile with `g++ -std=c++17 -O2 -o ex1722 ex1722.cpp`.

```cpp
// Exercise 17.2-2: redo Exercise 17.1-3 by the accounting method.
// Every operation is charged 3.  This program tracks the bank balance
// after each operation and checks it against the closed form of the text,
//     if i = 2^k + j  with 0 <= j < 2^k, then credit_i = 2j + k + 2.
#include <iostream>
#include <iomanip>
using namespace std;

bool isPowerOfTwo(long long i) { return i > 0 && (i & (i - 1)) == 0; }
long long actualCost(long long i) { return isPowerOfTwo(i) ? i : 1; }

int floorLg(long long n) { int k = 0; while ((1LL << (k + 1)) <= n) ++k; return k; }

int main() {
    const long long N = 1000000;
    long long credit = 0, minCredit = 0, actualTotal = 0;
    bool formulaHolds = true;
    long long firstBad = 0;

    cout << "  i  actual  charged  credit   2j+k+2\n";
    for (long long i = 1; i <= N; ++i) {
        long long c = actualCost(i);
        actualTotal += c;
        credit += 3 - c;                        // charge 3, pay out c
        int k = floorLg(i);
        long long j = i - (1LL << k);           // so that i = 2^k + j, 0 <= j < 2^k
        long long predicted = 2 * j + k + 2;
        if (credit != predicted && formulaHolds) { formulaHolds = false; firstBad = i; }
```

```
        if (credit < minCredit) minCredit = credit;
        if (i <= 17)
            cout << setw(3) << i << setw(8) << c << setw(9) << 3
                 << setw(8) << credit << setw(9) << predicted << "\n";
    }
    cout << "\nclosed form 2j+k+2 agrees for every i up to " << N << ": "
         << (formulaHolds ? "yes" : "no, first failure at i = ") ;
    if (!formulaHolds) cout << firstBad;
    cout << "\nlowest credit ever seen = " << minCredit
         << "  (the accounting method needs this to be >= 0)\n";
    cout << "total actual cost = " << actualTotal
         << ", total charged = " << 3 * N
         << ", so 3n is indeed an upper bound.\n";
    return 0;
}
```

```
  i  actual  charged  credit   2j+k+2
  1       1        3       2        2
  2       2        3       3        3
  3       1        3       5        5
  4       4        3       4        4
  5       1        3       6        6
  6       1        3       8        8
  7       1        3      10       10
  8       8        3       5        5
  9       1        3       7        7
 10       1        3       9        9
 11       1        3      11       11
 12       1        3      13       13
 13       1        3      15       15
 14       1        3      17       17
 15       1        3      19       19
 16      16        3       6        6
 17       1        3       8        8

closed form 2j+k+2 agrees for every i up to 1000000: yes
lowest credit ever seen = 0  (the accounting method needs this to be >= 0)
total actual cost = 2048555, total charged = 3000000, so 3n is indeed an upper bound.
```

Listing 14.2 confirms Equation (14.9) for a million operations. The balance climbs by 2 on each cheap operation and drops to $4, 5, 6$ at $i = 4, 8, 16$: exactly $k+2$.

Book: Section 14.5.5, "Checked against a program".

The Potential Method

The accounting method stores credit *on objects*. The potential method stores it as **one number for the whole structure**, which suits savings that are a property of the structure's *shape*.

SET-UP. D_0 is the initial state and D_i the state after operation i; c_i is the actual cost of operation i; Φ is a **potential function** we choose, mapping each state to a real number; and the **amortized cost** is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}). \tag{14.10}$$

Read Equation (14.10) **in words:** actual cost plus however much the operation raised the potential. An operation that leaves the structure more dangerous is charged for the danger; one that defuses it gets a refund.

$\Phi(D_i) - \Phi(D_{i-1})$ *is the change in potential, often written* $\Delta\Phi_i$. *Book: Section 14.6.1.*

The Telescoping Sum

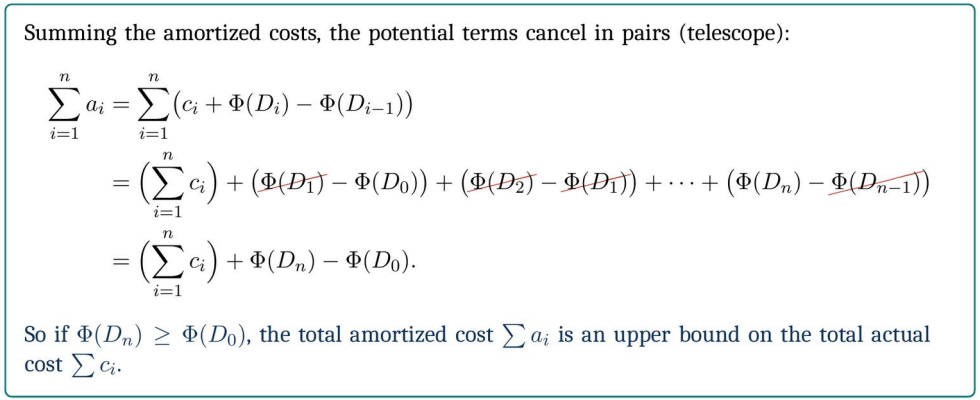


Figure 14.11 Summing the amortized cost over $i=1,\ldots,n$ makes the potential terms cancel in pairs, leaving only the first and the last. This calculation makes the whole method work.

In Figure 14.11, write the potential terms out for $n=4$: $(\Phi_1-\Phi_0)+(\Phi_2-\Phi_1)+(\Phi_3-\Phi_2)+(\Phi_4-\Phi_3)$. Everything in the middle cancels, like a collapsing telescope:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0). \tag{14.11}$$

Book: Section 14.6.2.

The One Condition

Rearranging Equation (14.11), the real total is at most the invented total exactly when

$$\Phi(D_n) \geq \Phi(D_0), \tag{14.12}$$

and, because the sequence can stop anywhere, we require Equation (14.12) at every i.

> **The credit invariant in disguise.** By telescoping, $\sum_{i \leq k} \hat{c}_i - \sum_{i \leq k} c_i = \Phi(D_k) - \Phi(D_0)$, so Equation (14.8) and "the potential never drops below its start" are **the same statement**.

The practical shortcut. Arrange $\Phi(D_0) = 0$ and $\Phi(D) \geq 0$ for every reachable D; state both facts explicitly. **Finding Φ**: it should measure *how much trouble the structure is storing up*.

Book: Section 14.6.3.

The Stack, by Potential

- Potential: $\Phi(D_i)$ = the number of objects on the stack after the i-th operation.
- The stack starts empty, so $\Phi(D_0) = 0$; a stack never holds a negative number of objects, so $\Phi(D_i) \geq 0 = \Phi(D_0)$.
- `Push` onto a stack of s objects: $a_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (s+1) - s = 2$.
- `Multipop`(S, k) with $k' = \min(k, s)$: $a_i = k' + (s - k') - s = 0$; likewise `Pop` costs $1 - 1 = 0$.

Figure 14.12 The stack by the potential method: with Φ = the number of elements on the stack, `Push` has amortized cost $1+(s+1)-s=2$ and `Multipop` has amortized cost $k'-k'=0$, where $k'=\min(k,s)$.

Φ = number of items. $\Phi(D_0)=0$ and $\Phi\geq 0$. With s items before the operation, as in Figure 14.12: `Push` $1+(s+1)-s=2$; `Pop` $1+(s-1)-s=0$; `Multipop` with $k'=\min(k,s)$: $k'+(s-k')-s=0$.

> The potential counts *future work stored up*. `Multipop` does k' units now but removes k' units of stored work, so it is charged nothing: it is **spending savings**, not creating obligations.

Third method, same $2n$: the answer is a property of the structure, not of the method. Book: Section 14.6.4.

The Counter, by Potential

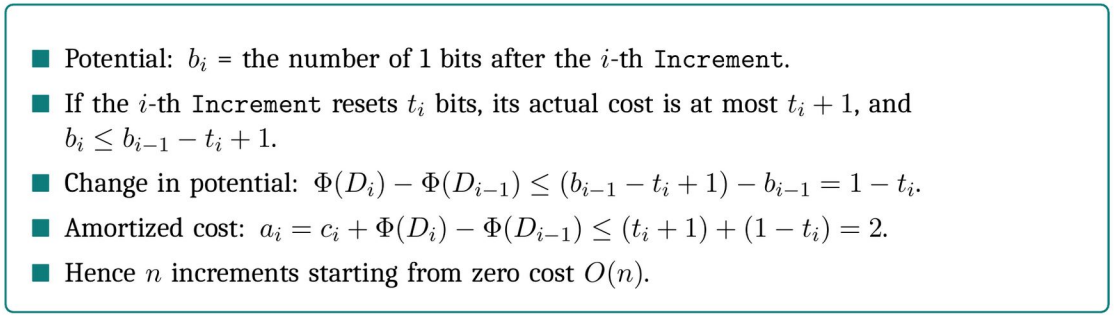


Figure 14.13 The counter by the potential method: with b_i = the number of 1 bits after the i-th `Increment` and t_i = the number of bits that increment reset, the potential change is at most $1-t_i$ and the amortized cost is at most $(t_i+1)+(1-t_i)=2$.

$\Phi(D_i)=b_i$, the number of 1 bits; $b_0=0$. The i-th increment resets t_i bits: $c_i \le t_i+1$ and $b_i \le b_{i-1}-t_i+1$, so $\Delta\Phi_i \le 1-t_i$ and, as Figure 14.13 shows,

$$\hat{c}_i \le (t_i+1)+(1-t_i)=2.$$

The t_i cancels. However many bits are reset, the potential falls by exactly as much. Measured on 16 increments: $\sum c_i=31$, $\Phi(D_n)=1$, and $31+1=32=\sum \hat{c}_i$.

Book: Section 14.6.5.

Exercise 17.3-6: a Queue from Two Stacks

Implement a queue with two stacks so that `Enqueue` and `Dequeue` are $O(1)$ amortized. An input stack S_{IN} and an output stack S_{OUT}: `Enqueue` pushes onto S_{IN}; `Dequeue` pops from S_{OUT}, first draining all of S_{IN} into it **if S_{OUT} is empty**.

```
ENQUEUE(e)
1  PUSH(S_IN, e)
```

```
DEQUEUE()
1  if STACK-EMPTY(S_OUT) then
2      while not STACK-EMPTY(S_IN) do
3          PUSH(S_OUT, POP(S_IN))
4  if STACK-EMPTY(S_OUT) then
5      error "underflow"
6  else
7      return POP(S_OUT)
```

Figure 14.14 `Enqueue` and `Dequeue` for the two-stack queue: `Enqueue` is one push, and `Dequeue` pops from S_{OUT} after first draining all of S_{IN} into it, **only if S_{OUT} is empty**. The underflow test comes after the transfer.

Two details in Figure 14.14. Transfer only when S_{OUT} is empty, or the order is reversed again and again. Test underflow *after* the transfer: the queue is empty only when both stacks are.

Reversing a stack turns it into a queue: the oldest item, at the bottom of S_{IN}, ends on top of S_{OUT}. Book: Section 14.6.6.

The Potential, and the Three Cases

- Potential: $\Phi(D_i) = 2 \times \textit{elements}_i$, where $\textit{elements}_i$ is the number of items in the **input** stack S_{IN} after the i-th operation.
- `Enqueue`: $a_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2 = 3$.
- `Dequeue` with no transfer (S_{OUT} not empty): $a_i = 1 + 0 = 1$.
- `Dequeue` with a transfer of $e = \textit{elements}_{i-1}$ items: $a_i = (2e + 1) + 0 - 2e = 1$.
- Hence both `Enqueue` and `Dequeue` have amortized cost $O(1)$.

Figure 14.15 The potential analysis of the two-stack queue. With $\Phi(D_i) = 2 \times \text{elements}_i$, where elements_i counts the items in the **input** stack only, `Enqueue` costs 3 amortized and both kinds of `Dequeue` cost 1.

$\Phi = 2 \times |S_{IN}|$: a transfer touches each item of S_{IN} twice, and items in S_{OUT} have already paid. With $e = |S_{IN}|$ before the operation, Figure 14.15 gives: `Enqueue` $1+2=3$; `Dequeue` without transfer $1+0=1$; `Dequeue` with transfer $(2e+1)+0-2e=1$.

> **Every item makes exactly one journey**: pushed onto S_{IN}, popped, pushed onto S_{OUT}, popped. `Enqueue` prepays three of those steps; the `Dequeue` that removes the item pays the last.

Book: Section 14.6.6, "The potential function".

The Two-Stack Queue in C++

Listing 14.3 Exercise 17.3-6: the two-stack queue, with its potential function computed after every operation. Compile with `g++ -std=c++17 -O2 -o queue2 queue2.cpp`.

```cpp
// Exercise 17.3-6: a FIFO queue built from two LIFO stacks, with the
// potential function Phi(D) = 2 * (number of items in the input stack)
// computed after every operation, so that the amortized cost
//     a_i = c_i + Phi(D_i) - Phi(D_{i-1})
// can be printed next to the actual cost and checked against the claim
// that it never exceeds 3.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;

class TwoStackQueue {
public:
    long long lastActual = 0, lastAmortized = 0;
    long long totalActual = 0, totalAmortized = 0;
    long long worstActual = 0, worstAmortized = 0;

    void enqueue(int x) {
        long long before = potential();
        sIn.push_back(x);                    // ENQUEUE(e, S_IN): push(S_IN, e)
        account(1, before);
    }

    // Returns the dequeued value; sets `ok` to false on an empty queue.
    int dequeue(bool& ok) {
        long long before = potential();
        long long c = 0;
```

```
        if (sOut.empty()) {                    // move everything across
            // Each element moved costs 2: one POP from S_IN, one PUSH to S_OUT.
            while (!sIn.empty()) { sOut.push_back(sIn.back()); sIn.pop_back(); c += 2; }
        }
        if (sOut.empty()) { ok = false; account(c, before); return 0; }
        int v = sOut.back(); sOut.pop_back(); ++c;
        ok = true;
        account(c, before);
        return v;
    }

    long long potential() const { return 2 * (long long)sIn.size(); }
    size_t size() const { return sIn.size() + sOut.size(); }
    size_t inSize() const { return sIn.size(); }
    size_t outSize() const { return sOut.size(); }
```

Listing 14.3 (Continued) *Exercise 17.3-6: the two-stack queue, with its potential function computed after every operation. Compile with g++ -std=c++17 -O2 -o queue2 queue2.cpp.*

```
private:
    vector<int> sIn, sOut;
    void account(long long c, long long before) {
        lastActual = c;
        lastAmortized = c + potential() - before;
        totalActual += lastActual;
        totalAmortized += lastAmortized;
        if (lastActual > worstActual) worstActual = lastActual;
        if (lastAmortized > worstAmortized) worstAmortized = lastAmortized;
    }
};

int main() {
    TwoStackQueue Q;
    cout << "op          returned  |S_IN|  |S_OUT|   Phi   actual   amortized\n";
    auto line = [&](const string& what, const string& ret) {
        cout << left << setw(12) << what << setw(10) << ret << right
             << setw(6) << Q.inSize() << setw(9) << Q.outSize()
             << setw(7) << Q.potential()
             << setw(8) << Q.lastActual << setw(12) << Q.lastAmortized << "\n";
    };
    for (int i = 1; i <= 5; ++i) { Q.enqueue(i); line("ENQUEUE " + to_string(i), "-"); }
    bool ok;
    for (int i = 0; i < 3; ++i) { int v = Q.dequeue(ok); line("DEQUEUE", to_string(v)); }
    Q.enqueue(6); line("ENQUEUE 6", "-");
    for (int i = 0; i < 3; ++i) { int v = Q.dequeue(ok); line("DEQUEUE", to_string(v)); }

    // A long adversarial run: n/2 enqueues, then n/2 dequeues, repeated.
```

```
    TwoStackQueue R;
    const int n = 200000;
    long long ops = 0;
    while (ops < n) {
        for (int i = 0; i < 1000 && ops < n; ++i) { R.enqueue(i); ++ops; }
        for (int i = 0; i < 1000 && ops < n; ++i) { R.dequeue(ok); ++ops; }
    }
    cout << "\n" << n << " operations in blocks of 1000:\n";
    cout << "  worst single actual cost   = " << R.worstActual << "\n";
    cout << "  worst single amortized cost= " << R.worstAmortized << "\n";
    cout << "  total actual cost          = " << R.totalActual << "\n";
    cout << "  total amortized cost       = " << R.totalAmortized << "\n";
    cout << "  actual cost per operation  = " << fixed << setprecision(6)
         << (double)R.totalActual / ops << "\n";
    cout << "The single worst operation costs " << R.worstActual
         << ", yet the average is at most 3: that gap is what amortization buys.\n";
    return 0;
}
```

Listing 14.3 implements the pseudocode of the two-stack queue line for line. Book: Section 14.6.6, "The program".

What the Queue Prints

```
op          returned  |S_IN|  |S_OUT|   Phi   actual   amortized
ENQUEUE 1   -              1         0     2        1           3
ENQUEUE 2   -              2         0     4        1           3
ENQUEUE 3   -              3         0     6        1           3
ENQUEUE 4   -              4         0     8        1           3
ENQUEUE 5   -              5         0    10        1           3
DEQUEUE     1              0         4     0       11           1
DEQUEUE     2              0         3     0        1           1
DEQUEUE     3              0         2     0        1           1
ENQUEUE 6   -              1         2     2        1           3
DEQUEUE     4              1         1     2        1           1
DEQUEUE     5              1         0     2        1           1
DEQUEUE     6              0         0     0        3           1

200000 operations in blocks of 1000:
  worst single actual cost   = 2001
  worst single amortized cost= 3
  total actual cost          = 400000
  total amortized cost       = 400000
  actual cost per operation  = 2.000000
The single worst operation costs 2001, yet the average is at most 3: that gap is what amortization buys.
```

The `returned` column reads $1,2,3,4,5,6$: first-in-first-out. The sixth line is the expensive `Dequeue`: five items transferred at 2 each plus one pop, $c_i = 11$, and amortized cost 1 because Φ fell from 10 to 0.

At scale: one operation costs 2001, and the average over 200,000 operations is 2.

Book: Section 14.6.6, "The program".

Dynamic Tables: the Problem

The structure you use every day: `std::vector`, Python's `list`, Java's `ArrayList`. Memory comes in fixed blocks, so to grow you allocate a bigger block, copy, and free the old one.

Two goals. *Speed:* $O(1)$ amortized per insertion and deletion. *Space:* the unused part never exceeds a constant fraction of the allocation.

Vocabulary. `TABLE-INSERT`, `TABLE-DELETE`; $num[T]$ items stored, $size[T]$ slots allocated; **load factor** $\alpha(T) = num[T]/size[T]$, with $\alpha = 1$ for an empty table of size 0. **First policy: when $\alpha(T) = 1$, double the table.**

> **A clash of notation.** Here α is a load factor between 0 and 1; in Section 14.8 α is the inverse Ackermann function. They are unrelated.

Book: Section 14.7.1.

TABLE-INSERT

```
TABLE-INSERT(T, x)
 1  if size[T] = 0
 2      then allocate table[T] with 1 slot
 3           size[T] ← 1
 4  if num[T] = size[T]
 5      then allocate new-table with 2 · size[T] slots
 6           insert all items of table[T] into new-table
 7           free table[T]
 8           table[T] ← new-table
 9           size[T] ← 2 · size[T]
10  insert x into table[T]
11  num[T] ← num[T] + 1
```

Figure 14.16 `TABLE-INSERT(T,x)`: lines 1–3 handle the first insertion into a table with no slots, lines 4–9 expand (allocate double, copy, free the old block), and lines 10–11 store x and count it.

Cost of Figure 14.16, one unit per elementary insertion: $c_i = 1$ if there is room; if the table is full, all items are copied and one written, so $c_i = i$.

The expansion condition. The table is full at sizes $1, 2, 4, 8, \ldots$, so insertion i expands exactly when $i - 1$ is a power of 2: insertions $2, 3, 5, 9, 17, \ldots$

The crude bound is again $O(n^2)$. Book: Section 14.7.2.

Aggregate: the Sum

$$\begin{aligned}\sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n\end{aligned}$$

Figure 14.17 The aggregate analysis of `TABLE-INSERT`. One unit for every insertion, plus the expansion costs $2^0+2^1+\cdots+2^{\lfloor \lg n \rfloor}$, which is less than $2n$; total under $3n$.

Baseline n, plus an extra 2^j for the expansion at insertion 2^j+1, $j \le \lfloor \lg n \rfloor$:

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j. \tag{14.13}$$

By Equation (14.6), the sum in Equation (14.13) is $2^{\lfloor \lg n \rfloor + 1} - 1 < 2n$, so the total is under $3n$, as Figure 14.17 states.

> **Why doubling and not adding.** Doubling makes the copying a *geometric* series, under twice the size the table ends with. Growing by a fixed 100 slots makes it *arithmetic*, $\Theta(n^2)$ in total. **The doubling is not an implementation detail; it is the whole idea.**

Book: Section 14.7.3.

Accounting: Three Units per Item

Charge every insertion 3. One unit writes the item; two are banked on it. Each item pays for (1) its own insertion, (2) its own relocation at the next expansion, (3) the relocation of one *older* item that has already been moved once.

Why clause (3). When a full table of size s doubles, $s/2$ of its items are *new* since the last expansion and still hold their savings: $2 \times s/2 = s$ units, exactly enough to copy all s items.

The invariant. The stored credit is $2\times$ (items inserted since the last expansion), a count, hence never negative. So n insertions cost at most $3n$.

Book: Section 14.7.4.

Potential: $\Phi = 2num - size$

The potential.

$$\Phi(T) = 2 \cdot num[T] - size[T]. \tag{14.14}$$

Right after an expansion the table is half full and Equation (14.14) is 0; each insertion raises it by 2; when full it equals $size$, exactly what the next expansion must copy.

Requirements. An empty table gives $\Phi = 0$; the table is always at least half full, so $2num \geq size$ and $\Phi \geq 0$.

Potential: $\Phi(T) = 2 \cdot num[T] - size[T]$.

No expansion in the i-th insertion ($size_i = size_{i-1}$, actual cost 1):

$$a_i = 1 + (2\,num_i - size_i) - (2\,num_{i-1} - size_{i-1}) = 1 + 2 = 3.$$

Expansion in the i-th insertion ($size_i/2 = size_{i-1} = num_i - 1$, actual cost num_i):

$$\begin{aligned} a_i &= num_i + (2\,num_i - size_i) - (2\,num_{i-1} - size_{i-1}) \\ &= num_i + \big(2\,num_i - 2(num_i - 1)\big) - \big(2(num_i - 1) - (num_i - 1)\big) \\ &= num_i + 2 - (num_i - 1) = 3. \end{aligned}$$

Either way the amortized cost of `TABLE-INSERT` is 3, that is $O(1)$.

Figure 14.18 Both cases of the potential analysis of `TABLE-INSERT` come to exactly 3. In the no-expansion case the actual cost is 1 and the potential rises by 2; in the expansion case the actual cost is num_i and the potential collapses by $num_i - 3$.

Figure 14.18 summarizes both cases; the next page writes them out.

The potential has saved up exactly the cost of the disaster it can see coming. Book: Section 14.7.5.

Both Cases Come to Exactly 3

No expansion: $size_i = size_{i-1}$, $num_{i-1} = num_i - 1$, $c_i = 1$:

$$\hat{c}_i = 1 + (2num_i - size_i) - (2(num_i - 1) - size_i) = 1 + 2 = 3.$$

Expansion: $size_i/2 = size_{i-1} = num_i - 1$ and $c_i = num_i$:

$$\hat{c}_i = num_i + (2num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) = num_i + 2 - (num_i - 1) = 3.$$

Both are **equalities**, not bounds with slack: the amortized cost of `TABLE-INSERT` is exactly 3.

Book: Section 14.7.5.

The Picture Is the Proof

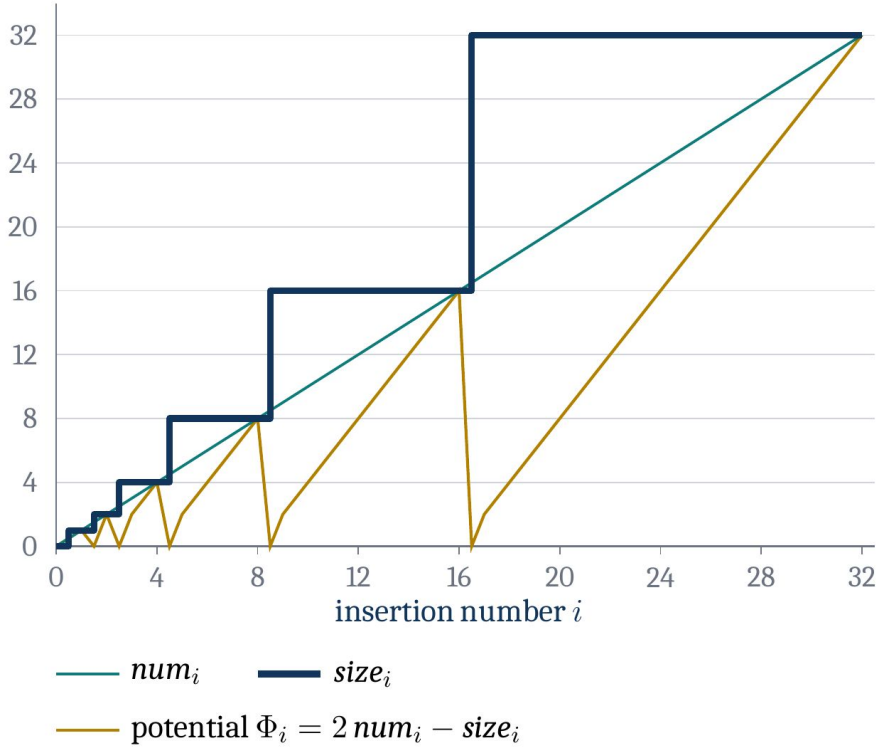


Figure 14.19 A sequence of insertions: the thin diagonal is num, the heavy staircase is $size$, and the sawtooth is $\Phi = 2\,num - size$, which falls to 0 at each expansion and climbs at slope 2 until it meets $size$ again.

Trace the sawtooth of Figure 14.19: just before an expansion $\Phi = size$; the expansion doubles $size$ and Φ drops to 0; then it climbs again at slope 2. **The potential is the savings account**, filled steadily and emptied in one instant by the expansion.

Book: Section 14.7.5, "The picture".

The Table in C++

Listing 14.4 `TABLE-INSERT` with doubling, instrumented with the potential function. Compile with `g++ -std=c++17 -O2 -o dyntable dyntable.cpp`.

```cpp
// TABLE-INSERT with doubling, instrumented three ways at once:
//   * the actual cost c_i of every insertion,
//   * the running total, which the aggregate method bounds by 3n,
//   * the potential Phi(T) = 2*num - size and the amortized cost
//     a_i = c_i + Phi(T_i) - Phi(T_{i-1}), which the text proves is 3.
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <cstring>
using namespace std;

class DynamicTable {
public:
    ~DynamicTable() { free(slot); }

    // Returns the actual cost of this insertion: 1 if there was room,
    // otherwise num_i (one write for x plus num_{i-1} copies).
    long long insert(int x) {
        long long c = 0;
        if (size_ == 0) { slot = (int*)malloc(sizeof(int)); size_ = 1; }
        if (num_ == size_) {
            int* fresh = (int*)malloc(sizeof(int) * 2 * size_);
            memcpy(fresh, slot, sizeof(int) * size_);   // num_ == size_ copies
            c += size_;
            free(slot); slot = fresh; size_ *= 2;
            ++expansions;
        }
        slot[num_++] = x; c += 1;
```

```
        totalActual += c; ++nOps;
        return c;
    }

    long long potential() const { return 2 * num_ - size_; }
    long long num() const { return num_; }
    long long size() const { return size_; }
    long long actual() const { return totalActual; }
    long long ops() const { return nOps; }
    long long expansionCount() const { return expansions; }

private:
    int* slot = nullptr;
    long long num_ = 0, size_ = 0, totalActual = 0, nOps = 0, expansions = 0;
};
```

Listing 14.4 (Continued) *TABLE-INSERT with doubling, instrumented with the potential function. Compile with g++ -std=c++17 -O2 -o dyntable dyntable.cpp.*

```
int main() {
    DynamicTable T;
    cout << " i   num  size  actual c_i   Phi before  Phi after  amortized a_i  running total\n";
    for (int i = 1; i <= 17; ++i) {
        long long before = T.potential();
        long long c = T.insert(i);
        long long after = T.potential();
        cout << setw(3) << i << setw(6) << T.num() << setw(6) << T.size()
             << setw(12) << c << setw(13) << before << setw(11) << after
             << setw(15) << c + after - before << setw(15) << T.actual() << "\n";
    }

    DynamicTable U;
    const long long n = 1000000;
    for (long long i = 0; i < n; ++i) U.insert((int)i);
    cout << "\nn = " << n << " insertions into a table that starts empty:\n";
    cout << "  expansions performed   = " << U.expansionCount() << "\n";
    cout << "  total actual cost      = " << U.actual() << "\n";
    cout << "  aggregate bound 3n     = " << 3 * n << "\n";
    cout << "  measured cost per insert = " << fixed << setprecision(6)
         << (double)U.actual() / (double)U.ops() << "\n";
    cout << "  final num = " << U.num() << ", final size = " << U.size()
         << ", final potential = " << U.potential() << "\n";
    return 0;
}
```

Listing 14.4 uses real `malloc`, `memcpy` and `free`, so the copying is real work. Book: Section 14.7.5, "The program".

What the Table Prints

```
 i   num  size  actual c_i   Phi before  Phi after  amortized a_i  running total
  1     1     1           1            0          1              2              1
  2     2     2           2            1          2              3              3
  3     3     4           3            2          2              3              6
  4     4     4           1            2          4              3              7
  5     5     8           5            4          2              3             12
  6     6     8           1            2          4              3             13
  7     7     8           1            4          6              3             14
  8     8     8           1            6          8              3             15
  9     9    16           9            8          2              3             24
 10    10    16           1            2          4              3             25
 11    11    16           1            4          6              3             26
 12    12    16           1            6          8              3             27
 13    13    16           1            8         10              3             28
 14    14    16           1           10         12              3             29
 15    15    16           1           12         14              3             30
 16    16    16           1           14         16              3             31
 17    17    32          17           16          2              3             48

n = 1000000 insertions into a table that starts empty:
  expansions performed   = 20
  total actual cost      = 2048575
  aggregate bound 3n     = 3000000
  measured cost per insert = 2.048575
  final num = 1000000, final size = 1048576, final potential = 951424
```

The amortized column is a wall of 3s (except the bootstrap insertion). The actual column is expensive exactly at $i = 2, 3, 5, 9, 17$. At $i = 16$ the potential reaches 16, what the next expansion copies, and at $i = 17$ it collapses to 2.

A million insertions: 2,048,575 units against the proved $3,000,000$, with only 20 expansions.

Book: Section 14.7.5, "The program".

Deletion, and Halving at Half

Without contraction, a table that grew to a million and shrank to one still holds a million slots: the space goal fails.

Policy A: when a deletion leaves the table less than half full, halve it.

The thrashing block from a full table, $num = size = s$: `INSERT` (expand, cost $s+1$); `DELETE` ($num = s$, exactly half, no contraction, cost 1); `DELETE` (below half, contract, cost s); `INSERT` (cost 1). Back where we started.

Four operations cost $(s+1)+1+s+1=2s+3$. Fill to $n/4$, then repeat: about $\frac{3n}{16}$ blocks at $\frac{n}{2}$ each, $\frac{3n^2}{32} = \Theta(n^2)$. **Policy A is $\Theta(n)$ amortized.**

> **Why it fails: no slack.** After an expansion the load factor is exactly 1/2, the very point where contraction triggers. **An amortized bound needs room between the trigger points.**

Book: Section 14.7.6.

The Fix: Halve at Quarter-Full

Policy B. Double when full; halve only when a deletion leaves the table **less than a quarter full**. From load $1/2$, $size/4$ deletions must pass before a contraction and $size/2$ insertions before an expansion.

```
THRASHING TEST.  Fill the table to exactly full, then repeat the
block  INSERT, DELETE, DELETE, INSERT  until n/2 operations are done.

n =    1024
  A: shrink below 1/2  table=     256 ops=      512 cost=       65920 cost/op=     128.75  expansions=
128 contractions=128
  B: shrink below 1/4  table=     256 ops=      512 cost=         768 cost/op=       1.50  expansions=
1 contractions=0
n =    4096
  A: shrink below 1/2  table=    1024 ops=     2048 cost=     1050112 cost/op=     512.75  expansions=
512 contractions=512
  B: shrink below 1/4  table=    1024 ops=     2048 cost=        3072 cost/op=       1.50  expansions=
1 contractions=0
n =   16384
  A: shrink below 1/2  table=    4096 ops=     8192 cost=    16783360 cost/op=    2048.75  expansions=
2048 contractions=2048
  B: shrink below 1/4  table=    4096 ops=     8192 cost=       12288 cost/op=       1.50  expansions=
1 contractions=0
n =   65536
  A: shrink below 1/2  table=   16384 ops=    32768 cost=   268460032 cost/op=    8192.75  expansions=
8192 contractions=8192
  B: shrink below 1/4  table=   16384 ops=    32768 cost=       49152 cost/op=       1.50  expansions=
1 contractions=0
n =  262144
  A: shrink below 1/2  table=   65536 ops=   131072 cost=  4295065600 cost/op=   32768.75  expansions=
32768 contractions=32768
  B: shrink below 1/4  table=   65536 ops=   131072 cost=      196608 cost/op=       1.50  expansions=
1 contractions=0
```

```
Potential and worst case on a long mixed run (400000 inserts, then 400000 deletes):
  shrink below 1/2   cost=  1848555  cost/op=2.3107  worst single op=  262145  lowest potential=0  final
num=0 size=1
  shrink below 1/4   cost=  1586412  cost/op=1.9830  worst single op=  262145  lowest potential=0  final
num=0 size=2
```

Measured: Policy A's cost per operation quadruples each time n does (128.75 up to 32768.75); Policy B costs 1.50 at every scale, with one expansion and no contractions. The load factor stays in $[1/4, 1]$.

> **Hysteresis.** When a structure reorganizes at a threshold, the thresholds for growing and shrinking must not coincide, just as a thermostat does not switch every second.

Output of the program of Section 14.7.6 comparing both policies. Book: Section 14.7.6, "The fix" and "Measuring both policies".

The Combined Potential

With $\alpha = num/size$:

$$\Phi(T) = \begin{cases} 2\,num[T] - size[T] & \text{if } \alpha(T) \geq \frac{1}{2}, \\ \frac{size[T]}{2} - num[T] & \text{if } \alpha(T) < \frac{1}{2}. \end{cases} \tag{14.15}$$

Equation (14.15) is **zero at half full** (both branches agree), **non-negative everywhere**, and **rises toward both disasters**: to $size$ as $\alpha \to 1$ (the expansion's cost) and to $size/4$ as $\alpha \to 1/4$ (the contraction's cost).

THEOREM (dynamic tables). Doubling when full and halving below quarter-full, from an empty table, any n `TABLE-INSERT` and `TABLE-DELETE` operations take $O(n)$ in total; each is $O(1)$ amortized, and at most three quarters of the space is ever wasted.

The eight-case analysis is Exercise 14.12, with a full solution in the book. Book: Section 14.7.7.

Union–Find: the Promise Kept

The structure. Sets as trees; each root is the representative; `MakeSet`, `FindSet` (follow parents to the root) and `Union`. The cost of `FindSet(x)` is the depth of x, so the question is: **how deep can trees get?**

Union by rank. Hang the root of smaller rank under the larger. A tree of rank r has at least 2^r nodes (two rank-$(r-1)$ trees make one of rank r), so $r \le \lg n$: every `FindSet` costs at most $\lg n$, a genuine worst-case bound.

Path compression. On the way back from the root, point every node on the path straight at the root. A find can still cost $\Theta(\lg n)$, but only once per path: the work is *invested*.

> $O(\alpha(n))$ **is amortized.** A single `FindSet` can cost $\Theta(\lg n)$; many cannot, because the first one flattens the path. As always, the bound is on the sum.

Tarjan proved the $O(m\,\alpha(n))$ bound in 1975 with a sophisticated potential function; we state it and measure it. Book: Sections 14.8.1 to 14.8.4.

What α Actually Is

The Ackermann family, where $A_{k-1}^{(j+1)}$ applies A_{k-1} to its own output $j+1$ times:

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1. \end{cases} \tag{14.16}$$

The inverse of Equation (14.16):

$$\alpha(n) = \min\left\{k : A_k(1) \geq n\right\}. \tag{14.17}$$

From the definition: $A_0(1)=2$, $A_1(1)=3$, $A_2(1)=7$, $A_3(1)=A_2(7)=2^8 \cdot 8 - 1 = 2047$.

Table 14.1 The values $A_k(1)$ that define α. The jump from $k=3$ to $k=4$ is a jump in kind: 2047 is a number you can write down, and $A_4(1)$ would not fit in the observable universe.

k	$A_k(1)$
0	2
1	3
2	7
3	2047
4	far greater than 2^{2048}

Reading Equation (14.17) off Table 14.1: $A_4(1) = A_3(2047)$, and the very first of its 2048 applications of A_2 already has about 620 digits.

Book: Section 14.8.5.

Why $\alpha(n) \le 4$, and the Measurement

Table 14.2 The inverse Ackermann function. For $\alpha(n)$ to reach 5, you would need more than $A_4(1) > 2^{2048}$ elements.

Range of n	$\alpha(n)$
$n \le 2$	0
$n \le 3$	1
$n \le 7$	2
$n \le 2047$	3
$n \le A_4(1)$, which exceeds 2^{2048}	4

The observable universe holds about $10^{80} \approx 2^{266}$ atoms, far below $A_4(1)$. So by Table 14.2, $\alpha(n) \le 4$ for every input that could physically exist. Fredman and Saks proved $\Omega(\alpha(n))$ amortized is unavoidable for pointer structures.

Measured on $n = 32768$. The naive forest averages 10,922 hops per operation (height 32,767). Union by rank alone is forced to height 15, exactly $\lg n$, at 2.5 hops; rank plus path compression flattens it to height 1 at **1.000** hop per operation.

Treat `FindSet` and `Union` as constant in an engineering estimate, while knowing the honest bound is $O(\alpha(n))$. Book: Sections 14.8.6 and 14.8.7.

Choosing a Method

Table 14.3 The three methods compared. All three are equivalent in power for the examples in this chapter; they differ in how much work each is and in how much they tell you.

	Aggregate	Accounting	Potential
What you must find	a counting argument for the total $T(n)$	a charge per operation type, and a rule saying where credit sits	a function Φ of the structure's state
What you get	one amortized cost, the same for every operation	a different amortized cost for each operation type	a different amortized cost for each operation, which may depend on the state
What you must prove	the counting argument	credit never goes negative	$\Phi(D_i) \geq \Phi(D_0)$ for all i
Best when	there is a clean "each object is touched at most k times" argument	the prepayment attaches naturally to individual objects	the prepayment is a property of the shape of the structure
Scales to	simple structures only	moderately complex structures	splay trees, Fibonacci heaps, union–find

Order of attack from Table 14.3**.** Try aggregate first; reach for accounting when operation types genuinely differ; reach for potential when the trouble is in the structure's *shape*, or when the other two fail.

> **They are not different theorems.** All three prove $\sum c_i \leq \sum \hat{c}_i$. If two methods give different answers, one contains an arithmetic error.

Book: Section 14.9.

Profitina in Practice

The crawl frontier is a dynamic table: a crawl of m pages does $O(m)$ copying, not $O(m^2)$. **The signal buffer is a two-stack queue**: a producer never blocks, `Enqueue` is amortized 3. **Near-duplicate grouping is union–find.**

Profitina Laundry. The intake register is a dynamic table, the pending pile a stack loaded into machines by `Multipop`, the collection line a two-stack queue.

```
ordinary Tuesday           garments=  20000 events=   68324 work=   145646 work/event=   2.1317
expansions=15
worst day (one big load)   garments=  20000 events=   80001 work=   172767 work/event=   2.1596
expansions=15

Intake register alone: 20000 garments logged.
  most expensive single log entry = 16385 slot writes (a table expansion)
  total work                      = 52767
  amortized cost per garment      = 2.6383 slot writes
  aggregate bound proved in 14.7  = 3 per garment, i.e. 60000 in total
  final potential 2*num - size    = 7232
```

The worst day costs **2.16** units per event, an ordinary Tuesday **2.13**: the order of events does not matter, only their number. The single worst event is one expansion of **16,385** slot writes, harmless as a memory copy but a real latency risk under a hard deadline.

The app advises; the user decides. Book: Section 14.10.

Common Mistakes

1. "Amortized" for "average-case": ask where the probability distribution is. **2. "Amortized" for "usually fast"**: a single operation can still cost $\Theta(n)$; a deadline needs a worst-case bound.

3. Amortizing across resets: the bound covers one uninterrupted sequence on one structure. **4. Forgetting $\Phi(D_0)$**: write "$\Phi(D_0)=0$" and "$\Phi \geq 0$" before computing any amortized cost.

5. Asserting the credit stays non-negative instead of saying where it sits. **6. Halving at half-full** (Figure 14.19 only works with slack). **7. Growing by a constant** instead of a constant factor.

8. Forgetting a non-empty start: add $\Phi(D_0)$ to the total. **9. Confusing the two α's. 10. Writing $O(1)$ for $O(\alpha(n))$** in a proof.

Book: Section 14.11.

Summary — Five Things

1. **The question.** What does the *whole sequence* cost? Worst-single-times-n can overstate it by a factor of n. Amortized costs are valid when every prefix of actual costs is covered.
2. **Not average-case.** No probability; the bound holds for every sequence, an adversary's included.
3. **Three methods, one answer.** Aggregate (stack $2n$, counter $< 2n$, Exercise 17.1-3 $\le 3n$); accounting (credit on items, on 1 bits, $2j+k+2$); potential ($\hat{c}_i = c_i + \Delta\Phi_i$, valid when $\Phi(D_i) \ge \Phi(D_0)$).
4. **Dynamic tables.** Doubling gives exactly 3 per insertion with $\Phi = 2num - size$; contract at quarter-full, never at half-full.
5. **Union–find.** Union by rank bounds height by $\lg n$; path compression makes it amortized $O(\alpha(n))$, and $\alpha(n) \le 4$ for every n that could exist.

Book: Section 14.12.

Next Time

Chapter 15 — All-Pairs Shortest Paths. Back to graphs: $\delta(u, v)$ for every pair at once, with matrix-like repeated relaxation.

Shortest paths as matrix multiplication over $(\min, +)$, repeated squaring, Floyd–Warshall's $\Theta(V^3)$ dynamic program, transitive closure, and Johnson's reweighting for sparse graphs.

The habit to keep: *ask what the whole sequence costs, not what the worst step costs*. Every `push_back`, every rehash, every Fibonacci-heap merge relies on an amortized bound.

Homework. Exercises 14.1–14.17; start with 14.4 (powers of 3), 14.9 (growth factor 3) and 14.17 (increment and decrement).

Book: Sections 14.12.1 and 14.13.

Lecture 15 — All-Pairs Shortest Paths

Design and Analysis of Algorithms (EF234405)

Chapter 15 of the book, in miniature, for the classroom.

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*.

Lecturer: Irfan Subakti — yifana@gmail.com

Today: the cheapest route between every ordered pair of vertices, in one table, by four algorithms whose costs we can compare with numbers rather than opinions. Book: Chapter 15, opening paragraph.

Agenda

1. **The problem.** Input matrix W, output matrices D and Π, and printing a route.
2. **The naive price.** $|V|$ runs of Dijkstra or Bellman–Ford, and why Dijkstra may not be used anyway.
3. **Shortest paths as matrix multiplication** over $(\min, +)$: SLOW-APSP in $\Theta(n^4)$, then repeated squaring in $\Theta(n^3 \lg n)$.
4. **Floyd–Warshall** in $\Theta(n^3)$: recurrence, loop order, predecessors, two rounds by hand, in-place safety, negative-cycle test.
5. **Transitive closure** by swapping two operators; **6. Johnson's reweighting** for sparse graphs; then which algorithm when, Profitina, mistakes, summary.

The pages follow the book in order; the exercises at the end of the chapter are the homework. Book: Sections 15.2 to 15.10.

Where We Left Off

Chapter 13, single source. $w(p)=\sum_{i=1}^{k} w(v_{i-1},v_i)$; $\delta(u,v)$ the least weight of a path, ∞ if none; estimates $d[v]$ and predecessors $\pi[v]$, pushed down only by **relaxation**.

Three strategies. Dijkstra (weights ≥ 0): $O(V^2+E)$ with an array, $O(E \lg V)$ with a binary heap, $O(V \lg V+E)$ with a Fibonacci heap. Bellman–Ford (any weights): $O(V E)$, with a negative-cycle test. The DAG algorithm: $\Theta(V+E)$.

> **THEOREM 13.1, RECALLED.** Subpaths of shortest paths are shortest paths. It is the entire reason dynamic programming applies to shortest paths, and today it is used in two different disguises.

Negative edges welcome, negative cycles not. Today's running example has two negative edges and no negative cycle; Section 15.6.14 shows how to check that for free.

Book: Section 15.1.1.

The New Question

> **DEFINITION (all-pairs shortest paths).** Given $G=(V,E)$ with $w: E \to R$, find for **every ordered pair** $u, v \in V$ a shortest path from u to v and its weight $\delta(u,v)$. The output is a table: row u, column v.

Ordered: in a directed graph $u \to v$ and $v \to u$ are different journeys. The table has $|V|^2$ entries and is not symmetric.

Why the whole table? Where to put a depot; the **diameter** of a network (the largest $\delta(u,v)$); the two pages furthest apart in clicks; transitive closure; and preprocessing for $O(1)$ lookups.

From here on $n=|V|$, and we drop the bars inside asymptotic notation: $O(V^3)$ means $O(|V|^3)$.

Book: Section 15.1.2.

The Input: an Adjacency Matrix

Number the vertices $1, 2, \ldots, n$ and store the edge weights of the directed graph $G = (V, E)$ in an $n \times n$ matrix $W = (w_{ij})$:

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of edge } (i,j) & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

Negative-weight edges are allowed; negative-weight cycles are not.

Figure 15.1 The input to every algorithm in this chapter: the $n \times n$ adjacency matrix $W = (w_{ij})$ of Equation (15.1). Negative-weight edges are allowed; negative-weight cycles are not.

Written out, the matrix of Figure 15.1 is

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ \text{the weight of directed edge } (i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases} \tag{15.1}$$

In Equation (15.1), $w_{ii} = 0$ is load-bearing twice today; ∞ never wins a min and poisons any sum, so the listings use `INF = 1000000000LL` and guard every addition against `INF + INF`.

Book: Section 15.2.1.

The Running Example

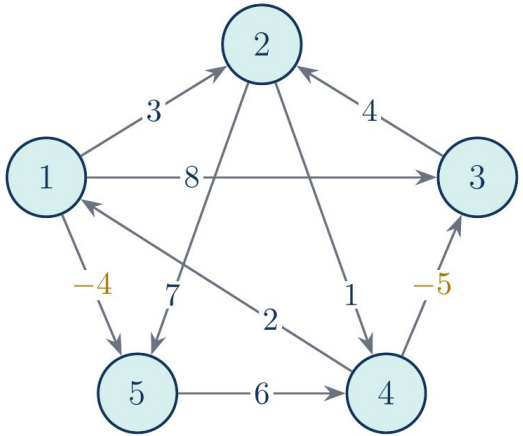


Figure 15.2 The running example: a five-vertex directed graph with nine edges, two of them negative ($1 \to 5$ of weight -4 and $4 \to 3$ of weight -5). It has no negative-weight cycle (§15.6.14 verifies this by machine), so every $\delta(i, j)$ is finite.

Applying Equation (15.1) to Figure 15.2:

$$W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}.$$

Row 1: stay at 0, reach 2 at 3, 3 at 8, no edge to 4, reach 5 at -4. W is not symmetric: $w_{15} = -4$ but $w_{51} = \infty$.

Storing W costs $\Theta(n^2)$ whatever the edge count, which is right because the output is $\Theta(n^2)$ anyway. Book: Section 15.2.1.

The Output: Two Matrices

Distances. $D=(d_{ij})$ with, at termination,

$$d_{ij}=\delta(i,j)\text{ for all }i,j\in V.$$

(15.2)

> **DEFINITION.** The **predecessor matrix** $\Pi=(\pi_{ij})$ has $\pi_{ij}=$ NIL if $i=j$ or there is no path, and otherwise **the predecessor of** j **on some shortest path from** i: the *second-to-last* vertex, not the second.

Row i of Π encodes the **predecessor subgraph** $G_{\pi,i}$, with $V_{\pi,i}=\{j:\pi_{ij}\neq\text{NIL}\}\cup\{i\}$ and $E_{\pi,i}=\{(\pi_{ij},j):j\in V_{\pi,i}-\{i\}\}$: a shortest-paths tree rooted at i. We build n of them at once.

> D **tells you *how much*;** Π **tells you *how*.** D is a price list, Π a pile of n road maps, each folded into one row. Both are $\Theta(n^2)$, so we keep both.

The matrix method of Section 15.4 calls its matrices $L^{(m)}$; the last L is the same D as in Equation (15.2). Book: Section 15.2.2.

Printing a Path out of Π

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
1  if i == j
2      print i
3  elseif π_ij == NIL
4      print "no path from" i "to" j "exists"
5  else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
6      print j
```

Figure 15.3 PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j) recovers the actual route from the predecessor matrix, printing it forward even though Π stores it backward.

In Figure 15.3 the recursive call comes *before* the print: $i = j$ print i; π_{ij} = **NIL** say there is no path; **otherwise** print the route from i to π_{ij}, then print j.

Linear in the path length, at most n; printing all n^2 paths costs $O(n^3)$, the same as computing them.

Scale check: *at* $n = 1000$ *the answer is a million numbers; at* $n = 100,000$ *the output alone is eighty gigabytes and all-pairs is the wrong question. Book: Sections 15.2.3 and 15.2.4.*

The Obvious Algorithm, and Its Price

The naive approach: run a single-source algorithm once per vertex and stack the answers. Correct, simple, and sometimes the fastest thing known.

Table 15.1 The cost of solving all-pairs shortest paths by running a single-source algorithm once per vertex. The last row is the one that hurts, and it is the row the rest of this chapter attacks.

Situation	Single-source algorithm, run V times	Total cost	On a dense graph ($E=\Theta(V^2)$)
all weights ≥ 0, array queue	Dijkstra	$O(V^3+V\,E)$	$O(V^3)$
all weights ≥ 0, binary heap	Dijkstra	$O(V\,E\lg V)$	$O(V^3\lg V)$
all weights ≥ 0, Fibonacci heap	Dijkstra	$O(V^2\lg V+V$	$O(V^3)$
some weight <0	Bellman–Ford	$O(V^2E)$	$O(V^4)$

In Table 15.1 the heap wins only when $E=o(V^2/\lg V)$; the Fibonacci row is the bound Johnson inherits. The disaster is the last row: $O(V^4)$, about 6×10^{10} relaxations at $n=500$.

Floyd–Warshall will match $O(V^3)$ with negative edges allowed; Johnson will beat it on sparse graphs. Book: Sections 15.3.1, 15.3.2 and 15.3.4.

Why Not Dijkstra Anyway?

Dijkstra finalizes the vertex of least estimate, because going *further* can only cost more. A negative edge breaks that: travelling further can make a route cheaper. The program of Section 15.3.3 audits Dijkstra from every vertex against Floyd–Warshall:

```
(a) the five-vertex running example:
   entries Dijkstra got wrong: 0 out of 25

(b) a four-vertex graph built to expose the flaw:
   delta(1,2): Dijkstra says 1, the truth is -1  <-- WRONG
   delta(1,4): Dijkstra says 2, the truth is 0  <-- WRONG
   entries Dijkstra got wrong: 2 out of 16

Moral: on (a) Dijkstra happened to be right, and that is worse than
being wrong, because it invites you to trust it on (b).
```

On graph (b), from vertex 1 Dijkstra finalizes vertex 2 at 1 and relaxes $2 \to 4$ to 2; only then does it find $1 \to 3 \to 2$ at $2+(-3)=-1$, too late to reach vertex 4. The truth is $\delta(1,4)=0$.

> **An algorithm that is wrong only sometimes is more dangerous than one that is always wrong,** because testing will not find it. Reweight first (Section 15.8), or use Floyd–Warshall.

Book: Section 15.3.3.

Step 1: the Structure, by Number of Edges

The dynamic-programming recipe of Chapter 7: characterize an optimal solution; define its value recursively; compute bottom up; construct the solution. We follow it twice today, with two different characterizations.

- All subpaths of a shortest path are shortest paths (Theorem 13.1).
- Let p be a shortest path from i to j with at most m edges; with no negative-weight cycles, such an m is finite.
- If $i = j$, then p has weight 0 and no edges.
- If $i \neq j$, split p as $i \overset{p'}{\rightsquigarrow} k \to j$, where p' has at most $m-1$ edges. By Theorem 13.1, p' is a shortest path from i to k, so $\delta(i,j) = \delta(i,k) + w_{kj}$.

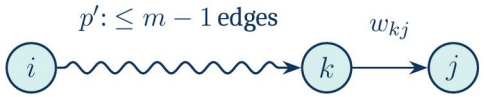


Figure 15.4 The structure of a shortest path with at most m edges. If $i \neq j$, chop the last edge off and what remains must itself be a shortest path — this is Theorem 13.1 in action.

As Figure 15.4 shows: if $i = j$ the best path is empty, weight 0. If $i \neq j$, split off the last edge $k \to j$; the rest p' has at most $m-1$ edges and is shortest by Theorem 13.1, so $\delta(i,j) = \delta(i,k) + w_{kj}$. We do not know k, but there are only n candidates.

Cover the last road segment with your thumb: what is left must be the cheapest way to where the thumb starts. Book: Sections 15.4.1 and 15.4.2.

Step 2: Defining $l_{ij}^{(m)}$

- Let $l_{ij}^{(m)}$ be the minimum weight of any path from i to j that contains **at most** m edges.
- For $m = 0$ there is a path from i to j with no edges if and only if $i = j$, so

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

- For $m \geq 1$, $l_{ij}^{(m)}$ is the smaller of $l_{ij}^{(m-1)}$ (a best path with at most $m-1$ edges) and the best path with exactly m edges, found by trying every possible predecessor k of j.

Figure 15.5 The definition of $l_{ij}^{(m)}$ and its base case $l_{ij}^{(0)}$. Notice that the superscript is in parentheses — it counts edges, it is not an exponent.

DEFINITION. $l_{ij}^{(m)}$ is the **minimum weight of any path from i to j with at most m edges**, ∞ if there is none. The superscript (m) in Figure 15.5 is an *edge budget*, not a power.

Base case: with no edges you cannot move,

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j. \end{cases} \tag{15.3}$$

In matrix form, Equation (15.3) is 0 on the diagonal and ∞ elsewhere: Section 15.4.9 shows it is the identity of our new multiplication.

Book: Section 15.4.3.

The Recurrence, and Its Simplification

Recursively:

$$l_{ij}^{(m)} = \min\Big(l_{ij}^{(m-1)},\ \min_{1\le k\le n}\{l_{ik}^{(m-1)} + w_{kj}\}\Big)$$
$$= \min_{1\le k\le n}\{l_{ik}^{(m-1)} + w_{kj}\} \qquad \text{since } w_{jj} = 0 \text{ for all } j$$

With no negative-weight cycles, whenever $\delta(i,j) < \infty$ there is a shortest path from i to j that is simple, so it has at most $n-1$ edges; a path with more edges cannot weigh less. Therefore

$$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots$$

Figure 15.6 The recurrence of Equation (15.4), its one-line simplification, and the fact, Equation (15.6), that the sequence $l_{ij}^{(m)}$ stops changing at $m = n-1$.

$$l_{ij}^{(m)} = \min\left(l_{ij}^{(m-1)},\ \min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right). \tag{15.4}$$

In Equation (15.4), summarized in Figure 15.6, the first argument covers an extra edge that is not needed; the second tries every last hop $k \to j$.

The candidate $k = j$ gives $l_{ij}^{(m-1)} + w_{jj} = l_{ij}^{(m-1)}$, so the first argument is already inside the inner minimum:

$$l_{ij}^{(m)} = \min_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}, w_{jj} = 0. \tag{15.5}$$

One minimum over one index is the shape of a matrix product; with ∞ on the diagonal of W the collapse to Equation (15.5) would fail. Check: $l_{24}^{(2)} = \min\{\infty, 1, \infty, 1, 13\} = 1$. Book: Sections 15.4.4 and 15.4.5.

When Does the Sequence Stop Changing?

Every shortest path has at most $n-1$ edges (Theorem 13.4): delete any cycle, weight ≥ 0, until the path is simple.

Extra edges never help: nothing beats a shortest path. So

$$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots \tag{15.6}$$

> **Where "no negative cycles" is spent.** In the first half of the argument for Equation (15.6), and only there. With a negative cycle, $l_{ij}^{(m)}$ would keep falling for ever, correctly, because $\delta(i,j) = -\infty$.

The sequence $l_{ij}^{(0)}, l_{ij}^{(1)}, \ldots$ is non-increasing and constant from $n-1$ on. Book: Section 15.4.6.

Step 3: EXTEND-SHORTEST-PATHS

Start from $L^{(1)} = W$ (with Equation (15.3), only $k = i$ survives, giving w_{ij}) and compute $L^{(2)}, \ldots, L^{(n-1)}$, one edge of budget at a time.

```
EXTEND-SHORTEST-PATHS(L, W)
1  n = L.rows
2  let L' = (l'_ij) be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          l'_ij = ∞
6          for k = 1 to n
7              l'_ij = min(l'_ij, l_ik + w_kj)
8  return L'
```

Figure 15.7 EXTEND-SHORTEST-PATHS(L, W), which takes a matrix of best-routes-so-far and extends every one of them by one more edge. Three nested loops, so $\Theta(n^3)$ time.

In Figure 15.7, lines 3–4 visit every cell, line 5 sets it to ∞ (the identity for $\min$), and lines 6–7 keep the running minimum $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$. Three nested loops: $\Theta(n^3)$.

Book: Section 15.4.7.

It Is Matrix Multiplication

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C = (c_ij) be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

Figure 15.8 SQUARE-MATRIX-MULTIPLY(A, B), the $\Theta(n^3)$ schoolbook algorithm for $C = A \cdot B$. Set it beside Figure 15.7 and the two differ in exactly two places.

Table 15.2 EXTEND-SHORTEST-PATHS is SQUARE-MATRIX-MULTIPLY with $\min$ substituted for $+$, $+$ substituted for $\times$, and ∞ substituted for 0 as the starting value; lines 1–4 are identical.

Line	SQUARE-MATRIX-MULTIPLY	EXTEND-SHORTEST-PATHS
5	$c_{ij} = 0$	$l'_{ij} = \infty$
6	**for** $k = 1$ **to** n	**for** $k = 1$ **to** n
7	$c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$	$l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$
8	**return** C	**return** L'

Table 15.2, read against Figure 15.8: $+ \to \min$, $\times \to +$, $0 \to \infty$. Writing $A \odot B$ for the result, $L^{(m)} = L^{(m-1)} \odot W = W^m$, and $L^{(n-1)} = W^{n-1}$ holds the answer.

Superscripts on W are genuine powers under $\odot$; parenthesized superscripts on L are edge budgets. They agree in value. Book: Section 15.4.8.

The $(\min, +)$ Semiring

A **semiring** has an associative, commutative "addition" $\oplus$ with identity $\acute{0}$, an associative "multiplication" $\otimes$ with identity $\acute{1}$, and distributivity. $R \cup \{\infty\}$ with $\oplus = \min$, $\otimes = +$ is one: $\acute{0} = \infty$, $\acute{1} = 0$, and $a + \min(b, c) = \min(a + b, a + c)$.

Three consequences for free. The identity matrix is $L^{(0)}$ of Equation (15.3). The product $\odot$ is **associative**, which Section 15.5 sells. And other semirings solve other problems with the same triple loop.

$(\vee, \wedge)$ gives reachability (Section 15.7); $(\max, \min)$ gives widest paths; $(+, \times)$ over path counts counts shortest paths (Exercise 15.14).

Book: Section 15.4.9.

SLOW-ALL-PAIRS-SHORTEST-PATHS

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1  n = W.rows
2  L^(1) = W
3  for m = 2 to n − 1
4      let L^(m) be a new n × n matrix
5      L^(m) = EXTEND-SHORTEST-PATHS(L^(m−1), W)
6  return L^(n−1)
```

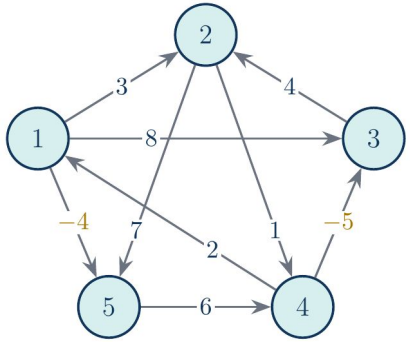
Figure 15.9 SLOW-ALL-PAIRS-SHORTEST-PATHS(W). It performs $n-2$ calls to EXTEND-SHORTEST-PATHS, each costing $\Theta(n^3)$, for a total of $\Theta(n^4)$.

Figure 15.9: line 2 sets $L^{(1)}=W$; lines 3–5 compute $L^{(m)}=L^{(m-1)}\odot W$ for $m=2,\dots,n-1$; line 6 returns $L^{(n-1)}$. $(n-2)\cdot\Theta(n^3)=\Theta(n^4)$.

No better than $|V|\times$ Bellman–Ford on a dense graph. Its purpose is *insight*, which the next pages turn into a saving.

Book: Section 15.4.10.

Every Intermediate Matrix



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Figure 15.10 The five-vertex graph and the sequence $L^{(1)}, L^{(2)}, L^{(3)}, L^{(4)}$ computed by SLOW-ALL-PAIRS-SHORTEST-PATHS. Since $n = 5$, the last matrix $L^{(4)} = L^{(n-1)}$ holds the shortest-path weights, and $L^{(5)} = L^{(4)} \odot W$ equals $L^{(4)}$.

Read l_{12} across Figure 15.10: 3 in $L^{(1)}$, $L^{(2)}$, $L^{(3)}$, then 1 in $L^{(4)}$, because the cheapest route $1 \to 5 \to 4 \to 3 \to 2$, of weight $-4+6-5+4=1$, needs all four edges. That entry is why the algorithm cannot stop early.

The program of Section 15.4.11 reproduces every matrix, and a fifth product changes nothing. Book: Sections 15.4.11 and 15.4.13.

Row 4 of $L^{(2)}$, Cell by Cell

By Equation (15.5) with $m=2$ and row 4 of W equal to $(2,\infty,-5,0,\infty)$, only $k\in\{1,3,4\}$ can contribute.

$l_{41}^{(2)}=\min\{2+0,\infty,\infty,0+2,\infty\}=2$. $l_{42}^{(2)}=\min\{2+3,\infty,-5+4,\infty,\infty\}=-1$, via $4\to3\to2$: the negative edge pays.

$l_{43}^{(2)}=\min\{2+8,\infty,-5+0,0+(-5),\infty\}=-5$. $l_{44}^{(2)}=0$. $l_{45}^{(2)}=\min\{2+(-4),\infty,\infty,\infty,\infty\}=-2$, via $4\to1\to5$.

Row 4 of $L^{(2)}$ is $(2,-1,-5,0,-2)$; row 2 is $(3,0,-4,1,7)$. *Each product lets every route grow by one more edge.*

Book: Section 15.4.12.

Repeated Squaring

We only want the last matrix, and by Equation (15.6) any budget $\geq n-1$ gives it: we are free to overshoot. **Associativity** lets us write $W^4 = (W \odot W) \odot (W \odot W)$: two products instead of three.

We need only $L^{(n-1)}$, and $L^{(m)} = L^{(n-1)}$ for every $m \geq n-1$. Because the "multiplication" of EXTEND-SHORTEST-PATHS is associative, square repeatedly:

$$
\begin{aligned}
L^{(1)} &= W, \\
L^{(2)} &= W^2 = W \cdot W, \\
L^{(4)} &= W^4 = W^2 \cdot W^2, \\
L^{(8)} &= W^8 = W^4 \cdot W^4, \\
&\vdots \\
L^{\left(2^{\lceil \lg(n-1) \rceil}\right)} &= W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}.
\end{aligned}
$$

Since $2^{\lceil \lg(n-1) \rceil} \geq n-1$, the last product equals $L^{(n-1)}$ after only $\lceil \lg(n-1) \rceil$ products.

Figure 15.11 Only $\lceil \lg(n-1) \rceil$ products are needed, because each one doubles the edge budget. The final product $L^{\left(2^{\lceil \lg(n-1) \rceil}\right)}$ equals $L^{(n-1)}$ by Equation (15.6).

Figure 15.11: budgets $1, 2, 4, 8, \ldots$; after t squarings the budget is 2^t, and the least t with $2^t \geq n-1$ is $\lceil \lg(n-1) \rceil$. For $n = 1000$: 10 products instead of 998.

The same trick computes a^b in $\lg b$ multiplications, here in a different semiring. Book: Sections 15.5.1 to 15.5.4.

FASTER-ALL-PAIRS-SHORTEST-PATHS

FASTER-ALL-PAIRS-SHORTEST-PATHS(W)

```
1  n = W.rows
2  L^(1) = W
3  m = 1
4  while m < n − 1
5      let L^(2m) be a new n × n matrix
6      L^(2m) = EXTEND-SHORTEST-PATHS(L^(m), L^(m))
7      m = 2m
8  return L^(m)
```

Figure 15.12 FASTER-ALL-PAIRS-SHORTEST-PATHS(W). Line 6 squares the current matrix rather than multiplying it by W, which is the whole difference from Figure 15.9.

Table 15.3 The two schedules on the five-vertex running example. The slow version adds one to the edge budget per product; the fast version doubles it.

Step	SLOW-APSP computes	Budget	FASTER-APSP computes	Budget
start	$L^{(1)} = W$	1	$L^{(1)} = W$	1
1	$L^{(2)} = L^{(1)} \odot W$	2	$L^{(2)} = L^{(1)} \odot L^{(1)}$	2
2	$L^{(3)} = L^{(2)} \odot W$	3	$L^{(4)} = L^{(2)} \odot L^{(2)}$	4 — stop
3	$L^{(4)} = L^{(3)} \odot W$	4 — stop	—	—
products	$n-2 = 3$		$\lceil \lg(n-1) \rceil = 2$	

Figure 15.12 changes two lines: square instead of multiplying by W, and double m. Table 15.3 compares the schedules. Cost: $\Theta(n^3 \lceil \lg(n-1) \rceil) = \Theta(n^3 \lg n)$.

Book: Sections 15.5.5 and 15.5.6.

Repeated Squaring in C++

Listing 15.1 FASTER-ALL-PAIRS-SHORTEST-PATHS, the repeated-squaring version. Compile with `g++ -std=c++17 -O2 -o faster_apsp faster_apsp.cpp`.

```cpp
// Chapter 15, Listing: FASTER-ALL-PAIRS-SHORTEST-PATHS -- repeated squaring.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;
typedef vector<vector<long long>> Matrix;

Matrix extendShortestPaths(const Matrix& A, const Matrix& B) {
    int n = (int)A.size();
    Matrix C(n, vector<long long>(n, INF));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            C[i][j] = INF;
            for (int k = 0; k < n; ++k)
                if (A[i][k] < INF && B[k][j] < INF)
                    C[i][j] = min(C[i][j], A[i][k] + B[k][j]);
        }
    return C;
}

void printMatrix(const string& name, const Matrix& M) {
    int n = (int)M.size();
    cout << name << " =\n";
    for (int i = 0; i < n; ++i) {
        cout << "   ";
```

```
        for (int j = 0; j < n; ++j) {
            if (M[i][j] >= INF) cout << setw(5) << "inf";
            else                cout << setw(5) << M[i][j];
        }
        cout << '\n';
    }
    cout << '\n';
}
```

Listing 15.1 (Continued) *FASTER-ALL-PAIRS-SHORTEST-PATHS, the repeated-squaring version. Compile with g++ -std=c++17 -O2 -o faster_apsp faster_apsp.cpp.*

```cpp
int main() {
    const int n = 5;
    Matrix W = {
        {  0,   3,   8, INF,  -4},
        {INF,   0, INF,   1,   7},
        {INF,   4,   0, INF, INF},
        {  2, INF,  -5,   0, INF},
        {INF, INF, INF,   6,   0}
    };

    // FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
    Matrix L = W;            // line 2: L^(1) = W
    int m = 1;               // line 3
    int products = 0;
    printMatrix("L^(1)", L);
    while (m < n - 1) {                     // line 4
        L = extendShortestPaths(L, L);      // line 6: L^(2m) = L^(m) . L^(m)
        ++products;
        m = 2 * m;                          // line 7
        printMatrix("L^(" + to_string(m) + ")", L);
    }
    cout << "n = " << n
         << ", matrix products performed = " << products
         << ", final exponent m = " << m << '\n';
    return 0;
}
```

```
L^(1) =
       0    3    8  inf   -4
     inf    0  inf    1    7
     inf    4    0  inf  inf
       2  inf   -5    0  inf
     inf  inf  inf    6    0

L^(2) =
       0    3    8    2   -4
       3    0   -4    1    7
     inf    4    0    5   11
       2   -1   -5    0   -2
       8  inf    1    6    0

L^(4) =
       0    1   -3    2   -4
       3    0   -4    1   -1
       7    4    0    5    3
       2   -1   -5    0   -2
       8    5    1    6    0

n = 5, matrix products performed = 2, final exponent m = 4
```

Listing 15.1 leaps from budget 2 to budget 4; $L^{(3)}$ is never computed, and $L^{(4)}$ is identical to the slow version's.

Book: Section 15.5.6.

What Squaring Does Not Give Us

It is still $\Theta(n^3 \lg n)$, and Floyd–Warshall does $\Theta(n^3)$ with a smaller constant. The value of the matrix view is the *idea*: shortest paths live in a semiring.

Strassen does not help. Fast matrix multiplication relies on **subtraction**, and $(\min, +)$ has none: nothing undoes a min. The sub-cubic all-pairs algorithms that exist take a different route.

> **The idea generalizes.** The same triple loop, over different semirings, gives transitive closure, widest paths and path counts; Section 15.7 is the first example.

Book: Section 15.5.7.

Floyd–Warshall: a Different Cut

A dynamic program in $\Theta(V^3)$ that allows negative edges but not negative cycles. Section 15.4 cut a path by **how many edges** it used; Floyd and Warshall cut it by **which vertices it may pass through**.

DEFINITION. An **intermediate vertex** of a simple path $\langle v_1, \ldots, v_l \rangle$ is any vertex other than v_1 and v_l.

Let p be a shortest path from i to j whose intermediate vertices all lie in $\{1, 2, \ldots, k\}$. The relationship depends on whether k is an intermediate vertex of p:

- k **is not on** p. Then every intermediate vertex of p is in $\{1, \ldots, k-1\}$, and p is also a shortest path from i to j with all intermediate vertices in $\{1, \ldots, k-1\}$.
- k **is on** p. Split p as $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$; both p_1 and p_2 have all intermediate vertices in $\{1, \ldots, k-1\}$.

Figure 15.13 The structure of a shortest path whose intermediate vertices lie in $\{1, \ldots, k\}$, split according to whether k itself is used.

Figure 15.13: let p be a shortest path from i to j with all intermediates in $\{1, \ldots, k\}$. **Case 1:** k is not intermediate, so p already lives in $\{1, \ldots, k-1\}$. **Case 2:** k is intermediate, so $p = i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$.

Book: Sections 15.6.1 and 15.6.2.

Why Both Halves Use Only $\{1,\ldots,k-1\}$

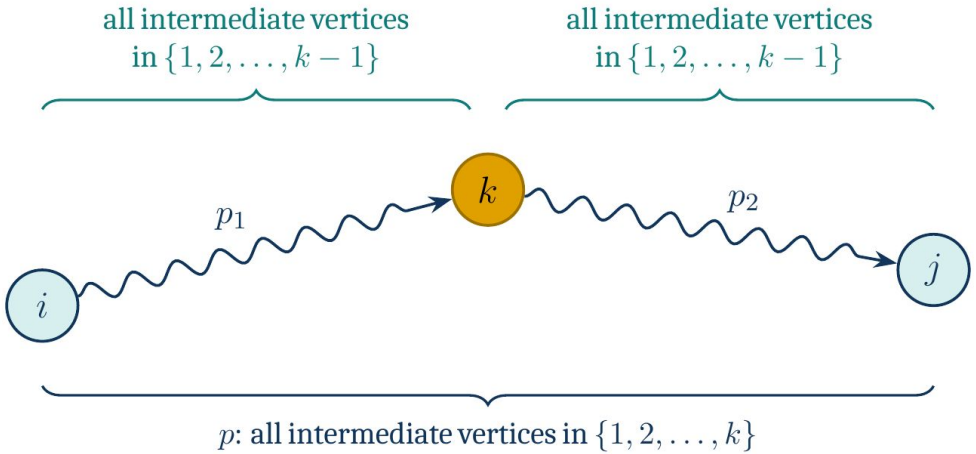


Figure 15.14 Path p is a shortest path from i to j and k is its highest-numbered intermediate vertex. Both halves p_1 and p_2 then have all their intermediate vertices in $\{1,2,\ldots,k-1\}$.

In Figure 15.14, p_1 is shortest by Theorem 13.1. And k is **not intermediate on** p_1: it is p_1's endpoint, and it cannot reappear inside because p is simple. The same holds for p_2.

> Cut the cheapest route through the first seven vertices at vertex 7: each half is cheapest, and neither passes *through* 7. **Two smaller problems of the same shape**, which is optimal substructure.

Book: Section 15.6.3.

The Floyd–Warshall Recurrence

Let $d_{ij}^{(k)}$ be the weight of a shortest path from i to j all of whose intermediate vertices are in $\{1, 2, \ldots, k\}$.

For $k = 0$ the path has no intermediate vertices at all, so it has at most one edge and $d_{ij}^{(0)} = w_{ij}$. Recursively,

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

Every path has all its intermediate vertices in $\{1, \ldots, n\}$, so the answer is $D^{(n)} = (d_{ij}^{(n)})$: $d_{ij}^{(n)} = \delta(i, j)$ for all $i, j \in V$.

Figure 15.15 The definition of $d_{ij}^{(k)}$, the base case $d_{ij}^{(0)} = w_{ij}$, and recurrence of Equation (15.7). The last matrix $D^{(n)} = (d_{ij}^{(n)})$ is the answer.

With $d_{ij}^{(k)}$ the least weight of a path from i to j whose intermediates lie in $\{1, \ldots, k\}$, as in Figure 15.15:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1. \end{cases} \tag{15.7}$$

Any path's intermediates lie in $\{1, \ldots, n\}$, so

$$d_{ij}^{(n)} = \delta(i, j) \text{ for all } i, j \in V. \tag{15.8}$$

Both superscripts in the second term of Equation (15.7) **are** $(k-1)$**.** Writing (k) would be circular. And Equation (15.7) takes a minimum over **two** candidates, n times, against n candidates in Equation (15.5): that is the whole speed difference.

Equation (15.8) meets the goal Equation (15.2). Book: Section 15.6.4.

The Algorithm

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

Figure 15.16 FLOYD-WARSHALL(W): five words of state and three nested loops. Line 7 is recurrence of Equation (15.7) written out.

Figure 15.16: line 2 sets $D^{(0)}=W$; line 3 runs $k=1..n$, enlarging the permitted set; lines 5–6 visit every pair; line 7 applies Equation (15.7); line 8 returns $D^{(n)}$.

Running time. Line 7 costs $O(1)$ and runs n^3 times: $\Theta(n^3)$, with a tiny constant and contiguous memory access along rows i and k.

The values $d_{ij}^{(k)}$ are computed in order of increasing k, and that order is a correctness requirement. Book: Sections 15.6.5 and 15.6.12.

Why k Must Be the Outermost Loop

Every value read in round k must carry superscript $(k-1)$. With k outermost that is automatic. With i outermost and k innermost, cell (i, j) may read d_{kj} from a row this pass has not visited yet. The program of Section 15.6.6 shows both failure modes:

```
(a) the wrong order loses the path entirely:
  k outermost (correct):
        0     3     2     1
      inf     0   inf   inf
      inf     1     0   inf
      inf     2     1     0
  i outermost, k innermost (wrong):
        0   inf     2     1
      inf     0   inf   inf
      inf     1     0   inf
      inf     2     1     0
  the two agree: NO
  delta(1,2): correct = 3, wrong = inf (no path found at all)
  passes of the wrong order needed before it matches: 2

(b) the wrong order finds a path, but overcharges:
  k outermost (correct):
        0    10     9     6
      inf     0   inf   inf
      inf     1     0   inf
      inf     4     3     0
  i outermost, k innermost (wrong):
        0    12     9     6
      inf     0   inf   inf
      inf     1     0   inf
      inf     4     3     0
  the two agree: NO
  delta(1,2): correct = 10, wrong = 12
```

```
passes of the wrong order needed before it matches: 2
```

Graph (a), the path $1 \to 4 \to 3 \to 2$: the wrong order reports **no path** from 1 to 2 where one of weight 3 exists. Graph (b): it reports **12** instead of **10**, a plausible wrong number that is harder to notice.

> **The most common Floyd–Warshall bug:** `for (i) for (j) for (k)` instead of `for (k) for (i) for (j)`. It compiles, it runs, and it is wrong. k **goes first.**

Repeating the wrong order converges here after two passes, but the number of passes depends on the numbering, so the $\Theta(n^3)$ guarantee is lost. Book: Section 15.6.6.

The Predecessor Matrix

$\pi_{ij}^{(k)}$ is the predecessor of j on a shortest path from i with intermediates in $\{1,\ldots,k\}$. **Base case:** only the direct edge,

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty. \end{cases} \tag{15.9}$$

Recursive case, starting from Equation (15.9): unchanged if we did not route through k; otherwise the predecessor of j on the best route **from** k:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases} \tag{15.10}$$

The natural guess $\pi_{ij}^{(k)}=k$ is wrong. Vertex k is in the *middle* of the route. In code, Equation (15.10) is `P[i][j] = P[k][j];`, not `P[i][j] = k;`.

Book: Section 15.6.7.

The Complete Trace

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figure 15.17 The complete sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by Floyd–Warshall for the graph of Figure 15.2. Row $k=0$ is the input; row $k=5$ is the answer.

Figure 15.17 shows six distance matrices and six predecessor matrices. $D^{(5)}$ equals the $L^{(4)}$ of the matrix method: two different algorithms, one answer.

Book: Section 15.6.8.

Floyd–Warshall in C++

Listing 15.2 Floyd–Warshall with the predecessor matrix, printing every $D^{(k)}$ and $\Pi^{(k)}$ and then every route. Compile with `g++ -std=c++17 -O2 -o floyd_warshall floyd_warshall.cpp`.

```cpp
// Chapter 15, Listing: the Floyd-Warshall algorithm, with the predecessor matrix
// and path reconstruction, traced on the running example graph.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

const long long INF = 1000000000LL;
const int NIL = -1;

typedef vector<vector<long long>> Matrix;
typedef vector<vector<int>>       PiMatrix;

void printD(const string& name, const Matrix& D) {
    int n = (int)D.size();
    cout << name << " =\n";
    for (int i = 0; i < n; ++i) {
        cout << "   ";
        for (int j = 0; j < n; ++j) {
            if (D[i][j] >= INF) cout << setw(6) << "inf";
            else                cout << setw(6) << D[i][j];
        }
        cout << '\n';
    }
}

void printPi(const string& name, const PiMatrix& P) {
```

```
    int n = (int)P.size();
    cout << name << " =\n";
    for (int i = 0; i < n; ++i) {
        cout << "   ";
        for (int j = 0; j < n; ++j) {
            if (P[i][j] == NIL) cout << setw(6) << "NIL";
            else                cout << setw(6) << P[i][j] + 1;   // 1-based for printing
        }
        cout << '\n';
    }
}
```

Listing 15.2 (Continued) *Floyd–Warshall with the predecessor matrix, printing every D^{(k)} and \Pi^{(k)} and then every route. Compile with g++ -std=c++17 -O2 -o floyd_warshall floyd_warshall.cpp.*

```
// PRINT-ALL-PAIRS-SHORTEST-PATH(Pi, i, j), exactly as in the figure of Section 15.2.3.
void printAllPairsShortestPath(const PiMatrix& P, int i, int j) {
    if (i == j) {
        cout << i + 1;
    } else if (P[i][j] == NIL) {
        cout << "no path from " << i + 1 << " to " << j + 1 << " exists";
    } else {
        printAllPairsShortestPath(P, i, P[i][j]);
        cout << " -> " << j + 1;
    }
}

int main() {
    const int n = 5;
    Matrix W = {
        {  0,   3,   8, INF,  -4},
        {INF,   0, INF,   1,   7},
        {INF,   4,   0, INF, INF},
        {  2, INF,  -5,   0, INF},
        {INF, INF, INF,   6,   0}
    };

    // D^(0) = W, and Pi^(0) from the base case for Pi in Section 15.6.7.
    Matrix   D = W;
    PiMatrix P(n, vector<int>(n, NIL));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (i != j && W[i][j] < INF) P[i][j] = i;
```

```
    printD("D^(0)", D);   printPi("Pi^(0)", P);   cout << '\n';

    // FLOYD-WARSHALL(W), done in place: line 7 of the pseudocode.
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (D[i][k] < INF && D[k][j] < INF && D[i][k] + D[k][j] < D[i][j]) {
                    D[i][j] = D[i][k] + D[k][j];
                    P[i][j] = P[k][j];            // j's predecessor on the path through k
                }
        printD("D^(" + to_string(k + 1) + ")", D);
        printPi("Pi^(" + to_string(k + 1) + ")", P);
        cout << '\n';
    }
```

Listing 15.2 (Continued) *Floyd–Warshall with the predecessor matrix, printing every D^{(k)} and \Pi^{(k)} and then every route. Compile with g++ -std=c++17 -O2 -o floyd_warshall floyd_warshall.cpp.*

```
    // Negative-weight cycles would show up as a negative entry on the diagonal.
    bool negativeCycle = false;
    for (int i = 0; i < n; ++i) if (D[i][i] < 0) negativeCycle = true;
    cout << "negative-weight cycle detected? " << (negativeCycle ? "yes" : "no") << "\n\n";

    cout << "Shortest paths read out of Pi^(5):\n";
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            if (i == j) continue;
            cout << "  " << i + 1 << " to " << j + 1 << " (weight " << setw(3) << D[i][j] << "): ";
            printAllPairsShortestPath(P, i, j);
            cout << '\n';
        }
    return 0;
}
```

Listing 15.2 keeps one matrix `D` and overwrites it in place, which Section 15.6.13 proves safe. Book: Section 15.6.8.

What It Prints

```
D^(0) =
       0     3     8   inf    -4
     inf     0   inf     1     7
     inf     4     0   inf   inf
       2   inf    -5     0   inf
     inf   inf   inf     6     0
Pi^(0) =
     NIL     1     1   NIL     1
     NIL   NIL   NIL     2     2
     NIL     3   NIL   NIL   NIL
       4   NIL     4   NIL   NIL
     NIL   NIL   NIL     5   NIL

D^(1) =
       0     3     8   inf    -4
     inf     0   inf     1     7
     inf     4     0   inf   inf
       2     5    -5     0    -2
     inf   inf   inf     6     0
Pi^(1) =
     NIL     1     1   NIL     1
     NIL   NIL   NIL     2     2
     NIL     3   NIL   NIL   NIL
       4     1     4   NIL     1
     NIL   NIL   NIL     5   NIL

D^(2) =
       0     3     8     4    -4
```

```
      inf     0   inf     1     7
      inf     4     0     5    11
        2     5    -5     0    -2
      inf   inf   inf     6     0
Pi^(2) =
      NIL     1     1     2     1
      NIL   NIL   NIL     2     2
      NIL     3   NIL     2     2
        4     1     4   NIL     1
      NIL   NIL   NIL     5   NIL

D^(3) =
        0     3     8     4    -4
      inf     0   inf     1     7
      inf     4     0     5    11
        2    -1    -5     0    -2
      inf   inf   inf     6     0
Pi^(3) =
      NIL     1     1     2     1
      NIL   NIL   NIL     2     2
      NIL     3   NIL     2     2
        4     3     4   NIL     1
      NIL   NIL   NIL     5   NIL
```

```
D^(4) =
       0     3    -1     4    -4
       3     0    -4     1    -1
       7     4     0     5     3
       2    -1    -5     0    -2
       8     5     1     6     0
Pi^(4) =
     NIL     1     4     2     1
       4   NIL     4     2     1
       4     3   NIL     2     1
       4     3     4   NIL     1
       4     3     4     5   NIL

D^(5) =
       0     1    -3     2    -4
       3     0    -4     1    -1
       7     4     0     5     3
       2    -1    -5     0    -2
       8     5     1     6     0
Pi^(5) =
     NIL     3     4     5     1
       4   NIL     4     2     1
       4     3   NIL     2     1
       4     3     4   NIL     1
       4     3     4     5   NIL

negative-weight cycle detected? no
```

```
Shortest paths read out of Pi^(5):
  1 to 2 (weight   1): 1 -> 5 -> 4 -> 3 -> 2
  1 to 3 (weight  -3): 1 -> 5 -> 4 -> 3
  1 to 4 (weight   2): 1 -> 5 -> 4
  1 to 5 (weight  -4): 1 -> 5
  2 to 1 (weight   3): 2 -> 4 -> 1
  2 to 3 (weight  -4): 2 -> 4 -> 3
  2 to 4 (weight   1): 2 -> 4
  2 to 5 (weight  -1): 2 -> 4 -> 1 -> 5
  3 to 1 (weight   7): 3 -> 2 -> 4 -> 1
  3 to 2 (weight   4): 3 -> 2
  3 to 4 (weight   5): 3 -> 2 -> 4
  3 to 5 (weight   3): 3 -> 2 -> 4 -> 1 -> 5
  4 to 1 (weight   2): 4 -> 1
  4 to 2 (weight  -1): 4 -> 3 -> 2
  4 to 3 (weight  -5): 4 -> 3
  4 to 5 (weight  -2): 4 -> 1 -> 5
  5 to 1 (weight   8): 5 -> 4 -> 1
  5 to 2 (weight   5): 5 -> 4 -> 3 -> 2
  5 to 3 (weight   1): 5 -> 4 -> 3
  5 to 4 (weight   6): 5 -> 4
```

All twelve matrices agree with Figure 15.17, and the routes are read out of $\Pi^{(5)}$ by PRINT-ALL-PAIRS-SHORTEST-PATH.

Book: Sections 15.6.8 and 15.6.11.

Round $k=1$, Cell by Cell

Candidate $d_{i1}^{(0)}+d_{1j}^{(0)}$ against incumbent $d_{ij}^{(0)}$. **Row 1:** the candidate is $0+d_{1j}$, a tie; **row k and column k never change in round k.**

Rows 2, 3, 5: $d_{21}=d_{31}=d_{51}=\infty$, so every candidate is ∞: **if $d_{ik}=\infty$, row i cannot change in round k.**

Row 4 ($d_{41}=2$): $j=2$: $\min(\infty, 2+3)=5$, and $\pi_{42}^{(1)}=\pi_{12}^{(0)}=1$. $j=3$: $\min(-5, 10)=-5$. $j=5$: $\min(\infty, 2+(-4))=-2$, and $\pi_{45}^{(1)}=1$.

Two cells changed in the whole round, both in row 4: exactly the printed $D^{(1)}$.

Book: Section 15.6.9.

Round $k=4$, Where the Work Happens

From $D^{(3)}$ with rows $(0,3,8,4,-4)$, $(\infty,0,\infty,1,7)$, $(\infty,4,0,5,11)$, $(2,-1,-5,0,-2)$, $(\infty,\infty,\infty,6,0)$.

Row 1 ($d_{14}=4$): $d_{13}=\min(8,4-5)=-1$, $\pi_{13}=4$; d_{12} ties at 3. **Row 2** ($d_{24}=1$): $d_{21}=3$, $d_{23}=-4$, $d_{25}=-1$ with $\pi_{25}=\pi_{45}=1$.

Row 3 ($d_{34}=5$): $d_{31}=7$, $d_{35}=3$; d_{33} ties at 0, so the cycle $3\to2\to4\to3$ weighs exactly 0. **Row 4** is row k: unchanged. **Row 5** ($d_{54}=6$): $d_{51}=8$, $d_{52}=5$ with $\pi_{52}=\pi_{42}=3$, $d_{53}=1$.

Nine cells changed; afterward every entry is finite. Vertex 4 is the hub that joins the pieces.

Book: Section 15.6.10.

Reading a Route, and Working in Place

Route 1 to 2. $\pi_{12}=3$, $\pi_{13}=4$, $\pi_{14}=5$, $\pi_{15}=1$: unwinding prints `1 5 4 3 2`, and $-4+6-5+4=1=d_{12}^{(5)}$.

> **THEOREM 15.1 (in place is safe).** Running the triple loop on one matrix, overwriting d_{ij}, produces the same $D^{(n)}$ as keeping $n+1$ matrices.

Proof. Only row k and column k are read in round k and could have changed. The candidate for d_{kj} is $d_{kk}+d_{kj}\geq d_{kj}$, since $d_{kk}\geq 0$ when no cycle is negative; the same for column k. So they never change, and every value read still carries its round-$(k-1)$ value. ∎

Memory drops from $\Theta(n^3)$ to $\Theta(n^2)$: at $n=2000$, 32 megabytes instead of 64 gigabytes.

Book: Sections 15.6.11 and 15.6.13.

Negative Cycles, Detected for Free

THE TEST. After the triple loop, the graph contains a negative-weight cycle **if and only if** $d_{ii}<0$ for some i.

d_{ii} is the cheapest closed walk from i. Without negative cycles it is 0; a negative closed walk contains a negative simple cycle; and once k reaches the highest vertex of such a cycle, its diagonal entry goes negative. The program of Section 15.6.14:

```
(a) the running example (edges of weight -4 and -5, but no negative cycle):
   diagonal after the triple loop:    0   0   0   0   0
   negative-weight cycle? no

(b) a graph whose cycle 1 -> 2 -> 3 -> 1 has weight 2 + 3 + (-6) = -1:
   diagonal after the triple loop:   -1  -1  -2
   negative-weight cycle? YES

(c) a graph whose cycle 1 -> 2 -> 1 has weight 1 + (-4) = -3:
   diagonal after the triple loop:   -3  -6
   negative-weight cycle? YES
```

(a) is the running example: all zeros, the machine-checked proof that its results are meaningful. In (b) and (c) only the **sign test** is trustworthy; the off-diagonal numbers are artifacts.

A cross-check of $|V|\times$ Bellman–Ford, SLOW-APSP, repeated squaring, Floyd–Warshall and Johnson on the running example gives one answer. Book: Sections 15.6.14 and 15.6.15.

Transitive Closure

DEFINITION. The **transitive closure** of G is $G^*=(V, E^*)$ with $E^*=\{(i,j)$: there is a path from i to j in $G\}$.

The lazy answer: weight every edge 1 and run Floyd–Warshall; $d_{ij}<n$ exactly when a path exists. **The better answer:** replace min by ∨ and $+$ by ∧.

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E \\ 1 & \text{if } i = j \text{ or } (i,j) \in E \end{cases}$$

and for $k \geq 1$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right).$$

Figure 15.18 The base case $t_{ij}^{(0)}$ and the recurrence for $t_{ij}^{(k)}$, which is the Floyd–Warshall recurrence with ∨ for min and ∧ for $+$.

Written out, Figure 15.18 is

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E, \\ 1 & \text{if } i = j \text{ or } (i,j) \in E, \end{cases} \tag{15.11}$$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right). \tag{15.12}$$

Read Equation (15.12) aloud: reachable already, or reachable to k and from k. In Equation (15.11), $t_{ii}^{(0)}=1$ is the boolean $w_{ii}=0$. Book: Sections 15.7.1 and 15.7.2.

TRANSITIVE-CLOSURE

TRANSITIVE-CLOSURE(G)

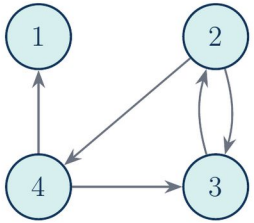
```
1  n = |G.V|
2  let T^(0) = (t_ij^(0)) be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          if i == j or (i, j) ∈ G.E
6              t_ij^(0) = 1
7          else t_ij^(0) = 0
8  for k = 1 to n
9      let T^(k) = (t_ij^(k)) be a new n × n matrix
10     for i = 1 to n
11         for j = 1 to n
12             t_ij^(k) = t_ij^(k−1) ∨ (t_ik^(k−1) ∧ t_kj^(k−1))
13 return T^(n)
```

Figure 15.19 TRANSITIVE-CLOSURE(G). Lines 3–7 build $T^{(0)}$ from the edge set; lines 8–12 are the Floyd–Warshall triple loop with boolean operators. It runs in $\Theta(n^3)$ time.

Figure 15.19 is the boolean semiring $(\vee, \wedge)$ with $\acute{0}$ = FALSE and $\acute{1}$ = TRUE: *is either route available?* and *can I do this and then that?*

Book: Section 15.7.3.

The Closure, Worked



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Figure 15.20 A directed graph with edges $2 \to 3$, $2 \to 4$, $3 \to 2$, $4 \to 1$, $4 \to 3$, and the matrices $T^{(k)}$ computed by the transitive-closure algorithm.

In Figure 15.20: round $k = 1$ changes nothing (vertex 1 has no outgoing edges); round 2 sets t_{34} via $3 \to 2 \to 4$; round 3 sets t_{42} via $4 \to 3 \to 2$; round 4 fills rows 2 and 3, since both reach 4 and 4 reaches 1.

Row 1 stays $(1, 0, 0, 0)$ for ever: vertex 1 is a sink.

Book: Section 15.7.3.

Transitive Closure in C++

Listing 15.3 The transitive-closure algorithm in full. Compile with `g++ -std=c++17 -O2 -o transitive_closure transitive_closure.cpp`.

```cpp
// Chapter 15, Listing: TRANSITIVE-CLOSURE on the four-vertex example graph.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

typedef vector<vector<int>> BoolMatrix;   // 0/1 entries, so that | and & work directly

void printT(const string& name, const BoolMatrix& T) {
    int n = (int)T.size();
    cout << name << " =\n";
    for (int i = 0; i < n; ++i) {
        cout << "   ";
        for (int j = 0; j < n; ++j) cout << setw(3) << T[i][j];
        cout << '\n';
    }
    cout << '\n';
}

int main() {
    const int n = 4;
    // The example graph: edges 2->3, 2->4, 3->2, 4->1, 4->3 (vertices numbered 1..4).
    vector<pair<int,int>> edges = {{2,3},{2,4},{3,2},{4,1},{4,3}};

    // Lines 2-7: build T^(0).
    BoolMatrix T(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i) T[i][i] = 1;              // i == j
```

```
    for (auto& e : edges) T[e.first - 1][e.second - 1] = 1; // (i,j) in G.E
    printT("T^(0)", T);

    // Lines 8-12: the same triple loop as Floyd-Warshall, with OR for min and AND for +.
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                T[i][j] = T[i][j] | (T[i][k] & T[k][j]);
        printT("T^(" + to_string(k + 1) + ")", T);
    }

    cout << "Edges of the transitive closure G* (excluding self-loops):\n   ";
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (i != j && T[i][j]) cout << "(" << i + 1 << "," << j + 1 << ") ";
    cout << '\n';
    return 0;
}
```

```
T^(0) =
     1  0  0  0
     0  1  1  1
     0  1  1  0
     1  0  1  1

T^(1) =
     1  0  0  0
     0  1  1  1
     0  1  1  0
     1  0  1  1

T^(2) =
     1  0  0  0
     0  1  1  1
     0  1  1  1
     1  0  1  1

T^(3) =
     1  0  0  0
     0  1  1  1
     0  1  1  1
     1  1  1  1

T^(4) =
     1  0  0  0
     1  1  1  1
     1  1  1  1
```

```
     1  1  1  1

Edges of the transitive closure G* (excluding self-loops):
   (2,1) (2,3) (2,4) (3,1) (3,2) (3,4) (4,1) (4,2) (4,3)
```

Listing 15.3 reproduces all five matrices of Figure 15.20.

Book: Section 15.7.3.

Why Booleans Are Faster in Practice

Space. A distance needs a 32- or 64-bit word; a boolean needs one bit, a factor of 32 or 64, which decides whether the matrix fits in cache.

Time. Packed into `std::bitset`, the inner j-loop becomes an OR of two whole rows: 64 iterations in one instruction. Measured by the program of Section 15.7.4:

```
n = 1024, edges = 4096
byte-per-entry version: 231.7 ms
bitset version:         3.2 ms
the two closures agree: yes
reachable pairs found:  1015046 out of 1048576
```

A factor of more than **seventy** for a two-line change. Asymptotics choose the algorithm; they say nothing about the constant, and both matter.

Timings are from one run on one machine; the ratio is what carries over. Book: Section 15.7.4.

Johnson: the Wish, and the Plan

The wish: $O(V^2 \lg V + VE)$ on sparse graphs, faster than Floyd–Warshall or repeated squaring, with negative edges allowed. That bound is V runs of Dijkstra with a Fibonacci heap: *how do we make $V \times$ Dijkstra legal with negative edges?*

- If every edge weight is non-negative, running Dijkstra once from each vertex solves all pairs; with a Fibonacci heap this takes $O(V^2 \lg V + VE)$ time.
- If G has negative-weight edges but no negative-weight cycle, compute new non-negative weights $\hat{w}$ that let the same method be used.
- The new weights must satisfy two properties:
  1. For all $u, v \in V$, a path p is a shortest path from u to v under w **if and only if** it is a shortest path from u to v under $\hat{w}$.
  2. For every edge (u, v), the new weight $\hat{w}(u, v)$ is non-negative.

Figure 15.21 If a new weight function $\hat{w}$ preserves shortest paths and is non-negative everywhere, then $V \times$ Dijkstra on $\hat{w}$ answers the original question.

Figure 15.21: **(1)** p is shortest under w iff shortest under $\hat{w}$; **(2)** $\hat{w}(u,v) \geq 0$ for every edge.

Adding a constant fails. With $w(1,2) = -2$, $w(2,3) = -2$, $w(1,3) = -3$ the best route is $1 \to 2 \to 3$ at -4. Adding $c = 3$ gives 1, 1, 0: now the direct edge wins. A path gains c per edge; the fix must depend on the **endpoints**.

Book: Sections 15.8.1 and 15.8.2.

The Reweighting That Works

Choose any $h : V \to R$, an *altitude* for every vertex, and set

$$\hat{w}(u,v) = w(u,v) + h(u) - h(v). \tag{15.13}$$

LEMMA 15.2. Under Equation (15.13), for any path $p = \langle v_0, \ldots, v_k \rangle$, $\hat{w}(p) = w(p) + h(v_0) - h(v_k)$. So shortest paths are preserved, and negative cycles neither appear nor disappear.

Proof. Summing Equation (15.13) along p, the terms $h(v_{i-1}) - h(v_i)$ **telescope** to $h(v_0) - h(v_k)$. All paths between the same endpoints gain the same constant, so their order is unchanged; a cycle has $v_0 = v_k$ and keeps its weight. ▰

This proves property 1 for every h; the art is choosing an h that also gives property 2. Book: Section 15.8.3.

Choosing h: the Auxiliary Vertex

Property 2 is $h(v) \le h(u) + w(u,v)$ for every edge: the **triangle inequality**, Corollary 13.3. So take $h(v) = \delta(s,v)$ for a source s that reaches everything.

Invent that source. G' adds a new vertex s and an edge (s,v) of weight 0 to every v. s has **no incoming edges**, so it lies on no shortest path between original vertices and changes no distance among them.

One Bellman–Ford run from s computes $h(v) = \delta(s,v)$, and is at the same time the test for a negative cycle.

> **Why every weight becomes non-negative:** $\delta(s,v) \le \delta(s,u) + w(u,v)$ is $h(v) \le h(u) + w(u,v)$, which is $0 \le \hat{w}(u,v)$. One line, once you see the requirement *is* the triangle inequality.

Book: Section 15.8.4.

JOHNSON(G, w)

JOHNSON(G, w)

1 compute G', where $G'.V = G.V \cup \{s\}$,
  $G'.E = G.E \cup \{(s, v) : v \in G.V\}$, and $w(s, v) = 0$ for all $v \in G.V$
2 **if** BELLMAN-FORD$(G', w, s) ==$ FALSE
3 print "the input graph contains a negative-weight cycle"
4 **else for each** vertex $v \in G'.V$
5 set $h(v)$ to the value of $\delta(s, v)$ computed by BELLMAN-FORD
6 **for each** edge $(u, v) \in G'.E$
7 $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$
8 **let** $D = (d_{uv})$ be a new $n \times n$ matrix
9 **for each** vertex $u \in G.V$
10 run DIJKSTRA$(G, \widehat{w}, u)$ to compute $\widehat{\delta}(u, v)$ for all $v \in G.V$
11 **for each** vertex $v \in G.V$
12 $d_{uv} = \widehat{\delta}(u, v) + h(v) - h(u)$
13 **return** D

Figure 15.22 JOHNSON(G, w). Line 1 adds the auxiliary vertex, line 2 runs Bellman–Ford once, lines 6–7 reweight, lines 9–10 run Dijkstra $|V|$ times, and line 12 converts each answer back.

Line 12 of Figure 15.22 **undoes the reweighting**, by Lemma 15.2 on a shortest path:

$$\hat{\delta}(u,v)=\delta(u,v)+h(u)-h(v), \text{so } d_{uv}=\hat{\delta}(u,v)+h(v)-h(u). \tag{15.14}$$

Cost. Bellman–Ford on G': $O(VE)$. Reweighting: $O(E)$. V runs of Dijkstra with a Fibonacci heap: $O(V^2 \lg V + VE)$, which dominates. With a binary heap: $O(VE \lg V)$, usually better in practice.

In Equation (15.14) note the reversed signs: to recover the original, subtract what was added. Book: Section 15.8.5.

A Small Reweighting: the Graph

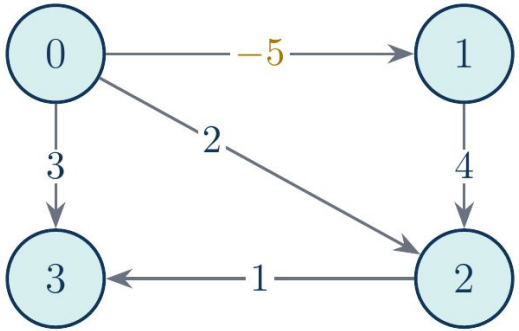


Figure 15.23 A four-vertex graph with one negative edge, $0 \to 1$ of weight -5. Dijkstra is not allowed here.

Figure 15.23 has vertices $0, 1, 2, 3$ and edges $0 \to 1$ (-5), $0 \to 2$ (2), $0 \to 3$ (3), $1 \to 2$ (4), $2 \to 3$ (1).

Book: Section 15.8.6.

Add s, Run Bellman–Ford

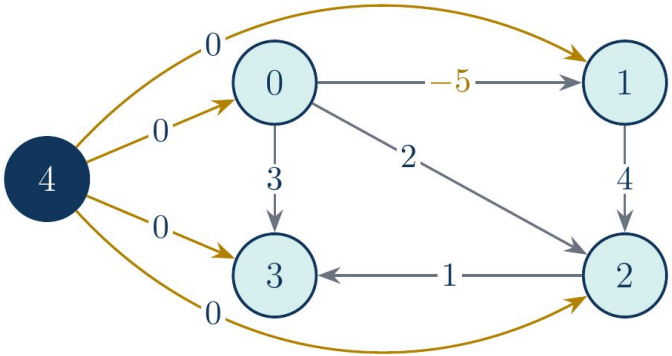


Figure 15.24 The same graph with the auxiliary vertex 4 added, joined to every original vertex by an edge of weight 0. Vertex 4 has no incoming edges, so it cannot appear inside any shortest path between original vertices.

From vertex 4 in Figure 15.24: $h[0]=0$; $h[1]=0+(-5)=-5$ via $4 \to 0 \to 1$; $h[2]=-5+4=-1$ via $4 \to 0 \to 1 \to 2$; $h[3]=0$. So $h=[0,-5,-1,0]$.

Book: Section 15.8.6.

Reweight: Every Weight Non-Negative

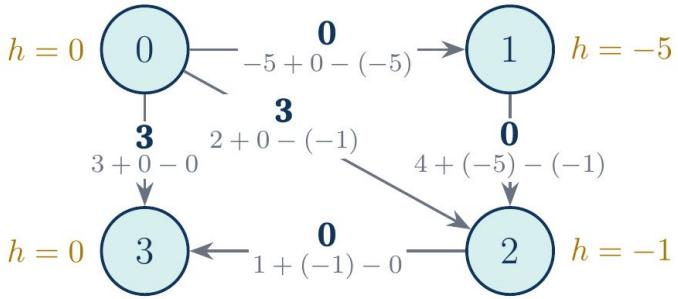


Figure 15.25 The four-vertex graph after reweighting. Every weight is now non-negative — three of the five are exactly zero — so Dijkstra's algorithm can be used from every vertex as the source.

By Equation (15.13): $(0,1)$: $-5+0+5=0$; $(0,2)$: $2+0+1=3$; $(0,3)$: 3; $(1,2)$: $4-5+1=0$; $(2,3)$: $1-1-0=0$, as Figure 15.25 shows.

Every edge on a shortest path from s becomes exactly 0. And weights do not all shrink: $(0,2)$ went from 2 up to 3. Reweighting is a fair redistribution, not a discount.

The zero-weight edges need not form a tree (Exercise 15.17). Book: Section 15.8.6.

Johnson in C++

Listing 15.4 Johnson's algorithm in full: Bellman–Ford from an auxiliary vertex, reweighting, $|V|$ runs of Dijkstra, and the conversion back. Compile with `g++ -std=c++17 -O2 -o johnson johnson.cpp`.

```cpp
// Chapter 15, Listing: Johnson's algorithm for sparse graphs, run on the running
// example graph. Bellman-Ford supplies h, then Dijkstra is run once per vertex.
#include <iostream>
#include <iomanip>
#include <vector>
#include <queue>
#include <string>
using namespace std;

const long long INF = 1000000000LL;

struct Edge { int u, v; long long w; };

// BELLMAN-FORD on G', returning false when a negative-weight cycle is reachable.
bool bellmanFord(int n, const vector<Edge>& E, int s, vector<long long>& d) {
    d.assign(n, INF);
    d[s] = 0;
    for (int pass = 1; pass <= n - 1; ++pass)          // |V'| - 1 passes
        for (const Edge& e : E)
            if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) d[e.v] = d[e.u] + e.w;
    for (const Edge& e : E)                            // one more pass: any improvement?
        if (d[e.u] < INF && d[e.u] + e.w < d[e.v]) return false;
    return true;
}

// DIJKSTRA with a binary min-heap, on nonnegative weights only.
vector<long long> dijkstra(int n, const vector<vector<pair<int,long long>>>& adj, int s) {
    vector<long long> d(n, INF);
```

```
    d[s] = 0;
    priority_queue<pair<long long,int>, vector<pair<long long,int>>, greater<>> Q;
    Q.push({0, s});
    while (!Q.empty()) {
        auto [du, u] = Q.top(); Q.pop();
        if (du > d[u]) continue;                        // a stale heap entry
        for (auto [v, w] : adj[u])
            if (d[u] + w < d[v]) { d[v] = d[u] + w; Q.push({d[v], v}); }
    }
    return d;
}
```

Listing 15.4 (Continued) *Johnson's algorithm in full: Bellman–Ford from an auxiliary vertex, reweighting, |V| runs of Dijkstra, and the conversion back. Compile with g++ -std=c++17 -O2 -o johnson johnson.cpp.*

```
int main() {
    const int n = 5;                                    // vertices 1..5, stored 0..4
    vector<Edge> E = {
        {0,1,3}, {0,2,8}, {0,4,-4},
        {1,3,1}, {1,4,7},
        {2,1,4},
        {3,0,2}, {3,2,-5},
        {4,3,6}
    };

    // Line 1: G' = G plus a new source s with a 0-weight edge to every vertex.
    int s = n;                                          // the auxiliary vertex, index 5
    vector<Edge> Eprime = E;
    for (int v = 0; v < n; ++v) Eprime.push_back({s, v, 0});

    // Lines 2-5: Bellman-Ford from s gives h(v) = delta(s, v).
    vector<long long> h;
    if (!bellmanFord(n + 1, Eprime, s, h)) {
        cout << "the input graph contains a negative-weight cycle\n";
        return 0;
    }
    cout << "h(v) = delta(s,v) for v = 1..5:  ";
    for (int v = 0; v < n; ++v) cout << h[v] << ' ';
    cout << "\n\n";

    // Lines 6-7: reweight. w-hat(u,v) = w(u,v) + h(u) - h(v).
    vector<vector<pair<int,long long>>> adj(n);
    cout << "reweighted edges:\n";
```

```
    for (const Edge& e : E) {
        long long wh = e.w + h[e.u] - h[e.v];
        adj[e.u].push_back({e.v, wh});
        cout << "   w-hat(" << e.u + 1 << "," << e.v + 1 << ") = "
             << setw(3) << e.w << " + (" << h[e.u] << ") - (" << h[e.v] << ") = " << wh << '\n';
    }
    cout << '\n';

    // Lines 8-12: one Dijkstra per vertex, then undo the reweighting.
    vector<vector<long long>> D(n, vector<long long>(n, INF));
    for (int u = 0; u < n; ++u) {
        vector<long long> dhat = dijkstra(n, adj, u);
        for (int v = 0; v < n; ++v)
            if (dhat[v] < INF) D[u][v] = dhat[v] + h[v] - h[u];
    }
```

Listing 15.4 (Continued) *Johnson's algorithm in full: Bellman–Ford from an auxiliary vertex, reweighting, |V| runs of Dijkstra, and the conversion back. Compile with g++ -std=c++17 -O2 -o johnson johnson.cpp.*

```
    cout << "D = matrix of shortest-path weights:\n";
    for (int i = 0; i < n; ++i) {
        cout << "   ";
        for (int j = 0; j < n; ++j) {
            if (D[i][j] >= INF) cout << setw(6) << "inf";
            else                cout << setw(6) << D[i][j];
        }
        cout << '\n';
    }
    return 0;
}
```

```
h(v) = delta(s,v) for v = 1..5:  0 -1 -5 0 -4

reweighted edges:
   w-hat(1,2) =   3 + (0) - (-1) = 4
   w-hat(1,3) =   8 + (0) - (-5) = 13
   w-hat(1,5) =  -4 + (0) - (-4) = 0
   w-hat(2,4) =   1 + (-1) - (0) = 0
   w-hat(2,5) =   7 + (-1) - (-4) = 10
   w-hat(3,2) =   4 + (-5) - (-1) = 0
   w-hat(4,1) =   2 + (0) - (0) = 2
   w-hat(4,3) =  -5 + (0) - (-5) = 0
   w-hat(5,4) =   6 + (-4) - (0) = 2

D = matrix of shortest-path weights:
        0     1    -3     2    -4
        3     0    -4     1    -1
        7     4     0     5     3
        2    -1    -5     0    -2
        8     5     1     6     0
```

Listing 15.4 on the running example: $h = (0, -1, -5, 0, -4)$, nine non-negative weights with four zeros, and the same D as Floyd–Warshall and the matrix method.

Book: Section 15.8.7.

Johnson in One Picture

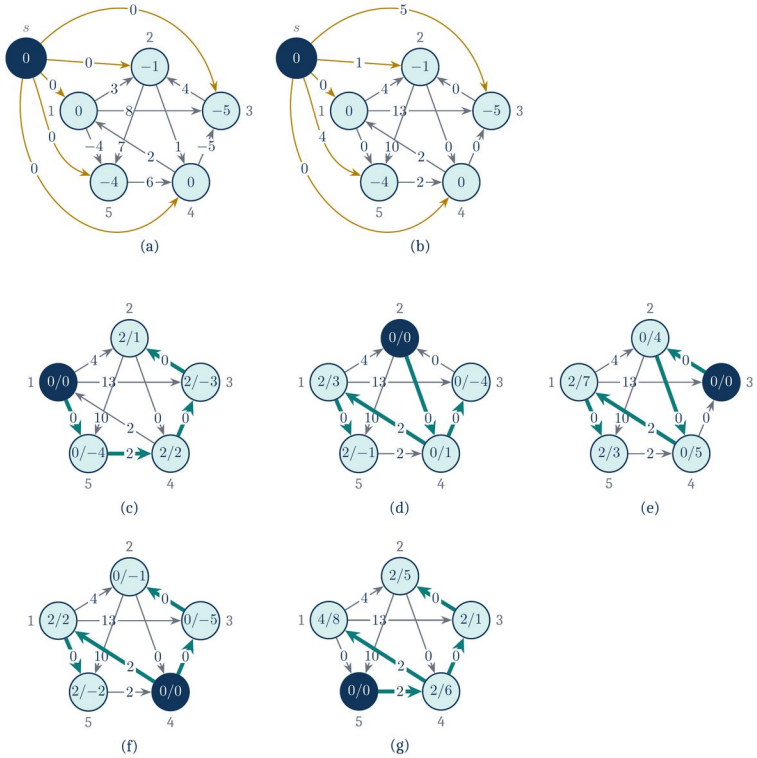


Figure 15.26 Johnson's algorithm on the graph of Figure 15.2. (a) G' with the original weights and $h(v) = \delta(s, v)$ at each vertex. (b) After reweighting, every weight is non-negative. (c)–(g) Dijkstra from each vertex in turn, each vertex showing $\hat{\delta}(u, v)$ and $\delta(u, v)$ separated by a slash.

Check part (c) of Figure 15.26 against Equation (15.14): source $u = 1$, $h(1) = 0$; vertex 3 shows $2/-3$, and $\delta(1, 3) = 2 + (-5) - 0 = -3$, row 1 column 3 of D.

The reweighted answer 2 *answers no question anybody asked; Equation (15.14) turns it back into one. Book: Section 15.8.7.*

Which Algorithm, and When

Table 15.4 Every algorithm in this chapter, with the cost and the case it is for. The two in bold are the two you will actually use.

Algorithm	Handles negative edges?	Running time	Space	Notes
$V \times$ Dijkstra (array)	no	$O(V^3 + VE)$	$\Theta(V^2)$	simplest correct option on dense non-negative graphs
$V \times$ Dijkstra (binary heap)	no	$O(VE \lg V)$	$\Theta(V+E)$	good on sparse non-negative graphs
$V \times$ Bellman–Ford	yes	$O(V^2E)$, dense $O(V^4)$	$\Theta(V+E)$	only use when nothing else is available
SLOW-APSP (repeated $\odot W$)	yes	$\Theta(V^4)$	$\Theta(V^2)$	pedagogical; never the right production choice
Repeated squaring	yes	$\Theta(V^3 \lg V)$	$\Theta(V^2)$	the semiring idea, made efficient
Floyd–Warshall	yes	$\Theta(V^3)$	$\Theta(V^2)$	the default; tiny constant, trivial to code
Transitive closure (boolean)	n/a	$\Theta(V^3)$, $\Theta(V^3/64)$ packed	$\Theta(V^2)$ bits	reachability only
Johnson	yes	$O(V^2 \lg V + VE)$	$\Theta(V+E)$	the default for sparse graphs

From Table 15.4, Johnson beats Floyd–Warshall asymptotically when $E = o(V^2)$. **Measured** at $V = 600$: Johnson wins up to $|E|/|V| \approx 40$, Floyd–Warshall from about 80, a crossover near **13 percent** density.

Floyd–Warshall's time is almost flat in E, as $\Theta(V^3)$ predicts; Johnson's grows with E. Book: Sections 15.9.1 and 15.9.2.

A Decision Procedure

Table 15.5 A decision procedure for the all-pairs shortest-paths problem. Work down the rows; the first "yes" is your answer.

Question	If yes	If no
Do you only need reachability, not distance?	boolean transitive closure, packed into bits (§15.7)	continue
Is n small (say $n \le 400$), or is the code going to be read by people?	Floyd–Warshall — five lines, no data structures (§15.6)	continue
Is the graph dense, $E \gtrsim 0.13\,V^2$ by the measurement of §15.9.2?	Floyd–Warshall	continue
Are all weights non-negative?	V × Dijkstra with a binary heap	Johnson: one Bellman–Ford, reweight, then V × Dijkstra (§15.8)
Do you need the routes and not just the costs?	keep Π alongside D (§15.6.7); it costs $\Theta(n^2)$ space and nothing in time	—
Might there be a negative-weight cycle?	check the diagonal after Floyd–Warshall (§15.6.14), or let Bellman–Ford in Johnson line 2 report it	—

Work down Table 15.5; the first "yes" is the answer.

Book: Section 15.9.3.

Profitina in Practice

Inside the GEO platform, a crawled site is a directed graph; the **diameter** of the internal link graph is an all-pairs question, and the **orphan report** (pages no row of the closure reaches) is transitive closure. Large sites are handled section by section.

Profitina Laundry. Eight districts of Surabaya with some one-way roads, so the matrix is not symmetric. Two depot questions: the smallest **worst** run (the **center**, least eccentricity), and the smallest **total** out-and-back driving.

Two answers. Klampis: no customer more than **17** minutes away. Manyar: total driving **137** minutes, one less than Klampis, but a worst run of 20. The algorithm replaces an argument with a table; the owner chooses.

Keputih reaches Manyar in 9 minutes, Manyar needs 12 to return; Π reveals the worst run from Klampis as the two-leg route Klampis → Keputih → Rungkut.

The platform advises; the user decides. Book: Section 15.10.

Common Mistakes

1. Loops in the wrong order: k goes first. **2. Flattening superscripts**: $l_{ij}^{(m)}$ budgets edges, $d_{ij}^{(k)}$ permits intermediate vertices; they are different algorithms.

3. Computing `INF + INF`: guard the addition, or pick a sentinel small enough that twice it fits. **4.** $\pi_{ij}=k$ **instead of** π_{kj}: test by summing weights along printed routes.

5. Trusting Dijkstra because the test graph worked. 6. Forgetting to undo the reweighting with Equation (15.14). **7. Adding a constant to every edge.**

8. Assuming D **is symmetric. 9. All-pairs when one source was needed. 10. Reading off-diagonal entries** before checking the diagonal for a negative cycle.

Book: Section 15.11.

Summary — Five Things

1. The problem. $\delta(i, j)$ for every ordered pair, from W with 0 on the diagonal and ∞ for a missing edge; output D and Π.

2. Naive. n single-source runs: $O(V^3)$ with Dijkstra if $w \geq 0$, $O(V^2 E)$, dense $O(V^4)$, with Bellman–Ford.

3. Matrix multiplication over $(\min, +)$**.** $L^{(m)} = W^m$ by edge budget: $\Theta(n^4)$ slowly, $\Theta(n^3 \lg n)$ by repeated squaring.

4. Floyd–Warshall. Permitted intermediates $\{1, \ldots, k\}$: $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$, $\Theta(n^3)$, k outermost, in place, diagonal test. Boolean version: transitive closure.

5. Johnson. $h = \delta(s, \cdot)$ from an auxiliary source, $\hat{w} = w + h(u) - h(v)$, $V \times$ Dijkstra, convert back: $O(V^2 \lg V + V E)$. Floyd–Warshall if dense or small, Johnson if sparse.

Book: Section 15.12.

Next Time

Chapter 16 — Evaluation and Discussion IV. Worked problems on Chapters 13 to 15 and on the whole book.

Every algorithm of today returns there. Rehearse two kinds of problem: *trace Floyd–Warshall on this graph, showing every* $D^{(k)}$, and *why must* k *be the outermost loop?*

Also Dijkstra and Bellman–Ford tables, the binary counter and the dynamic table three ways, and Johnson's reweighting carried out in full.

Homework. Exercises 15.1–15.18; start with 15.8 (round $k=2$ by hand) and 15.17 (the zero-weight edges after reweighting).

Cormen et al., Introduction to Algorithms, 4th ed., Chapter 23, covers the same ground. Book: Sections 15.12.1 and 15.13.

Lecture 16 — Evaluation and Discussion IV: Shortest Paths, Amortized Analysis, and the Whole Book

Design and Analysis of Algorithms (EF234405)

Chapter 16 of the book, in miniature, for the classroom.

Book: Irfan Subakti & Rully Soelaiman, *Design and Analysis of Algorithms*.

Lecturer: Irfan Subakti — yifana@gmail.com

No new theory today: twenty-one worked problems that consolidate Chapters 13 to 15 and integrate the whole book, each solved from nothing and checked by a program. Book: Chapter 16, opening paragraph.

Agenda

1. **Where we stand**, the whole book on one page, and a method for multi-part problems and stalled proofs.
2. **Five integrating problems:** fifteen true-or-false statements (16.1), the longest increasing subsequence (16.2), extended binary trees by induction (16.3), a 2-approximation for metric TSP (16.4), Bellman–Ford against Dijkstra (16.5).
3. **Seven problems on Chapters 13 to 15:** Dijkstra, Bellman–Ford with a negative cycle, DAG shortest paths (16.6–16.8); the binary counter and the dynamic table three ways (16.9–16.10); Floyd–Warshall and Johnson by hand (16.11–16.12).
4. **Nine further problems** (16.13–16.21) revisiting the rest of the book.
5. **Consolidation:** the problems compared, self-check drills, the algorithm selection table, Profitina, common mistakes, a summary and closing word, and where to go next.

Every problem has the same shape: problem, what it tests, solution, program, and a check yourself variant with its answer. Book: Section 16.1.3.

Where We Stand

Chapter 13: single-source shortest paths, the estimate $d[v]$ and predecessor $\pi[v]$, and the one primitive, **relaxation**:

$$\text{if } d[v] > d[u] + w(u,v) \text{ then } d[v] \leftarrow d[u] + w(u,v), \pi[v] \leftarrow u. \tag{16.1}$$

Dijkstra, Bellman–Ford and the DAG algorithm differ only in the *order* in which they apply Equation (16.1).

Chapter 14: amortized analysis of a *sequence* of operations, by the aggregate, accounting and potential methods, on the stack with `MULTIPOP`, the k-bit counter and the doubling table.

Chapter 15: all pairs, by repeated matrix "multiplication", Floyd–Warshall's recurrence over allowed interior vertices, and Johnson's reweighting so that Dijkstra becomes legal.

All three chapters follow Cormen et al., Introduction to Algorithms, 4th ed. Book: Section 16.1.1.

The Whole Book on One Page

Table 16.1 One idea per chapter, and where this chapter uses it. If you can state all sixteen from memory, the whole book is in your hands.

Ch	The one idea	Used here
1	A problem is solved in five phases: understand, abstract, design, analyze, implement. Coding is the last one, not the first.	Section 16.2
2	A loop invariant is how you know a loop is right: initialization, maintenance, termination.	Section 16.4.4
3	Compare growth, not stopwatches. $T(n)=aT(n/b)+f(n)$ is settled by comparing $f(n)$ with $n^{\log_b a}$.	Problem 16.1, statement (11)
4	A constraint is a time budget. $n \le 10^5$ means $O(n \lg n)$ or better; $n \le 500$ means $O(n^3)$ is fine.	Section 16.4.7
5	Heapsort trades the heap's cheap selection for $\Theta(n \lg n)$ always; quicksort trades a cheap partition for $\Theta(n \lg n)$ usually. $\Omega(n \lg n)$ is the comparison floor.	Problem 16.1, statements (1) and (5)
6	A balance invariant converts $O(n)$ into $O(\lg n)$, and rotation is the only tool needed to maintain it.	Problem 16.3
7	When subproblems overlap and the optimum is built from optimal pieces, remember instead of recomputing.	Problems 16.2 and 16.19
8	The answer is the derivation. Code without an invariant is a guess that happened to pass a test.	Section 16.2
9	A locally best choice is globally safe only when you have proved the greedy-choice property.	Problems 16.4 and 16.16
10	One traversal skeleton, two containers: a queue gives fewest edges, a stack gives structure.	Problems 16.1 (statement 13) and 16.8

Table 16.1 (Continued) *One idea per chapter, and where this chapter uses it. If you can state all sixteen from memory, the whole book is in your hands.*

Ch	The one idea	Used here
11	The cut property: the lightest edge crossing any cut that the current forest respects is safe.	Problems 16.4 and 16.21
12	An algorithm becomes an engineering artifact only when it is implemented, measured and defended.	Section 16.16
13	Relaxation plus an order. Dijkstra's order is by distance, Bellman–Ford's is "all edges, $\lvert V \rvert - 1$ times", the DAG's is topological.	Problems 16.5 to 16.8
14	The cost of a sequence is not the worst single step multiplied by the number of steps.	Problems 16.9 and 16.10
15	The same recurrence twice: over path length, and over which vertices may be interior. Reweighting makes Dijkstra legal on negative edges.	Problems 16.11, 16.12 and 16.17
16	Choosing the algorithm is the skill. Everything else is bookkeeping.	Section 16.14

Table 16.1 names, for each chapter, the one idea that would let you rebuild the rest, and where today's problems use it.

If you can state all sixteen ideas from memory, the whole book is in your hands. Book: Section 16.1.2.

How to Approach a Multi-Part Problem

A fixed order within any part: (1) the answer in one sentence; (2) the assumptions, with every word of the problem defined; (3) the derivation or proof; (4) a worked example, never omitted.

When a proof will not come out: state the claim formally; name the technique and the quantity; do the base case; state the hypothesis precisely; write the steps you can justify and **label the gap**; check on an example.

> **A stalled proof is repaired by structure, not by length.** Six labeled lines (claim, technique, base, hypothesis, what follows, what is missing) are worth more than three pages of unlabeled algebra.

Five habits: never write “obviously”; never state an O bound without its case; pseudocode before code; illustrate the algorithm, not the graph; check every number twice.

Book: Section 16.2.

Problem 16.1 — Fifteen Statements

Mark True or False and justify: (1) quicksort has worse asymptotic complexity than merge sort; (2) binary search is $O(\log n)$; (3) linear search is $O(n)$ unsorted but $O(\log n)$ sorted; (4) linear search in a sorted linked list is $O(n)$; (5) for one search, linear search beats sort-then-binary-search.

(6) Distinct arc weights make the MST unique; (7) binary search requires a sorted array; (8) insertion into an ordered list can be done in $O(\log n)$; (9) a good hash function spreads keys uniformly; (10) a hash-table search can be $O(1)$.

(11) $O(g(n))$ means at least $g(n)$ operations; (12) always choose the better asymptotic complexity; (13) DFS and BFS can never visit > 2 nodes in the same order; (14) a binary tree of height H has at most $2^{H+1} - 1$ nodes; (15) in a complete binary tree only leaves have no children.

What it tests: four pairs of ideas habitually merged: upper bound against exact count; worst against average case; what a structure's shape permits against what an algorithm does; guaranteed against typical.

Book: Sections 16.3.1 and 16.3.2.

Problem 16.1 — The Answers

Table 16.2 The fifteen answers to Problem 16.1, with a one-line justification for each. Nine are true and six are false.

#	Statement, abbreviated	Answer	Why
1	Quicksort has worse asymptotic complexity than merge sort	**T**	Worst case $\Theta(n^2)$ against $\Theta(n \lg n)$
2	Binary search is $O(\log n)$	**T**	$T(n) = T(n/2) + \Theta(1)$
3	Linear search: $O(n)$ unsorted, $O(\log n)$ sorted	**F**	Linear search is $\Theta(n)$ whatever the order
4	Linear search in a sorted linked list is $O(n)$	**T**	No random access, so no halving
5	For one search, linear beats sort-then-binary-search	**T**	$O(n)$ against $\Omega(n \lg n)$
6	Distinct arc weights $\Rightarrow$ the MST is unique	**T**	Exchange argument, Section 16.3.9
7	Binary search requires the array to be sorted	**T**	The discarded half must be known to be wrong
8	Insertion into an ordered list can be done in $O(\log n)$	**F**	Finding the place is $O(\lg n)$; making room is $\Theta(n)$
9	A good hash function spreads keys uniformly	**T**	That is the definition of simple uniform hashing
10	Good hash function + reasonable data $\Rightarrow O(1)$ search	**T**	Expected $\Theta(1+\alpha)$ with $\alpha = O(1)$
11	$O(g(n))$ means at least $g(n)$ operations are executed	**F**	O is an upper bound; Ω is the lower one
12	Always choose the better asymptotic complexity	**F**	Constants, memory, input size, stability

Table 16.2 (Continued) *The fifteen answers to Problem 16.1, with a one-line justification for each. Nine are true and six are false.*

#	Statement, abbreviated	Answer	Why
13	DFS and BFS can never agree on a graph of > 2 nodes	**F**	A path or a star; 26 of 38 four-vertex graphs, Section 16.3.11
14	A binary tree of height H has at most $2^{H+1} - 1$ nodes	**T**	$\sum_{i=0}^{H} 2^i = 2^{H+1} - 1$
15	In a complete binary tree only leaves have no children	**T**	True by the definition of *leaf*

Table 16.2: nine true, six false, and every false one is false for a reason taught in this book.

The defense against misreading: underline the verb and the qualifier in every statement (linear search, at least, always, not possible) before answering. Book: Sections 16.3.3 and 16.3.12.

Statements (1), (3)–(5), (7), (8)

(1) True, reading it as worst case: quicksort $T(n)=T(n-1)+\Theta(n)=\Theta(n^2)$ on a bad split against merge sort's $\Theta(n \lg n)$. In the average case both are $\Theta(n \lg n)$ and it would be false. "Faster in practice" is a different claim.

(3) False, (4) True, (7) True. Sorting changes what an algorithm *can* do, not what linear search *does*: it is $\Theta(n)$ either way. A linked list has no $\Theta(1)$ access to the middle, so no halving. Binary search's discard step is sound only if the data is sorted.

(5) True. Linear search costs $\Theta(n)$; sorting first costs $\Omega(n \lg n)+O(\lg n)$. With k searches the two cross near $k \approx \lg n$: about twenty lookups for a million elements.

(8) False. Finding the place is $O(\lg n)$ in an array; making room is $\Theta(n)$. In a list the halves swap roles. Either way $\Theta(n)$; balanced trees exist because of this.

Book: Sections 16.3.4 to 16.3.7.

Statements (6), (9)–(15)

(6) True, by exchange. If $T_1 \neq T_2$ are both minimum, let e be the lightest edge of their symmetric difference, say $e \in T_1$. Adding e to T_2 closes a cycle containing some $f \in T_2 \setminus T_1$ with $w(e) < w(f)$; then $T_2 \cup \{e\} \setminus \{f\}$ is lighter than T_2. Contradiction. ∎

(9), (10) True: simple uniform hashing gives expected $\Theta(1+\alpha)$. **(12) False:** constants, memory, input size and stability all outrank asymptotics sometimes. **(11) False:** O is an upper bound; "at least" is Ω.

(14) True: $\sum_{i=0}^{H} 2^i = 2^{H+1} - 1$. **(15) True** by the definition of a leaf. **(13) False:** a path $1-2-3$ or a star; the program of Section 16.3.11 checks every graph on four labeled vertices:

```
edges: 0-1 0-2 0-3   order: 0 1 2 3
edges: 0-1 0-2 0-3 1-2   order: 0 1 2 3
edges: 0-2 0-3 1-3   order: 0 2 3 1

connected graphs on 4 labeled vertices reachable from 0: 38
of those, DFS order = BFS order in 26 of them
```

The converse of (6) fails: ties permit a second MST but do not force one. Book: Sections 16.3.8 to 16.3.11.

Problem 16.2 — Longest Increasing Subsequence

Problem. Design a polynomial-time algorithm to find the longest monotonically increasing subsequence of n distinct integers; give assumptions, pseudocode and a time-complexity analysis.

Assumptions, stated properly. Input $S[1..n]$; all values **distinct**; increasing means **strictly**; a **subsequence** keeps order but need not be contiguous, $1 \le i_1 < \cdots < i_k \le n$; the answer has length at least 1.

> **The whole difficulty hides in the word *subsequence*.** Solving the contiguous problem gives a correct $O(n)$ algorithm for a problem nobody asked: on the example below the longest run has length 3, not 6.

Polynomial is a floor, not a target: $O(n^2)$ is the natural answer, and $O(n \lg n)$ is better.

Book: Sections 16.4.1 to 16.4.3.

The Recurrence and the Pseudocode

Strengthen the subproblem until it composes: $L[i]$ is the length of the longest increasing subsequence of $S[1..i]$ that **ends exactly at** $S[i]$. By cut and paste,

$$L[i] = \begin{cases} \max\limits_{j<i \,\wedge\, S[j]<S[i]} (L[j]+1) & \text{if such a } j \text{ exists,} \\ 1 & \text{otherwise.} \end{cases} \tag{16.2}$$

The answer is $\max_i L[i]$; $P[i]$ records the maximizing j.

Pseudocode 16.1 The $O(n^2)$ dynamic program for the longest increasing subsequence, followed by the backtracking that recovers the subsequence itself.

```
LIS(S, n)
 1  for i = 1 to n
 2      L[i] = 1
 3      P[i] = 0
 4      for j = 1 to i - 1
 5          if S[j] < S[i] and L[j] >= L[i]
 6              L[i] = L[j] + 1
 7              P[i] = j
 8  let k be an index with the largest value L[k] among L[1..n]
 9  i = 1                                          // backtracking
10  j = k
11  repeat
12      T[i] = S[j]
13      i = i + 1;  j = P[j]
14  until j == 0
15  return L[k] and the reverse of T[1..L[k]]
```

In Pseudocode 16.1, line 5's `L[j] >= L[i]` is the same test as `L[j] + 1 > L[i]` over integers, and keeps the *last* qualifying j. The loop invariant for Equation (16.2): $L[1..i-1]$ and $P[1..i-1]$ are correct at the start of iteration i.

Book: Sections 16.4.4 and 16.4.5.

A Worked Example, by Hand

$S=\langle 9,2,5,3,7,11,8,10,13,6\rangle$. For instance at $i=5$ ($S[5]=7$): $j=2$ gives $L=2$, $j=3$ gives $L=3$, $j=4$ does not improve; so $L[5]=3$, $P[5]=3$.

Table 16.3 The dynamic-programming table for $S=\langle 9,2,5,3,7,11,8,10,13,6\rangle$. $L[i]$ is the length of the longest increasing subsequence ending at $S[i]$; $P[i]$ is the index of the element before it.

i	1	2	3	4	5	6	7	8	9	10
$S[i]$	9	2	5	3	7	11	8	10	13	6
$L[i]$	1	1	2	2	3	4	4	5	6	3
$P[i]$	0	0	2	2	3	5	5	7	8	3

From Table 16.3: $k=9$ with $L[9]=6$; backtrack $9\to 8\to 7\to 5\to 3\to 2$ gives $\langle 2,5,7,8,10,13\rangle$, length 6, at positions $2<3<5<7<8<9$.

Time. $\sum_{i=1}^{n}(i-1)=\frac{n^2-n}{2}$ inner iterations, $\Theta(n^2)$; the backtrack is $\Theta(n)$ because $P[j]<j$. At $n=10^4$ well under a second; at $n=10^6$ hours.

Book: Sections 16.4.6 and 16.4.7.

The Program

Listing 16.1 The $O(n^2)$ algorithm of Problem 16.2 in C++, printing the L and P tables.

```cpp
// Problem 16.2: the longest monotonically increasing subsequence, by the
// O(n^2) dynamic program of Section 16.4.5, with the whole table shown.
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

int main() {
    vector<int> S = {9, 2, 5, 3, 7, 11, 8, 10, 13, 6};
    int n = (int)S.size();

    vector<int> L(n + 1, 1), P(n + 1, 0);          // 1-based, as in the pseudocode

    for (int i = 1; i <= n; ++i) {
        L[i] = 1;
        P[i] = 0;
        for (int j = 1; j <= i - 1; ++j)
            // "L[j] >= L[i]" is the same test as "L[j]+1 > L[i]"
            if (S[j - 1] < S[i - 1] && L[j] >= L[i]) {
                L[i] = L[j] + 1;
                P[i] = j;
            }
    }

    cout << " i   : ";
    for (int i = 1; i <= n; ++i) cout << setw(4) << i;
    cout << "\n S[i]: ";
    for (int i = 1; i <= n; ++i) cout << setw(4) << S[i - 1];
```

```
    cout << "\n L[i]: ";
    for (int i = 1; i <= n; ++i) cout << setw(4) << L[i];
    cout << "\n P[i]: ";
    for (int i = 1; i <= n; ++i) cout << setw(4) << P[i];
    cout << "\n";

    int k = 1;
    for (int i = 2; i <= n; ++i) if (L[i] > L[k]) k = i;   // first largest L[k]

    vector<int> T;
    for (int j = k; j != 0; j = P[j]) T.push_back(S[j - 1]);

    cout << "\nlength = " << L[k] << "\nsubsequence =";
    for (int i = (int)T.size() - 1; i >= 0; --i) cout << " " << T[i];
    cout << "\n";
    return 0;
}
```

```
 i   :    1   2   3   4   5   6   7   8   9  10
 S[i]:    9   2   5   3   7  11   8  10  13   6
 L[i]:    1   1   2   2   3   4   4   5   6   3
 P[i]:    0   0   2   2   3   5   5   7   8   3

length = 6
subsequence = 2 5 7 8 10 13
```

Listing 16.1 agrees entry by entry with the hand computation.

Book: Section 16.4.8.

A Second Method: $O(n \lg n)$

Keep `tail[k]`, **the smallest value that can end an increasing subsequence of length** $k+1$ so far. `tail` is sorted, and each new element replaces exactly one entry, found by binary search: n binary searches, $O(n \lg n)$.

Listing 16.2 The $O(n \lg n)$ method for Problem 16.2, printing the `tail` array after every element.

```cpp
// Problem 16.2, second method: the same longest increasing subsequence
// in O(n lg n), by patience sorting with a binary search.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> S = {9, 2, 5, 3, 7, 11, 8, 10, 13, 6};
    int n = (int)S.size();

    vector<int> tail;           // tail[k] = smallest possible tail of an increasing
                                // subsequence of length k+1 seen so far
    vector<int> tailIdx, parent(n, -1);

    for (int i = 0; i < n; ++i) {
        // first position whose tail is >= S[i]; strict increase, so lower_bound
        int k = (int)(lower_bound(tail.begin(), tail.end(), S[i]) - tail.begin());
        if (k > 0) parent[i] = tailIdx[k - 1];
        if (k == (int)tail.size()) { tail.push_back(S[i]); tailIdx.push_back(i); }
        else                       { tail[k] = S[i];       tailIdx[k] = i;       }
        cout << "after S[" << i + 1 << "]=" << S[i] << "  tail =";
        for (int t : tail) cout << " " << t;
        cout << "\n";
    }

    vector<int> out;
    for (int i = tailIdx.back(); i != -1; i = parent[i]) out.push_back(S[i]);
```

```
    reverse(out.begin(), out.end());

    cout << "\nlength = " << tail.size() << "\nsubsequence =";
    for (int v : out) cout << " " << v;
    cout << "\n";
    return 0;
}
```

```
after S[1]=9  tail = 9
after S[2]=2  tail = 2
after S[3]=5  tail = 2 5
after S[4]=3  tail = 2 3
after S[5]=7  tail = 2 3 7
after S[6]=11  tail = 2 3 7 11
after S[7]=8  tail = 2 3 7 8
after S[8]=10  tail = 2 3 7 8 10
after S[9]=13  tail = 2 3 7 8 10 13
after S[10]=6  tail = 2 3 6 8 10 13

length = 6
subsequence = 2 3 7 8 10 13
```

Listing 16.2 finds $\langle 2,3,7,8,10,13\rangle$: same length, a different subsequence. And `tail` is **not** an answer: $2,3,6,8,10,13$ puts 6 before 8 although it comes after 13 in S.

The two classic mistakes: solving the contiguous problem, and computing only $\max_i L[i]$ *without the backtrack. Book: Sections 16.4.9 and 16.4.10.*

Problem 16.3 — Extended Binary Trees

Replace every missing child of a binary tree by an **external node**; the original nodes are **internal**; the empty tree extends to one external node.

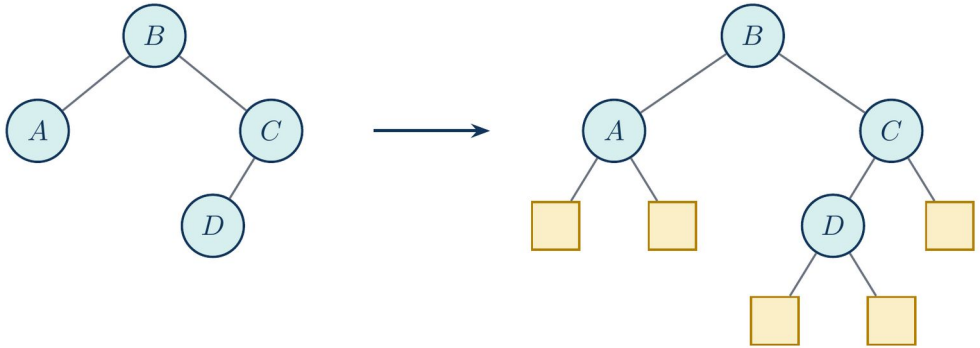


Figure 16.1 An ordinary binary tree with four nodes (left) and the extended binary tree obtained from it (right). Circles are internal nodes; squares are the external nodes that replace the null pointers. Four internal nodes, five external ones.

Problem 16.3. Prove by induction that an extended binary tree with n internal nodes has $n+1$ external nodes. In Figure 16.1: four internal nodes, five external ones.

*The trap is to induct on height; the right quantity is the number of internal nodes, and the right form is **strong** induction. Book: Sections 16.5.1 and 16.5.2.*

The Proof, and Two Checks

Claim $X(n)=n+1$. **Base** $n=0$: the empty tree extends to one external node. **Hypothesis:** $X(n')=n'+1$ for every $0\le n'<n$.
Step: remove the root; the subtrees have k and $(n-1)-k$ internal nodes, both $<n$, so

$$X(n)=X(k)+X((n-1)-k)=(k+1)+(n-k)=n+1. \tag{16.3}$$

In Equation (16.3) k **cancels**, which is why the result holds for every shape. On Figure 16.1: $k=1$, $X(1)=2$, $X(2)=3$, total 5.

Alternatives. Remove an internal node with two external children: two external nodes become one, so $X(n)-1=X(n-1)$. Or count edges: $2n$ from above, $n+X(n)-1$ from below.

> **Weak induction leaves a hole:** subtrees of sizes k and $n-1-k$ are usually not of size $n-1$. And define $X(n)$ for n **internal** nodes, exactly once and exactly right.

The total node count $2n+1$ is always odd, and the $n+1$ null links are what threaded trees reuse. Book: Sections 16.5.3 to 16.5.8.

Problem 16.4 — a 2-Approximation for TSP

Problem. On a complete graph with metric distances: choose a root r, build an MST T, list the vertices in **preorder**, return that list as a tour. Prove it (1) produces a tour, (2) runs in polynomial time, (3) has ratio bound $\rho(n)=2$.

DEFINITIONS. A **tour** is a Hamiltonian cycle; TSP asks for a shortest one. The weights satisfy the **triangle inequality** when

$$w(u,v)\le w(u,x)+w(x,v) \tag{16.4}$$

for all u, v, x. **Ratio bound** $\rho(n)$: $C/C^{*}\le\rho(n)$ on every input. A **full walk** lists a vertex each time it is met; the **preorder walk** is the full walk with repeats deleted.

Approximation is not a chapter of this book, yet every ingredient is: an MST (Ch. 11), a tree walk (Ch. 10) and Equation (16.4) (Ch. 13).

Book: Sections 16.6.1 and 16.6.2.

The Triangle Inequality, Again

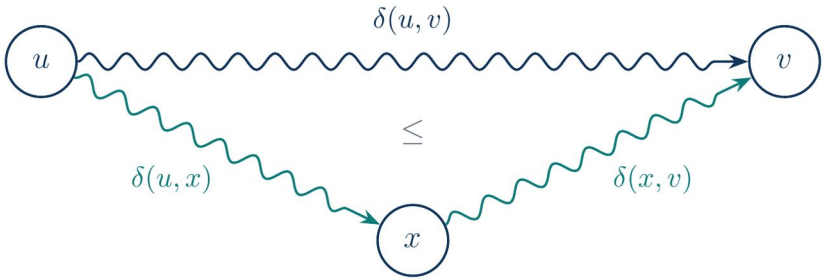


Figure 16.2 The triangle inequality: going from u to v directly is never worse than going from u to x and then from x to v. This single fact is what makes the shortcutting step of Problem 16.4 legal.

Figure 16.2 is the Chapter 13 drawing, used there for $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$. Here it makes each shortcut of the full walk no longer than the detour it replaces.

Straight-line, driving and Manhattan distance are metric; a fare table with a discounted connection may not be. Book: Section 16.6.2.

Parts 1 and 2: a Tour, in Polynomial Time

Part 1. T spans V, so its full walk visits every vertex at least once; keeping first occurrences leaves every vertex **exactly** once. G is complete, so consecutive vertices and the closing pair are joined: a Hamiltonian cycle. ∎

Table 16.4 The cost of each step of the approximation algorithm on a complete graph of n vertices. The minimum spanning tree dominates.

Step	What it does	Cost
1	Choose a root r	$\Theta(1)$
2	Build a minimum spanning tree	$\Theta(n^2)$ with Prim and an array; $O(E \lg V) = O(n^2 \lg n)$ with Kruskal or a binary heap
3	Preorder walk of T	$\Theta(n)$ — the tree has n vertices and $n-1$ edges
4	Return L	$\Theta(n)$

Part 2. By Table 16.4, $\Theta(n^2)$ with Prim and an array, the right choice on a complete graph, where it beats the $O(n^2 \lg n)$ heap version. ∎

Completeness is used once, in part 1; the triangle inequality is not used in part 1 at all. Exact TSP would need about 6×10^{16} *tours at* $n = 20$. *Book: Sections 16.6.3 and 16.6.4.*

Part 3: the Ratio Bound Is 2

Step 1, $d(T) \le d(L^*)$**:** deleting an edge of the optimal tour leaves a spanning tree T' with $d(T') \le d(L^*)$, and T is minimum. **Step 2,** $d(W) = 2\,d(T)$**:** the full walk uses every tree edge twice.

Step 3, $d(L) \le d(W)$**:** each deleted repeat replaces $u \to x \to v$ by $u \to v$, which by Equation (16.4) is no longer. **Together:**

$$d(L) \le d(W) = 2\,d(T) \le 2\,d(L^*), \tag{16.5}$$

so by Equation (16.5) $d(L)/d(L^*) \le 2$. ∎

> **Where each hypothesis is used.** Completeness in part 1; non-negativity and minimality of T in Step 1; the triangle inequality in Step 3 and nowhere else.

Without the triangle inequality no constant ratio is possible unless $P = NP$: weight 1 on the edges of H and $\rho n + 1$ elsewhere would decide Hamiltonian cycle.

Book: Sections 16.6.5 and 16.6.6.

A Complete Worked Instance

Six Profitina Laundry pickup points with the Manhattan metric: $A=(0,0)$, $B=(4,0)$, $C=(8,1)$, $D=(6,5)$, $E=(2,6)$, $F=(0,3)$.

Table 16.5 Manhattan distances in blocks between six Profitina Laundry pickup points. The matrix is symmetric, has a zero diagonal, and satisfies the triangle inequality — the three conditions that make it a metric.

	A	B	C	D	E	F
A	0	4	9	11	8	3
B	4	0	5	7	8	7
C	9	5	0	6	11	10
D	11	7	6	0	5	8
E	8	8	11	5	0	5
F	3	7	10	8	5	0

From Table 16.5: Prim from A takes $AF\,3$, $AB\,4$, $BC\,5$, $FE\,5$, $ED\,5$: $d(T)=22$. Full walk A,B,C,B,A,F,E,D,E,F,A of length $44=2\cdot 22$. Preorder A,B,C,F,E,D: tour $4+5+10+5+5+11=40$.

Brute force over the $5!/2=60$ tours: the optimum is $A\to B\to C\to D\to E\to F\to A$ at 28. Check: $22\le 28$, $44=44$, $40\le 44$, and $40/28=1.4286\le 2$.

The bound 2 is asymptotically tight for this algorithm; Christofides' algorithm achieves $3/2$*. Book: Sections 16.6.7 to 16.6.9.*

Problem 16.5 — Bellman–Ford against Dijkstra

Problem. (1) Describe their differences and applications; (2) write pseudocode and show where they differ; (3) illustrate each briefly. **What it tests:** that they are *the same idea with two orderings* of Equation (16.1).

Table 16.6 Dijkstra against Bellman–Ford. The last two rows explain the real-life split: routers use Bellman–Ford because it is local; map applications use Dijkstra because it is fast and road weights are never negative.

	Dijkstra	Bellman–Ford
Design paradigm	Greedy (Ch. 9)	Dynamic programming over path length (Ch. 7)
Negative edge weights	**Not allowed**	Allowed
Negative cycles	Undefined behavior	**Detected**; returns `false`
Order of relaxation	Edges out of the nearest unfinished vertex	All edges, $\lvert V \rvert - 1$ times
Each vertex finalized	Once, on extraction from the queue	Never — every estimate can fall again
Data structure needed	Priority queue supporting `Extract-Min` and `Decrease-Key`	None; a list of edges
Running time	$\Theta(V^2)$ with an array; $O(E \lg V)$ with a binary heap; $O(V \lg V + E)$ with a Fibonacci heap	$\Theta(V E)$
On a sparse graph, $E = \Theta(V)$	$O(V \lg V)$	$\Theta(V^2)$
Distributed implementation	Hard — needs a global minimum	Easy — each node needs only its neighbors
Fails when	Any edge weight is negative	A negative cycle is reachable (it says so)

Applications, with the reason. RIP routing is distributed Bellman–Ford, because relaxing an edge needs only its two routers (Table 16.6, "distributed"); currency arbitrage uses weights $-\ln r_{ij}$. Navigation and OSPF use Dijkstra: non-negative weights, sparse graphs, speed.

Book: Sections 16.7.1 to 16.7.3.

Dijkstra's Pseudocode

```
DIJKSTRA(G, w, s)
 1  for each v ∈ V
 2      do d[v] ← ∞
 3  d[s] ← 0
 4  S ← ∅        ▷ set of solved vertices
 5  Q ← V
 6  while Q ≠ ∅
 7      do u ← EXTRACT-MIN(Q)
 8          S ← S ∪ {u}
 9          for each v ∈ Adj[u]
10              do if d[v] > d[u] + w(u, v)
11                  then d[v] ← d[u] + w(u, v)
```

relaxing edges

Figure 16.3 Dijkstra's algorithm as given in Chapter 13. Lines 1–3 initialize, line 7 extracts the closest unfinished vertex, and the highlighted lines 9–11 are the relaxation.

In Figure 16.3, lines 4–7 keep a finished set S and a priority queue, and choose the next vertex by `Extract-Min`: the greedy choice. Each edge is relaxed **once**.

Book: Section 16.7.4.

Bellman–Ford's Pseudocode

```
BELLMAN-FORD(G, w, s)
01  for each v ∈ V[G]
02      d[v] ← ∞
03      π[v] ← NIL
04  d[s] ← 0
05  for i ← 1 to |V[G]| − 1 do
06      for each edge (u, v) ∈ E[G] do
07          RELAX(u, v, w)
08  for each edge (u, v) ∈ E[G] do
09      if d[v] > d[u] + w(u, v) then return false
10  return true
```

Figure 16.4 The Bellman–Ford algorithm as given in Chapter 13. Lines 1–4 initialize identically to Dijkstra; lines 5–7 are the $|V|-1$ passes over all edges; lines 8–10 are the negative-cycle test that Dijkstra has no counterpart for.

Four differences with Figure 16.3: (1) initialization is identical; (2) Figure 16.4 chooses nothing, it loops $|V|-1$ times over all edges; (3) so each edge is relaxed $|V|-1$ times, $\Theta(VE)$ against $O(E \lg V)$; (4) lines 8–10 detect a negative cycle, which Dijkstra cannot.

Why $|V|-1$ passes suffice: without a negative cycle a shortest path has at most $|V|-1$ edges, and pass i fixes every vertex whose shortest path has at most i edges.

Book: Section 16.7.4.

Dijkstra, Illustrated: the Start

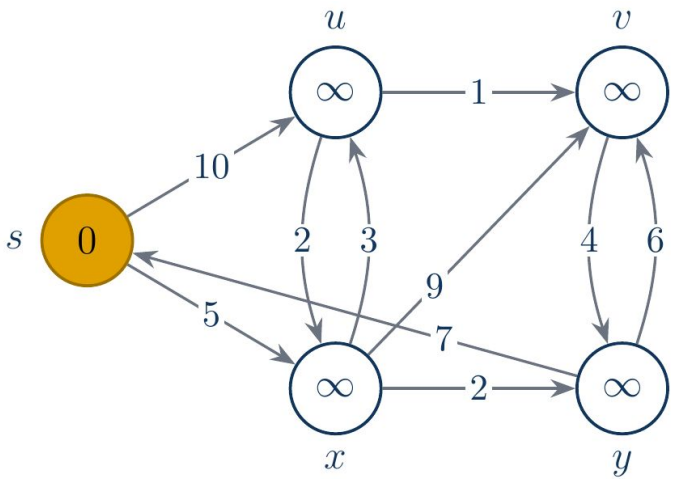


Figure 16.5 Dijkstra's example at initialization: $d[s]=0$, and $d[u]=d[v]=d[x]=d[y]=\infty$. Edge weights $s \to u=10$, $s \to x=5$, $u \to v=1$, $u \to x=2$, $x \to u=3$, $x \to v=9$, $x \to y=2$, $y \to v=6$, $y \to s=7$, $v \to y=4$.

From Figure 16.5: extract s: $d[u]=10$, $d[x]=5$. Extract x: $d[u]=8$, $d[v]=14$, $d[y]=7$. Extract y: $d[v]=13$. Extract u: $d[v]=9$. Extract v: nothing changes.

Book: Section 16.7.5.

Dijkstra, Illustrated: the End

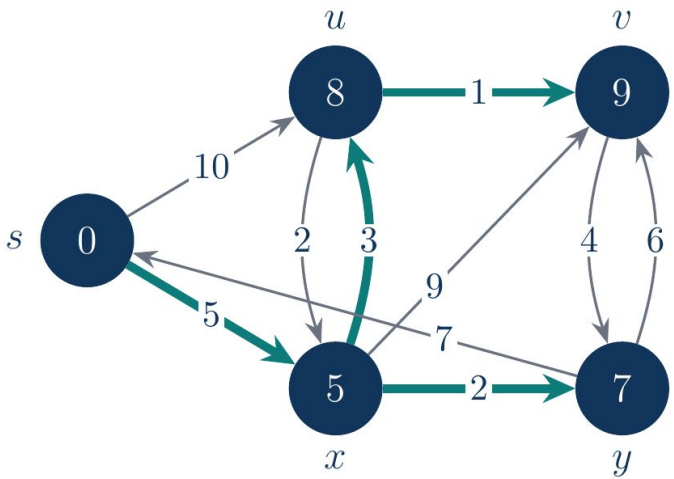


Figure 16.6 Dijkstra's example on termination: $d[s]=0$, $d[x]=5$, $d[y]=7$, $d[u]=8$, $d[v]=9$. The bold edges are the shortest-path tree. Compare the numbers with the hand trace in Section 16.7.5.

Figure 16.6 shows every vertex finalized and the shortest-path tree in bold. An illustration must show the estimates changing, at least two intermediate states.

Book: Section 16.7.5.

Bellman–Ford, Illustrated: Pass by Pass

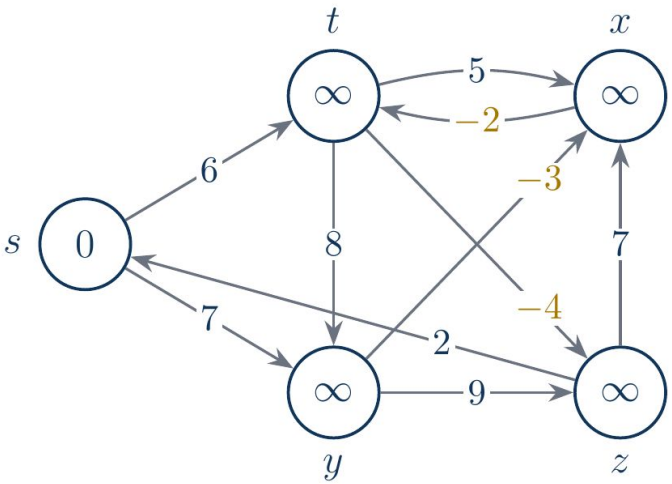


Figure 16.7 Bellman–Ford's example at initialization. Weights: $s \to t = 6$, $s \to y = 7$, $t \to x = 5$, $t \to y = 8$, $t \to z = -4$, $x \to t = -2$, $y \to x = -3$, $y \to z = 9$, $z \to s = 2$, $z \to x = 7$. Three of the ten weights are negative, so Dijkstra is not applicable.

Table 16.7 The four passes of Bellman–Ford on the graph of Figure 16.7. Note that $d[t]$ *falls* from 6 to 2 in pass 2 — an estimate Dijkstra would already have finalized — and that pass 4 changes nothing, which the closing check then confirms.

After	$d[s]$	$d[t]$	$d[x]$	$d[y]$	$d[z]$
initialize	0	∞	∞	∞	∞
pass 1	0	6	4	7	2
pass 2	0	2	4	7	2
pass 3	0	2	4	7	-2
pass 4	0	2	4	7	-2

Table 16.7 on Figure 16.7, relaxing in the listed order: pass 1 gives $(0, 6, 4, 7, 2)$ ($d[x]$ goes 11 then 4 within it); $d[t]$ falls from 6 to 2 in pass 2, an estimate Dijkstra would already have frozen.

Book: Section 16.7.5.

Bellman–Ford, the End, and Dijkstra's Failure

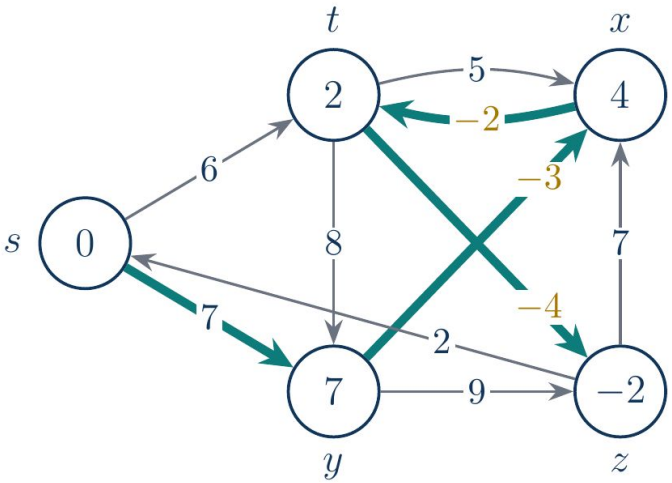


Figure 16.8 Bellman–Ford on termination: $d[s]=0$, $d[t]=2$, $d[x]=4$, $d[y]=7$, $d[z]=-2$. The bold edges are the shortest-path tree. The path to t is $s \to y \to x \to t$, costing $7-3-2=2$ — three edges, where the direct edge $s \to t$ costs 6.

Figure 16.8 is the end state. The program of Section 16.7.7 then shows *why* Dijkstra fails on a negative edge and Bellman–Ford refusing a negative cycle:

```
graph 1:  s->a = 2,  s->b = 3,  b->a = -2,  a->c = 1
  vertex   Dijkstra   Bellman-Ford
     s          0            0
     a          2            1
     b          3            3
     c          3            2
  the true shortest path to c is s->b->a->c = 3 + (-2) + 1 = 2;
  Dijkstra froze d[a] = 2 before the edge b->a could improve it.

graph 2:  s->a = 2,  a->b = 1,  b->a = -4   (cycle weight -3)
  Bellman-Ford returns false -- a negative-weight cycle is reachable from s
```

Dijkstra froze $d[a]=2$ *before* $b \to a$ *could improve it, so* $d[c]$ *stayed at 3 instead of 2. Book: Sections 16.7.5 to 16.7.7.*

Problem 16.6 — Dijkstra, the Whole Table

Problem. Dijkstra from s on $s \to a(4)$, $s \to b(1)$, $b \to a(2)$, $a \to c(5)$, $b \to c(8)$, $b \to d(10)$, $c \to d(2)$, $c \to e(6)$, $d \to e(1)$, $a \to d(9)$; give d/π after every extraction.

Table 16.8 Dijkstra's algorithm on the graph of Problem 16.6, one row per `Extract-Min`. Each cell is $d[v]/\pi[v]$; the bold cell is the vertex finalized in that row.

After	Extracted	s	a	b	c	d	e
init	—	$0/-$	$\infty/-$	$\infty/-$	$\infty/-$	$\infty/-$	$\infty/-$
1	s	**0/NIL**	$4/s$	$1/s$	$\infty/-$	$\infty/-$	$\infty/-$
2	b	$0/-$	$3/b$	**1/s**	$9/b$	$11/b$	$\infty/-$
3	a	$0/-$	**3/b**	$1/s$	$8/a$	$11/b$	$\infty/-$
4	c	$0/-$	$3/b$	$1/s$	**8/a**	$10/c$	$14/c$
5	d	$0/-$	$3/b$	$1/s$	$8/a$	**10/c**	$11/d$
6	e	$0/-$	$3/b$	$1/s$	$8/a$	$10/c$	**11/d**

In Table 16.8, iteration 2 improves a from 4 to 3 via $s \to b \to a$; iteration 3's $a \to d$ ($12 \not< 11$) is the only failed relaxation. Answer $\delta = (0, 3, 1, 8, 10, 11)$; the tree is the single path $s \to b \to a \to c \to d \to e$.

Independent check at d: the six routes cost $11, 13, 12, 11, 11, 10$; the minimum is 10.

Nine relaxations succeed, four of them `Decrease-Key` operations in a heap implementation. Book: Section 16.8.1.

Problem 16.6 in C++

Listing 16.3 Dijkstra's algorithm on the graph of Problem 16.6, printing the pair $d[v]/\pi[v]$ for every vertex after every extraction, and then every shortest path read back through π.

```cpp
// Problem 16.6: Dijkstra's algorithm with the whole (d, pi) table printed after
// every EXTRACT-MIN, so each row of the hand trace can be checked mechanically.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

const int INF = INT_MAX / 4;
struct Edge { int u, v, w; };

int main() {
    vector<string> nm = {"s", "a", "b", "c", "d", "e"};
    vector<Edge> E = {{0,1,4}, {0,2,1}, {2,1,2}, {1,3,5}, {2,3,8},
                      {2,4,10}, {3,4,2}, {3,5,6}, {4,5,1}, {1,4,9}};
    int n = (int)nm.size();
    vector<vector<pair<int,int>>> adj(n);
    for (auto& e : E) adj[e.u].push_back({e.v, e.w});

    vector<int> d(n, INF), pi(n, -1);
    vector<bool> done(n, false);
    d[0] = 0;
    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<>> Q;
    Q.push({0, 0});

    auto row = [&](const string& tag) {
```

```
        cout << setw(11) << left << tag << right;
        for (int v = 0; v < n; ++v) {
            string cell = (d[v] >= INF) ? "inf" : to_string(d[v]);
            cell += "/";
            cell += (pi[v] < 0) ? "-" : nm[pi[v]];
            cout << setw(8) << cell;
        }
        cout << "\n";
    };
    cout << setw(11) << left << "step" << right;
    for (int v = 0; v < n; ++v) cout << setw(8) << ("d/pi " + nm[v]);
    cout << "\n";
    row("init");
```

Listing 16.3 (Continued) *Dijkstra's algorithm on the graph of Problem 16.6, printing the pair d[v]/\pi[v] for every vertex after every extraction, and then every shortest path read back through \pi.*

```
    int step = 0;
    while (!Q.empty()) {
        auto [du, u] = Q.top(); Q.pop();
        if (done[u] || du > d[u]) continue;       // a stale queue entry
        done[u] = true;
        for (auto [v, w] : adj[u])
            if (d[u] + w < d[v]) { d[v] = d[u] + w; pi[v] = u; Q.push({d[v], v}); }
        row(to_string(++step) + ": " + nm[u]);
    }

    cout << "\nshortest paths from s:\n";
    for (int v = 0; v < n; ++v) {
        vector<int> path;
        for (int x = v; x != -1; x = pi[x]) path.push_back(x);
        cout << "  " << nm[v] << "  d = " << setw(2) << d[v] << "  path:";
        for (int i = (int)path.size() - 1; i >= 0; --i) cout << " " << nm[path[i]];
        cout << "\n";
    }
    return 0;
}
```

```
step         d/pi s  d/pi a  d/pi b  d/pi c  d/pi d  d/pi e
init            0/-   inf/-   inf/-   inf/-   inf/-   inf/-
1: s            0/-     4/s     1/s   inf/-   inf/-   inf/-
2: b            0/-     3/b     1/s     9/b    11/b   inf/-
3: a            0/-     3/b     1/s     8/a    11/b   inf/-
4: c            0/-     3/b     1/s     8/a    10/c    14/c
5: d            0/-     3/b     1/s     8/a    10/c    11/d
6: e            0/-     3/b     1/s     8/a    10/c    11/d

shortest paths from s:
  s  d =  0  path: s
  a  d =  3  path: s b a
  b  d =  1  path: s b
  c  d =  8  path: s b a c
  d  d = 10  path: s b a c d
  e  d = 11  path: s b a c d e
```

Row for row, Listing 16.3 prints the hand-computed table.

Check yourself: with $w(s,b)=5$ *the order becomes* s,a,b,c,d,e *and* b *ends as a leaf. Book: Section 16.8.1.*

Problem 16.7(a) — Bellman–Ford, No Cycle

Problem. Edges relaxed in the order $(b,c,-2)$, $(a,b,4)$, $(s,a,3)$, $(s,b,6)$, $(c,d,3)$, $(a,c,5)$, $(d,b,1)$. The only cycle $b \to c \to d \to b$ weighs $2>0$.

Table 16.9 Bellman–Ford on the graph of Problem 16.7(a), with $d[v]/\pi[v]$ after each pass. The estimates settle after pass 2; passes 3 and 4 change nothing and the closing check returns TRUE.

After	s	a	b	c	d	Changes
init	$0/-$	$\infty/-$	$\infty/-$	$\infty/-$	$\infty/-$	—
pass 1	$0/-$	$3/s$	$6/s$	$8/a$	$\infty/-$	3
pass 2	$0/-$	$3/s$	$6/s$	$4/b$	$7/c$	2
pass 3	$0/-$	$3/s$	$6/s$	$4/b$	$7/c$	0
pass 4	$0/-$	$3/s$	$6/s$	$4/b$	$7/c$	0

In pass 1 of Table 16.9, (c,d) is examined before (a,c) makes $d[c]$ finite: the edge order costs a pass. The shortest path $s \to b \to c \to d$ (weight 7) is fully relaxed by pass 2. Answer $(0,3,6,4,7)$, TRUE.

A pass with no change means nothing will ever change again, so an early exit is legal. Book: Section 16.8.2.

Problem 16.7(b) — a Negative Cycle Exhibited

Set $w(d,b)=-2$: the cycle $b \to c \to d \to b$ now weighs -1, reachable through $s \to b$.

Table 16.10 Bellman–Ford on the graph of Problem 16.7(b). The three estimates on the cycle fall by 1 in every pass, and the closing check finds that (b,c) can still be relaxed, since $3+(-2)=1<2$.

After	s	a	b	c	d	Changes
init	$0/-$	$\infty/-$	$\infty/-$	$\infty/-$	$\infty/-$	—
pass 1	$0/-$	$3/s$	$6/s$	$8/a$	$\infty/-$	3
pass 2	$0/-$	$3/s$	$5/d$	$4/b$	$7/c$	3
pass 3	$0/-$	$3/s$	$4/d$	$3/b$	$6/c$	3
pass 4	$0/-$	$3/s$	$3/d$	$2/b$	$5/c$	3

In Table 16.10 the estimates on the cycle fall by 1 every pass. **Closing check:** (b,c) gives $3+(-2)=1<2$, so the algorithm returns FALSE.

Exhibit the cycle. From c follow π back $|V|=5$ steps, $c \to b \to d \to c \to b \to d$, landing on the cycle at d; walk round to d again and reverse: $b \to c \to d \to b$, weight $-2+3-2=-1$.

Five steps suffice because a chain of six vertices from five must repeat one. Book: Section 16.8.2.

Problem 16.8 — Shortest Paths in a DAG

Pseudocode 16.2 Single-source shortest paths in a directed acyclic graph: one topological sort, then one relaxation per edge.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  for each vertex v in G.V
3      v.d = infinity;  v.pi = NIL
4  s.d = 0
5  for each vertex u, taken in topologically sorted order
6      for each vertex v in G.Adj[u]
7          RELAX(u, v, w)
```

(a) DFS from s gives finish times $e\,6$, $d\,7$, $c\,8$, $a\,9$, $b\,11$, $s\,12$; by *decreasing* finish time the order is s, b, a, c, d, e, and all ten edges go forward. **(b)** One pass of Pseudocode 16.2:

Table 16.11 The DAG algorithm on Problem 16.8, one row per vertex processed in topological order. Each vertex's value is final at the moment it is processed, because all of its in-edges come from vertices processed earlier.

Processed	s	a	b	c	d	e
s	$0/-$	$6/s$	$3/s$	$\infty/-$	$\infty/-$	$\infty/-$
b	$0/-$	$1/b$	$3/s$	$7/b$	$10/b$	$\infty/-$
a	$0/-$	$1/b$	$3/s$	$2/a$	$6/a$	$\infty/-$
c	$0/-$	$1/b$	$3/s$	$2/a$	$-1/c$	$4/c$
d	$0/-$	$1/b$	$3/s$	$2/a$	$-1/c$	$0/d$
e	$0/-$	$1/b$	$3/s$	$2/a$	$-1/c$	$0/d$

From Table 16.11: $\delta = (s:0, a:1, b:3, c:2, d:-1, e:0)$, and the path to e is $s \to b \to a \to c \to d \to e$ of weight 0. **(c)** Every in-edge of u is relaxed before u is processed, so there is nothing to choose and no queue. Dijkstra happens to succeed here, by luck: its extraction values $0, 3, 1, 2, -1, 0$ are not even increasing.

Book: Section 16.8.3.

Amortized Analysis: the Identities

Amortized cost with respect to a potential Φ:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}), \tag{16.6}$$

and summing Equation (16.6), the potentials telescope:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0), \text{so} \sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i \text{ whenever } \Phi(D_n) \ge \Phi(D_0). \tag{16.7}$$

Problem 16.9. A k-bit counter starts at zero and Pseudocode 16.3 runs n times; show the total is $< 2n$ by the aggregate, accounting and potential methods and by induction.

Pseudocode 16.3 `INCREMENT` on a k-bit binary counter: clear the trailing 1 bits, then set the first 0 bit.

```
INCREMENT(A, k)
1  i = 0
2  while i < k and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < k
6      A[i] = 1
```

The condition in Equation (16.7) must hold for every n, which is why one takes $\Phi(D_0) = 0$ and proves $\Phi \ge 0$. Book: Sections 16.9 and 16.9.1.

Problem 16.9 — the Aggregate Method

Counter value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 16.9 An 8-bit binary counter from 0 to 16. The shaded bits are those that flip on the way to the next value, and the last column is the running total of flips — 31 after sixteen increments, just under $2 \times 16 = 32$.

The naive bound nk charges every call as the worst. **Count by bit** (see Figure 16.9): bit i flips $\lfloor n/2^i \rfloor$ times, so

$$C(n) = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n. \tag{16.8}$$

At $n = 16$, Equation (16.8) gives $16+8+4+2+1 = 31 < 32$.

Book: Section 16.9.1, solution (a).

Accounting, Potential, and Induction

(b) Accounting. Fee 2 per call; the one set-to-1 spends 1 and leaves 1 on the bit; resets are paid by that credit. The bank equals the number of 1 bits, never negative. **(c) Potential** $\Phi = b_i$: $\hat{c}_i \le (t_i + 1) + (1 - t_i) = 2$; call 8 costs 4 and $\hat{c}_8 = 4 + 1 - 3 = 2$.

(d) Induction needs a stronger claim: $C(n) + v(n) \le 2n$, with v the number of 1 bits. Step: $C(n) + v(n) = C(n-1) + v(n-1) + 2$. **The potential method is an induction with a strengthened hypothesis, written as bookkeeping.**

Table 16.12 The binary counter by all three methods for the first eight calls. The running total $C(i)$ stays below $2i$ (aggregate), the bank never goes negative and always equals the number of 1 bits (accounting), and every amortized cost is exactly 2 (potential).

i	Counter	t_i	c_i	$C(i)$	$2i$	Bank	Φ_i	$c_i=c_i+\Phi_i-\Phi_{i-1}$
1	0001	0	1	1	2	1	1	$1+1-0=2$
2	0010	1	2	3	4	1	1	$2+1-1=2$
3	0011	0	1	4	6	2	2	$1+2-1=2$
4	0100	2	3	7	8	1	1	$3+1-2=2$
5	0101	0	1	8	10	2	2	$1+2-1=2$
6	0110	1	2	10	12	2	2	$2+2-2=2$
7	0111	0	1	11	14	3	3	$1+3-2=2$
8	1000	3	4	15	16	1	1	$4+1-3=2$

In Table 16.12 the *bank* and Φ_i columns are equal: the credit and the potential are the same number.

Book: Section 16.9.1, solutions (b) to (d).

Problem 16.9 in C++

Listing 16.4 The binary counter of Problem 16.9 with the aggregate total, the accounting bank and the potential-method amortized cost printed for every call, followed by the aggregate sum for larger n.

```cpp
// Problem 16.9: the k-bit binary counter analyzed three ways at once.
//   aggregate : the running total of real costs, compared with 2n
//   accounting: charge 2 per INCREMENT; the bank must never go negative
//   potential : Phi = number of 1 bits; amortized cost = c + Phi_i - Phi_{i-1}
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

const int K = 8;

int main() {
    int A[K] = {0};
    long long total = 0, bank = 0;
    int phiPrev = 0;
    cout << "  i  counter   t  cost  total   2i  bank  Phi  amortized\n";
    for (int i = 1; i <= 16; ++i) {
        int j = 0, t = 0;
        while (j < K && A[j] == 1) { A[j] = 0; ++j; ++t; }   // clear trailing 1s
        if (j < K) A[j] = 1;                                 // set one 0 to 1
        int cost = t + (j < K ? 1 : 0);
        total += cost;
        bank += 2 - cost;                  // charged 2, spent the real cost
        int phi = 0;
        for (int b = 0; b < K; ++b) phi += A[b];
        int amort = cost + phi - phiPrev;
        phiPrev = phi;
        string bits;
```

```
        for (int b = K - 1; b >= 0; --b) bits += char('0' + A[b]);
        cout << setw(3) << i << "  " << bits << setw(3) << t << setw(6) << cost
             << setw(7) << total << setw(5) << 2 * i << setw(6) << bank
             << setw(5) << phi << setw(8) << amort << "\n";
    }
    // the aggregate formula, checked for a larger n
    for (long long n : {16LL, 100LL, 1000LL, 1000000LL}) {
        long long c = 0;
        for (long long p = 1; p <= n; p *= 2) c += n / p;   // sum of floor(n / 2^i)
        cout << "n = " << setw(7) << n << ":  total flips = " << setw(7) << c
             << "  <  2n = " << 2 * n << "\n";
    }
    return 0;
}
```

```
  i  counter   t  cost  total   2i  bank  Phi  amortized
  1  00000001  0     1      1    2     1    1       2
  2  00000010  1     2      3    4     1    1       2
  3  00000011  0     1      4    6     2    2       2
  4  00000100  2     3      7    8     1    1       2
  5  00000101  0     1      8   10     2    2       2
  6  00000110  1     2     10   12     2    2       2
  7  00000111  0     1     11   14     3    3       2
  8  00001000  3     4     15   16     1    1       2
  9  00001001  0     1     16   18     2    2       2
 10  00001010  1     2     18   20     2    2       2
 11  00001011  0     1     19   22     3    3       2
 12  00001100  2     3     22   24     2    2       2
 13  00001101  0     1     23   26     3    3       2
 14  00001110  1     2     25   28     3    3       2
 15  00001111  0     1     26   30     4    4       2
 16  00010000  4     5     31   32     1    1       2
n =      16:  total flips =      31  <  2n = 32
n =     100:  total flips =     197  <  2n = 200
n =    1000:  total flips =    1994  <  2n = 2000
n = 1000000:  total flips = 1999993  <  2n = 2000000
```

Listing 16.4: at $n=1000$ the total is 1994, an amortized 1.994 per call.

Check yourself: if bit i costs 2^i, $n=2^m$ increments cost $n(\lg n+1)$, so amortization spreads cost but does not make it disappear. Book: Section 16.9.1.

Problem 16.10 — the Dynamic Table

Pseudocode 16.4 `TABLE-INSERT` for a table that doubles when full. Lines 3–6 are the expensive expansion; line 7 is the elementary insertion.

```
TABLE-INSERT(T, x)
1  if T.size == 0
2      allocate T.table with 1 slot;  T.size = 1
3  if T.num == T.size
4      allocate new-table with 2 * T.size slots
5      insert all items in T.table into new-table
6      free T.table;  T.table = new-table;  T.size = 2 * T.size
7  insert x into T.table
8  T.num = T.num + 1
```

(a) For Pseudocode 16.4, $c_i = i$ when $i-1$ is a power of 2 ($i = 2, 3, 5, 9, 17, \ldots$), else 1. **(b) Aggregate:** $n + \sum_j 2^j < 3n$; at $n = 17$, $17 + 31 = 48 < 51$.

(c) Accounting, fee 3: one unit writes the item, one pays for moving it later, one for moving an older item that has already spent its credit. Just before an expansion of $2m$ items the bank holds exactly $2m$.

(d) Potential

$$\Phi(T) = 2 \cdot num[T] - size[T]. \tag{16.9}$$

Equation (16.9) is 0 when empty and ≥ 0 because the table is at least half full.

Book: Section 16.9.2, solutions (a) to (d).

Both Cases, and Nine Insertions

No expansion: $\hat{c}_i = 1 + (2num_i - size_i) - (2(num_i - 1) - size_i) = 3.$ **Expansion:** $\hat{c}_i = num_i + (2num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) = 3.$ ($\hat{c}_1 = 2.$)

Table 16.13 The dynamic table of Problem 16.10 by all three methods for the first nine insertions. The total stays below $3i$, the bank stays positive, and the amortized cost is 3 from the second insertion on — including at the expensive insertions $i = 2, 3, 5, 9$.

i	$size_i$	num_i	c_i	Σc	$3i$	Bank	$\Phi_i = 2num_i - size_i$	$\hat{c}_i$
1	1	1	1	1	3	2	$2 - 1 = 1$	$1 + 1 - 0 =$
2	2	2	2	3	6	3	$4 - 2 = 2$	$2 + 2 - 1 =$
3	4	3	3	6	9	3	$6 - 4 = 2$	$3 + 2 - 2 =$
4	4	4	1	7	12	5	$8 - 4 = 4$	$1 + 4 - 2 =$
5	8	5	5	12	15	3	$10 - 8 = 2$	$5 + 2 - 4 =$
6	8	6	1	13	18	5	$12 - 8 = 4$	$1 + 4 - 2 =$
7	8	7	1	14	21	7	$14 - 8 = 6$	$1 + 6 - 4 =$
8	8	8	1	15	24	9	$16 - 8 = 8$	$1 + 8 - 6 =$

Table 16.13 (Continued) *The dynamic table of Problem 16.10 by all three methods for the first nine insertions. The total stays below 3i, the bank stays positive, and the amortized cost is 3 from the second insertion on — including at the expensive insertions i=2,3,5,9.*

i	$size_i$	num_i	c_i	Σc	$3i$	Bank	$\Phi_i = 2num_i - size_i$	c_i
9	16	9	9	24	27	3	$18 - 16 = 2$	$9 + 2 - 8 =$

In Table 16.13 the bank is always $\Phi_i + 1$: the extra unit is the change left from the first insertion, charged 3 but costing 2 amortized.

The program of Section 16.9.2 prints the same ledgers and totals 48, 2023 and 2048575 for $n = 17,\ 1000,\ 10^6$. *Book: Section 16.9.2, solution (e).*

A Tempting Wrong Potential

Try $\Phi = num - size$**.** Case 1: $\hat{c}_i = 2$. Case 2: $\hat{c}_i = num_i + (2 - num_i) = 2$. Every amortized cost at most 2, "proving" $2n$. But at $n = 17$ the actual total is **48**, not at most 34.

Why it fails: this Φ is **negative**, $17 - 32 = -15$ at $n = 17$. By Equation (16.7), $\sum c_i = 33 + 15 = 48$: the amortized total understates the truth by exactly the debt the potential ran up.

$\Phi = num$ is non-negative but gives $\hat{c}_i = num_i + 1$ at an expansion: not constant, so no better than naive. A good potential is large *exactly when an expensive operation is imminent*.

> **Always check** $\Phi(D_i) \geq \Phi(D_0)$ **before believing an amortized bound.** A potential that goes negative borrows against the future.

Check yourself: growing by a fixed 4 slots costs 57 for 17 insertions and $\Theta(n^2)$ *in general. Book: Section 16.9.2, solution (f).*

Problem 16.11 — Floyd–Warshall by Hand

The graph for Problems 16.11 and 16.12: $1 \to 2(4)$, $1 \to 4(5)$, $2 \to 3(-2)$, $3 \to 1(3)$, $3 \to 4(2)$, $4 \to 2(1)$, $4 \to 3(6)$. Its five cycles all have positive weight.

The recurrence: a path with interior vertices in $\{1..k\}$ avoids k or passes it once,

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1, \end{cases} \tag{16.10}$$

and when the second argument of Equation (16.10) wins, $\pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$.

Table 16.14 The starting matrices of Problem 16.11: $D^{(0)}$ is the weight matrix, and $\Pi^{(0)}$ records the tail of each edge.

$D^{(0)}$	1	2	3	4		$\Pi^{(0)}$	1	2	3	4
1	0	4	∞	5		**1**	NIL	1	NIL	1
2	∞	0	−2	∞		**2**	NIL	NIL	2	NIL
3	3	∞	0	2		**3**	3	NIL	NIL	3
4	∞	1	6	0		**4**	NIL	4	4	NIL

Table 16.14 is round 0. **Shortcut:** in round k only entries with both d_{ik} and d_{kj} finite, and $i, j \neq k$, can change.

Book: Sections 16.10 and 16.10.1.

The Eight Changes, and the Answer

Table 16.15 Every entry that changes during Floyd–Warshall on Problem 16.11, with the comparison that changed it. Note that d_{32} changes twice: a route found in round 1 is beaten by a route found in round 4.

Round k	Entry	Old	Candidate $d_{ik}+d_{kj}$	New	New π_{ij}
1	d_{32}	∞	$3+4=7$	7	1
2	d_{13}	∞	$4+(-2)=2$	2	2
2	d_{43}	6	$1+(-2)=-1$	-1	2
3	d_{14}	5	$2+2=4$	4	3
3	d_{21}	∞	$-2+3=1$	1	3
3	d_{24}	∞	$-2+2=0$	0	3
3	d_{41}	∞	$-1+3=2$	2	3
4	d_{32}	7	$2+1=3$	3	4

Table 16.15: d_{32} changes twice, 7 via 3 → 1 → 2 in round 1, then 3 via 3 → 4 → 2 in round 4. Result

$$D^{(4)} = \begin{pmatrix} 0 & 4 & 2 & 4 \\ 1 & 0 & -2 & 0 \\ 3 & 3 & 0 & 2 \\ 2 & 1 & -1 & 0 \end{pmatrix}.$$

Paths from $\Pi^{(4)}$: $1 \to 2 \to 3 \to 4$ (weight 4), $4 \to 2 \to 3 \to 1$ (2), $3 \to 4 \to 2$ (3). **Negative cycles:** look at the diagonal; here it is all zero.

Book: Section 16.10.1.

Problem 16.11 in C++

Listing 16.5 Floyd–Warshall on the graph of Problem 16.11, printing $D^{(k)}$ and $\Pi^{(k)}$ for $k = 0, \ldots, 4$ and marking each changed entry, followed by the negative-cycle test on the diagonal.

```cpp
// Problem 16.11: Floyd-Warshall on a four-vertex graph with a negative edge,
// printing every distance matrix D(0)..D(4) and every predecessor matrix
// Pi(0)..Pi(4), and marking with * each entry that changed in the round.
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;

const int INF = 1000000;
const int n = 4;
struct Edge { int u, v, w; };

int D[n + 1][n + 1], P[n + 1][n + 1];          // 1-based; P = 0 means NIL
bool changed[n + 1][n + 1];

void print(int k) {
    cout << "D(" << k << ")                          Pi(" << k << ")\n";
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= n; ++j) {
            string c = (D[i][j] >= INF) ? "inf" : to_string(D[i][j]);
            if (changed[i][j]) c += "*";
            cout << setw(6) << c;
        }
        cout << "        ";
        for (int j = 1; j <= n; ++j) {
            if (P[i][j] == 0) cout << setw(4) << "NIL"; else cout << setw(4) << P[i][j];
        }
```

```
        cout << "\n";
    }
    cout << "\n";
}

int main() {
    vector<Edge> E = {{1,2,4}, {1,4,5}, {2,3,-2}, {3,1,3},
                      {3,4,2}, {4,2,1}, {4,3,6}};
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j) { D[i][j] = (i == j) ? 0 : INF; P[i][j] = 0; }
    for (auto& e : E) { D[e.u][e.v] = e.w; P[e.u][e.v] = e.u; }
    print(0);
```

Listing 16.5 (Continued) *Floyd–Warshall on the graph of Problem 16.11, printing D^{(k)} and \Pi^{(k)} for k = 0, \ldots, 4 and marking each changed entry, followed by the negative-cycle test on the diagonal.*

```
    for (int k = 1; k <= n; ++k) {
        for (int i = 1; i <= n; ++i)
            for (int j = 1; j <= n; ++j) {
                changed[i][j] = false;
                if (D[i][k] < INF && D[k][j] < INF && D[i][k] + D[k][j] < D[i][j]) {
                    D[i][j] = D[i][k] + D[k][j];   // go through k instead
                    P[i][j] = P[k][j];             // last hop copied from k's route
                    changed[i][j] = true;
                }
            }
        print(k);
    }

    bool negCycle = false;
    for (int i = 1; i <= n; ++i) if (D[i][i] < 0) negCycle = true;
    cout << "negative diagonal entry: " << (negCycle ? "yes" : "no") << "\n";
    return 0;
}
```

```
D(0)                          Pi(0)
     0     4   inf     5          NIL   1 NIL   1
   inf     0    -2   inf          NIL NIL   2 NIL
     3   inf     0     2            3 NIL NIL   3
   inf     1     6     0          NIL   4   4 NIL

D(1)                          Pi(1)
     0     4   inf     5          NIL   1 NIL   1
   inf     0    -2   inf          NIL NIL   2 NIL
     3    7*     0     2            3   1 NIL   3
   inf     1     6     0          NIL   4   4 NIL

D(2)                          Pi(2)
     0     4    2*     5          NIL   1   2   1
   inf     0    -2   inf          NIL NIL   2 NIL
     3     7     0     2            3   1 NIL   3
   inf     1   -1*     0          NIL   4   2 NIL

D(3)                          Pi(3)
     0     4     2    4*          NIL   1   2   3
    1*     0    -2    0*            3 NIL   2   3
     3     7     0     2            3   1 NIL   3
    2*     1    -1     0            3   4   2 NIL

D(4)                          Pi(4)
     0     4     2     4          NIL   1   2   3
     1     0    -2     0            3 NIL   2   3
     3    3*     0     2            3   4 NIL   3
```

```
     2     1    -1     0          3   4   2 NIL

negative diagonal entry: no
```

The starred entries printed by Listing 16.5 are exactly the eight rows of the change table: $4^3 = 64$ comparisons in all.

Check yourself: with $w(4,2) = -1$ *the diagonal becomes* $(0,-1,-1,-2)$*, flagging the cycle* $2 \to 3 \to 4 \to 2$*. Book: Section 16.10.1.*

Problem 16.12 — Johnson's Reweighting

(a) Add q with zero-weight edges to every vertex; Bellman–Ford from q gives $h=(0,0,-2,0)$: only vertex 3 is reached for less than free, through $2 \to 3$.

(b) Reweight every edge by

$$w'(u,v)=w(u,v)+h(u)-h(v). \tag{16.11}$$

Table 16.16 Johnson's reweighting on the graph of Problem 16.12. Every reweighted edge is non-negative, and the negative edge $2 \to 3$ has become an edge of weight 0.

Edge (u,v)	$w(u,v)$	$h(u)$	$h(v)$	$w'(u,v)=w+h(u)-h(v)$
$(1,2)$	4	0	0	$4+0-0=4$
$(1,4)$	5	0	0	$5+0-0=5$
$(2,3)$	-2	0	-2	$-2+0+2=0$
$(3,1)$	3	-2	0	$3-2-0=1$
$(3,4)$	2	-2	0	$2-2-0=0$
$(4,2)$	1	0	0	$1+0-0=1$
$(4,3)$	6	0	-2	$6+0+2=8$

Table 16.16 applies Equation (16.11): all non-negative, by the triangle inequality $h(v) \le h(u) + w(u,v)$; the zeros, $(2,3)$ and $(3,4)$, lie on shortest paths from q. **(c)** Dijkstra from 4 gives $\hat{d} = (2,1,1,0)$, and $\delta(u,v) = \hat{d}(u,v) - h(u) + h(v)$ gives $(2,1,-1,0)$.

(d) Along any path the corrections telescope:

$$w'(p) = \sum_{i=1}^{k} \left(w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i) \right) = w(p) + h(v_0) - h(v_k), \tag{16.12}$$

so by Equation (16.12) every path between the same endpoints shifts by the same amount.

The program of Section 16.10.2 recovers exactly $D^{(4)}$ of Problem 16.11: two different ideas, sixteen agreeing entries. Book: Section 16.10.2.

Problems 16.13 and 16.18 — Thirty More Statements

16.13, the traps. (2) **F**: any cycle, even of length 2, forbids a topological sort. (4) **F**: greedy 0/1 knapsack gets 160 where $\{2,3\}$ gives 220. (7) **F**: undirected DFS has no cross edges. (11) **F**: Φ must stay $\geq \Phi(D_0)$ at *every* step. (13) **F**: relaxation never increases $d[v]$.

16.13, true ones worth a reason. (1) Kruskal $O(E \lg V)$; (9) Johnson beats Floyd–Warshall on sparse graphs; (10) an amortized cost can exceed the worst case (`PUSH` 2 against 1); (14) every MST edge is lightest across some cut; (15) shortest paths never repeat a vertex.

16.18, the traps. (2) **F**: `BUILD-MAX-HEAP` is $\Theta(n)$. (6) **F**: memoization may skip subproblems. (7) **F**: optimal substructure without the greedy-choice property is not enough. (10) **F**: Kruskal keeps a forest. (11) **F**: edges from unreachable vertices are never relaxed. (13) **F**: $\Theta(nW)$ is pseudo-polynomial.

16.18, true ones. (1) heapsort in place, $\Theta(n \lg n)$; (3) AVL $N(h) = 1 + N(h-1) + N(h-2)$; (5) $\Omega(n \lg n)$ comparisons; (8) $O(m\alpha(n))$; (9) Prim grows one tree; (12) the methods may share the cost out differently; (14) more than $V-1$ edges force a cycle; (15) the cycle property.

Book: Sections 16.11.1 and 16.11.6.

Problem 16.14 — the Longest Path in a DAG

Design. Process vertices in topological order and relax with **max** instead of min: Problem 16.8 turned inside out, since the DAG algorithm accepts any weights, including negated ones.

Pseudocode 16.5 Longest paths from s in a weighted DAG: the DAG shortest-path algorithm with the comparison reversed.

```
LONGEST-PATH-DAG(G, w, s)
1  topologically sort V(G)
2  for each v in V:  dist[v] = -infinity;  par[v] = NIL
3  dist[s] = 0
4  for each u in topological order
5      if dist[u] > -infinity
6          for each edge (u,v) in Adj[u]
7              if dist[u] + w(u,v) > dist[v]
8                  dist[v] = dist[u] + w(u,v);  par[v] = u
9  return dist, par
```

Pseudocode 16.5 is $\Theta(V+E)$. **Worked instance** in the order $1,3,2,5,4,6,7$: $dist[5]=15$, $dist[4]=11$, $dist[6]=19$, $dist[7]=22$; the longest path is $1 \to 2 \to 5 \to 6 \to 7$ of weight $6+9+4+3=22$.

On a general digraph it fails twice: there is no topological order, and longest simple path is NP-hard (a Hamiltonian path is a longest path with unit weights).

This is the critical-path method of Section 13.9.7. Book: Section 16.11.2.

Problems 16.15 and 16.16 — Trees and Huffman

16.15: a tree on n vertices has $n-1$ edges. Strong induction: a tree with $n \geq 2$ vertices has a leaf ℓ; removing ℓ and its edge leaves a tree T' on $n-1$ vertices, connected and acyclic; so $|E(T)| = (n-2)+1 = n-1$. ∎

16.16 (1) Prefix-free: symbols sit only at leaves, and a leaf has no descendants, so no codeword is a prefix of another. **(2)** $n-1$ merges of two `Extract-Min` and one `Insert`: $O(n \lg n)$.

(3) Optimal. *Greedy choice:* swapping the two rarest symbols to the deepest siblings changes the cost by $-(f(a)-f(x))(d(a)-d(x)) \leq 0$. *Optimal substructure:* merging them gives $cost(T) = cost(T') + f(x) + f(y)$. Induction on n finishes it. ∎

The tree fact is used silently in Problem 16.4: a tour minus one edge is a spanning tree. Book: Sections 16.11.3 and 16.11.4.

Problem 16.17 — Floyd–Warshall against Johnson

Differences. Floyd–Warshall: a dynamic program over allowed interior vertices, $\Theta(V^3)$ time, $\Theta(V^2)$ space, negative diagonal for negative cycles. Johnson: a reduction, one Bellman–Ford and V Dijkstras, $O(V^2 \lg V + V E)$. **Dense or small: Floyd–Warshall; sparse: Johnson.**

Pseudocode 16.6 Floyd–Warshall (left) and Johnson (right). Floyd–Warshall never changes the weights; Johnson changes them once, in line 4, so that Dijkstra becomes legal.

```
FLOYD-WARSHALL(W)
1 D = W
2 for k = 1 to n
3   for i = 1 to n
4     for j = 1 to n
5       D[i][j] = min(D[i][j],
6                     D[i][k]+D[k][j])
7 return D
```

```
JOHNSON(G, w)
1 add vertex q joined to all v with w(q,v)=0
2 if BELLMAN-FORD(G', w, q) = false: report neg. cycle
3 h(v) = delta(q,v) for all v
4 w'(u,v) = w(u,v) + h(u) - h(v)  for all (u,v)
5 for each u in V
6   run DIJKSTRA(G, w', u) giving dhat(u,v)
7   D[u][v] = dhat(u,v) - h(u) + h(v)
8 return D
```

Pseudocode 16.6: Johnson's reweighting is sound by Equation (16.12). At $V = 1000$, $E = 3000$, Floyd–Warshall does 10^9 steps and Johnson about 10^7. On a five-vertex graph the program of Section 16.11.5 prints non-negative reweighted edges $4, 13, 0, 0, 10, 0, 2, 0, 2$.

Book: Section 16.11.5.

Problem 16.19 — 0/1 Knapsack

Recurrence. Item i is left out or taken:

$$c[i][j] = \begin{cases} 0 & i = 0 \text{ or } j = 0, \\ c[i-1][j] & w_i > j, \\ \max(c[i-1][j],\ v_i + c[i-1][j-w_i]) & \text{otherwise.} \end{cases} \tag{16.13}$$

Pseudocode 16.7 The 0/1 knapsack dynamic program: n rows of $W+1$ entries, each computed in constant time.

```
KNAPSACK(w, v, n, W)
1 for j = 0 to W:  c[0][j] = 0
2 for i = 1 to n
3   c[i][0] = 0
4   for j = 1 to W
5     if w[i] > j: c[i][j] = c[i-1][j]
6     else:        c[i][j] = max(c[i-1][j], v[i] + c[i-1][j-w[i]])
7 return c[n][W]
```

Pseudocode 16.7 fills Equation (16.13) in $\Theta(nW)$. **Is it polynomial? No.** W takes $\lceil \lg(W+1) \rceil$ bits, so $\Theta(nW)$ is exponential in the input *size*: **pseudo-polynomial**. 0/1 knapsack is NP-hard.

Listing 16.6 The 0/1 knapsack dynamic program of Problem 16.19, printing the whole c table and reconstructing the chosen set by walking it backward.

```cpp
// Problem 16.19: the 0/1 knapsack by dynamic programming, whole table shown.
// Five add-on services, each costing machine-minutes and returning rupiah.
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
using namespace std;

int main() {
    const int n = 5, W = 10;
    string name[n + 1] = {"", "starch", "express", "perfume", "fold", "iron"};
    int w[n + 1] = {0, 3, 4, 5, 2, 6};           // machine-minutes
    int v[n + 1] = {0, 30, 50, 60, 20, 70};      // thousand rupiah

    vector<vector<int>> c(n + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= n; ++i)
        for (int j = 0; j <= W; ++j)
            c[i][j] = (w[i] > j) ? c[i-1][j]
                                 : max(c[i-1][j], c[i-1][j - w[i]] + v[i]);

    cout << "      j =";
    for (int j = 0; j <= W; ++j) cout << setw(4) << j;
    cout << "\n";
    for (int i = 0; i <= n; ++i) {
        cout << setw(8) << (i == 0 ? string("-") : name[i]) << " ";
        for (int j = 0; j <= W; ++j) cout << setw(4) << c[i][j];
        cout << "\n";
    }
```

```
    cout << "\nbest value = " << c[n][W] << "\nchosen:";
    int j = W, used = 0;
    for (int i = n; i >= 1; --i)
        if (c[i][j] != c[i-1][j]) { cout << " " << name[i]; used += w[i]; j -= w[i]; }
    cout << "\nminutes used = " << used << " of " << W << "\n";
    return 0;
}
```

```
      j =   0   1   2   3   4   5   6   7   8   9  10
        -   0   0   0   0   0   0   0   0   0   0   0
   starch   0   0   0  30  30  30  30  30  30  30  30
  express   0   0   0  30  50  50  50  80  80  80  80
  perfume   0   0   0  30  50  60  60  80  90 110 110
     fold   0   0  20  30  50  60  70  80  90 110 110
     iron   0   0  20  30  50  60  70  80  90 110 120

best value = 120
chosen: iron express
minutes used = 10 of 10
```

Listing 16.6: express and iron give 120 in 10 minutes; $c[i][j] \neq c[i-1][j]$ means item i was taken. Book: Section 16.11.7.

Problem 16.20 — a 2-Approximation for Vertex Cover

`APPROX-VERTEX-COVER` picks any uncovered edge (u, v), adds **both** endpoints, and deletes every edge they touch. **(1)** It stops only when every edge is covered. **(2)** Each edge is examined a constant number of times: $O(V+E)$.

(3) Ratio 2. The picked edges A form a **matching**: no two share an endpoint. Any cover contains an endpoint of each, all distinct, so $|C^*| \geq |A|$. The algorithm returns $|C| = 2|A| \leq 2|C^*|$. ∎

On the program's example: $|C| = 6$, $|C^*| = 4$, ratio 1.5, matching size 3.

> **The same skeleton as Problem 16.4:** find a quantity that lower-bounds the optimum ($|A|$ here, $d(T)$ there) and show the answer is at most twice it.

Book: Section 16.11.8.

Problem 16.21 — Prim against Kruskal

Both add **safe** edges by the cut property. **Prim** grows one tree with a priority queue: $\Theta(V^2)$ with an array, for dense graphs. **Kruskal** grows a forest with a sort and union–find: $O(E \lg V)$, for sparse graphs or pre-sorted edges.

Pseudocode 16.8 Prim (left) and Kruskal (right). Line 5 of each is where they differ: nearest vertex to the tree, against the next edge that joins two components.

```
PRIM(G, w, r)                          KRUSKAL(G, w)
1 for each v: key[v]=inf, pi[v]=NIL    1 A = empty
2 key[r] = 0                           2 for each v: MAKE-SET(v)
3 Q = V                                3 sort E by weight, ascending
4 while Q not empty                    4 for each (u,v) in E, in order
5   u = EXTRACT-MIN(Q)                 5   if FIND-SET(u) != FIND-SET(v)
6   for each v in Adj[u]               6     A = A + {(u,v)}
7     if v in Q and w(u,v) < key[v]    7     UNION(u,v)
8       pi[v] = u; key[v] = w(u,v)     8 return A
```

Table 16.17 Prim and Kruskal on the six-vertex graph of Table 16.5. Prim adds $F-E$ then $E-D$; Kruskal adds $D-E$ then $E-F$. The *order* differs; the edge *set* $\{AF, AB, BC, DE, EF\}$ and the weight 22 do not.

Prim from A	Edge	w	Cum.		Kruskal	Edge	w	Verdict	Cum.
1	(root A)	0	0		1	$A-F$	3	accept	3
2	$A-F$	3	3		2	$A-B$	4	accept	7
3	$A-B$	4	7		3	$B-C$	5	accept	12
4	$B-C$	5	12		4	$D-E$	5	accept	17
5	$F-E$	5	17		5	$E-F$	5	accept	22
6	$E-D$	5	22		—	stop: 5 edges			22

Pseudocode 16.8: Prim asks *which vertex is nearest the tree?*, Kruskal *does this edge join two components?* Table 16.17, on the six laundry points of Table 16.5: different order, same edge set, weight 22.

Three edges tie at weight 5, yet exhaustive search finds exactly one MST: ties permit a second tree, they do not produce one. Book: Section 16.11.9.

The Problems Compared

Table 16.18 The twenty-one problems of this chapter at a glance: what kind of problem each is, the tool that solves it and where the book taught it, and what the solution costs.

Problem	Section	What it is	Main tool, and where it was taught	Cost of the solution
16.1 fifteen statements	16.3	recognition across the book	O against Ω, worst against average case (Ch. 3, 5, 6, 11)	—
16.2 increasing subsequence	16.4	design and analysis	dynamic programming, strengthened subproblem (Ch. 7)	$\Theta(n^2)$; $O(n \lg n)$
16.3 extended binary trees	16.5	proof by induction	strong induction on internal nodes (Ch. 3, 6)	—
16.4 metric TSP	16.6	three-part guarantee	MST, preorder walk, triangle inequality (Ch. 10, 11, 13)	$\Theta(n^2)$, ratio 2
16.5 Bellman–Ford vs Dijkstra	16.7	comparison	relaxation and its order (Ch. 13)	$\Theta(V E)$; $O(E \lg V)$
16.6 Dijkstra table	16.8.1	trace	`Extract-Min` plus relaxation (§13.7)	$O(E \lg V)$
16.7 negative cycle	16.8.2	trace and detection	$\lvert V \rvert - 1$ passes, closing check, π walk (§13.8)	$\Theta(V E)$
16.8 DAG shortest paths	16.8.3	trace	DFS topological sort plus relaxation (§10, §13.9)	$\Theta(V + E)$
16.9 binary counter	16.9.1	amortized, four ways	aggregate, accounting, potential, induction (§14.3–14.6)	$< 2n$ in total

Table 16.18 (Continued) *The twenty-one problems of this chapter at a glance: what kind of problem each is, the tool that solves it and where the book taught it, and what the solution costs.*

Problem	Section	What it is	Main tool, and where it was taught	Cost of the solution
16.10 dynamic table	16.9.2	amortized, three ways	potential $2\,num - size$ (§14.7)	$< 3n$ in total
16.11 Floyd–Warshall	16.10.1	trace	recurrence over interior vertices (§15.6)	$\Theta(V^3)$
16.12 Johnson	16.10.2	trace	reweighting by potentials (§15.8)	$O(V^2 \lg V + V E)$
16.13 fifteen statements	16.11.1	recognition across the book	Ch. 9–15	—
16.14 longest path in a DAG	16.11.2	design and analysis	topological order, maximizing relaxation (§13.9)	$\Theta(V+E)$
16.15 tree edges	16.11.3	proof by induction	removing a leaf (Ch. 10, 11)	—
16.16 Huffman	16.11.4	three-part guarantee	exchange argument, optimal substructure (Ch. 9)	$O(n \lg n)$
16.17 Floyd–Warshall vs Johnson	16.11.5	comparison	dense against sparse (Ch. 15)	$\Theta(V^3)$; $O(V^2 \lg V + V E)$
16.18 fifteen statements	16.11.6	recognition across the book	Ch. 5–14	—
16.19 0/1 knapsack	16.11.7	design and analysis	dynamic programming; pseudo-polynomial time (Ch. 7)	$\Theta(nW)$

Table 16.18 (Continued) *The twenty-one problems of this chapter at a glance: what kind of problem each is, the tool that solves it and where the book taught it, and what the solution costs.*

Problem	Section	What it is	Main tool, and where it was taught	Cost of the solution
16.20 vertex cover	16.11.8	three-part guarantee	a matching lower-bounds the optimum	$O(V+E)$, ratio 2
16.21 Prim vs Kruskal	16.11.9	comparison	cut property; heap against union–find (Ch. 11, 14)	$\Theta(V^2)$; $O(E \lg V)$

What Table 16.18 **shows.** Every problem has an obvious approach that is wrong; every answer was checked twice; and a few ideas recur: relaxation in seven problems, a strengthened hypothesis in two, a lower bound on the optimum in two.

One idea, two chapters. Johnson's h and the amortized Φ both add a correction that telescopes: compare Equation (16.12) with Equation (16.7). Only the two ends survive.

Book: Section 16.12.

Quick Self-Check Drills

Chapters 1–4. The five phases? Understand, abstract, design, analyze, implement. The three obligations of a loop invariant? Initialization, maintenance, termination. Define $f(n) = O(g(n))$. The three Master Theorem cases.

Chapters 5–8. Why is `BUILD-MAX-HEAP` $\Theta(n)$? What does a randomized pivot buy? The comparison-sort lower bound, and why? What makes a problem fit dynamic programming?

Chapters 9–12. The two hypotheses of a greedy proof? Why is Huffman optimal? BFS and DFS differ in what, exactly? State the cut property. Prim or Kruskal?

Chapters 13–16. What is relaxation? Why does Dijkstra fail on negative edges? Why $|V| - 1$ passes? The condition on a potential? Why does Johnson's reweighting preserve shortest paths? The one question before writing any algorithm: *what is the shape of the input, and of the answer?*

The book lists forty drills with one-line answers; anything you miss names the chapter to reread. Book: Section 16.13.

The Algorithm Selection Table

Table 16.19 The algorithm selection table. Read the left column until a row matches the problem in front of you; the other three columns are the answer, the justification you will be asked for, and the budget you must fit.

The problem looks like this	Use	Why	Cost
Sort n items, memory is tight	Heapsort	In place, and $\Theta(n \lg n)$ with no bad case	$\Theta(n \lg n)$, $O(1)$ space
Sort n items, average speed matters most	Randomized quicksort	Small constants, good locality	$\Theta(n \lg n)$ expected
Sort n items, equal keys must keep their order	Merge sort	The only stable one of the three	$\Theta(n \lg n)$, $\Theta(n)$ space
Sort n small integers in a known range	Counting or radix sort	Not a comparison sort, so the $\Omega(n \lg n)$ floor does not apply	$\Theta(n+k)$
Search a static sorted array	Binary search	Halving is sound only because it is sorted	$\Theta(\lg n)$
Search with insertions and deletions, ordered traversal needed	AVL or red–black tree	Keeps order *and* $O(\lg n)$ updates	$O(\lg n)$
Search with insertions and deletions, order never needed	Hash table	Expected $\Theta(1)$; no order	$\Theta(1)$ expected
Repeatedly take the smallest or largest	Binary heap	`Extract-Min` in $\Theta(\lg n)$, `Build` in $\Theta(n)$	$\Theta(\lg n)$ per operation
Choose the most items that do not overlap	Greedy by earliest finishing time	Greedy-choice property holds; proved in Ch. 9	$\Theta(n \lg n)$

Table 16.19 (Continued) *The algorithm selection table. Read the left column until a row matches the problem in front of you; the other three columns are the answer, the justification you will be asked for, and the budget you must fit.*

The problem looks like this	Use	Why	Cost
Fill a capacity with divisible goods	Greedy by value density	Fractional knapsack; exchange argument	$\Theta(n \lg n)$
Fill a capacity with indivisible goods	Dynamic programming over capacity	Greedy fails; $c[i][j]$ recurrence	$\Theta(nW)$, pseudo-polynomial
Compress symbols with known frequencies	Huffman	Optimal prefix code	$O(n \lg n)$
Subproblems overlap, answer built from optimal pieces	Dynamic programming	Remember, do not recompute	problem-specific
Subproblems are independent	Divide and conquer	Nothing to remember	Master Theorem
Reach every vertex, fewest edges from one source	BFS	The queue gives non-decreasing distance	$\Theta(V+E)$
Detect a cycle, order tasks, find components	DFS	Discovery/finish times expose structure	$\Theta(V+E)$
Order tasks with prerequisites	Topological sort	Exists exactly for a DAG	$\Theta(V+E)$
Connect everything at least cost	Prim (dense) or Kruskal (sparse)	Cut property makes the greedy edge safe	$\Theta(V^2)$ / $O(E \lg V)$
Shortest path, one source, weights ≥ 0	Dijkstra	Greedy is safe when extending cannot shorten	$O(E \lg V)$

Table 16.19 (Continued) *The algorithm selection table. Read the left column until a row matches the problem in front of you; the other three columns are the answer, the justification you will be asked for, and the budget you must fit.*

The problem looks like this	Use	Why	Cost
Shortest path, one source, some weights negative	Bellman–Ford	Relaxes everything $\lvert V\rvert - 1$ times; detects negative cycles	$\Theta(V E)$
Shortest path, one source, graph is a DAG	Relax in topological order	One pass suffices; also solves *longest* path	$\Theta(V+E)$
Shortest path between every pair, dense graph	Floyd–Warshall	Three loops, $\Theta(V^2)$ space	$\Theta(V^3)$
Shortest path between every pair, sparse graph	Johnson	Reweight once, then V Dijkstras	$O(V^2 \lg V + V E)$
A sequence of operations, one of them occasionally expensive	Amortized analysis	The worst step is not the average step	—
The exact answer is NP-hard and n is large	An approximation with a proved ratio	A guarantee beats a heuristic	problem-specific
Something must be maintained as items arrive	A disjoint-set forest, a heap, or a balanced tree	Choose by the operation you do most	$O(\alpha(n))$ / $O(\lg n)$

Table 16.19 is the page to keep. **Rule 1:** the shape comes from the input and the required output, not from the words. **Rule 2:** a row is a starting point, not a verdict: measure before you commit.

"Connect the villages cheaply" and "wire a chip with the least copper" are one row; "visit every village once" is the NP-hard row. Book: Section 16.14.

Profitina in Practice

In the GEO platform: the crawl queue is a priority queue; page depth is a BFS distance, chosen so a budget-cut crawl has covered the shallow pages; the re-scan cadence is a priority queue and activity selection; the weekly score history is a doubling table.

Checking twice is a release gate: over a thousand automated tests must pass, and a release succeeds only after a health check. The platform *advises*; the customer *decides*.

Profitina Laundry's last night: three questions, three algorithms. Which press jobs fit one machine? Greedy activity selection. In what order should the van collect? MST plus preorder walk, with the ratio of Problem 16.4. How far is each neighborhood by road? Dijkstra.

The greedy schedule skips the long job J4 (08:45–10:30) and runs four jobs, provably optimal. The Manhattan grid and the road network give the same distances here only by coincidence: they are two different graphs.

The platform advises; the customer decides, and a score is an estimate, never a guarantee. Book: Sections 16.16 and 16.16.1.

Common Mistakes

Reading. Answering the statement you expected; an O bound without its case; subsequence confused with substring; computing a DP value but not the solution.

Proof. Weak induction where strong is needed; a step asserted without its hypothesis; a ratio bound without a lower bound on the optimum.

Shortest paths. Dijkstra treated as universal; extracting the wrong vertex; forgetting Bellman–Ford's closing check; topological order from *increasing* finish times; letting $\infty + w$ overflow; k in the wrong Floyd–Warshall loop; not undoing Johnson's reweighting; adding a constant to every edge.

Amortized and beyond. A potential that goes negative; amortized confused with average-case; pseudo-polynomial called polynomial; illustrating a graph instead of an algorithm; reading Table 16.19 as a verdict.

Book: Section 16.17.

Summary, and a Closing Word

Problems 16.1–16.5: nine true and six false; $L[i]$ *ending at* $S[i]$; $X(n) = n+1$ with k canceling; $d(L) \le d(W) = 2\,d(T) \le 2\,d(L^*)$; two algorithms that differ only in the order of relaxation.

Problems 16.6–16.12: Dijkstra, Bellman–Ford with its negative-cycle check and π walk, one DAG pass; counter $< 2n$ and table $< 3n$ by all three methods; Floyd–Warshall and Johnson reach the same matrix.

Problems 16.13–16.21 revisited the whole book: longest DAG paths, tree edges, Huffman, the all-pairs choice, pseudo-polynomial knapsack, vertex cover, Prim against Kruskal.

> **What you now have** is not a catalog of algorithms but the ability to **recognize the shape** of a problem you have never seen: Problem 16.4 came from a subject this book never taught, and everything you needed was already in your hands. Keep the habit of checking twice, and the arithmetic of consequence. Good luck, and stay awesome.

Book: Section 16.18, including "A closing word".

Where to Go Next

Network flow (Ford–Fulkerson, max-flow min-cut; BFS gives Edmonds–Karp) and **NP-completeness and approximation** (reductions; ratio bounds like Problems 16.4 and 16.20).

String matching (KMP in $\Theta(n+m)$, analyzed like the binary counter; suffix arrays) and **computational geometry** (closest pair by divide and conquer, Graham's scan, sweep lines).

Linear programming (Johnson's potentials are a dual solution) and **randomized and online algorithms** (bad input becomes bad luck; competitive ratios).

The habit of measuring: every asymptotic statement in this book can be checked on your own machine in ten minutes.

Homework. Exercises 1–14 of Chapter 16; start with 4 (another metric TSP instance), 12 (counter with `RESET`) and 14 (Johnson on Exercise 13's graph).

Book: Sections 16.15 and 16.19.